```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                  include/ansi.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/* The <ansi.h> header attempts to decide whether the compiler has enough
 * conformance to Standard C for Minix to take advantage of.  If so, the
 * symbol _ANSI is defined (as 31459).  Otherwise _ANSI is not defined
 * here, but it may be defined by applications that want to bend the rules.
 * The magic number in the definition is to inhibit unnecessary bending
 * of the rules.  (For consistency with the new '#ifdef _ANSI" tests in
 * the headers, _ANSI should really be defined as nothing, but that would
 * break many library routines that use "#if _ANSI".)

 * If _ANSI ends up being defined, a macro
 *
 *      _PROTOTYPE(function, params)
 *
 * is defined.  This macro expands in different ways, generating either
 * ANSI Standard C prototypes or old-style K&R (Kernighan & Ritchie)
 * prototypes, as needed.  Finally, some programs use _CONST, _VOIDSTAR etc
 * in such a way that they are portable over both ANSI and K&R compilers.
 * The appropriate macros are defined here.
 */

#ifndef _ANSI_H
#define _ANSI_H

#if __STDC__ == 1
#define _ANSI           31459   /* compiler claims full ANSI conformance */
#endif

#ifdef __GNUC__
#define _ANSI           31459   /* gcc conforms enough even in non-ANSI mode */
#endif

#ifdef _ANSI

/* Keep everything for ANSI prototypes. */
#define _PROTOTYPE(function, params)    function params
#define _ARGS(params)                   params

#define _VOIDSTAR       void *
#define _VOID           void
#define _CONST          const
#define _VOLATILE       volatile
#define _SIZET          size_t

#else

/* Throw away the parameters for K&R prototypes. */
#define _PROTOTYPE(function, params)    function()
#define _ARGS(params)                   ()

#define _VOIDSTAR       void *
#define _VOID           void
#define _CONST
#define _VOLATILE
#define _SIZET          int

```

```
#endif /* _ANSI */

/* This should be defined as restrict when a C99 compiler is used. */
#define _RESTRICT

/* Setting any of _MINIX, _POSIX_C_SOURCE or _POSIX2_SOURCE implies
 * _POSIX_SOURCE.  (Seems wrong to put this here in ANSI space.)
 */
#if defined(_MINIX) || _POSIX_C_SOURCE > 0 || defined(_POSIX2_SOURCE)
#undef _POSIX_SOURCE
#define _POSIX_SOURCE   1
#endif

#endif /* ANSI_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                  include/limits.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/* The <limits.h> header defines some basic sizes, both of the language types
 * (e.g., the number of bits in an integer), and of the operating system (e.g.
 * the number of characters in a file name.
 */

#ifndef _LIMITS_H
#define _LIMITS_H

/* Definitions about chars (8 bits in MINIX, and signed). */
#define CHAR_BIT           8    /* # bits in a char */
#define CHAR_MIN        -128    /* minimum value of a char */
#define CHAR_MAX         127    /* maximum value of a char */
#define SCHAR_MIN       -128    /* minimum value of a signed char */
#define SCHAR_MAX        127    /* maximum value of a signed char */
#define UCHAR_MAX        255    /* maximum value of an unsigned char */
#define MB_LEN_MAX         1    /* maximum length of a multibyte char */

/* Definitions about shorts (16 bits in MINIX). */
#define SHRT_MIN  (-32767-1)    /* minimum value of a short */
#define SHRT_MAX       32767    /* maximum value of a short */
#define USHRT_MAX     0xFFFF    /* maximum value of unsigned short */

/* _EM_WSIZE is a compiler-generated symbol giving the word size in bytes. */
#define INT_MIN (-2147483647-1) /* minimum value of a 32-bit int */
#define INT_MAX   2147483647    /* maximum value of a 32-bit int */
#define UINT_MAX  0xFFFFFFFF    /* maximum value of an unsigned 32-bit int */

/*Definitions about longs (32 bits in MINIX). */
#define LONG_MIN (-2147483647L-1)/* minimum value of a long */
#define LONG_MAX  2147483647L   /* maximum value of a long */
#define ULONG_MAX 0xFFFFFFFFL   /* maximum value of an unsigned long */

#include <sys/dir.h>

/* Minimum sizes required by the POSIX P1003.1 standard (Table 2-3). */
#ifdef _POSIX_SOURCE            /* these are only visible for POSIX */
#define _POSIX_ARG_MAX    4096  /* exec() may have 4K worth of args */
#define _POSIX_CHILD_MAX     6  /* a process may have 6 children */
#define _POSIX_LINK_MAX      8  /* a file may have 8 links */
#define _POSIX_MAX_CANON   255  /* size of the canonical input queue */
```

```
#define _POSIX_MAX_INPUT   255  /* you can type 255 chars ahead */
#define _POSIX_NAME_MAX DIRSIZ  /* a file name may have 14 chars */
#define _POSIX_NGROUPS_MAX   0  /* supplementary group IDs are optional */
#define _POSIX_OPEN_MAX     16  /* a process may have 16 files open */
#define _POSIX_PATH_MAX    255  /* a pathname may contain 255 chars */
#define _POSIX_PIPE_BUF    512  /* pipes writes of 512 bytes must be atomic */
#define _POSIX_STREAM_MAX    8  /* at least 8 FILEs can be open at once */
#define _POSIX_TZNAME_MAX    3  /* time zone names can be at least 3 chars */
#define _POSIX_SSIZE_MAX 32767  /* read() must support 32767 byte reads */

/* Values actually implemented by MINIX (Tables 2-4, 2-5, 2-6, and 2-7). */
/* Some of these old names had better be defined when not POSIX. */
#define _NO_LIMIT          100  /* arbitrary number; limit not enforced */

#define NGROUPS_MAX          0  /* supplemental group IDs not available */
#define ARG_MAX          16384  /* # bytes of args + environ for exec() */
#define CHILD_MAX    _NO_LIMIT  /* MINIX does not limit children */
#define OPEN_MAX            20  /* # open files a process may have */
#define LINK_MAX     SHRT_MAX   /* # links a file may have */
#define MAX_CANON          255  /* size of the canonical input queue */
#define MAX_INPUT          255  /* size of the type-ahead buffer */
#define NAME_MAX        DIRSIZ  /* # chars in a file name */
#define PATH_MAX           255  /* # chars in a path name */
#define PIPE_BUF          7168  /* # bytes in atomic write to a pipe */
#define STREAM_MAX          20  /* must be the same as FOPEN_MAX in stdio.h */
#define TZNAME_MAX           3  /* maximum bytes in a time zone name is 3 */
#define SSIZE_MAX        32767  /* max defined byte count for read() */

#endif /* _POSIX_SOURCE */

#endif /* _LIMITS_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                    include/errno.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/* The <errno.h> header defines the numbers of the various errors that can
 * occur during program execution.  They are visible to user programs and
 * should be small positive integers.  However, they are also used within
 * MINIX, where they must be negative.  For example, the READ system call is
 * executed internally by calling do_read().  This function returns either a
 * (negative) error number or a (positive) number of bytes actually read.
 *
 * To solve the problem of having the error numbers be negative inside the
 * the system and positive outside, the following mechanism is used.  All the
 * definitions are are the form:
 *
 *      #define EPERM           (_SIGN 1)
 *
 * If the macro _SYSTEM is defined, then  _SIGN is set to "-", otherwise it is
 * set to "".  Thus when compiling the operating system, the  macro _SYSTEM
 * will be defined, setting EPERM to (- 1), whereas when when this
 * file is included in an ordinary user program, EPERM has the value ( 1).
 */

#ifndef _ERRNO_H                /* check if <errno.h> is already included */
```

```
#define _ERRNO_H                /* it is not included; note that fact */

/* Now define _SIGN as "" or "-" depending on _SYSTEM. */
#ifdef _SYSTEM
#   define _SIGN         -
#   define OK            0
#else
#   define _SIGN
#endif

extern int errno;                 /* place where the error numbers go */

/* Here are the numerical values of the error numbers. */
#define _NERROR              70  /* number of errors */

#define EGENERIC      (_SIGN 99)  /* generic error */
#define EPERM         (_SIGN  1)  /* operation not permitted */
#define ENOENT        (_SIGN  2)  /* no such file or directory */
#define ESRCH         (_SIGN  3)  /* no such process */
#define EINTR         (_SIGN  4)  /* interrupted function call */
#define EIO           (_SIGN  5)  /* input/output error */
#define ENXIO         (_SIGN  6)  /* no such device or address */
#define E2BIG         (_SIGN  7)  /* arg list too long */
#define ENOEXEC       (_SIGN  8)  /* exec format error */
#define EBADF         (_SIGN  9)  /* bad file descriptor */
#define ECHILD        (_SIGN 10)  /* no child process */
#define EAGAIN        (_SIGN 11)  /* resource temporarily unavailable */
#define ENOMEM        (_SIGN 12)  /* not enough space */
#define EACCES        (_SIGN 13)  /* permission denied */
#define EFAULT        (_SIGN 14)  /* bad address */
#define ENOTBLK       (_SIGN 15)  /* Extension: not a block special file */
#define EBUSY         (_SIGN 16)  /* resource busy */
#define EEXIST        (_SIGN 17)  /* file exists */
#define EXDEV         (_SIGN 18)  /* improper link */
#define ENODEV        (_SIGN 19)  /* no such device */
#define ENOTDIR       (_SIGN 20)  /* not a directory */
#define EISDIR        (_SIGN 21)  /* is a directory */
#define EINVAL        (_SIGN 22)  /* invalid argument */
#define ENFILE        (_SIGN 23)  /* too many open files in system */
#define EMFILE        (_SIGN 24)  /* too many open files */
#define ENOTTY        (_SIGN 25)  /* inappropriate I/O control operation */
#define ETXTBSY       (_SIGN 26)  /* no longer used */
#define EFBIG         (_SIGN 27)  /* file too large */
#define ENOSPC        (_SIGN 28)  /* no space left on device */
#define ESPIPE        (_SIGN 29)  /* invalid seek */
#define EROFS         (_SIGN 30)  /* read-only file system */
#define EMLINK        (_SIGN 31)  /* too many links */
#define EPIPE         (_SIGN 32)  /* broken pipe */
#define EDOM          (_SIGN 33)  /* domain error       (from ANSI C std) */
#define ERANGE        (_SIGN 34)  /* result too large   (from ANSI C std) */
#define EDEADLK       (_SIGN 35)  /* resource deadlock avoided */
#define ENAMETOOLONG  (_SIGN 36)  /* file name too long */
#define ENOLCK        (_SIGN 37)  /* no locks available */
#define ENOSYS        (_SIGN 38)  /* function not implemented */
#define ENOTEMPTY     (_SIGN 39)  /* directory not empty */

/* The following errors relate to networking. */
#define EPACKSIZE     (_SIGN 50)  /* invalid packet size for some protocol */
#define EOUTOFBUFS    (_SIGN 51)  /* not enough buffers left */
#define EBADIOCTL     (_SIGN 52)  /* illegal ioctl for device */
```

```
#define EBADMODE      (_SIGN 53)  /* badmode in ioctl */
#define EWOULDBLOCK   (_SIGN 54)
#define EBADDEST      (_SIGN 55)  /* not a valid destination address */
#define EDSTNOTRCH    (_SIGN 56)  /* destination not reachable */
#define EISCONN       (_SIGN 57)  /* all ready connected */
#define EADDRINUSE    (_SIGN 58)  /* address in use */
#define ECONNREFUSED  (_SIGN 59)  /* connection refused */
#define ECONNRESET    (_SIGN 60)  /* connection reset */
#define ETIMEDOUT     (_SIGN 61)  /* connection timed out */
#define EURG          (_SIGN 62)  /* urgent data present */
#define ENOURG        (_SIGN 63)  /* no urgent data present */
#define ENOTCONN      (_SIGN 64)  /* no connection (yet or anymore) */
#define ESHUTDOWN     (_SIGN 65)  /* a write call to a shutdown connection */
#define ENOCONN       (_SIGN 66)  /* no such connection */
#define EAFNOSUPPORT  (_SIGN 67)  /* address family not supported */
#define EPROTONOSUPPORT (_SIGN 68) /* protocol not supported by AF */
#define EPROTOTYPE    (_SIGN 69)  /* Protocol wrong type for socket */
#define EINPROGRESS   (_SIGN 70)  /* Operation now in progress */
#define EADDRNOTAVAIL (_SIGN 71)  /* Can't assign requested address */
#define EALREADY      (_SIGN 72)  /* Connection already in progress */
#define EMSGSIZE      (_SIGN 73)  /* Message too long */

/* The following are not POSIX errors, but they can still happen.
 * All of these are generated by the kernel and relate to message passing.
 */
#define ELOCKED      (_SIGN 101)  /* can't send message due to deadlock */
#define EBADCALL     (_SIGN 102)  /* illegal system call number */
#define EBADSRCDST   (_SIGN 103)  /* bad source or destination process */
#define ECALLDENIED  (_SIGN 104)  /* no permission for system call */
#define EDEADDST     (_SIGN 105)  /* send destination is not alive */
#define ENOTREADY    (_SIGN 106)  /* source or destination is not ready */
#define EBADREQUEST  (_SIGN 107)  /* destination cannot handle request */
#define EDONTREPLY   (_SIGN 201)  /* pseudo-code: don't send a reply */

#endif /* _ERRNO_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                      include/unistd.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/* The <unistd.h> header contains a few miscellaneous manifest constants. */

#ifndef _UNISTD_H
#define _UNISTD_H

#ifndef _TYPES_H
#include <sys/types.h>
#endif

/* Values used by access().  POSIX Table 2-8. */
#define F_OK              0    /* test if file exists */
#define X_OK              1    /* test if file is executable */
#define W_OK              2    /* test if file is writable */
#define R_OK              4    /* test if file is readable */

/* Values used for whence in lseek(fd, offset, whence).  POSIX Table 2-9. */
#define SEEK_SET          0    /* offset is absolute  */
#define SEEK_CUR          1    /* offset is relative to current position */
#define SEEK_END          2    /* offset is relative to end of file */

```

```
/* This value is required by POSIX Table 2-10. */
#define _POSIX_VERSION 199009L  /* which standard is being conformed to */

/* These three definitions are required by POSIX Sec. 8.2.1.2. */
#define STDIN_FILENO       0    /* file descriptor for stdin */
#define STDOUT_FILENO      1    /* file descriptor for stdout */
#define STDERR_FILENO      2    /* file descriptor for stderr */

#ifdef _MINIX
/* How to exit the system or stop a server process. */
#define RBT_HALT           0
#define RBT_REBOOT         1
#define RBT_PANIC          2    /* a server panics */
#define RBT_MONITOR        3    /* let the monitor do this */
#define RBT_RESET          4    /* hard reset the system */
#endif

/* What system info to retrieve with sysgetinfo(). */
#define SI_KINFO           0    /* get kernel info via PM */
#define SI_PROC_ADDR       1    /* address of process table */
#define SI_PROC_TAB        2    /* copy of entire process table */
#define SI_DMAP_TAB        3    /* get device <-> driver mappings */

/* NULL must be defined in <unistd.h> according to POSIX Sec. 2.7.1. */
#define NULL    ((void *)0)

/* The following relate to configurable system variables. POSIX Table 4-2. */
#define _SC_ARG_MAX        1
#define _SC_CHILD_MAX      2
#define _SC_CLOCKS_PER_SEC 3
#define _SC_CLK_TCK        3
#define _SC_NGROUPS_MAX    4
#define _SC_OPEN_MAX       5
#define _SC_JOB_CONTROL    6
#define _SC_SAVED_IDS      7
#define _SC_VERSION        8
#define _SC_STREAM_MAX     9
#define _SC_TZNAME_MAX    10

/* The following relate to configurable pathname variables. POSIX Table 5-2. */
#define _PC_LINK_MAX       1    /* link count */
#define _PC_MAX_CANON      2    /* size of the canonical input queue */
#define _PC_MAX_INPUT      3    /* type-ahead buffer size */
#define _PC_NAME_MAX       4    /* file name size */
#define _PC_PATH_MAX       5    /* pathname size */
#define _PC_PIPE_BUF       6    /* pipe size */
#define _PC_NO_TRUNC       7    /* treatment of long name components */
#define _PC_VDISABLE       8    /* tty disable */
#define _PC_CHOWN_RESTRICTED 9  /* chown restricted or not */

/* POSIX defines several options that may be implemented or not, at the
 * implementer's whim.  This implementer has made the following choices:
 *
 * _POSIX_JOB_CONTROL       not defined:        no job control
 * _POSIX_SAVED_IDS         not defined:        no saved uid/gid
 * _POSIX_NO_TRUNC          defined as -1:      long path names are truncated
 * _POSIX_CHOWN_RESTRICTED  defined:            you can't give away files
 * _POSIX_VDISABLE          defined:            tty functions can be disabled
 */
#define _POSIX_NO_TRUNC       (-1)
```

```
#define _POSIX_CHOWN_RESTRICTED  1

/* Function Prototypes. */
_PROTOTYPE( void _exit, (int _status)                                   );
_PROTOTYPE( int access, (const char *_path, int _amode)                 );
_PROTOTYPE( unsigned int alarm, (unsigned int _seconds)                 );
_PROTOTYPE( int chdir, (const char *_path)                              );
_PROTOTYPE( int fchdir, (int fd)                                        );
_PROTOTYPE( int chown, (const char *_path, _mnx_Uid_t _owner, _mnx_Gid_t _group)        );
_PROTOTYPE( int close, (int _fd)                                        );
_PROTOTYPE( char *ctermid, (char *_s)                                   );
_PROTOTYPE( char *cuserid, (char *_s)                                   );
_PROTOTYPE( int dup, (int _fd)                                          );
_PROTOTYPE( int dup2, (int _fd, int _fd2)                               );
_PROTOTYPE( int execl, (const char *_path, const char *_arg, ...)       );
_PROTOTYPE( int execle, (const char *_path, const char *_arg, ...)      );
_PROTOTYPE( int execlp, (const char *_file, const char *arg, ...)       );
_PROTOTYPE( int execv, (const char *_path, char *const _argv[])         );
_PROTOTYPE( int execve, (const char *_path, char *const _argv[],
                                                char *const _envp[])    );
_PROTOTYPE( int execvp, (const char *_file, char *const _argv[])        );
_PROTOTYPE( pid_t fork, (void)                                          );
_PROTOTYPE( long fpathconf, (int _fd, int _name)                        );
_PROTOTYPE( char *getcwd, (char *_buf, size_t _size)                    );
_PROTOTYPE( gid_t getegid, (void)                                       );
_PROTOTYPE( uid_t geteuid, (void)                                       );
_PROTOTYPE( gid_t getgid, (void)                                        );
_PROTOTYPE( int getgroups, (int _gidsetsize, gid_t _grouplist[])        );
_PROTOTYPE( char *getlogin, (void)                                      );
_PROTOTYPE( pid_t getpgrp, (void)                                       );
_PROTOTYPE( pid_t getpid, (void)                                        );
_PROTOTYPE( pid_t getppid, (void)                                       );
_PROTOTYPE( uid_t getuid, (void)                                        );
_PROTOTYPE( int isatty, (int _fd)                                       );
_PROTOTYPE( int link, (const char *_existing, const char *_new)         );
_PROTOTYPE( off_t lseek, (int _fd, off_t _offset, int _whence)          );
_PROTOTYPE( long pathconf, (const char *_path, int _name)               );
_PROTOTYPE( int pause, (void)                                           );
_PROTOTYPE( int pipe, (int _fildes[2])                                  );
_PROTOTYPE( ssize_t read, (int _fd, void *_buf, size_t _n)              );
_PROTOTYPE( int rmdir, (const char *_path)                              );
_PROTOTYPE( int setgid, (_mnx_Gid_t _gid)                               );
_PROTOTYPE( int setpgid, (pid_t _pid, pid_t _pgid)                      );
_PROTOTYPE( pid_t setsid, (void)                                        );
_PROTOTYPE( int setuid, (_mnx_Uid_t _uid)                               );
_PROTOTYPE( unsigned int sleep, (unsigned int _seconds)                 );
_PROTOTYPE( long sysconf, (int _name)                                   );
_PROTOTYPE( pid_t tcgetpgrp, (int _fd)                                  );
_PROTOTYPE( int tcsetpgrp, (int _fd, pid_t _pgrp_id)                    );
_PROTOTYPE( char *ttyname, (int _fd)                                    );
_PROTOTYPE( int unlink, (const char *_path)                             );
_PROTOTYPE( ssize_t write, (int _fd, const void *_buf, size_t _n)       );

/* Open Group Base Specifications Issue 6 (not complete) */
_PROTOTYPE( int symlink, (const char *path1, const char *path2)         );
_PROTOTYPE( int getopt, (int _argc, char **_argv, char *_opts)          );
extern char *optarg;
extern int optind, opterr, optopt;
_PROTOTYPE( int usleep, (useconds_t _useconds)                          );

```

```
#ifdef _MINIX
#ifndef _TYPE_H
#include <minix/type.h>
#endif
_PROTOTYPE( int brk, (char *_addr)                                      );
_PROTOTYPE( int chroot, (const char *_name)                             );
_PROTOTYPE( int mknod, (const char *_name, _mnx_Mode_t _mode, Dev_t _addr)      );
_PROTOTYPE( int mknod4, (const char *_name, _mnx_Mode_t _mode, Dev_t _addr,
            long _size)                                                 );
_PROTOTYPE( char *mktemp, (char *_template)                             );
_PROTOTYPE( int mount, (char *_spec, char *_name, int _flag)            );
_PROTOTYPE( long ptrace, (int _req, pid_t _pid, long _addr, long _data) );
_PROTOTYPE( char *sbrk, (int _incr)                                     );
_PROTOTYPE( int sync, (void)                                            );
_PROTOTYPE( int fsync, (int fd)                                         );
_PROTOTYPE( int umount, (const char *_name)                             );
_PROTOTYPE( int reboot, (int _how, ...)                                 );
_PROTOTYPE( int gethostname, (char *_hostname, size_t _len)             );
_PROTOTYPE( int getdomainname, (char *_domain, size_t _len)             );
_PROTOTYPE( int ttyslot, (void)                                         );
_PROTOTYPE( int fttyslot, (int _fd)                                     );
_PROTOTYPE( char *crypt, (const char *_key, const char *_salt)          );
_PROTOTYPE( int getsysinfo, (int who, int what, void *where)            );
_PROTOTYPE( int getprocnr, (void)                                       );
_PROTOTYPE( int findproc, (char *proc_name, int *proc_nr)               );
_PROTOTYPE( int allocmem, (phys_bytes size, phys_bytes *base)           );
_PROTOTYPE( int freemem, (phys_bytes size, phys_bytes base)             );
#define DEV_MAP 1
#define DEV_UNMAP 2
#define mapdriver(driver, device, style) devctl(DEV_MAP, driver, device, style)
#define unmapdriver(device) devctl(DEV_UNMAP, 0, device, 0)
_PROTOTYPE( int devctl, (int ctl_req, int driver, int device, int style));

/* For compatibility with other Unix systems */
_PROTOTYPE( int getpagesize, (void)                                     );
_PROTOTYPE( int setgroups, (int ngroups, const gid_t *gidset)           );

#endif

_PROTOTYPE( int readlink, (const char *, char *, int));
_PROTOTYPE( int getopt, (int, char **, char *));
extern int optind, opterr, optopt;

#endif /* _UNISTD_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                 include/string.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/* The <string.h> header contains prototypes for the string handling
 * functions.
 */

#ifndef _STRING_H
#define _STRING_H

#define NULL    ((void *)0)

#ifndef _SIZE_T
```

```
#define _SIZE_T
typedef unsigned int size_t;    /* type returned by sizeof */
#endif /*_SIZE_T */

/* Function Prototypes. */
#ifndef _ANSI_H
#include <ansi.h>
#endif

_PROTOTYPE( void *memchr, (const void *_s, int _c, size_t _n)          );
_PROTOTYPE( int memcmp, (const void *_s1, const void *_s2, size_t _n)  );
_PROTOTYPE( void *memcpy, (void *_s1, const void *_s2, size_t _n)      );
_PROTOTYPE( void *memmove, (void *_s1, const void *_s2, size_t _n)     );
_PROTOTYPE( void *memset, (void *_s, int _c, size_t _n)                );
_PROTOTYPE( char *strcat, (char *_s1, const char *_s2)                 );
_PROTOTYPE( char *strchr, (const char *_s, int _c)                     );
_PROTOTYPE( int strncmp, (const char *_s1, const char *_s2, size_t _n) );
_PROTOTYPE( int strcmp, (const char *_s1, const char *_s2)             );
_PROTOTYPE( int strcoll, (const char *_s1, const char *_s2)            );
_PROTOTYPE( char *strcpy, (char *_s1, const char *_s2)                 );
_PROTOTYPE( size_t strcspn, (const char *_s1, const char *_s2)         );
_PROTOTYPE( char *strerror, (int _errnum)                              );
_PROTOTYPE( size_t strlen, (const char *_s)                            );
_PROTOTYPE( char *strncat, (char *_s1, const char *_s2, size_t _n)     );
_PROTOTYPE( char *strncpy, (char *_s1, const char *_s2, size_t _n)     );
_PROTOTYPE( char *strpbrk, (const char *_s1, const char *_s2)          );
_PROTOTYPE( char *strrchr, (const char *_s, int _c)                    );
_PROTOTYPE( size_t strspn, (const char *_s1, const char *_s2)          );
_PROTOTYPE( char *strstr, (const char *_s1, const char *_s2)           );
_PROTOTYPE( char *strtok, (char *_s1, const char *_s2)                 );
_PROTOTYPE( size_t strxfrm, (char *_s1, const char *_s2, size_t _n)    );

#ifdef _POSIX_SOURCE
/* Open Group Base Specifications Issue 6 (not complete) */
 char *strdup(const char *_s1);
#endif

#ifdef _MINIX
/* For backward compatibility. */
_PROTOTYPE( char *index, (const char *_s, int _charwanted)             );
_PROTOTYPE( char *rindex, (const char *_s, int _charwanted)            );
_PROTOTYPE( void bcopy, (const void *_src, void *_dst, size_t _length) );
_PROTOTYPE( int bcmp, (const void *_s1, const void *_s2, size_t _length));
_PROTOTYPE( void bzero, (void *_dst, size_t _length)                   );
_PROTOTYPE( void *memccpy, (char *_dst, const char *_src, int _ucharstop,
                                                    size_t _size)      );

/* Misc. extra functions */
_PROTOTYPE( int strcasecmp, (const char *_s1, const char *_s2)         );
_PROTOTYPE( int strncasecmp, (const char *_s1, const char *_s2,
                                                        size_t _len)   );
_PROTOTYPE( size_t strnlen, (const char *_s, size_t _n)                );
#endif

#endif /* _STRING_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                include/signal.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/* The <signal.h> header defines all the ANSI and POSIX signals.
 * MINIX supports all the signals required by POSIX. They are defined below.
 * Some additional signals are also supported.
 */

#ifndef _SIGNAL_H
#define _SIGNAL_H

#ifndef _ANSI_H
#include <ansi.h>
#endif
#ifdef _POSIX_SOURCE
#ifndef _TYPES_H
#include <sys/types.h>
#endif
#endif

/* Here are types that are closely associated with signal handling. */
typedef int sig_atomic_t;

#ifdef _POSIX_SOURCE
#ifndef _SIGSET_T
#define _SIGSET_T
typedef unsigned long sigset_t;
#endif
#endif

#define _NSIG             20    /* number of signals used */

#define SIGHUP             1    /* hangup */
#define SIGINT             2    /* interrupt (DEL) */
#define SIGQUIT            3    /* quit (ASCII FS) */
#define SIGILL             4    /* illegal instruction */
#define SIGTRAP            5    /* trace trap (not reset when caught) */
#define SIGABRT            6    /* IOT instruction */
#define SIGIOT             6    /* SIGABRT for people who speak PDP-11 */
#define SIGUNUSED          7    /* spare code */
#define SIGFPE             8    /* floating point exception */
#define SIGKILL            9    /* kill (cannot be caught or ignored) */
#define SIGUSR1           10    /* user defined signal # 1 */
#define SIGSEGV           11    /* segmentation violation */
#define SIGUSR2           12    /* user defined signal # 2 */
#define SIGPIPE           13    /* write on a pipe with no one to read it */
#define SIGALRM           14    /* alarm clock */
#define SIGTERM           15    /* software termination signal from kill */
#define SIGCHLD           17    /* child process terminated or stopped */

#define SIGEMT             7    /* obsolete */
#define SIGBUS            10    /* obsolete */

/* MINIX specific signals. These signals are not used by user proceses,
 * but meant to inform system processes, like the PM, about system events.
 */
#define SIGKMESS          18    /* new kernel message */
#define SIGKSIG           19    /* kernel signal pending */
```

```
#define SIGKSTOP          20    /* kernel shutting down */

/* POSIX requires the following signals to be defined, even if they are
 * not supported.  Here are the definitions, but they are not supported.
 */
#define SIGCONT           18    /* continue if stopped */
#define SIGSTOP           19    /* stop signal */
#define SIGTSTP           20    /* interactive stop signal */
#define SIGTTIN           21    /* background process wants to read */
#define SIGTTOU           22    /* background process wants to write */

/* The sighandler_t type is not allowed unless _POSIX_SOURCE is defined. */
typedef void _PROTOTYPE( (*__sighandler_t), (int) );

/* Macros used as function pointers. */
#define SIG_ERR    ((__sighandler_t) -1)        /* error return */
#define SIG_DFL    ((__sighandler_t)  0)        /* default signal handling */
#define SIG_IGN    ((__sighandler_t)  1)        /* ignore signal */
#define SIG_HOLD   ((__sighandler_t)  2)        /* block signal */
#define SIG_CATCH  ((__sighandler_t)  3)        /* catch signal */
#define SIG_MESS   ((__sighandler_t)  4)        /* pass as message (MINIX) */

#ifdef _POSIX_SOURCE
struct sigaction {
  __sighandler_t sa_handler;    /* SIG_DFL, SIG_IGN, or pointer to function */
  sigset_t sa_mask;             /* signals to be blocked during handler */
  int sa_flags;                 /* special flags */
};

/* Fields for sa_flags. */
#define SA_ONSTACK   0x0001     /* deliver signal on alternate stack */
#define SA_RESETHAND 0x0002     /* reset signal handler when signal caught */
#define SA_NODEFER   0x0004     /* don't block signal while catching it */
#define SA_RESTART   0x0008     /* automatic system call restart */
#define SA_SIGINFO   0x0010     /* extended signal handling */
#define SA_NOCLDWAIT 0x0020     /* don't create zombies */
#define SA_NOCLDSTOP 0x0040     /* don't receive SIGCHLD when child stops */

/* POSIX requires these values for use with sigprocmask(2). */
#define SIG_BLOCK          0    /* for blocking signals */
#define SIG_UNBLOCK        1    /* for unblocking signals */
#define SIG_SETMASK        2    /* for setting the signal mask */
#define SIG_INQUIRE        4    /* for internal use only */
#endif  /* _POSIX_SOURCE */

/* POSIX and ANSI function prototypes. */
_PROTOTYPE( int raise, (int _sig)                                        );
_PROTOTYPE( __sighandler_t signal, (int _sig, __sighandler_t _func)     );

#ifdef _POSIX_SOURCE
_PROTOTYPE( int kill, (pid_t _pid, int _sig)                            );
_PROTOTYPE( int sigaction,
    (int _sig, const struct sigaction *_act, struct sigaction *_oact)   );
_PROTOTYPE( int sigaddset, (sigset_t *_set, int _sig)                   );
_PROTOTYPE( int sigdelset, (sigset_t *_set, int _sig)                   );
_PROTOTYPE( int sigemptyset, (sigset_t *_set)                           );
_PROTOTYPE( int sigfillset, (sigset_t *_set)                            );
_PROTOTYPE( int sigismember, (const sigset_t *_set, int _sig)           );
_PROTOTYPE( int sigpending, (sigset_t *_set)                            );
_PROTOTYPE( int sigprocmask,
            (int _how, const sigset_t *_set, sigset_t *_oset)          );
_PROTOTYPE( int sigsuspend, (const sigset_t *_sigmask)                 );
#endif

#endif /* _SIGNAL_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                      include/fcntl.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/* The <fcntl.h> header is needed by the open() and fcntl() system calls,
 * which  have a variety of parameters and flags.  They are described here.
 * The formats of the calls to each of these are:
 *
 *      open(path, oflag [,mode])       open a file
 *      fcntl(fd, cmd [,arg])           get or set file attributes
 *
 */

#ifndef _FCNTL_H
#define _FCNTL_H

#ifndef _TYPES_H
#include <sys/types.h>
#endif

/* These values are used for cmd in fcntl().  POSIX Table 6-1.  */
#define F_DUPFD            0    /* duplicate file descriptor */
#define F_GETFD            1    /* get file descriptor flags */
#define F_SETFD            2    /* set file descriptor flags */
#define F_GETFL            3    /* get file status flags */
#define F_SETFL            4    /* set file status flags */
#define F_GETLK            5    /* get record locking information */
#define F_SETLK            6    /* set record locking information */
#define F_SETLKW           7    /* set record locking info; wait if blocked */

/* File descriptor flags used for fcntl().  POSIX Table 6-2. */
#define FD_CLOEXEC         1    /* close on exec flag for third arg of fcntl */

/* L_type values for record locking with fcntl().  POSIX Table 6-3. */
#define F_RDLCK            1    /* shared or read lock */
#define F_WRLCK            2    /* exclusive or write lock */
#define F_UNLCK            3    /* unlock */

/* Oflag values for open().  POSIX Table 6-4. */
#define O_CREAT        00100    /* creat file if it doesn't exist */
#define O_EXCL         00200    /* exclusive use flag */
#define O_NOCTTY       00400    /* do not assign a controlling terminal */
#define O_TRUNC        01000    /* truncate flag */

/* File status flags for open() and fcntl().  POSIX Table 6-5. */
#define O_APPEND       02000    /* set append mode */
#define O_NONBLOCK     04000    /* no delay */

/* File access modes for open() and fcntl().  POSIX Table 6-6. */
#define O_RDONLY           0    /* open(name, O_RDONLY) opens read only */
#define O_WRONLY           1    /* open(name, O_WRONLY) opens write only */
#define O_RDWR             2    /* open(name, O_RDWR) opens read/write */

/* Mask for use with file access modes.  POSIX Table 6-7. */
```

```
#define O_ACCMODE         03    /* mask for file access modes */

/* Struct used for locking.  POSIX Table 6-8. */
struct flock {
  short l_type;                 /* type: F_RDLCK, F_WRLCK, or F_UNLCK */
  short l_whence;               /* flag for starting offset */
  off_t l_start;                /* relative offset in bytes */
  off_t l_len;                  /* size; if 0, then until EOF */
  pid_t l_pid;                  /* process id of the locks' owner */
};

/* Function Prototypes. */
_PROTOTYPE( int creat, (const char *_path, _mnx_Mode_t _mode)           );
_PROTOTYPE( int fcntl, (int _filedes, int _cmd, ...)                    );
_PROTOTYPE( int open,  (const char *_path, int _oflag, ...)             );

#endif /* _FCNTL_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                include/termios.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/* The <termios.h> header is used for controlling tty modes. */

#ifndef _TERMIOS_H
#define _TERMIOS_H

typedef unsigned short tcflag_t;
typedef unsigned char cc_t;
typedef unsigned int speed_t;

#define NCCS               20   /* size of cc_c array, some extra space
                                 * for extensions. */

/* Primary terminal control structure. POSIX Table 7-1. */
struct termios {
  tcflag_t c_iflag;             /* input modes */
  tcflag_t c_oflag;             /* output modes */
  tcflag_t c_cflag;             /* control modes */
  tcflag_t c_lflag;             /* local modes */
  speed_t  c_ispeed;            /* input speed */
  speed_t  c_ospeed;            /* output speed */
  cc_t c_cc[NCCS];              /* control characters */
};

/* Values for termios c_iflag bit map.  POSIX Table 7-2. */
#define BRKINT          0x0001  /* signal interrupt on break */
#define ICRNL           0x0002  /* map CR to NL on input */
#define IGNBRK          0x0004  /* ignore break */
#define IGNCR           0x0008  /* ignore CR */
#define IGNPAR          0x0010  /* ignore characters with parity errors */
#define INLCR           0x0020  /* map NL to CR on input */
#define INPCK           0x0040  /* enable input parity check */
#define ISTRIP          0x0080  /* mask off 8th bit */
#define IXOFF           0x0100  /* enable start/stop input control */
#define IXON            0x0200  /* enable start/stop output control */
#define PARMRK          0x0400  /* mark parity errors in the input queue */
```

```

/* Values for termios c_oflag bit map.  POSIX Sec. 7.1.2.3. */
#define OPOST           0x0001  /* perform output processing */

/* Values for termios c_cflag bit map.  POSIX Table 7-3. */
#define CLOCAL          0x0001  /* ignore modem status lines */
#define CREAD           0x0002  /* enable receiver */
#define CSIZE           0x000C  /* number of bits per character */
#define         CS5     0x0000  /* if CSIZE is CS5, characters are 5 bits */
#define         CS6     0x0004  /* if CSIZE is CS6, characters are 6 bits */
#define         CS7     0x0008  /* if CSIZE is CS7, characters are 7 bits */
#define         CS8     0x000C  /* if CSIZE is CS8, characters are 8 bits */
#define CSTOPB          0x0010  /* send 2 stop bits if set, else 1 */
#define HUPCL           0x0020  /* hang up on last close */
#define PARENB          0x0040  /* enable parity on output */
#define PARODD          0x0080  /* use odd parity if set, else even */

/* Values for termios c_lflag bit map.  POSIX Table 7-4. */
#define ECHO            0x0001  /* enable echoing of input characters */
#define ECHOE           0x0002  /* echo ERASE as backspace */
#define ECHOK           0x0004  /* echo KILL */
#define ECHONL          0x0008  /* echo NL */
#define ICANON          0x0010  /* canonical input (erase and kill enabled) */
#define IEXTEN          0x0020  /* enable extended functions */
#define ISIG            0x0040  /* enable signals */
#define NOFLSH          0x0080  /* disable flush after interrupt or quit */
#define TOSTOP          0x0100  /* send SIGTTOU (job control, not implemented*/

/* Indices into c_cc array.  Default values in parentheses. POSIX Table 7-5. */
#define VEOF                 0  /* cc_c[VEOF] = EOF char (^D) */
#define VEOL                 1  /* cc_c[VEOL] = EOL char (undef) */
#define VERASE               2  /* cc_c[VERASE] = ERASE char (^H) */
#define VINTR                3  /* cc_c[VINTR] = INTR char (DEL) */
#define VKILL                4  /* cc_c[VKILL] = KILL char (^U) */
#define VMIN                 5  /* cc_c[VMIN] = MIN value for timer */
#define VQUIT                6  /* cc_c[VQUIT] = QUIT char (^\) */
#define VTIME                7  /* cc_c[VTIME] = TIME value for timer */
#define VSUSP                8  /* cc_c[VSUSP] = SUSP (^Z, ignored) */
#define VSTART               9  /* cc_c[VSTART] = START char (^S) */
#define VSTOP               10  /* cc_c[VSTOP] = STOP char (^Q) */

#define _POSIX_VDISABLE   (cc_t)0xFF    /* You can't even generate this
                                         * character with 'normal' keyboards.
                                         * But some language specific keyboards
                                         * can generate 0xFF. It seems that all
                                         * 256 are used, so cc_t should be a
                                         * short...
                                         */

/* Values for the baud rate settings.  POSIX Table 7-6. */
#define B0              0x0000  /* hang up the line */
#define B50             0x1000  /* 50 baud */
#define B75             0x2000  /* 75 baud */
#define B110            0x3000  /* 110 baud */
#define B134            0x4000  /* 134.5 baud */
#define B150            0x5000  /* 150 baud */
#define B200            0x6000  /* 200 baud */
#define B300            0x7000  /* 300 baud */
#define B600            0x8000  /* 600 baud */
#define B1200           0x9000  /* 1200 baud */
#define B1800           0xA000  /* 1800 baud */
#define B2400           0xB000  /* 2400 baud */
#define B4800           0xC000  /* 4800 baud */
#define B9600           0xD000  /* 9600 baud */
#define B19200          0xE000  /* 19200 baud */
#define B38400          0xF000  /* 38400 baud */

/* Optional actions for tcsetattr().  POSIX Sec. 7.2.1.2. */
#define TCSANOW            1    /* changes take effect immediately */
#define TCSADRAIN          2    /* changes take effect after output is done */
#define TCSAFLUSH          3    /* wait for output to finish and flush input */

/* Queue_selector values for tcflush().  POSIX Sec. 7.2.2.2. */
#define TCIFLUSH           1    /* flush accumulated input data */
#define TCOFLUSH           2    /* flush accumulated output data */
#define TCIOFLUSH          3    /* flush accumulated input and output data */

/* Action values for tcflow().  POSIX Sec. 7.2.2.2. */
#define TCOOFF             1    /* suspend output */
#define TCOON              2    /* restart suspended output */
#define TCIOFF             3    /* transmit a STOP character on the line */
#define TCION              4    /* transmit a START character on the line */

/* Function Prototypes. */
#ifndef _ANSI_H
#include <ansi.h>
#endif

_PROTOTYPE( int tcsendbreak, (int _fildes, int _duration)                     );
_PROTOTYPE( int tcdrain, (int _filedes)                                       );
_PROTOTYPE( int tcflush, (int _filedes, int _queue_selector)                  );
_PROTOTYPE( int tcflow, (int _filedes, int _action)                           );
_PROTOTYPE( speed_t cfgetispeed, (const struct termios *_termios_p)           );
_PROTOTYPE( speed_t cfgetospeed, (const struct termios *_termios_p)           );
_PROTOTYPE( int cfsetispeed, (struct termios *_termios_p, speed_t _speed)     );
_PROTOTYPE( int cfsetospeed, (struct termios *_termios_p, speed_t _speed)     );
_PROTOTYPE( int tcgetattr, (int _filedes, struct termios *_termios_p)         );
_PROTOTYPE( int tcsetattr, \
        (int _filedes, int _opt_actions, const struct termios *_termios_p)   );

#define cfgetispeed(termios_p)          ((termios_p)->c_ispeed)
#define cfgetospeed(termios_p)          ((termios_p)->c_ospeed)
#define cfsetispeed(termios_p, speed)   ((termios_p)->c_ispeed = (speed), 0)
#define cfsetospeed(termios_p, speed)   ((termios_p)->c_ospeed = (speed), 0)

#ifdef _MINIX
/* Here are the local extensions to the POSIX standard for Minix. Posix
 * conforming programs are not able to access these, and therefore they are
 * only defined when a Minix program is compiled.
 */

/* Extensions to the termios c_iflag bit map.  */
#define IXANY           0x0800  /* allow any key to continue ouptut */

/* Extensions to the termios c_oflag bit map. They are only active iff
 * OPOST is enabled. */
#define ONLCR           0x0002  /* Map NL to CR-NL on output */
#define XTABS           0x0004  /* Expand tabs to spaces */
#define ONOEOT          0x0008  /* discard EOT's (^D) on output) */

```

```
/* Extensions to the termios c_lflag bit map.  */
#define LFLUSHO         0x0200  /* Flush output. */

/* Extensions to the c_cc array. */
#define VREPRINT          11    /* cc_c[VREPRINT] (ˆR) */
#define VLNEXT            12    /* cc_c[VLNEXT] (ˆV) */
#define VDISCARD          13    /* cc_c[VDISCARD] (ˆO) */

/* Extensions to baud rate settings. */
#define B57600          0x0100  /* 57600 baud */
#define B115200         0x0200  /* 115200 baud */

/* These are the default settings used by the kernel and by 'stty sane' */

#define TCTRL_DEF       (CREAD | CS8 | HUPCL)
#define TINPUT_DEF      (BRKINT | ICRNL | IXON | IXANY)
#define TOUTPUT_DEF     (OPOST | ONLCR)
#define TLOCAL_DEF      (ISIG | IEXTEN | ICANON | ECHO | ECHOE)
#define TSPEED_DEF      B9600

#define TEOF_DEF        '\4'    /* ˆD */
#define TEOL_DEF        _POSIX_VDISABLE
#define TERASE_DEF      '\10'   /* ˆH */
#define TINTR_DEF       '\3'    /* ˆC */
#define TKILL_DEF       '\25'   /* ˆU */
#define TMIN_DEF        1
#define TQUIT_DEF       '\34'   /* ˆ\ */
#define TSTART_DEF      '\21'   /* ˆQ */
#define TSTOP_DEF       '\23'   /* ˆS */
#define TSUSP_DEF       '\32'   /* ˆZ */
#define TTIME_DEF       0
#define TREPRINT_DEF    '\22'   /* ˆR */
#define TLNEXT_DEF      '\26'   /* ˆV */
#define TDISCARD_DEF    '\17'   /* ˆO */

/* Window size. This information is stored in the TTY driver but not used.
 * This can be used for screen based applications in a window environment.
 * The ioctls TIOCGWINSZ and TIOCSWINSZ can be used to get and set this
 * information.
 */

struct winsize
{
        unsigned short  ws_row;         /* rows, in characters */
        unsigned short  ws_col;         /* columns, in characters */
        unsigned short  ws_xpixel;      /* horizontal size, pixels */
        unsigned short  ws_ypixel;      /* vertical size, pixels */
};
#endif /* _MINIX */

#endif /* _TERMIOS_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                  include/timers.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

/* This library provides generic watchdog timer management functionality.
 * The functions operate on a timer queue provided by the caller. Note that
 * the timers must use absolute time to allow sorting. The library provides:
 *
 *    tmrs_settimer:     (re)set a new watchdog timer in the timers queue
 *    tmrs_clrtimer:     remove a timer from both the timers queue
 *    tmrs_exptimers:    check for expired timers and run watchdog functions
 *
 * Author:
 *    Jorrit N. Herder <jnherder@cs.vu.nl>
 *    Adapted from tmr_settimer and tmr_clrtimer in src/kernel/clock.c.
 *    Last modified: September 30, 2004.
 */

#ifndef _TIMERS_H
#define _TIMERS_H

#include <limits.h>
#include <sys/types.h>

struct timer;
typedef void (*tmr_func_t)(struct timer *tp);
typedef union { int ta_int; long ta_long; void *ta_ptr; } tmr_arg_t;

/* A timer_t variable must be declare for each distinct timer to be used.
 * The timers watchdog function and expiration time are automatically set
 * by the library function tmrs_settimer, but its argument is not.
 */
typedef struct timer
{
  struct timer  *tmr_next;      /* next in a timer chain */
  clock_t       tmr_exp_time;   /* expiration time */
  tmr_func_t    tmr_func;       /* function to call when expired */
  tmr_arg_t     tmr_arg;        /* random argument */
} timer_t;

/* Used when the timer is not active. */
#define TMR_NEVER    ((clock_t) -1 < 0) ? ((clock_t) LONG_MAX) : ((clock_t) -1)
#undef TMR_NEVER
#define TMR_NEVER       ((clock_t) LONG_MAX)

/* These definitions can be used to set or get data from a timer variable. */
#define tmr_arg(tp) (&(tp)->tmr_arg)
#define tmr_exp_time(tp) (&(tp)->tmr_exp_time)

/* Timers should be initialized once before they are being used. Be careful
 * not to reinitialize a timer that is in a list of timers, or the chain
 * will be broken.
 */
#define tmr_inittimer(tp) (void)((tp)->tmr_exp_time = TMR_NEVER, \
        (tp)->tmr_next = NULL)

/* The following generic timer management functions are available. They
 * can be used to operate on the lists of timers. Adding a timer to a list
 * automatically takes care of removing it.
 */
_PROTOTYPE( clock_t tmrs_clrtimer, (timer_t **tmrs, timer_t *tp, clock_t *new_head)            );
_PROTOTYPE( void tmrs_exptimers, (timer_t **tmrs, clock_t now, clock_t *new_head)              );
_PROTOTYPE( clock_t tmrs_settimer, (timer_t **tmrs, timer_t *tp,
        clock_t exp_time, tmr_func_t watchdog, clock_t *new_head)                              );

#endif /* _TIMERS_H */

```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                    include/sys/types.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/* The <sys/types.h> header contains important data type definitions.
 * It is considered good programming practice to use these definitions,
 * instead of the underlying base type.  By convention, all type names end
 * with _t.
 */

#ifndef _TYPES_H
#define _TYPES_H

#ifndef _ANSI_H
#include <ansi.h>
#endif

/* The type size_t holds all results of the sizeof operator.  At first glance,
 * it seems obvious that it should be an unsigned int, but this is not always
 * the case. For example, MINIX-ST (68000) has 32-bit pointers and 16-bit
 * integers. When one asks for the size of a 70K struct or array, the result
 * requires 17 bits to express, so size_t must be a long type.  The type
 * ssize_t is the signed version of size_t.
 */
#ifndef _SIZE_T
#define _SIZE_T
typedef unsigned int size_t;
#endif

#ifndef _SSIZE_T
#define _SSIZE_T
typedef int ssize_t;
#endif

#ifndef _TIME_T
#define _TIME_T
typedef long time_t;               /* time in sec since 1 Jan 1970 0000 GMT */
#endif

#ifndef _CLOCK_T
#define _CLOCK_T
typedef long clock_t;              /* unit for system accounting */
#endif

#ifndef _SIGSET_T
#define _SIGSET_T
typedef unsigned long sigset_t;
#endif

```

```
/* Open Group Base Specifications Issue 6 (not complete) */
typedef long useconds_t;        /* Time in microseconds */

/* Types used in disk, inode, etc. data structures. */
typedef short          dev_t;      /* holds (major|minor) device pair */
typedef char           gid_t;      /* group id */
typedef unsigned long  ino_t;      /* i-node number (V3 filesystem) */
typedef unsigned short mode_t;     /* file type and permissions bits */
typedef short        nlink_t;      /* number of links to a file */
typedef unsigned long  off_t;      /* offset within a file */
typedef int            pid_t;      /* process id (must be signed) */
typedef short          uid_t;      /* user id */
typedef unsigned long zone_t;      /* zone number */
typedef unsigned long block_t;     /* block number */
typedef unsigned long  bit_t;      /* bit number in a bit map */
typedef unsigned short zone1_t;    /* zone number for V1 file systems */
typedef unsigned short bitchunk_t; /* collection of bits in a bitmap */

typedef unsigned char   u8_t;      /* 8 bit type */
typedef unsigned short u16_t;      /* 16 bit type */
typedef unsigned long  u32_t;      /* 32 bit type */

typedef char            i8_t;      /* 8 bit signed type */
typedef short          i16_t;      /* 16 bit signed type */
typedef long           i32_t;      /* 32 bit signed type */

typedef struct { u32_t _[2]; } u64_t;

/* The following types are needed because MINIX uses K&R style function
 * definitions (for maximum portability).  When a short, such as dev_t, is
 * passed to a function with a K&R definition, the compiler automatically
 * promotes it to an int.  The prototype must contain an int as the parameter,
 * not a short, because an int is what an old-style function definition
 * expects.  Thus using dev_t in a prototype would be incorrect.  It would be
 * sufficient to just use int instead of dev_t in the prototypes, but Dev_t
 * is clearer.
 */
typedef int            Dev_t;
typedef int       _mnx_Gid_t;
typedef int          Nlink_t;
typedef int       _mnx_Uid_t;
typedef int             U8_t;
typedef unsigned long  U32_t;
typedef int             I8_t;
typedef int            I16_t;
typedef long           I32_t;

/* ANSI C makes writing down the promotion of unsigned types very messy.  When
 * sizeof(short) == sizeof(int), there is no promotion, so the type stays
 * unsigned.  When the compiler is not ANSI, there is usually no loss of
 * unsignedness, and there are usually no prototypes so the promoted type
 * doesn't matter.  The use of types like Ino_t is an attempt to use ints
 * (which are not promoted) while providing information to the reader.
 */

typedef unsigned long  Ino_t;

#if _EM_WSIZE == 2
/*typedef unsigned int      Ino_t; Ino_t is now 32 bits */
typedef unsigned int    Zone1_t;
typedef unsigned int Bitchunk_t;
typedef unsigned int      U16_t;
typedef unsigned int  _mnx_Mode_t;

#else /* _EM_WSIZE == 4, or _EM_WSIZE undefined */
/*typedef int             Ino_t; Ino_t is now 32 bits */
typedef int             Zone1_t;
typedef int          Bitchunk_t;
typedef int               U16_t;
typedef int         _mnx_Mode_t;

#endif /* _EM_WSIZE == 2, etc */

/* Signal handler type, e.g. SIG_IGN */
typedef void _PROTOTYPE( (*sighandler_t), (int) );

/* Compatibility with other systems */
typedef unsigned char   u_char;
typedef unsigned short  u_short;
typedef unsigned int    u_int;
typedef unsigned long   u_long;
typedef char            *caddr_t;

#endif /* _TYPES_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                      include/sys/sigcontext.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
#ifndef _SIGCONTEXT_H
#define _SIGCONTEXT_H

/* The sigcontext structure is used by the sigreturn(2) system call.
 * sigreturn() is seldom called by user programs, but it is used internally
 * by the signal catching mechanism.
 */

#ifndef _ANSI_H
#include <ansi.h>
#endif

#ifndef _MINIX_SYS_CONFIG_H
#include <minix/sys_config.h>
#endif

#if !defined(_MINIX_CHIP)
#include "error, configuration is not known"
#endif

/* The following structure should match the stackframe_s structure used
 * by the kernel's context switching code.  Floating point registers should
 * be added in a different struct.
 */
struct sigregs {
  short sr_gs;
  short sr_fs;
  short sr_es;
  short sr_ds;
  int sr_di;
  int sr_si;
  int sr_bp;
  int sr_st;                   /* stack top -- used in kernel */
  int sr_bx;
  int sr_dx;
  int sr_cx;
  int sr_retreg;
  int sr_retadr;               /* return address to caller of save -- used
                                * in kernel */
  int sr_pc;
  int sr_cs;
  int sr_psw;
  int sr_sp;
  int sr_ss;
};

struct sigframe {              /* stack frame created for signalled process */
  _PROTOTYPE( void (*sf_retadr), (void) );
  int sf_signo;
  int sf_code;
  struct sigcontext *sf_scp;
  int sf_fp;
  _PROTOTYPE( void (*sf_retadr2), (void) );
  struct sigcontext *sf_scpcopy;
};

struct sigcontext {
  int sc_flags;                /* sigstack state to restore */
  long sc_mask;                /* signal mask to restore */
  struct sigregs sc_regs;      /* register set to restore */
};

#define sc_gs sc_regs.sr_gs
#define sc_fs sc_regs.sr_fs
#define sc_es sc_regs.sr_es
#define sc_ds sc_regs.sr_ds
#define sc_di sc_regs.sr_di
#define sc_si sc_regs.sr_si
#define sc_fp sc_regs.sr_bp
#define sc_st sc_regs.sr_st            /* stack top -- used in kernel */
#define sc_bx sc_regs.sr_bx
#define sc_dx sc_regs.sr_dx
#define sc_cx sc_regs.sr_cx
#define sc_retreg sc_regs.sr_retreg
#define sc_retadr sc_regs.sr_retadr    /* return address to caller of
                                       save -- used in kernel */
#define sc_pc sc_regs.sr_pc
#define sc_cs sc_regs.sr_cs
#define sc_psw sc_regs.sr_psw
#define sc_sp sc_regs.sr_sp
#define sc_ss sc_regs.sr_ss

/* Values for sc_flags.  Must agree with <minix/jmp_buf.h>. */
#define SC_SIGCONTEXT   2      /* nonzero when signal context is included */
#define SC_NOREGLOCALS  4      /* nonzero when registers are not to be
                                       saved and restored */

_PROTOTYPE( int sigreturn, (struct sigcontext *_scp)                   );

#endif /* _SIGCONTEXT_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                      include/sys/stat.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

/* The <sys/stat.h> header defines a struct that is used in the stat() and
 * fstat functions.  The information in this struct comes from the i-node of
 * some file.  These calls are the only approved way to inspect i-nodes.
 */

#ifndef _STAT_H
#define _STAT_H

#ifndef _TYPES_H
#include <sys/types.h>
#endif

struct stat {
  dev_t st_dev;                 /* major/minor device number */
  ino_t st_ino;                 /* i-node number */
  mode_t st_mode;               /* file mode, protection bits, etc. */
  short int st_nlink;           /* # links; TEMPORARY HACK: should be nlink_t*/
  uid_t st_uid;                 /* uid of the file's owner */
  short int st_gid;             /* gid; TEMPORARY HACK: should be gid_t */
  dev_t st_rdev;
  off_t st_size;                /* file size */
  time_t st_atime;              /* time of last access */
  time_t st_mtime;              /* time of last data modification */
  time_t st_ctime;              /* time of last file status change */
};

/* Traditional mask definitions for st_mode. */
/* The ugly casts on only some of the definitions are to avoid suprising sign
 * extensions such as S_IFREG != (mode_t) S_IFREG when ints are 32 bits.
 */
#define S_IFMT  ((mode_t) 0170000)      /* type of file */
#define S_IFLNK ((mode_t) 0120000)      /* symbolic link, not implemented */
#define S_IFREG ((mode_t) 0100000)      /* regular */
#define S_IFBLK 0060000         /* block special */
#define S_IFDIR 0040000         /* directory */
#define S_IFCHR 0020000         /* character special */
#define S_IFIFO 0010000         /* this is a FIFO */
#define S_ISUID 0004000         /* set user id on execution */
#define S_ISGID 0002000         /* set group id on execution */
                                /* next is reserved for future use */
#define S_ISVTX   01000         /* save swapped text even after use */

/* POSIX masks for st_mode. */
#define S_IRWXU   00700         /* owner:  rwx------ */
#define S_IRUSR   00400         /* owner:  r-------- */
#define S_IWUSR   00200         /* owner:  -w------- */
#define S_IXUSR   00100         /* owner:  --x------ */

#define S_IRWXG   00070         /* group:  ---rwx--- */
#define S_IRGRP   00040         /* group:  ---r----- */
#define S_IWGRP   00020         /* group:  ----w---- */
#define S_IXGRP   00010         /* group:  -----x--- */

#define S_IRWXO   00007         /* others: ------rwx */
#define S_IROTH   00004         /* others: ------r-- */
```

```
#define S_IWOTH   00002         /* others: -------w- */
#define S_IXOTH   00001         /* others: --------x */

/* The following macros test st_mode (from POSIX Sec. 5.6.1.1). */
#define S_ISREG(m)      (((m) & S_IFMT) == S_IFREG)     /* is a reg file */
#define S_ISDIR(m)      (((m) & S_IFMT) == S_IFDIR)     /* is a directory */
#define S_ISCHR(m)      (((m) & S_IFMT) == S_IFCHR)     /* is a char spec */
#define S_ISBLK(m)      (((m) & S_IFMT) == S_IFBLK)     /* is a block spec */
#define S_ISFIFO(m)     (((m) & S_IFMT) == S_IFIFO)     /* is a pipe/FIFO */
#define S_ISLNK(m)      (((m) & S_IFMT) == S_IFLNK)     /* is a sym link */

/* Function Prototypes. */
_PROTOTYPE( int chmod, (const char *_path, _mnx_Mode_t _mode)           );
_PROTOTYPE( int fstat, (int _fildes, struct stat *_buf)                 );
_PROTOTYPE( int mkdir, (const char *_path, _mnx_Mode_t _mode)           );
_PROTOTYPE( int mkfifo, (const char *_path, _mnx_Mode_t _mode)          );
_PROTOTYPE( int stat, (const char *_path, struct stat *_buf)            );
_PROTOTYPE( mode_t umask, (_mnx_Mode_t _cmask)                          );

/* Open Group Base Specifications Issue 6 (not complete) */
_PROTOTYPE( int lstat, (const char *_path, struct stat *_buf)           );

#endif /* _STAT_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                include/sys/dir.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/* The <dir.h> header gives the layout of a directory. */

#ifndef _DIR_H
#define _DIR_H

#include <sys/types.h>

#define DIRBLKSIZ       512     /* size of directory block */

#ifndef DIRSIZ
#define DIRSIZ  60
#endif

struct direct {
  ino_t d_ino;
  char d_name[DIRSIZ];
};

#endif /* _DIR_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                include/sys/wait.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/* The <sys/wait.h> header contains macros related to wait(). The value
 * returned by wait() and waitpid() depends on whether the process
 * terminated by an exit() call, was killed by a signal, or was stopped
 * due to job control, as follows:
 *
 *                              High byte   Low byte
 *                             +---------------------+
 *      exit(status)           |  status  |    0     |
 *                             +---------------------+
 *      killed by signal       |    0     |  signal  |
 *                             +---------------------+
 *      stopped (job control)  |  signal  |   0177   |
 *                             +---------------------+
 */

#ifndef _WAIT_H
#define _WAIT_H

#ifndef _TYPES_H
#include <sys/types.h>
#endif

#define _LOW(v)         ( (v) & 0377)
#define _HIGH(v)        ( ((v) >> 8) & 0377)

#define WNOHANG         1       /* do not wait for child to exit */
#define WUNTRACED       2       /* for job control; not implemented */

#define WIFEXITED(s)    (_LOW(s) == 0)                      /* normal exit */
#define WEXITSTATUS(s)  (_HIGH(s))                          /* exit status */
#define WTERMSIG(s)     (_LOW(s) & 0177)                    /* sig value */
#define WIFSIGNALED(s)  (((unsigned int)(s)-1 & 0xFFFF) < 0xFF) /* signaled */
#define WIFSTOPPED(s)   (_LOW(s) == 0177)                   /* stopped */
#define WSTOPSIG(s)     (_HIGH(s) & 0377)                   /* stop signal */

/* Function Prototypes. */
_PROTOTYPE( pid_t wait, (int *_stat_loc)                                   );
_PROTOTYPE( pid_t waitpid, (pid_t _pid, int *_stat_loc, int _options)      );

#endif /* _WAIT_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                include/sys/ioctl.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/*      sys/ioctl.h - All ioctl() command codes.        Author: Kees J. Bot
 *                                                              23 Nov 2002
 *
 * This header file includes all other ioctl command code headers.
 */

#ifndef _S_IOCTL_H
#define _S_IOCTL_H

/* A driver that uses ioctls claims a character for its series of commands.
 * For instance:  #define TCGETS  _IOR('T',  8, struct termios)
 * This is a terminal ioctl that uses the character 'T'.  The character(s)
 * used in each header file are shown in the comment following.
 */

#include <sys/ioc_tty.h>        /* 'T' 't' 'k'          */
#include <sys/ioc_disk.h>       /* 'd'                  */
#include <sys/ioc_memory.h>     /* 'm'                  */
#include <sys/ioc_cmos.h>       /* 'c'                  */

```

```
#endif /* _S_IOCTL_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                      include/sys/ioc_disk.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/*      sys/ioc_disk.h - Disk ioctl() command codes.    Author: Kees J. Bot
 *                                                              23 Nov 2002
 *
 */

#ifndef _S_I_DISK_H
#define _S_I_DISK_H

#include <minix/ioctl.h>

#define DIOCSETP        _IOW('d', 3, struct partition)
#define DIOCGETP        _IOR('d', 4, struct partition)
#define DIOCEJECT       _IO ('d', 5)
#define DIOCTIMEOUT     _IOW('d', 6, int)
#define DIOCOPENCT      _IOR('d', 7, int)

#endif /* _S_I_DISK_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                      include/minix/ioctl.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/*      minix/ioctl.h - Ioctl helper definitions.       Author: Kees J. Bot
 *                                                              23 Nov 2002
 *
 * This file is included by every header file that defines ioctl codes.
 */

#ifndef _M_IOCTL_H
#define _M_IOCTL_H

#ifndef _TYPES_H
#include <sys/types.h>
#endif

#if _EM_WSIZE >= 4
/* Ioctls have the command encoded in the low-order word, and the size
 * of the parameter in the high-order word. The 3 high bits of the high-
 * order word are used to encode the in/out/void status of the parameter.
 */
#define _IOCPARM_MASK   0x1FFF
#define _IOC_VOID       0x20000000
#define _IOCTYPE_MASK   0xFFFF
#define _IOC_IN         0x40000000
#define _IOC_OUT        0x80000000
#define _IOC_INOUT      (_IOC_IN | _IOC_OUT)

```

```
#define _IO(x,y)        ((x << 8) | y | _IOC_VOID)
#define _IOR(x,y,t)     ((x << 8) | y | ((sizeof(t) & _IOCPARM_MASK) << 16) |\
                                _IOC_OUT)
#define _IOW(x,y,t)     ((x << 8) | y | ((sizeof(t) & _IOCPARM_MASK) << 16) |\
                                _IOC_IN)
#define _IORW(x,y,t)    ((x << 8) | y | ((sizeof(t) & _IOCPARM_MASK) << 16) |\
                                _IOC_INOUT)
#else
/* No fancy encoding on a 16-bit machine. */

#define _IO(x,y)        ((x << 8) | y)
#define _IOR(x,y,t)     _IO(x,y)
#define _IOW(x,y,t)     _IO(x,y)
#define _IORW(x,y,t)    _IO(x,y)
#endif

int ioctl(int _fd, int _request, void *_data);

#endif /* _M_IOCTL_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                      include/minix/config.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
#ifndef _CONFIG_H
#define _CONFIG_H

/* Minix release and version numbers. */
#define OS_RELEASE "3"
#define OS_VERSION "1.0"

/* This file sets configuration parameters for the MINIX kernel, FS, and PM.
 * It is divided up into two main sections.  The first section contains
 * user-settable parameters.  In the second section, various internal system
 * parameters are set based on the user-settable parameters.
 *
 * Parts of config.h have been moved to sys_config.h, which can be included
 * by other include files that wish to get at the configuration data, but
 * don't want to pollute the users namespace. Some editable values have
 * gone there.
 *
 * This is a modified version of config.h for compiling a small Minix system
 * with only the options described in the text, Operating Systems Design and
 * Implementation, 3rd edition. See the version of config.h in the full
 * source code directory for information on alternatives omitted here.
 */

/* The MACHINE (called _MINIX_MACHINE) setting can be done
 * in <minix/machine.h>.
 */
#include <minix/sys_config.h>

#define MACHINE      _MINIX_MACHINE

#define IBM_PC       _MACHINE_IBM_PC
#define SUN_4        _MACHINE_SUN_4
#define SUN_4_60     _MACHINE_SUN_4_60
#define ATARI        _MACHINE_ATARI
#define MACINTOSH    _MACHINE_MACINTOSH
```

```

/* Number of slots in the process table for non-kernel processes. The number
 * of system processes defines how many processes with special privileges
 * there can be. User processes share the same properties and count for one.
 *
 * These can be changed in sys_config.h.
 */
#define NR_PROCS          _NR_PROCS
#define NR_SYS_PROCS      _NR_SYS_PROCS

#define NR_BUFS 128
#define NR_BUF_HASH 128

/* Number of controller tasks (/dev/cN device classes). */
#define NR_CTRLRS          2

/* Enable or disable the second level file system cache on the RAM disk. */
#define ENABLE_CACHE2      0

/* Enable or disable swapping processes to disk. */
#define ENABLE_SWAP        0

/* Include or exclude an image of /dev/boot in the boot image.
 * Please update the makefile in /usr/src/tools/ as well.
 */
#define ENABLE_BOOTDEV     0    /* load image of /dev/boot at boot time */

/* DMA_SECTORS may be increased to speed up DMA based drivers. */
#define DMA_SECTORS        1    /* DMA buffer size (must be >= 1) */

/* Include or exclude backwards compatibility code. */
#define ENABLE_BINCOMPAT   0    /* for binaries using obsolete calls */
#define ENABLE_SRCCOMPAT   0    /* for sources using obsolete calls */

/* Which process should receive diagnostics from the kernel and system?
 * Directly sending it to TTY only displays the output. Sending it to the
 * log driver will cause the diagnostics to be buffered and displayed.
 */
#define OUTPUT_PROC_NR  LOG_PROC_NR     /* TTY_PROC_NR or LOG_PROC_NR */

/* NR_CONS, NR_RS_LINES, and NR_PTYS determine the number of terminals the
 * system can handle.
 */
#define NR_CONS            4    /* # system consoles (1 to 8) */
#define NR_RS_LINES        0    /* # rs232 terminals (0 to 4) */
#define NR_PTYS            0    /* # pseudo terminals (0 to 64) */

/*===========================================================================*
 *      There are no user-settable parameters after this line               *
 *===========================================================================*/
/* Set the CHIP type based on the machine selected. The symbol CHIP is actually
 * indicative of more than just the CPU.  For example, machines for which
 * CHIP == INTEL are expected to have 8259A interrrupt controllers and the
 * other properties of IBM PC/XT/AT/386 types machines in general. */
#define INTEL             _CHIP_INTEL   /* CHIP type for PC, XT, AT, 386 and clones */
#define M68000            _CHIP_M68000  /* CHIP type for Atari, Amiga, Macintosh    */
#define SPARC             _CHIP_SPARC   /* CHIP type for SUN-4 (e.g. SPARCstation)  */

/* Set the FP_FORMAT type based on the machine selected, either hw or sw    */
#define FP_NONE  _FP_NONE       /* no floating point support                */
```

```
#define FP_IEEE  _FP_IEEE       /* conform IEEE floating point standard     */

/* _MINIX_CHIP is defined in sys_config.h. */
#define CHIP    _MINIX_CHIP

/* _MINIX_FP_FORMAT is defined in sys_config.h. */
#define FP_FORMAT       _MINIX_FP_FORMAT

/* _ASKDEV and _FASTLOAD are defined in sys_config.h. */
#define ASKDEV _ASKDEV
#define FASTLOAD _FASTLOAD

#endif /* _CONFIG_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                      include/minix/sys_config.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
#ifndef _MINIX_SYS_CONFIG_H
#define _MINIX_SYS_CONFIG_H 1

/* This is a modified sys_config.h for compiling a small Minix system
 * with only the options described in the text, Operating Systems Design and
 * Implementation, 3rd edition. See the sys_config.h in the full
 * source code directory for information on alternatives omitted here.
 */

/*===========================================================================*
 *              This section contains user-settable parameters               *
 *===========================================================================*/
#define _MINIX_MACHINE       _MACHINE_IBM_PC

#define _MACHINE_IBM_PC             1   /* any  8088 or 80x86-based system */

/* Word size in bytes (a constant equal to sizeof(int)). */
#if __ACK__ || __GNUC__
#define _WORD_SIZE      _EM_WSIZE
#define _PTR_SIZE       _EM_WSIZE
#endif

#define _NR_PROCS       64
#define _NR_SYS_PROCS   32

/* Set the CHIP type based on the machine selected. The symbol CHIP is actually
 * indicative of more than just the CPU.  For example, machines for which
 * CHIP == INTEL are expected to have 8259A interrrupt controllers and the
 * other properties of IBM PC/XT/AT/386 types machines in general. */
#define _CHIP_INTEL     1       /* CHIP type for PC, XT, AT, 386 and clones */

/* Set the FP_FORMAT type based on the machine selected, either hw or sw    */
#define _FP_NONE        0       /* no floating point support                */
#define _FP_IEEE        1       /* conform IEEE floating point standard     */

#define _MINIX_CHIP          _CHIP_INTEL

#define _MINIX_FP_FORMAT   _FP_NONE

#ifndef _MINIX_MACHINE
```

```
error "In <minix/sys_config.h> please define _MINIX_MACHINE"
#endif

#ifndef _MINIX_CHIP
error "In <minix/sys_config.h> please define _MINIX_MACHINE to have a legal value"
#endif

#if (_MINIX_MACHINE == 0)
error "_MINIX_MACHINE has incorrect value (0)"
#endif

#endif /* _MINIX_SYS_CONFIG_H */


```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                      include/minix/const.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/* Copyright (C) 2001 by Prentice-Hall, Inc.  See the copyright notice in
 * the file /usr/src/LICENSE.
 */

#ifndef CHIP
#error CHIP is not defined
#endif

#define EXTERN        extern    /* used in *.h files */
#define PRIVATE       static    /* PRIVATE x limits the scope of x */
#define PUBLIC                  /* PUBLIC is the opposite of PRIVATE */
#define FORWARD       static    /* some compilers require this to be 'static'*/

#define TRUE               1    /* used for turning integers into Booleans */
#define FALSE              0    /* used for turning integers into Booleans */

#define HZ                60    /* clock freq (software settable on IBM-PC) */

#define SUPER_USER (uid_t) 0    /* uid_t of superuser */

/* Devices. */
#define MAJOR              8    /* major device = (dev>>MAJOR) & 0377 */
#define MINOR              0    /* minor device = (dev>>MINOR) & 0377 */

#define NULL     ((void *)0)    /* null pointer */
#define CPVEC_NR          16    /* max # of entries in a SYS_VCOPY request */
#define CPVVEC_NR         64    /* max # of entries in a SYS_VCOPY request */
#define NR_IOREQS       MIN(NR_BUFS, 64)
                                /* maximum number of entries in an iorequest */

/* Message passing constants. */
#define MESS_SIZE (sizeof(message))     /* might need usizeof from FS here */
#define NIL_MESS ((message *) 0)        /* null pointer */

/* Memory related constants. */
#define SEGMENT_TYPE  0xFF00    /* bit mask to get segment type */
#define SEGMENT_INDEX 0x00FF    /* bit mask to get segment index */

#define LOCAL_SEG     0x0000    /* flags indicating local memory segment */
#define NR_LOCAL_SEGS      3    /* # local segments per process (fixed) */
```

```
#define T                 0    /* proc[i].mem_map[T] is for text */
#define D                 1    /* proc[i].mem_map[D] is for data */
#define S                 2    /* proc[i].mem_map[S] is for stack */

#define REMOTE_SEG    0x0100    /* flags indicating remote memory segment */
#define NR_REMOTE_SEGS     3    /* # remote memory regions (variable) */

#define BIOS_SEG      0x0200    /* flags indicating BIOS memory segment */
#define NR_BIOS_SEGS       3    /* # BIOS memory regions (variable) */

#define PHYS_SEG      0x0400    /* flag indicating entire physical memory */

/* Labels used to disable code sections for different reasons. */
#define DEAD_CODE          0    /* unused code in normal configuration */
#define FUTURE_CODE        0    /* new code to be activated + tested later */
#define TEMP_CODE          1    /* active code to be removed later */

/* Process name length in the PM process table, including '\0'. */
#define PROC_NAME_LEN   16

/* Miscellaneous */
#define BYTE            0377    /* mask for 8 bits */
#define READING            0    /* copy data to user */
#define WRITING            1    /* copy data from user */
#define NO_NUM        0x8000    /* used as numerical argument to panic() */
#define NIL_PTR   (char *) 0    /* generally useful expression */
#define HAVE_SCATTERED_IO  1    /* scattered I/O is now standard */

/* Macros. */
#define MAX(a, b)   ((a) > (b) ? (a) : (b))
#define MIN(a, b)   ((a) < (b) ? (a) : (b))

/* Memory is allocated in clicks. */
#if (CHIP == INTEL)
#define CLICK_SIZE      1024    /* unit in which memory is allocated */
#define CLICK_SHIFT       10    /* log2 of CLICK_SIZE */
#endif

#if (CHIP == SPARC) || (CHIP == M68000)
#define CLICK_SIZE      4096    /* unit in which memory is allocated */
#define CLICK_SHIFT       12    /* log2 of CLICK_SIZE */
#endif

/* Click to byte conversions (and vice versa). */
#define HCLICK_SHIFT       4    /* log2 of HCLICK_SIZE */
#define HCLICK_SIZE       16    /* hardware segment conversion magic */
#if CLICK_SIZE >= HCLICK_SIZE
#define click_to_hclick(n) ((n) << (CLICK_SHIFT - HCLICK_SHIFT))
#else
#define click_to_hclick(n) ((n) >> (HCLICK_SHIFT - CLICK_SHIFT))
#endif
#define hclick_to_physb(n) ((phys_bytes) (n) << HCLICK_SHIFT)
#define physb_to_hclick(n) ((n) >> HCLICK_SHIFT)

#define ABS             -999    /* this process means absolute memory */

/* Flag bits for i_mode in the inode. */
#define I_TYPE          0170000 /* this field gives inode type */
#define I_REGULAR       0100000 /* regular file, not dir or special */
#define I_BLOCK_SPECIAL 0060000 /* block special file */
```

```
#define I_DIRECTORY     0040000 /* file is a directory */
#define I_CHAR_SPECIAL  0020000 /* character special file */
#define I_NAMED_PIPE    0010000 /* named pipe (FIFO) */
#define I_SET_UID_BIT   0004000 /* set effective uid_t on exec */
#define I_SET_GID_BIT   0002000 /* set effective gid_t on exec */
#define ALL_MODES       0006777 /* all bits for user, group and others */
#define RWX_MODES       0000777 /* mode bits for RWX only */
#define R_BIT           0000004 /* Rwx protection bit */
#define W_BIT           0000002 /* rWx protection bit */
#define X_BIT           0000001 /* rwX protection bit */
#define I_NOT_ALLOC     0000000 /* this inode is free */

/* Flag used only in flags argument of dev_open. */
#define RO_BIT          0200000 /* Open device readonly; fail if writable. */

/* Some limits. */
#define MAX_BLOCK_NR  ((block_t) 077777777)     /* largest block number */
#define HIGHEST_ZONE   ((zone_t) 077777777)     /* largest zone number */
#define MAX_INODE_NR ((ino_t) 037777777777)     /* largest inode number */
#define MAX_FILE_POS ((off_t) 037777777777)     /* largest legal file offset */

#define NO_BLOCK              ((block_t) 0)     /* absence of a block number */
#define NO_ENTRY                ((ino_t) 0)     /* absence of a dir entry */
#define NO_ZONE                ((zone_t) 0)     /* absence of a zone number */
#define NO_DEV                  ((dev_t) 0)     /* absence of a device numb */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                      include/minix/type.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
#ifndef _TYPE_H
#define _TYPE_H

#ifndef _MINIX_SYS_CONFIG_H
#include <minix/sys_config.h>
#endif

#ifndef _TYPES_H
#include <sys/types.h>
#endif

/* Type definitions. */
typedef unsigned int vir_clicks;        /*  virtual addr/length in clicks */
typedef unsigned long phys_bytes;       /* physical addr/length in bytes */
typedef unsigned int phys_clicks;       /* physical addr/length in clicks */

#if (_MINIX_CHIP == _CHIP_INTEL)
typedef unsigned int vir_bytes; /* virtual addresses and lengths in bytes */
#endif

#if (_MINIX_CHIP == _CHIP_M68000)
typedef unsigned long vir_bytes;/* virtual addresses and lengths in bytes */
#endif

#if (_MINIX_CHIP == _CHIP_SPARC)
typedef unsigned long vir_bytes;/* virtual addresses and lengths in bytes */
#endif

/* Memory map for local text, stack, data segments. */
struct mem_map {
  vir_clicks mem_vir;           /* virtual address */
  phys_clicks mem_phys;         /* physical address */
  vir_clicks mem_len;           /* length */
};

/* Memory map for remote memory areas, e.g., for the RAM disk. */
struct far_mem {
  int in_use;                   /* entry in use, unless zero */
  phys_clicks mem_phys;         /* physical address */
  vir_clicks mem_len;           /* length */
};

/* Structure for virtual copying by means of a vector with requests. */
struct vir_addr {
  int proc_nr;
  int segment;
  vir_bytes offset;
};

#define phys_cp_req vir_cp_req
struct vir_cp_req {
  struct vir_addr src;
  struct vir_addr dst;
  phys_bytes count;
};

typedef struct {
  vir_bytes iov_addr;           /* address of an I/O buffer */
  vir_bytes iov_size;           /* sizeof an I/O buffer */
} iovec_t;

/* PM passes the address of a structure of this type to KERNEL when
 * sys_sendsig() is invoked as part of the signal catching mechanism.
 * The structure contain all the information that KERNEL needs to build
 * the signal stack.
 */
struct sigmsg {
  int sm_signo;                 /* signal number being caught */
  unsigned long sm_mask;        /* mask to restore when handler returns */
  vir_bytes sm_sighandler;      /* address of handler */
  vir_bytes sm_sigreturn;       /* address of _sigreturn in C library */
  vir_bytes sm_stkptr;          /* user stack pointer */
};

/* This is used to obtain system information through SYS_GETINFO. */
struct kinfo {
  phys_bytes code_base;         /* base of kernel code */
  phys_bytes code_size;
  phys_bytes data_base;         /* base of kernel data */
  phys_bytes data_size;
  vir_bytes proc_addr;          /* virtual address of process table */
  phys_bytes kmem_base;         /* kernel memory layout (/dev/kmem) */
  phys_bytes kmem_size;
  phys_bytes bootdev_base;      /* boot device from boot image (/dev/boot) */
  phys_bytes bootdev_size;
  phys_bytes bootdev_mem;
  phys_bytes params_base;       /* parameters passed by boot monitor */
  phys_bytes params_size;
  int nr_procs;                 /* number of user processes */
  int nr_tasks;                 /* number of kernel tasks */
  char release[6];              /* kernel release number */
  char version[6];              /* kernel version number */
  int relocking;                /* relocking check (for debugging) */
};

struct machine {
  int pc_at;
  int ps_mca;
  int processor;
  int protected;
  int vdu_ega;
  int vdu_vga;
};

#endif /* _TYPE_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                      include/minix/ipc.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
#ifndef _IPC_H
#define _IPC_H

/*==========================================================================*
 * Types relating to messages.                                              *
 *==========================================================================*/

#define M1                 1
#define M3                 3
#define M4                 4
#define M3_STRING         14

typedef struct {int m1i1, m1i2, m1i3; char *m1p1, *m1p2, *m1p3;} mess_1;
typedef struct {int m2i1, m2i2, m2i3; long m2l1, m2l2; char *m2p1;} mess_2;
typedef struct {int m3i1, m3i2; char *m3p1; char m3ca1[M3_STRING];} mess_3;
typedef struct {long m4l1, m4l2, m4l3, m4l4, m4l5;} mess_4;
typedef struct {short m5c1, m5c2; int m5i1, m5i2; long m5l1, m5l2, m5l3;}mess_5;
typedef struct {int m7i1, m7i2, m7i3, m7i4; char *m7p1, *m7p2;} mess_7;
typedef struct {int m8i1, m8i2; char *m8p1, *m8p2, *m8p3, *m8p4;} mess_8;

typedef struct {
  int m_source;                 /* who sent the message */
  int m_type;                   /* what kind of message is it */
  union {
        mess_1 m_m1;
        mess_2 m_m2;
        mess_3 m_m3;
        mess_4 m_m4;
        mess_5 m_m5;
        mess_7 m_m7;
        mess_8 m_m8;
  } m_u;
} message;

/* The following defines provide names for useful members. */
#define m1_i1  m_u.m_m1.m1i1
#define m1_i2  m_u.m_m1.m1i2
#define m1_i3  m_u.m_m1.m1i3
#define m1_p1  m_u.m_m1.m1p1
#define m1_p2  m_u.m_m1.m1p2
```

```
#define m1_p3  m_u.m_m1.m1p3

#define m2_i1  m_u.m_m2.m2i1
#define m2_i2  m_u.m_m2.m2i2
#define m2_i3  m_u.m_m2.m2i3
#define m2_l1  m_u.m_m2.m2l1
#define m2_l2  m_u.m_m2.m2l2
#define m2_p1  m_u.m_m2.m2p1

#define m3_i1  m_u.m_m3.m3i1
#define m3_i2  m_u.m_m3.m3i2
#define m3_p1  m_u.m_m3.m3p1
#define m3_ca1 m_u.m_m3.m3ca1

#define m4_l1  m_u.m_m4.m4l1
#define m4_l2  m_u.m_m4.m4l2
#define m4_l3  m_u.m_m4.m4l3
#define m4_l4  m_u.m_m4.m4l4
#define m4_l5  m_u.m_m4.m4l5

#define m5_c1  m_u.m_m5.m5c1
#define m5_c2  m_u.m_m5.m5c2
#define m5_i1  m_u.m_m5.m5i1
#define m5_i2  m_u.m_m5.m5i2
#define m5_l1  m_u.m_m5.m5l1
#define m5_l2  m_u.m_m5.m5l2
#define m5_l3  m_u.m_m5.m5l3

#define m7_i1  m_u.m_m7.m7i1
#define m7_i2  m_u.m_m7.m7i2
#define m7_i3  m_u.m_m7.m7i3
#define m7_i4  m_u.m_m7.m7i4
#define m7_p1  m_u.m_m7.m7p1
#define m7_p2  m_u.m_m7.m7p2

#define m8_i1  m_u.m_m8.m8i1
#define m8_i2  m_u.m_m8.m8i2
#define m8_p1  m_u.m_m8.m8p1
#define m8_p2  m_u.m_m8.m8p2
#define m8_p3  m_u.m_m8.m8p3
#define m8_p4  m_u.m_m8.m8p4

/*==========================================================================*
 * Minix run-time system (IPC).                                             *
 *==========================================================================*/

/* Hide names to avoid name space pollution. */
#define echo            _echo
#define notify          _notify
#define sendrec         _sendrec
#define receive         _receive
#define send            _send
#define nb_receive      _nb_receive
#define nb_send         _nb_send

_PROTOTYPE( int echo, (message *m_ptr)                                  );
_PROTOTYPE( int notify, (int dest)                                      );
_PROTOTYPE( int sendrec, (int src_dest, message *m_ptr)                 );
_PROTOTYPE( int receive, (int src, message *m_ptr)                      );
_PROTOTYPE( int send, (int dest, message *m_ptr)                        );
_PROTOTYPE( int nb_receive, (int src, message *m_ptr)                   );
_PROTOTYPE( int nb_send, (int dest, message *m_ptr)                     );

#endif /* _IPC_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                      include/minix/syslib.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/* Prototypes for system library functions. */

#ifndef _SYSLIB_H
#define _SYSLIB_H

#ifndef _TYPES_H
#include <sys/types.h>
#endif

#ifndef _IPC_H
#include <minix/ipc.h>
#endif

#ifndef _DEVIO_H
#include <minix/devio.h>
#endif

/* Forward declaration */
struct reg86u;

#define SYSTASK SYSTEM

/*==========================================================================*
 * Minix system library.                                                    *
 *==========================================================================*/
_PROTOTYPE( int _taskcall, (int who, int syscallnr, message *msgptr));

_PROTOTYPE( int sys_abort, (int how, ...));
_PROTOTYPE( int sys_exec, (int proc, char *ptr,
                                char *aout, vir_bytes initpc));
_PROTOTYPE( int sys_fork, (int parent, int child));
_PROTOTYPE( int sys_newmap, (int proc, struct mem_map *ptr));
_PROTOTYPE( int sys_exit, (int proc));
_PROTOTYPE( int sys_trace, (int req, int proc, long addr, long *data_p));

_PROTOTYPE( int sys_svrctl, (int proc, int req, int priv,vir_bytes argp));
_PROTOTYPE( int sys_nice, (int proc, int priority));

_PROTOTYPE( int sys_int86, (struct reg86u *reg86p));

/* Shorthands for sys_sdevio() system call. */
#define sys_insb(port, proc_nr, buffer, count) \
        sys_sdevio(DIO_INPUT, port, DIO_BYTE, proc_nr, buffer, count)
#define sys_insw(port, proc_nr, buffer, count) \
        sys_sdevio(DIO_INPUT, port, DIO_WORD, proc_nr, buffer, count)
#define sys_outsb(port, proc_nr, buffer, count) \
        sys_sdevio(DIO_OUTPUT, port, DIO_BYTE, proc_nr, buffer, count)
#define sys_outsw(port, proc_nr, buffer, count) \
        sys_sdevio(DIO_OUTPUT, port, DIO_WORD, proc_nr, buffer, count)
_PROTOTYPE( int sys_sdevio, (int req, long port, int type, int proc_nr,
        void *buffer, int count));

/* Clock functionality: get system times or (un)schedule an alarm call. */
_PROTOTYPE( int sys_times, (int proc_nr, clock_t *ptr));
_PROTOTYPE(int sys_setalarm, (clock_t exp_time, int abs_time));

/* Shorthands for sys_irqctl() system call. */
#define sys_irqdisable(hook_id) \
    sys_irqctl(IRQ_DISABLE, 0, 0, hook_id)
#define sys_irqenable(hook_id) \
    sys_irqctl(IRQ_ENABLE, 0, 0, hook_id)
#define sys_irqsetpolicy(irq_vec, policy, hook_id) \
    sys_irqctl(IRQ_SETPOLICY, irq_vec, policy, hook_id)
#define sys_irqrmpolicy(irq_vec, hook_id) \
    sys_irqctl(IRQ_RMPOLICY, irq_vec, 0, hook_id)
_PROTOTYPE ( int sys_irqctl, (int request, int irq_vec, int policy,
    int *irq_hook_id) );

/* Shorthands for sys_vircopy() and sys_physcopy() system calls. */
#define sys_biosin(bios_vir, dst_vir, bytes) \
        sys_vircopy(SELF, BIOS_SEG, bios_vir, SELF, D, dst_vir, bytes)
#define sys_biosout(src_vir, bios_vir, bytes) \
        sys_vircopy(SELF, D, src_vir, SELF, BIOS_SEG, bios_vir, bytes)
#define sys_datacopy(src_proc, src_vir, dst_proc, dst_vir, bytes) \
        sys_vircopy(src_proc, D, src_vir, dst_proc, D, dst_vir, bytes)
#define sys_textcopy(src_proc, src_vir, dst_proc, dst_vir, bytes) \
        sys_vircopy(src_proc, T, src_vir, dst_proc, T, dst_vir, bytes)
#define sys_stackcopy(src_proc, src_vir, dst_proc, dst_vir, bytes) \
        sys_vircopy(src_proc, S, src_vir, dst_proc, S, dst_vir, bytes)
_PROTOTYPE(int sys_vircopy, (int src_proc, int src_seg, vir_bytes src_vir,
        int dst_proc, int dst_seg, vir_bytes dst_vir, phys_bytes bytes));

#define sys_abscopy(src_phys, dst_phys, bytes) \
        sys_physcopy(NONE, PHYS_SEG, src_phys, NONE, PHYS_SEG, dst_phys, bytes)
_PROTOTYPE(int sys_physcopy, (int src_proc, int src_seg, vir_bytes src_vir,
        int dst_proc, int dst_seg, vir_bytes dst_vir, phys_bytes bytes));
_PROTOTYPE(int sys_memset, (unsigned long pattern,
                phys_bytes base, phys_bytes bytes));

/* Vectored virtual / physical copy calls. */
#if DEAD_CODE           /* library part not yet implemented */
_PROTOTYPE(int sys_virvcopy, (phys_cp_req *vec_ptr,int vec_size,int *nr_ok));
_PROTOTYPE(int sys_physvcopy, (phys_cp_req *vec_ptr,int vec_size,int *nr_ok));
#endif

_PROTOTYPE(int sys_umap, (int proc_nr, int seg, vir_bytes vir_addr,
         vir_bytes bytes, phys_bytes *phys_addr));
_PROTOTYPE(int sys_segctl, (int *index, u16_t *seg, vir_bytes *off,
        phys_bytes phys, vir_bytes size));

/* Shorthands for sys_getinfo() system call. */
#define sys_getkmessages(dst)   sys_getinfo(GET_KMESSAGES, dst, 0,0,0)
#define sys_getkinfo(dst)       sys_getinfo(GET_KINFO, dst, 0,0,0)
#define sys_getmachine(dst)     sys_getinfo(GET_MACHINE, dst, 0,0,0)
#define sys_getproctab(dst)     sys_getinfo(GET_PROCTAB, dst, 0,0,0)
#define sys_getprivtab(dst)     sys_getinfo(GET_PRIVTAB, dst, 0,0,0)
#define sys_getproc(dst,nr)     sys_getinfo(GET_PROC, dst, 0,0, nr)
#define sys_getrandomness(dst)  sys_getinfo(GET_RANDOMNESS, dst, 0,0,0)
#define sys_getimage(dst)       sys_getinfo(GET_IMAGE, dst, 0,0,0)
#define sys_getirqhooks(dst)    sys_getinfo(GET_IRQHOOKS, dst, 0,0,0)
```

```
#define sys_getmonparams(v,vl)  sys_getinfo(GET_MONPARAMS, v,vl, 0,0)
#define sys_getschedinfo(v1,v2) sys_getinfo(GET_SCHEDINFO, v1,0, v2,0)
#define sys_getlocktimings(dst) sys_getinfo(GET_LOCKTIMING, dst, 0,0,0)
#define sys_getbiosbuffer(virp, sizep) sys_getinfo(GET_BIOSBUFFER, virp, \
        sizeof(*virp), sizep, sizeof(*sizep))
_PROTOTYPE(int sys_getinfo, (int request, void *val_ptr, int val_len,
                                 void *val_ptr2, int val_len2)          );

/* Signal control. */
_PROTOTYPE(int sys_kill, (int proc, int sig) );
_PROTOTYPE(int sys_sigsend, (int proc_nr, struct sigmsg *sig_ctxt) );
_PROTOTYPE(int sys_sigreturn, (int proc_nr, struct sigmsg *sig_ctxt) );
_PROTOTYPE(int sys_getksig, (int *k_proc_nr, sigset_t *k_sig_map) );
_PROTOTYPE(int sys_endksig, (int proc_nr) );

/* NOTE: two different approaches were used to distinguish the device I/O
 * types 'byte', 'word', 'long': the latter uses #define and results in a
 * smaller implementation, but looses the static type checking.
 */
_PROTOTYPE(int sys_voutb, (pvb_pair_t *pvb_pairs, int nr_ports)         );
_PROTOTYPE(int sys_voutw, (pvw_pair_t *pvw_pairs, int nr_ports)         );
_PROTOTYPE(int sys_voutl, (pvl_pair_t *pvl_pairs, int nr_ports)         );
_PROTOTYPE(int sys_vinb, (pvb_pair_t *pvb_pairs, int nr_ports)          );
_PROTOTYPE(int sys_vinw, (pvw_pair_t *pvw_pairs, int nr_ports)          );
_PROTOTYPE(int sys_vinl, (pvl_pair_t *pvl_pairs, int nr_ports)          );

/* Shorthands for sys_out() system call. */
#define sys_outb(p,v)   sys_out((p), (unsigned long) (v), DIO_BYTE)
#define sys_outw(p,v)   sys_out((p), (unsigned long) (v), DIO_WORD)
#define sys_outl(p,v)   sys_out((p), (unsigned long) (v), DIO_LONG)
_PROTOTYPE(int sys_out, (int port, unsigned long value, int type)       );

/* Shorthands for sys_in() system call. */
#define sys_inb(p,v)    sys_in((p), (unsigned long*) (v), DIO_BYTE)
#define sys_inw(p,v)    sys_in((p), (unsigned long*) (v), DIO_WORD)
#define sys_inl(p,v)    sys_in((p), (unsigned long*) (v), DIO_LONG)
_PROTOTYPE(int sys_in, (int port, unsigned long *value, int type)       );

#endif /* _SYSLIB_H */

```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                include/minix/sysutil.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
#ifndef _EXTRALIB_H
#define _EXTRALIB_H

/* Extra system library definitions to support device drivers and servers.
 *
 * Created:
 *      Mar 15, 2004 by Jorrit N. Herder
 *
 * Changes:
 *      May 31, 2005: added printf, kputc (relocated from syslib)
 *      May 31, 2005: added getuptime
 *      Mar 18, 2005: added tickdelay
 *      Oct 01, 2004: added env_parse, env_prefix, env_panic
 *      Jul 13, 2004: added fkey_ctl
 *      Apr 28, 2004: added report, panic
 *      Mar 31, 2004: setup like other libraries, such as syslib
 */

/*==========================================================================*
 * Miscellaneous helper functions.
 *==========================================================================*/

/* Environment parsing return values. */
#define EP_BUF_SIZE   128       /* local buffer for env value */
#define EP_UNSET        0       /* variable not set */
#define EP_OFF          1       /* var = off */
#define EP_ON           2       /* var = on (or field left blank) */
#define EP_SET          3       /* var = 1:2:3 (nonblank field) */
#define EP_EGETKENV     4       /* sys_getkenv() failed ... */

_PROTOTYPE( void env_setargs, (int argc, char *argv[])                  );
_PROTOTYPE( int env_get_param, (char *key, char *value, int max_size)   );
_PROTOTYPE( int env_prefix, (char *env, char *prefix)                   );
_PROTOTYPE( void env_panic, (char *key)                                 );
_PROTOTYPE( int env_parse, (char *env, char *fmt, int field, long *param,
                                long min, long max)                     );

#define fkey_map(fkeys, sfkeys) fkey_ctl(FKEY_MAP, (fkeys), (sfkeys))
#define fkey_unmap(fkeys, sfkeys) fkey_ctl(FKEY_UNMAP, (fkeys), (sfkeys))
#define fkey_events(fkeys, sfkeys) fkey_ctl(FKEY_EVENTS, (fkeys), (sfkeys))
_PROTOTYPE( int fkey_ctl, (int req, int *fkeys, int *sfkeys)            );

_PROTOTYPE( int printf, (const char *fmt, ...));
_PROTOTYPE( void kputc, (int c));
_PROTOTYPE( void report, (char *who, char *mess, int num));
_PROTOTYPE( void panic, (char *who, char *mess, int num));
_PROTOTYPE( int getuptime, (clock_t *ticks));
_PROTOTYPE( int tickdelay, (clock_t ticks));

#endif /* _EXTRALIB_H */

```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                              include/minix/callnr.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
#define NCALLS           91    /* number of system calls allowed */

#define EXIT              1
#define FORK              2
#define READ              3
#define WRITE             4
#define OPEN              5
#define CLOSE             6
#define WAIT              7
#define CREAT             8
#define LINK              9
#define UNLINK           10
#define WAITPID          11
#define CHDIR            12
#define TIME             13
```

```
#define MKNOD            14
#define CHMOD            15
#define CHOWN            16
#define BRK              17
#define STAT             18
#define LSEEK            19
#define GETPID           20
#define MOUNT            21
#define UMOUNT           22
#define SETUID           23
#define GETUID           24
#define STIME            25
#define PTRACE           26
#define ALARM            27
#define FSTAT            28
#define PAUSE            29
#define UTIME            30
#define ACCESS           33
#define SYNC             36
#define KILL             37
#define RENAME           38
#define MKDIR            39
#define RMDIR            40
#define DUP              41
#define PIPE             42
#define TIMES            43
#define SETGID           46
#define GETGID           47
#define SIGNAL           48
#define IOCTL            54
#define FCNTL            55
#define EXEC             59
#define UMASK            60
#define CHROOT           61
#define SETSID           62
#define GETPGRP          63

/* The following are not system calls, but are processed like them. */
#define UNPAUSE          65    /* to MM or FS: check for EINTR */
#define REVIVE           67    /* to FS: revive a sleeping process */
#define TASK_REPLY       68    /* to FS: reply code from tty task */

/* Posix signal handling. */
#define SIGACTION        71
#define SIGSUSPEND       72
#define SIGPENDING       73
#define SIGPROCMASK      74
#define SIGRETURN        75

#define REBOOT           76    /* to PM */

/* MINIX specific calls, e.g., to support system services. */
#define SVRCTL           77
                               /* unused */
#define GETSYSINFO       79    /* to PM or FS */
#define GETPROCNR        80    /* to PM */
#define DEVCTL           81    /* to FS */
#define FSTATFS          82    /* to FS */
#define ALLOCMEM         83    /* to PM */
#define FREEMEM          84    /* to PM */
```

```
#define SELECT           85    /* to FS */
#define FCHDIR           86    /* to FS */
#define FSYNC            87    /* to FS */
#define GETPRIORITY      88    /* to PM */
#define SETPRIORITY      89    /* to PM */
#define GETTIMEOFDAY     90    /* to PM */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                              include/minix/com.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
#ifndef _MINIX_COM_H
#define _MINIX_COM_H

/*===========================================================================*
 *                        Magic process numbers                              *
 *===========================================================================*/

#define ANY             0x7ace  /* used to indicate 'any process' */
#define NONE            0x6ace  /* used to indicate 'no process at all' */
#define SELF            0x8ace  /* used to indicate 'own process' */

/*===========================================================================*
 *              Process numbers of processes in the system image             *
 *===========================================================================*/

/* The values of several task numbers depend on whether they or other tasks
 * are enabled. They are defined as (PREVIOUS_TASK - ENABLE_TASK) in general.
 * ENABLE_TASK is either 0 or 1, so a task either gets a new number, or gets
 * the same number as the previous task and is further unused. Note that the
 * order should correspond to the order in the task table defined in table.c.
 */

/* Kernel tasks. These all run in the same address space. */
#define IDLE             -4     /* runs when no one else can run */
#define CLOCK            -3     /* alarms and other clock functions */
#define SYSTEM           -2     /* request system functionality */
#define KERNEL           -1     /* pseudo-process for IPC and scheduling */
#define HARDWARE     KERNEL     /* for hardware interrupt handlers */

/* Number of tasks. Note that NR_PROCS is defined in <minix/config.h>. */
#define NR_TASKS          4

/* User-space processes, that is, device drivers, servers, and INIT. */
#define PM_PROC_NR        0     /* process manager */
#define FS_PROC_NR        1     /* file system */
#define RS_PROC_NR        2     /* reincarnation server */
#define MEM_PROC_NR       3     /* memory driver (RAM disk, null, etc.) */
#define LOG_PROC_NR       4     /* log device driver */
#define TTY_PROC_NR       5     /* terminal (TTY) driver */
#define DRVR_PROC_NR      6     /* device driver for boot medium */
#define INIT_PROC_NR      7     /* init -- goes multiuser */

/* Number of processes contained in the system image. */
#define NR_BOOT_PROCS   (NR_TASKS + INIT_PROC_NR + 1)

```

```
/*===========================================================================*
 *                          Kernel notification types                        *
 *===========================================================================*/

/* Kernel notification types. In principle, these can be sent to any process,
 * so make sure that these types do not interfere with other message types.
 * Notifications are prioritized because of the way they are unhold() and
 * blocking notifications are delivered. The lowest numbers go first. The
 * offset are used for the per-process notification bit maps.
 */
#define NOTIFY_MESSAGE            0x1000
#define NOTIFY_FROM(p_nr)        (NOTIFY_MESSAGE | ((p_nr) + NR_TASKS))
#  define SYN_ALARM     NOTIFY_FROM(CLOCK)      /* synchronous alarm */
#  define SYS_SIG       NOTIFY_FROM(SYSTEM)     /* system signal */
#  define HARD_INT      NOTIFY_FROM(HARDWARE)   /* hardware interrupt */
#  define NEW_KSIG      NOTIFY_FROM(HARDWARE)   /* new kernel signal */
#  define FKEY_PRESSED  NOTIFY_FROM(TTY_PROC_NR)/* function key press */

/* Shorthands for message parameters passed with notifications. */
#define NOTIFY_SOURCE           m_source
#define NOTIFY_TYPE             m_type
#define NOTIFY_ARG              m2_l1
#define NOTIFY_TIMESTAMP        m2_l2
#define NOTIFY_FLAGS            m2_i1

/*===========================================================================*
 *                Messages for BLOCK and CHARACTER device drivers            *
 *===========================================================================*/

/* Message types for device drivers. */
#define DEV_RQ_BASE   0x400     /* base for device request types */
#define DEV_RS_BASE   0x500     /* base for device response types */

#define CANCEL          (DEV_RQ_BASE +  0) /* general req to force a task to cancel */
#define DEV_READ        (DEV_RQ_BASE +  3) /* read from minor device */
#define DEV_WRITE       (DEV_RQ_BASE +  4) /* write to minor device */
#define DEV_IOCTL       (DEV_RQ_BASE +  5) /* I/O control code */
#define DEV_OPEN        (DEV_RQ_BASE +  6) /* open a minor device */
#define DEV_CLOSE       (DEV_RQ_BASE +  7) /* close a minor device */
#define DEV_SCATTER     (DEV_RQ_BASE +  8) /* write from a vector */
#define DEV_GATHER      (DEV_RQ_BASE +  9) /* read into a vector */
#define TTY_SETPGRP     (DEV_RQ_BASE + 10) /* set process group */
#define TTY_EXIT        (DEV_RQ_BASE + 11) /* process group leader exited */
#define DEV_SELECT      (DEV_RQ_BASE + 12) /* request select() attention */
#define DEV_STATUS      (DEV_RQ_BASE + 13) /* request driver status */

#define DEV_REPLY       (DEV_RS_BASE + 0) /* general task reply */
#define DEV_CLONED      (DEV_RS_BASE + 1) /* return cloned minor */
#define DEV_REVIVE      (DEV_RS_BASE + 2) /* driver revives process */
#define DEV_IO_READY    (DEV_RS_BASE + 3) /* selected device ready */
#define DEV_NO_STATUS   (DEV_RS_BASE + 4) /* empty status reply */

/* Field names for messages to block and character device drivers. */
#define DEVICE          m2_i1   /* major-minor device */
#define PROC_NR         m2_i2   /* which (proc) wants I/O? */
#define COUNT           m2_i3   /* how many bytes to transfer */
#define REQUEST         m2_i3   /* ioctl request code */
#define POSITION        m2_l1   /* file offset */
#define ADDRESS         m2_p1   /* core buffer address */

```

```
/* Field names for DEV_SELECT messages to device drivers. */
#define DEV_MINOR       m2_i1   /* minor device */
#define DEV_SEL_OPS     m2_i2   /* which select operations are requested */
#define DEV_SEL_WATCH   m2_i3   /* request notify if no operations are ready */

/* Field names used in reply messages from tasks. */
#define REP_PROC_NR     m2_i1   /* # of proc on whose behalf I/O was done */
#define REP_STATUS      m2_i2   /* bytes transferred or error number */
#  define SUSPEND        -998   /* status to suspend caller, reply later */

/* Field names for messages to TTY driver. */
#define TTY_LINE        DEVICE  /* message parameter: terminal line */
#define TTY_REQUEST     COUNT   /* message parameter: ioctl request code */
#define TTY_SPEK        POSITION/* message parameter: ioctl speed, erasing */
#define TTY_FLAGS       m2_l2   /* message parameter: ioctl tty mode */
#define TTY_PGRP        m2_i3   /* message parameter: process group */

/* Field names for the QIC 02 status reply from tape driver */
#define TAPE_STAT0      m2_l1
#define TAPE_STAT1      m2_l2

/*===========================================================================*
 *                          Messages for networking layer                    *
 *===========================================================================*/

/* Message types for network layer requests. This layer acts like a driver. */
#define NW_OPEN         DEV_OPEN
#define NW_CLOSE        DEV_CLOSE
#define NW_READ         DEV_READ
#define NW_WRITE        DEV_WRITE
#define NW_IOCTL        DEV_IOCTL
#define NW_CANCEL       CANCEL

/* Base type for data link layer requests and responses. */
#define DL_RQ_BASE      0x800
#define DL_RS_BASE      0x900

/* Message types for data link layer requests. */
#define DL_WRITE        (DL_RQ_BASE + 3)
#define DL_WRITEV       (DL_RQ_BASE + 4)
#define DL_READ         (DL_RQ_BASE + 5)
#define DL_READV        (DL_RQ_BASE + 6)
#define DL_INIT         (DL_RQ_BASE + 7)
#define DL_STOP         (DL_RQ_BASE + 8)
#define DL_GETSTAT      (DL_RQ_BASE + 9)

/* Message type for data link layer replies. */
#define DL_INIT_REPLY   (DL_RS_BASE + 20)
#define DL_TASK_REPLY   (DL_RS_BASE + 21)

/* Field names for data link layer messages. */
#define DL_PORT         m2_i1
#define DL_PROC         m2_i2
#define DL_COUNT        m2_i3
#define DL_MODE         m2_l1
#define DL_CLCK         m2_l2
#define DL_ADDR         m2_p1
#define DL_STAT         m2_l1

/* Bits in 'DL_STAT' field of DL replies. */
```

```
#  define DL_PACK_SEND         0x01
#  define DL_PACK_RECV         0x02
#  define DL_READ_IP           0x04

/* Bits in 'DL_MODE' field of DL requests. */
#  define DL_NOMODE            0x0
#  define DL_PROMISC_REQ       0x2
#  define DL_MULTI_REQ         0x4
#  define DL_BROAD_REQ         0x8

/*===========================================================================*
 *                 SYSTASK request types and field names                    *
 *===========================================================================*/

/* System library calls are dispatched via a call vector, so be careful when
 * modifying the system call numbers. The numbers here determine which call
 * is made from the call vector.
 */
#define KERNEL_CALL     0x600   /* base for kernel calls to SYSTEM */

#  define SYS_FORK       (KERNEL_CALL + 0)      /* sys_fork() */
#  define SYS_EXEC       (KERNEL_CALL + 1)      /* sys_exec() */
#  define SYS_EXIT       (KERNEL_CALL + 2)      /* sys_exit() */
#  define SYS_NICE       (KERNEL_CALL + 3)      /* sys_nice() */
#  define SYS_PRIVCTL    (KERNEL_CALL + 4)      /* sys_privctl() */
#  define SYS_TRACE      (KERNEL_CALL + 5)      /* sys_trace() */
#  define SYS_KILL       (KERNEL_CALL + 6)      /* sys_kill() */

#  define SYS_GETKSIG    (KERNEL_CALL + 7)      /* sys_getsig() */
#  define SYS_ENDKSIG    (KERNEL_CALL + 8)      /* sys_endsig() */
#  define SYS_SIGSEND    (KERNEL_CALL + 9)      /* sys_sigsend() */
#  define SYS_SIGRETURN  (KERNEL_CALL + 10)     /* sys_sigreturn() */

#  define SYS_NEWMAP     (KERNEL_CALL + 11)     /* sys_newmap() */
#  define SYS_SEGCTL     (KERNEL_CALL + 12)     /* sys_segctl() */
#  define SYS_MEMSET     (KERNEL_CALL + 13)     /* sys_memset() */

#  define SYS_UMAP       (KERNEL_CALL + 14)     /* sys_umap() */
#  define SYS_VIRCOPY    (KERNEL_CALL + 15)     /* sys_vircopy() */
#  define SYS_PHYSCOPY   (KERNEL_CALL + 16)     /* sys_physcopy() */
#  define SYS_VIRVCOPY   (KERNEL_CALL + 17)     /* sys_virvcopy() */
#  define SYS_PHYSVCOPY  (KERNEL_CALL + 18)     /* sys_physvcopy() */

#  define SYS_IRQCTL     (KERNEL_CALL + 19)     /* sys_irqctl() */
#  define SYS_INT86      (KERNEL_CALL + 20)     /* sys_int86() */
#  define SYS_DEVIO      (KERNEL_CALL + 21)     /* sys_devio() */
#  define SYS_SDEVIO     (KERNEL_CALL + 22)     /* sys_sdevio() */
#  define SYS_VDEVIO     (KERNEL_CALL + 23)     /* sys_vdevio() */

#  define SYS_SETALARM   (KERNEL_CALL + 24)     /* sys_setalarm() */
#  define SYS_TIMES      (KERNEL_CALL + 25)     /* sys_times() */
#  define SYS_GETINFO    (KERNEL_CALL + 26)     /* sys_getinfo() */
#  define SYS_ABORT      (KERNEL_CALL + 27)     /* sys_abort() */

#define NR_SYS_CALLS    28      /* number of system calls */

/* Field names for SYS_MEMSET, SYS_SEGCTL. */
#define MEM_PTR         m2_p1   /* base */
#define MEM_COUNT       m2_l1   /* count */
#define MEM_PATTERN     m2_l2   /* pattern to write */
```

```
#define MEM_CHUNK_BASE  m4_l1   /* physical base address */
#define MEM_CHUNK_SIZE  m4_l2   /* size of mem chunk */
#define MEM_TOT_SIZE    m4_l3   /* total memory size */
#define MEM_CHUNK_TAG   m4_l4   /* tag to identify chunk of mem */

/* Field names for SYS_DEVIO, SYS_VDEVIO, SYS_SDEVIO. */
#define DIO_REQUEST     m2_i3   /* device in or output */
#   define DIO_INPUT        0   /* input */
#   define DIO_OUTPUT       1   /* output */
#define DIO_TYPE        m2_i1   /* flag indicating byte, word, or long */
#   define DIO_BYTE       'b'   /* byte type values */
#   define DIO_WORD       'w'   /* word type values */
#   define DIO_LONG       'l'   /* long type values */
#define DIO_PORT        m2_l1   /* single port address */
#define DIO_VALUE       m2_l2   /* single I/O value */
#define DIO_VEC_ADDR    m2_p1   /* address of buffer or (p,v)-pairs */
#define DIO_VEC_SIZE    m2_l2   /* number of elements in vector */
#define DIO_VEC_PROC    m2_i2   /* number of process where vector is */

/* Field names for SYS_SIGNARLM, SYS_FLAGARLM, SYS_SYNCALRM. */
#define ALRM_EXP_TIME   m2_l1   /* expire time for the alarm call */
#define ALRM_ABS_TIME   m2_i2   /* set to 1 to use absolute alarm time */
#define ALRM_TIME_LEFT  m2_l1   /* how many ticks were remaining */
#define ALRM_PROC_NR    m2_i1   /* which process wants the alarm? */
#define ALRM_FLAG_PTR   m2_p1   /* virtual address of timeout flag */

/* Field names for SYS_IRQCTL. */
#define IRQ_REQUEST     m5_c1   /* what to do? */
#  define IRQ_SETPOLICY     1   /* manage a slot of the IRQ table */
#  define IRQ_RMPOLICY      2   /* remove a slot of the IRQ table */
#  define IRQ_ENABLE        3   /* enable interrupts */
#  define IRQ_DISABLE       4   /* disable interrupts */
#define IRQ_VECTOR      m5_c2   /* irq vector */
#define IRQ_POLICY      m5_i1   /* options for IRQCTL request */
#  define IRQ_REENABLE  0x001   /* reenable IRQ line after interrupt */
#  define IRQ_BYTE      0x100   /* byte values */
#  define IRQ_WORD      0x200   /* word values */
#  define IRQ_LONG      0x400   /* long values */
#define IRQ_PROC_NR     m5_i2   /* process number, SELF, NONE */
#define IRQ_HOOK_ID     m5_l3   /* id of irq hook at kernel */

/* Field names for SYS_SEGCTL. */
#define SEG_SELECT      m4_l1   /* segment selector returned */
#define SEG_OFFSET      m4_l2   /* offset in segment returned */
#define SEG_PHYS        m4_l3   /* physical address of segment */
#define SEG_SIZE        m4_l4   /* segment size */
#define SEG_INDEX       m4_l5   /* segment index in remote map */

/* Field names for SYS_VIDCOPY. */
#define VID_REQUEST     m4_l1   /* what to do? */
#  define VID_VID_COPY     1    /* request vid_vid_copy() */
#  define MEM_VID_COPY     2    /* request mem_vid_copy() */
#define VID_SRC_ADDR    m4_l2   /* virtual address in memory */
#define VID_SRC_OFFSET  m4_l3   /* offset in video memory */
#define VID_DST_OFFSET  m4_l4   /* offset in video memory */
#define VID_CP_COUNT    m4_l5   /* number of words to be copied */

/* Field names for SYS_ABORT. */
#define ABRT_HOW        m1_i1   /* RBT_REBOOT, RBT_HALT, etc. */
#define ABRT_MON_PROC   m1_i2   /* process where monitor params are */
```

```
#define ABRT_MON_LEN    m1_i3   /* length of monitor params */
#define ABRT_MON_ADDR   m1_p1   /* virtual address of monitor params */

/* Field names for _UMAP, _VIRCOPY, _PHYSCOPY. */
#define CP_SRC_SPACE    m5_c1   /* T or D space (stack is also D) */
#define CP_SRC_PROC_NR  m5_i1   /* process to copy from */
#define CP_SRC_ADDR     m5_l1   /* address where data come from */
#define CP_DST_SPACE    m5_c2   /* T or D space (stack is also D) */
#define CP_DST_PROC_NR  m5_i2   /* process to copy to */
#define CP_DST_ADDR     m5_l2   /* address where data go to */
#define CP_NR_BYTES     m5_l3   /* number of bytes to copy */

/* Field names for SYS_VCOPY and SYS_VVIRCOPY. */
#define VCP_NR_OK       m1_i2   /* number of successfull copies */
#define VCP_VEC_SIZE    m1_i3   /* size of copy vector */
#define VCP_VEC_ADDR    m1_p1   /* pointer to copy vector */

/* Field names for SYS_GETINFO. */
#define I_REQUEST      m7_i3    /* what info to get */
#   define GET_KINFO       0    /* get kernel information structure */
#   define GET_IMAGE       1    /* get system image table */
#   define GET_PROCTAB     2    /* get kernel process table */
#   define GET_RANDOMNESS  3    /* get randomness buffer */
#   define GET_MONPARAMS   4    /* get monitor parameters */
#   define GET_KENV        5    /* get kernel environment string */
#   define GET_IRQHOOKS    6    /* get the IRQ table */
#   define GET_KMESSAGES   7    /* get kernel messages */
#   define GET_PRIVTAB     8    /* get kernel privileges table */
#   define GET_KADDRESSES  9    /* get various kernel addresses */
#   define GET_SCHEDINFO  10    /* get scheduling queues */
#   define GET_PROC       11    /* get process slot if given process */
#   define GET_MACHINE    12    /* get machine information */
#   define GET_LOCKTIMING 13    /* get lock()/unlock() latency timing */
#   define GET_BIOSBUFFER 14    /* get a buffer for BIOS calls */
#define I_PROC_NR      m7_i4    /* calling process */
#define I_VAL_PTR      m7_p1    /* virtual address at caller */
#define I_VAL_LEN      m7_i1    /* max length of value */
#define I_VAL_PTR2     m7_p2    /* second virtual address */
#define I_VAL_LEN2     m7_i2    /* second length, or proc nr */

/* Field names for SYS_TIMES. */
#define T_PROC_NR      m4_l1    /* process to request time info for */
#define T_USER_TIME    m4_l1    /* user time consumed by process */
#define T_SYSTEM_TIME  m4_l2    /* system time consumed by process */
#define T_CHILD_UTIME  m4_l3    /* user time consumed by process' children */
#define T_CHILD_STIME  m4_l4    /* sys time consumed by process' children */
#define T_BOOT_TICKS   m4_l5    /* number of clock ticks since boot time */

/* Field names for SYS_TRACE, SYS_SVRCTL. */
#define CTL_PROC_NR    m2_i1    /* process number of the caller */
#define CTL_REQUEST    m2_i2    /* server control request */
#define CTL_MM_PRIV    m2_i3    /* privilege as seen by PM */
#define CTL_ARG_PTR    m2_p1    /* pointer to argument */
#define CTL_ADDRESS    m2_l1    /* address at traced process' space */
#define CTL_DATA       m2_l2    /* data field for tracing */

/* Field names for SYS_KILL, SYS_SIGCTL */
#define SIG_REQUEST    m2_l2    /* PM signal control request */
#define S_GETSIG           0    /* get pending kernel signal */
#define S_ENDSIG           1    /* finish a kernel signal */
```

```
#define S_SENDSIG          2    /* POSIX style signal handling */
#define S_SIGRETURN        3    /* return from POSIX handling */
#define S_KILL             4    /* servers kills process with signal */
#define SIG_PROC       m2_i1    /* process number for inform */
#define SIG_NUMBER     m2_i2    /* signal number to send */
#define SIG_FLAGS      m2_i3    /* signal flags field */
#define SIG_MAP        m2_l1    /* used by kernel to pass signal bit map */
#define SIG_CTXT_PTR   m2_p1    /* pointer to info to restore signal context */

/* Field names for SYS_FORK, _EXEC, _EXIT, _NEWMAP. */
#define PR_PROC_NR     m1_i1    /* indicates a (child) process */
#define PR_PRIORITY    m1_i2    /* process priority */
#define PR_PPROC_NR    m1_i2    /* indicates a (parent) process */
#define PR_PID         m1_i3    /* process id at process manager */
#define PR_STACK_PTR   m1_p1    /* used for stack ptr in sys_exec, sys_getsp */
#define PR_TRACING     m1_i3    /* flag to indicate tracing is on/ off */
#define PR_NAME_PTR    m1_p2    /* tells where program name is for dmp */
#define PR_IP_PTR      m1_p3    /* initial value for ip after exec */
#define PR_MEM_PTR     m1_p1    /* tells where memory map is for sys_newmap */

/* Field names for SYS_INT86 */
#define INT86_REG86    m1_p1    /* pointer to registers */

/* Field names for SELECT (FS). */
#define SEL_NFDS       m8_i1
#define SEL_READFDS    m8_p1
#define SEL_WRITEFDS   m8_p2
#define SEL_ERRORFDS   m8_p3
#define SEL_TIMEOUT    m8_p4

/*===========================================================================*
 *                Messages for system management server                      *
 *===========================================================================*/

#define SRV_RQ_BASE             0x700

#define SRV_UP          (SRV_RQ_BASE + 0)       /* start system service */
#define SRV_DOWN        (SRV_RQ_BASE + 1)       /* stop system service */
#define SRV_STATUS      (SRV_RQ_BASE + 2)       /* get service status */

#  define SRV_PATH_ADDR         m1_p1           /* path of binary */
#  define SRV_PATH_LEN          m1_i1           /* length of binary */
#  define SRV_ARGS_ADDR         m1_p2           /* arguments to be passed */
#  define SRV_ARGS_LEN          m1_i2           /* length of arguments */
#  define SRV_DEV_MAJOR         m1_i3           /* major device number */
#  define SRV_PRIV_ADDR         m1_p3           /* privileges string */
#  define SRV_PRIV_LEN          m1_i3           /* length of privileges */

/*===========================================================================*
 *                Miscellaneous messages used by TTY                         *
 *===========================================================================*/

/* Miscellaneous request types and field names, e.g. used by IS server. */
#define PANIC_DUMPS            97      /* debug dumps at the TTY on RBT_PANIC */
#define FKEY_CONTROL           98      /* control a function key at the TTY */
#  define FKEY_REQUEST       m2_i1      /* request to perform at TTY */
#  define    FKEY_MAP          10      /* observe function key */
#  define    FKEY_UNMAP        11      /* stop observing function key */
#  define    FKEY_EVENTS       12      /* request open key presses */
#  define FKEY_FKEYS          m2_l1     /* F1-F12 keys pressed */
#  define FKEY_SFKEYS         m2_l2     /* Shift-F1-F12 keys pressed */
#define DIAGNOSTICS     100     /* output a string without FS in between */
#  define DIAG_PRINT_BUF      m1_p1
#  define DIAG_BUF_COUNT      m1_i1
#  define DIAG_PROC_NR        m1_i2

#endif /* _MINIX_COM_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                include/minix/devio.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/* This file provides basic types and some constants for the
 * SYS_DEVIO and SYS_VDEVIO system calls, which allow user-level
 * processes to perform device I/O.
 *
 * Created:
 *      Apr 08, 2004 by Jorrit N. Herder
 */

#ifndef _DEVIO_H
#define _DEVIO_H

#include <minix/sys_config.h>     /* needed to include <minix/type.h> */
#include <sys/types.h>        /* u8_t, u16_t, u32_t needed */

typedef u16_t port_t;
typedef U16_t Port_t;

/* We have different granularities of port I/O: 8, 16, 32 bits.
 * Also see <ibm/portio.h>, which has functions for bytes, words,
 * and longs. Hence, we need different (port,value)-pair types.
 */
typedef struct { u16_t port;  u8_t value; } pvb_pair_t;
typedef struct { u16_t port; u16_t value; } pvw_pair_t;
typedef struct { u16_t port; u32_t value; } pvl_pair_t;

/* Macro shorthand to set (port,value)-pair. */
#define pv_set(pv, p, v) ((pv).port = (p), (pv).value = (v))
#define pv_ptr_set(pv_ptr, p, v) ((pv_ptr)->port = (p), (pv_ptr)->value = (v))

#endif  /* _DEVIO_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                include/minix/dmap.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
#ifndef _DMAP_H
#define _DMAP_H

#include <minix/sys_config.h>
#include <minix/ipc.h>

```

```
/*===========================================================================*
 *                         Device <-> Driver Table                          *
 *===========================================================================*/

/* Device table.  This table is indexed by major device number.  It provides
 * the link between major device numbers and the routines that process them.
 * The table can be update dynamically. The field 'dmap_flags' describe an
 * entry's current status and determines what control options are possible.
 */
#define DMAP_MUTABLE            0x01    /* mapping can be overtaken */
#define DMAP_BUSY               0x02    /* driver busy with request */

enum dev_style { STYLE_DEV, STYLE_NDEV, STYLE_TTY, STYLE_CLONE };

extern struct dmap {
  int _PROTOTYPE ((*dmap_opcl), (int, Dev_t, int, int) );
  void _PROTOTYPE ((*dmap_io), (int, message *) );
  int dmap_driver;
  int dmap_flags;
} dmap[];

/*===========================================================================*
 *                      Major and minor device numbers                      *
 *===========================================================================*/

/* Total number of different devices. */
#define NR_DEVICES                32                    /* number of (major) devices */

/* Major and minor device numbers for MEMORY driver. */
#define MEMORY_MAJOR               1    /* major device for memory devices */
#  define RAM_DEV                  0    /* minor device for /dev/ram */
#  define MEM_DEV                  1    /* minor device for /dev/mem */
#  define KMEM_DEV                 2    /* minor device for /dev/kmem */
#  define NULL_DEV                 3    /* minor device for /dev/null */
#  define BOOT_DEV                 4    /* minor device for /dev/boot */
#  define ZERO_DEV                 5    /* minor device for /dev/zero */

#define CTRLR(n) ((n)==0 ? 3 : (8 + 2*((n)-1))) /* magic formula */

/* Full device numbers that are special to the boot monitor and FS. */
#  define DEV_RAM             0x0100    /* device number of /dev/ram */
#  define DEV_BOOT            0x0104    /* device number of /dev/boot */

#define FLOPPY_MAJOR               2    /* major device for floppy disks */
#define TTY_MAJOR                  4    /* major device for ttys */
#define CTTY_MAJOR                 5    /* major device for /dev/tty */

#define INET_MAJOR                 7    /* major device for inet */

#define LOG_MAJOR                 15    /* major device for log driver */
#  define IS_KLOG_DEV              0    /* minor device for /dev/klog */

#endif /* _DMAP_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                      include/ibm/portio.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/*
ibm/portio.h

Created:        Jan 15, 1992 by Philip Homburg
*/

#ifndef _PORTIO_H_
#define _PORTIO_H_

#ifndef _TYPES_H
#include <sys/types.h>
#endif

unsigned inb(U16_t _port);
unsigned inw(U16_t _port);
unsigned inl(U32_t _port);
void outb(U16_t _port, U8_t _value);
void outw(U16_t _port, U16_t _value);
void outl(U16_t _port, U32_t _value);
void insb(U16_t _port, void *_buf, size_t _count);
void insw(U16_t _port, void *_buf, size_t _count);
void insl(U16_t _port, void *_buf, size_t _count);
void outsb(U16_t _port, void *_buf, size_t _count);
void outsw(U16_t _port, void *_buf, size_t _count);
void outsl(U16_t _port, void *_buf, size_t _count);
void intr_disable(void);
void intr_enable(void);

#endif /* _PORTIO_H_ */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                      include/ibm/interrupt.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/* Interrupt numbers and hardware vectors. */

#ifndef _INTERRUPT_H
#define _INTERRUPT_H

#if (CHIP == INTEL)

/* 8259A interrupt controller ports. */
#define INT_CTL         0x20    /* I/O port for interrupt controller */
#define INT_CTLMASK     0x21    /* setting bits in this port disables ints */
#define INT2_CTL        0xA0    /* I/O port for second interrupt controller */
#define INT2_CTLMASK    0xA1    /* setting bits in this port disables ints */

/* Magic numbers for interrupt controller. */
#define END_OF_INT      0x20    /* code used to re-enable after an interrupt */

/* Interrupt vectors defined/reserved by processor. */
#define DIVIDE_VECTOR      0    /* divide error */
#define DEBUG_VECTOR       1    /* single step (trace) */
#define NMI_VECTOR         2    /* non-maskable interrupt */
```

```
#define BREAKPOINT_VECTOR  3    /* software breakpoint */
#define OVERFLOW_VECTOR    4    /* from INTO */

/* Fixed system call vector. */
#define SYS_VECTOR        32    /* system calls are made with int SYSVEC */
#define SYS386_VECTOR     33    /* except 386 system calls use this */
#define LEVEL0_VECTOR     34    /* for execution of a function at level 0 */

/* Suitable irq bases for hardware interrupts.  Reprogram the 8259(s) from
 * the PC BIOS defaults since the BIOS doesn't respect all the processor's
 * reserved vectors (0 to 31).
 */
#define BIOS_IRQ0_VEC   0x08    /* base of IRQ0-7 vectors used by BIOS */
#define BIOS_IRQ8_VEC   0x70    /* base of IRQ8-15 vectors used by BIOS */
#define IRQ0_VECTOR     0x50    /* nice vectors to relocate IRQ0-7 to */
#define IRQ8_VECTOR     0x70    /* no need to move IRQ8-15 */

/* Hardware interrupt numbers. */
#define NR_IRQ_VECTORS    16
#define CLOCK_IRQ          0
#define KEYBOARD_IRQ       1
#define CASCADE_IRQ        2    /* cascade enable for 2nd AT controller */
#define ETHER_IRQ          3    /* default ethernet interrupt vector */
#define SECONDARY_IRQ      3    /* RS232 interrupt vector for port 2 */
#define RS232_IRQ          4    /* RS232 interrupt vector for port 1 */
#define XT_WINI_IRQ        5    /* xt winchester */
#define FLOPPY_IRQ         6    /* floppy disk */
#define PRINTER_IRQ        7
#define AT_WINI_0_IRQ     14    /* at winchester controller 0 */
#define AT_WINI_1_IRQ     15    /* at winchester controller 1 */

/* Interrupt number to hardware vector. */
#define BIOS_VECTOR(irq)        \
        (((irq) < 8 ? BIOS_IRQ0_VEC : BIOS_IRQ8_VEC) + ((irq) & 0x07))
#define VECTOR(irq)     \
        (((irq) < 8 ? IRQ0_VECTOR : IRQ8_VECTOR) + ((irq) & 0x07))

#endif /* (CHIP == INTEL) */

#endif /* _INTERRUPT_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                include/ibm/ports.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/* Addresses and magic numbers for miscellaneous ports. */

#ifndef _PORTS_H
#define _PORTS_H

#if (CHIP == INTEL)

/* Miscellaneous ports. */
#define PCR             0x65    /* Planar Control Register */
#define PORT_B          0x61    /* I/O port for 8255 port B (kbd, beeper...) */
#define TIMER0          0x40    /* I/O port for timer channel 0 */
#define TIMER2          0x42    /* I/O port for timer channel 2 */
#define TIMER_MODE      0x43    /* I/O port for timer mode control */

#endif /* (CHIP == INTEL) */
```

```

#endif /* _PORTS_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                      kernel/kernel.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
#ifndef KERNEL_H
#define KERNEL_H

/* This is the master header for the kernel.  It includes some other files
 * and defines the principal constants.
 */
#define _POSIX_SOURCE      1    /* tell headers to include POSIX stuff */
#define _MINIX             1    /* tell headers to include MINIX stuff */
#define _SYSTEM            1    /* tell headers that this is the kernel */

/* The following are so basic, all the *.c files get them automatically. */
#include <minix/config.h>       /* global configuration, MUST be first */
#include <ansi.h>               /* C style: ANSI or K&R, MUST be second */
#include <sys/types.h>          /* general system types */
#include <minix/const.h>        /* MINIX specific constants */
#include <minix/type.h>         /* MINIX specific types, e.g. message */
#include <minix/ipc.h>          /* MINIX run-time system */
#include <timers.h>             /* watchdog timer management */
#include <errno.h>              /* return codes and error numbers */
#include <ibm/portio.h>         /* device I/O and toggle interrupts */

/* Important kernel header files. */
#include "config.h"             /* configuration, MUST be first */
#include "const.h"              /* constants, MUST be second */
#include "type.h"               /* type definitions, MUST be third */
#include "proto.h"              /* function prototypes */
#include "glo.h"                /* global variables */
#include "ipc.h"                /* IPC constants */
/* #include "debug.h" */        /* debugging, MUST be last kernel header */

#endif /* KERNEL_H */

```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                      kernel/config.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
#ifndef CONFIG_H
#define CONFIG_H

/* This file defines the kernel configuration. It allows to set sizes of some
 * kernel buffers and to enable or disable debugging code, timing features,
 * and individual kernel calls.
 *
 * Changes:
 *   Jul 11, 2005       Created.  (Jorrit N. Herder)
 */
```

```

/* In embedded and sensor applications, not all the kernel calls may be
 * needed. In this section you can specify which kernel calls are needed
 * and which are not. The code for unneeded kernel calls is not included in
 * the system binary, making it smaller. If you are not sure, it is best
 * to keep all kernel calls enabled.
 */
#define USE_FORK          1    /* fork a new process */
#define USE_NEWMAP        1    /* set a new memory map */
#define USE_EXEC          1    /* update process after execute */
#define USE_EXIT          1    /* clean up after process exit */
#define USE_TRACE         1    /* process information and tracing */
#define USE_GETKSIG       1    /* retrieve pending kernel signals */
#define USE_ENDKSIG       1    /* finish pending kernel signals */
#define USE_KILL          1    /* send a signal to a process */
#define USE_SIGSEND       1    /* send POSIX-style signal */
#define USE_SIGRETURN     1    /* sys_sigreturn(proc_nr, ctxt_ptr, flags) */
#define USE_ABORT         1    /* shut down MINIX */
#define USE_GETINFO       1    /* retrieve a copy of kernel data */
#define USE_TIMES         1    /* get process and system time info */
#define USE_SETALARM      1    /* schedule a synchronous alarm */
#define USE_DEVIO         1    /* read or write a single I/O port */
#define USE_VDEVIO        1    /* process vector with I/O requests */
#define USE_SDEVIO        1    /* perform I/O request on a buffer */
#define USE_IRQCTL        1    /* set an interrupt policy */
#define USE_SEGCTL        1    /* set up a remote segment */
#define USE_PRIVCTL       1    /* system privileges control */
#define USE_NICE          1    /* change scheduling priority */
#define USE_UMAP          1    /* map virtual to physical address */
#define USE_VIRCOPY       1    /* copy using virtual addressing */
#define USE_VIRVCOPY      1    /* vector with virtual copy requests */
#define USE_PHYSCOPY      1    /* copy using physical addressing */
#define USE_PHYSVCOPY     1    /* vector with physical copy requests */
#define USE_MEMSET        1    /* write char to a given memory area */

/* Length of program names stored in the process table. This is only used
 * for the debugging dumps that can be generated with the IS server. The PM
 * server keeps its own copy of the program name.
 */
#define P_NAME_LEN        8

/* Kernel diagnostics are written to a circular buffer. After each message,
 * a system server is notified and a copy of the buffer can be retrieved to
 * display the message. The buffers size can safely be reduced.
 */
#define KMESS_BUF_SIZE   256

/* Buffer to gather randomness. This is used to generate a random stream by
 * the MEMORY driver when reading from /dev/random.
 */
#define RANDOM_ELEMENTS   32

/* This section contains defines for valuable system resources that are used
 * by device drivers. The number of elements of the vectors is determined by
 * the maximum needed by any given driver. The number of interrupt hooks may
 * be incremented on systems with many device drivers.
 */
#define NR_IRQ_HOOKS      16            /* number of interrupt hooks */
#define VDEVIO_BUF_SIZE   64            /* max elements per VDEVIO request */
#define VCOPY_VEC_SIZE    16            /* max elements per VCOPY request */
```

```

/* How many bytes for the kernel stack. Space allocated in mpx.s. */
#define K_STACK_BYTES   1024

/* This section allows to enable kernel debugging and timing functionality.
 * For normal operation all options should be disabled.
 */
#define DEBUG_SCHED_CHECK  0    /* sanity check of scheduling queues */
#define DEBUG_LOCK_CHECK   0    /* kernel lock() sanity check */
#define DEBUG_TIME_LOCKS   0    /* measure time spent in locks */

#endif /* CONFIG_H */

```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                    kernel/const.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/* General macros and constants used by the kernel. */
#ifndef CONST_H
#define CONST_H

#include <ibm/interrupt.h>      /* interrupt numbers and hardware vectors */
#include <ibm/ports.h>          /* port addresses and magic numbers */
#include <ibm/bios.h>           /* BIOS addresses, sizes and magic numbers */
#include <ibm/cpu.h>            /* BIOS addresses, sizes and magic numbers */
#include <minix/config.h>
#include "config.h"

/* To translate an address in kernel space to a physical address.  This is
 * the same as umap_local(proc_ptr, D, vir, sizeof(*vir)), but less costly.
 */
#define vir2phys(vir)   (kinfo.data_base + (vir_bytes) (vir))

/* Map a process number to a privilege structure id. */
#define s_nr_to_id(n)   (NR_TASKS + (n) + 1)

/* Translate a pointer to a field in a structure to a pointer to the structure
 * itself. So it translates '&struct_ptr->field' back to 'struct_ptr'.
 */
#define structof(type, field, ptr) \
        ((type *) (((char *) (ptr)) - offsetof(type, field)))

/* Constants used in virtual_copy(). Values must be 0 and 1, respectively. */
#define _SRC_   0
#define _DST_   1

/* Number of random sources */
#define RANDOM_SOURCES  16

/* Constants and macros for bit map manipulation. */
#define BITCHUNK_BITS   (sizeof(bitchunk_t) * CHAR_BIT)
#define BITMAP_CHUNKS(nr_bits) (((nr_bits)+BITCHUNK_BITS-1)/BITCHUNK_BITS)
#define MAP_CHUNK(map,bit) (map)[((bit)/BITCHUNK_BITS)]
#define CHUNK_OFFSET(bit) ((bit)%BITCHUNK_BITS))
#define GET_BIT(map,bit) ( MAP_CHUNK(map,bit) & (1 << CHUNK_OFFSET(bit) )
#define SET_BIT(map,bit) ( MAP_CHUNK(map,bit) |= (1 << CHUNK_OFFSET(bit) )
#define UNSET_BIT(map,bit) ( MAP_CHUNK(map,bit) &= ˜(1 << CHUNK_OFFSET(bit) )

#define get_sys_bit(map,bit) \
        ( MAP_CHUNK(map.chunk,bit) & (1 << CHUNK_OFFSET(bit) )
#define set_sys_bit(map,bit) \
        ( MAP_CHUNK(map.chunk,bit) |= (1 << CHUNK_OFFSET(bit) )
#define unset_sys_bit(map,bit) \
        ( MAP_CHUNK(map.chunk,bit) &= ˜(1 << CHUNK_OFFSET(bit) )
#define NR_SYS_CHUNKS   BITMAP_CHUNKS(NR_SYS_PROCS)

/* Program stack words and masks. */
#define INIT_PSW      0x0200    /* initial psw */
#define INIT_TASK_PSW 0x1200    /* initial psw for tasks (with IOPL 1) */
#define TRACEBIT      0x0100    /* OR this with psw in proc[] for tracing */
#define SETPSW(rp, new)         /* permits only certain bits to be set */ \
        ((rp)->p_reg.psw = (rp)->p_reg.psw & ˜0xCD5 | (new) & 0xCD5)
#define IF_MASK 0x00000200
#define IOPL_MASK 0x003000

/* Disable/ enable hardware interrupts. The parameters of lock() and unlock()
 * are used when debugging is enabled. See debug.h for more information.
 */
#define lock(c, v)      intr_disable();
#define unlock(c)       intr_enable();

/* Sizes of memory tables. The boot monitor distinguishes three memory areas,
 * namely low mem below 1M, 1M-16M, and mem after 16M. More chunks are needed
 * for DOS MINIX.
 */
#define NR_MEMS            8

#endif /* CONST_H */





```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                 kernel/type.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
#ifndef TYPE_H
#define TYPE_H

typedef _PROTOTYPE( void task_t, (void) );

/* Process table and system property related types. */
typedef int proc_nr_t;                  /* process table entry number */
typedef short sys_id_t;                 /* system process index */
typedef struct {                        /* bitmap for system indexes */
  bitchunk_t chunk[BITMAP_CHUNKS(NR_SYS_PROCS)];
} sys_map_t;

struct boot_image {
  proc_nr_t proc_nr;                    /* process number to use */
  task_t *initial_pc;                   /* start function for tasks */
  int flags;                           /* process flags */
  unsigned char quantum;               /* quantum (tick count) */
  int priority;                        /* scheduling priority */
  int stksize;                         /* stack size for tasks */
  short trap_mask;                     /* allowed system call traps */
  bitchunk_t ipc_to;                   /* send mask protection */
  long call_mask;                      /* system call protection */
  char proc_name[P_NAME_LEN];          /* name in process table */
};

struct memory {
  phys_clicks base;                    /* start address of chunk */
  phys_clicks size;                    /* size of memory chunk */
};

/* The kernel outputs diagnostic messages in a circular buffer. */
struct kmessages {
  int km_next;                         /* next index to write */
  int km_size;                         /* current size in buffer */
  char km_buf[KMESS_BUF_SIZE];         /* buffer for messages */
};

struct randomness {
  struct {
        int r_next;                            /* next index to write */
        int r_size;                            /* number of random elements */
        unsigned short r_buf[RANDOM_ELEMENTS]; /* buffer for random info */
  } bin[RANDOM_SOURCES];
};

#if (CHIP == INTEL)
typedef unsigned reg_t;         /* machine register */

/* The stack frame layout is determined by the software, but for efficiency
 * it is laid out so the assembly code to use it is as simple as possible.
 * 80286 protected mode and all real modes use the same frame, built with
 * 16-bit registers.  Real mode lacks an automatic stack switch, so little
 * is lost by using the 286 frame for it.  The 386 frame differs only in
 * having 32-bit registers and more segment registers.  The same names are
 * used for the larger registers to avoid differences in the code.
 */
struct stackframe_s {           /* proc_ptr points here */
#if _WORD_SIZE == 4
  u16_t gs;                     /* last item pushed by save */
  u16_t fs;                     /*  ^ */
#endif
  u16_t es;                     /*  | */
  u16_t ds;                     /*  | */
  reg_t di;                     /* di through cx are not accessed in C */
  reg_t si;                     /* order is to match pusha/popa */
  reg_t fp;                     /* bp */
  reg_t st;                     /* hole for another copy of sp */
  reg_t bx;                     /*  | */
  reg_t dx;                     /*  | */
  reg_t cx;                     /*  | */
  reg_t retreg;                 /* ax and above are all pushed by save */
  reg_t retadr;                 /* return address for assembly code save() */
  reg_t pc;                     /*  ^  last item pushed by interrupt */
  reg_t cs;                     /*  | */
  reg_t psw;                    /*  | */
  reg_t sp;                     /*  | */
  reg_t ss;                     /* these are pushed by CPU during interrupt */
};

struct segdesc_s {              /* segment descriptor for protected mode */
  u16_t limit_low;
  u16_t base_low;
  u8_t base_middle;
  u8_t access;                  /* |P|DL|1|X|E|R|A| */
  u8_t granularity;             /* |G|X|0|A|LIMT| */
  u8_t base_high;
};

typedef unsigned long irq_policy_t;
typedef unsigned long irq_id_t;

typedef struct irq_hook {
  struct irq_hook *next;                /* next hook in chain */
  int (*handler)(struct irq_hook *);    /* interrupt handler */
  int irq;                              /* IRQ vector number */
  int id;                               /* id of this hook */
  int proc_nr;                          /* NONE if not in use */
  irq_id_t notify_id;                   /* id to return on interrupt */
  irq_policy_t policy;                  /* bit mask for policy */
} irq_hook_t;

typedef int (*irq_handler_t)(struct irq_hook *);

#endif /* (CHIP == INTEL) */

#if (CHIP == M68000)
/* M68000 specific types go here. */
#endif /* (CHIP == M68000) */

#endif /* TYPE_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                kernel/proto.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/* Function prototypes. */

#ifndef PROTO_H
#define PROTO_H

/* Struct declarations. */
struct proc;
struct timer;

/* clock.c */
_PROTOTYPE( void clock_task, (void)                                         );
_PROTOTYPE( void clock_stop, (void)                                         );
_PROTOTYPE( clock_t get_uptime, (void)                                      );
_PROTOTYPE( unsigned long read_clock, (void)                                );
_PROTOTYPE( void set_timer, (struct timer *tp, clock_t t, tmr_func_t f) );
_PROTOTYPE( void reset_timer, (struct timer *tp)                            );

/* main.c */
_PROTOTYPE( void main, (void)                                               );
_PROTOTYPE( void prepare_shutdown, (int how)                                );
```

```

/* utility.c */
_PROTOTYPE( void kprintf, (const char *fmt, ...)                       );
_PROTOTYPE( void panic, (_CONST char *s, int n)                        );

/* proc.c */
_PROTOTYPE( int sys_call, (int function, int src_dest, message *m_ptr)  );
_PROTOTYPE( int lock_notify, (int src, int dst)                        );
_PROTOTYPE( int lock_send, (int dst, message *m_ptr)                   );
_PROTOTYPE( void lock_enqueue, (struct proc *rp)                       );
_PROTOTYPE( void lock_dequeue, (struct proc *rp)                       );

/* start.c */
_PROTOTYPE( void cstart, (U16_t cs, U16_t ds, U16_t mds,
                                U16_t parmoff, U16_t parmsize)         );

/* system.c */
_PROTOTYPE( int get_priv, (register struct proc *rc, int proc_type)     );
_PROTOTYPE( void send_sig, (int proc_nr, int sig_nr)                   );
_PROTOTYPE( void cause_sig, (int proc_nr, int sig_nr)                  );
_PROTOTYPE( void sys_task, (void)                                      );
_PROTOTYPE( void get_randomness, (int source)                          );
_PROTOTYPE( int virtual_copy, (struct vir_addr *src, struct vir_addr *dst,
                                vir_bytes bytes)                       );
#define numap_local(proc_nr, vir_addr, bytes) \
        umap_local(proc_addr(proc_nr), D, (vir_addr), (bytes))
_PROTOTYPE( phys_bytes umap_local, (struct proc *rp, int seg,
                vir_bytes vir_addr, vir_bytes bytes)                   );
_PROTOTYPE( phys_bytes umap_remote, (struct proc *rp, int seg,
                vir_bytes vir_addr, vir_bytes bytes)                   );
_PROTOTYPE( phys_bytes umap_bios, (struct proc *rp, vir_bytes vir_addr,
                vir_bytes bytes)                                       );

/* exception.c */
_PROTOTYPE( void exception, (unsigned vec_nr)                          );

/* i8259.c */
_PROTOTYPE( void intr_init, (int mine)                                 );
_PROTOTYPE( void intr_handle, (irq_hook_t *hook)                       );
_PROTOTYPE( void put_irq_handler, (irq_hook_t *hook, int irq,
                                                irq_handler_t handler)  );
_PROTOTYPE( void rm_irq_handler, (irq_hook_t *hook)                    );

/* klib*.s */
_PROTOTYPE( void int86, (void)                                         );
_PROTOTYPE( void cp_mess, (int src,phys_clicks src_clicks,vir_bytes src_offset,
                phys_clicks dst_clicks, vir_bytes dst_offset)          );
_PROTOTYPE( void enable_irq, (irq_hook_t *hook)                        );
_PROTOTYPE( int disable_irq, (irq_hook_t *hook)                        );
_PROTOTYPE( u16_t mem_rdw, (U16_t segm, vir_bytes offset)              );
_PROTOTYPE( void phys_copy, (phys_bytes source, phys_bytes dest,
                phys_bytes count)                                      );
_PROTOTYPE( void phys_memset, (phys_bytes source, unsigned long pattern,
                phys_bytes count)                                      );
_PROTOTYPE( void phys_insb, (U16_t port, phys_bytes buf, size_t count)  );
_PROTOTYPE( void phys_insw, (U16_t port, phys_bytes buf, size_t count)  );
_PROTOTYPE( void phys_outsb, (U16_t port, phys_bytes buf, size_t count) );
_PROTOTYPE( void phys_outsw, (U16_t port, phys_bytes buf, size_t count) );
_PROTOTYPE( void reset, (void)                                         );
_PROTOTYPE( void level0, (void (*func)(void))                          );
_PROTOTYPE( void monitor, (void)                                        );
_PROTOTYPE( void read_tsc, (unsigned long *high, unsigned long *low)    );
_PROTOTYPE( unsigned long read_cpu_flags, (void)                        );

/* mpx*.s */
_PROTOTYPE( void idle_task, (void)                                      );
_PROTOTYPE( void restart, (void)                                        );

/* The following are never called from C (pure asm procs). */

/* Exception handlers (real or protected mode), in numerical order. */
void _PROTOTYPE( int00, (void) ), _PROTOTYPE( divide_error, (void) );
void _PROTOTYPE( int01, (void) ), _PROTOTYPE( single_step_exception, (void) );
void _PROTOTYPE( int02, (void) ), _PROTOTYPE( nmi, (void) );
void _PROTOTYPE( int03, (void) ), _PROTOTYPE( breakpoint_exception, (void) );
void _PROTOTYPE( int04, (void) ), _PROTOTYPE( overflow, (void) );
void _PROTOTYPE( int05, (void) ), _PROTOTYPE( bounds_check, (void) );
void _PROTOTYPE( int06, (void) ), _PROTOTYPE( inval_opcode, (void) );
void _PROTOTYPE( int07, (void) ), _PROTOTYPE( copr_not_available, (void) );
void                              _PROTOTYPE( double_fault, (void) );
void                              _PROTOTYPE( copr_seg_overrun, (void) );
void                              _PROTOTYPE( inval_tss, (void) );
void                              _PROTOTYPE( segment_not_present, (void) );
void                              _PROTOTYPE( stack_exception, (void) );
void                              _PROTOTYPE( general_protection, (void) );
void                              _PROTOTYPE( page_fault, (void) );
void                              _PROTOTYPE( copr_error, (void) );

/* Hardware interrupt handlers. */
_PROTOTYPE( void hwint00, (void) );
_PROTOTYPE( void hwint01, (void) );
_PROTOTYPE( void hwint02, (void) );
_PROTOTYPE( void hwint03, (void) );
_PROTOTYPE( void hwint04, (void) );
_PROTOTYPE( void hwint05, (void) );
_PROTOTYPE( void hwint06, (void) );
_PROTOTYPE( void hwint07, (void) );
_PROTOTYPE( void hwint08, (void) );
_PROTOTYPE( void hwint09, (void) );
_PROTOTYPE( void hwint10, (void) );
_PROTOTYPE( void hwint11, (void) );
_PROTOTYPE( void hwint12, (void) );
_PROTOTYPE( void hwint13, (void) );
_PROTOTYPE( void hwint14, (void) );
_PROTOTYPE( void hwint15, (void) );

/* Software interrupt handlers, in numerical order. */
_PROTOTYPE( void trp, (void) );
_PROTOTYPE( void s_call, (void) ), _PROTOTYPE( p_s_call, (void) );
_PROTOTYPE( void level0_call, (void) );

/* protect.c */
_PROTOTYPE( void prot_init, (void)                                      );
_PROTOTYPE( void init_codeseg, (struct segdesc_s *segdp, phys_bytes base,
                vir_bytes size, int privilege)                          );
_PROTOTYPE( void init_dataseg, (struct segdesc_s *segdp, phys_bytes base,
                vir_bytes size, int privilege)                          );
_PROTOTYPE( phys_bytes seg2phys, (U16_t seg)                            );
_PROTOTYPE( void phys2seg, (u16_t *seg, vir_bytes *off, phys_bytes phys));
_PROTOTYPE( void enable_iop, (struct proc *pp)                          );
_PROTOTYPE( void alloc_segments, (struct proc *rp)                      );

#endif /* PROTO_H */


++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                  kernel/glo.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

#ifndef GLO_H
#define GLO_H

/* Global variables used in the kernel. This file contains the declarations;
 * storage space for the variables is allocated in table.c, because EXTERN is
 * defined as extern unless the _TABLE definition is seen. We rely on the
 * compiler's default initialization (0) for several global variables.
 */
#ifdef _TABLE
#undef EXTERN
#define EXTERN
#endif

#include <minix/config.h>
#include "config.h"

/* Variables relating to shutting down MINIX. */
EXTERN char kernel_exception;           /* TRUE after system exceptions */
EXTERN char shutdown_started;           /* TRUE after shutdowns / reboots */

/* Kernel information structures. This groups vital kernel information. */
EXTERN phys_bytes aout;                 /* address of a.out headers */
EXTERN struct kinfo kinfo;              /* kernel information for users */
EXTERN struct machine machine;          /* machine information for users */
EXTERN struct kmessages kmess;          /* diagnostic messages in kernel */
EXTERN struct randomness krandom;       /* gather kernel random information */

/* Process scheduling information and the kernel reentry count. */
EXTERN struct proc *prev_ptr;   /* previously running process */
EXTERN struct proc *proc_ptr;   /* pointer to currently running process */
EXTERN struct proc *next_ptr;   /* next process to run after restart() */
EXTERN struct proc *bill_ptr;   /* process to bill for clock ticks */
EXTERN char k_reenter;          /* kernel reentry count (entry count less 1) */
EXTERN unsigned lost_ticks;     /* clock ticks counted outside clock task */

/* Interrupt related variables. */
EXTERN irq_hook_t irq_hooks[NR_IRQ_HOOKS];      /* hooks for general use */
EXTERN irq_hook_t *irq_handlers[NR_IRQ_VECTORS];/* list of IRQ handlers */
EXTERN int irq_actids[NR_IRQ_VECTORS];          /* IRQ ID bits active */
EXTERN int irq_use;                             /* map of all in-use irq's */

/* Miscellaneous. */
EXTERN reg_t mon_ss, mon_sp;            /* boot monitor stack */
EXTERN int mon_return;                  /* true if we can return to monitor */

/* Variables that are initialized elsewhere are just extern here. */
extern struct boot_image image[];       /* system image processes */
extern char *t_stack[];                 /* task stack space */
extern struct segdesc_s gdt[];          /* global descriptor table */

```

```
EXTERN _PROTOTYPE( void (*level0_func), (void) );

#endif /* GLO_H */





```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                  kernel/ipc.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
#ifndef IPC_H
#define IPC_H

/* This header file defines constants for MINIX inter-process communication.
 * These definitions are used in the file proc.c.
 */
#include <minix/com.h>

/* Masks and flags for system calls. */
#define SYSCALL_FUNC    0x0F    /* mask for system call function */
#define SYSCALL_FLAGS   0xF0    /* mask for system call flags */
#define NON_BLOCKING    0x10    /* prevent blocking, return error */

/* System call numbers that are passed when trapping to the kernel. The
 * numbers are carefully defined so that it can easily be seen (based on
 * the bits that are on) which checks should be done in sys_call().
 */
#define SEND               1    /* 0 0 0 1 : blocking send */
#define RECEIVE            2    /* 0 0 1 0 : blocking receive */
#define SENDREC            3    /* 0 0 1 1 : SEND + RECEIVE */
#define NOTIFY             4    /* 0 1 0 0 : nonblocking notify */
#define ECHO               8    /* 1 0 0 0 : echo a message */

/* The following bit masks determine what checks that should be done. */
#define CHECK_PTR       0x0B    /* 1 0 1 1 : validate message buffer */
#define CHECK_DST       0x05    /* 0 1 0 1 : validate message destination */
#define CHECK_SRC       0x02    /* 0 0 1 0 : validate message source */

#endif /* IPC_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                  kernel/proc.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
#ifndef PROC_H
#define PROC_H

/* Here is the declaration of the process table.  It contains all process
 * data, including registers, flags, scheduling priority, memory map,
 * accounting, message passing (IPC) information, and so on.
 *
 * Many assembly code routines reference fields in it.  The offsets to these
 * fields are defined in the assembler include file sconst.h.  When changing
 * struct proc, be sure to change sconst.h to match.
 */
#include <minix/com.h>
#include "protect.h"
#include "const.h"
#include "priv.h"

struct proc {
  struct stackframe_s p_reg;    /* process' registers saved in stack frame */
  reg_t p_ldt_sel;              /* selector in gdt with ldt base and limit */
  struct segdesc_s p_ldt[2+NR_REMOTE_SEGS]; /* CS, DS and remote segments */

  proc_nr_t p_nr;               /* number of this process (for fast access) */
  struct priv *p_priv;          /* system privileges structure */
  char p_rts_flags;             /* SENDING, RECEIVING, etc. */

  char p_priority;              /* current scheduling priority */
  char p_max_priority;          /* maximum scheduling priority */
  char p_ticks_left;            /* number of scheduling ticks left */
  char p_quantum_size;          /* quantum size in ticks */

  struct mem_map p_memmap[NR_LOCAL_SEGS];   /* memory map (T, D, S) */

  clock_t p_user_time;          /* user time in ticks */
  clock_t p_sys_time;           /* sys time in ticks */

  struct proc *p_nextready;     /* pointer to next ready process */
  struct proc *p_caller_q;      /* head of list of procs wishing to send */
  struct proc *p_q_link;        /* link to next proc wishing to send */
  message *p_messbuf;           /* pointer to passed message buffer */
  proc_nr_t p_getfrom;          /* from whom does process want to receive? */
  proc_nr_t p_sendto;           /* to whom does process want to send? */

  sigset_t p_pending;           /* bit map for pending kernel signals */

  char p_name[P_NAME_LEN];      /* name of the process, including \0 */
};

/* Bits for the runtime flags. A process is runnable iff p_rts_flags == 0. */
#define SLOT_FREE       0x01    /* process slot is free */
#define NO_MAP          0x02    /* keeps unmapped forked child from running */
#define SENDING         0x04    /* process blocked trying to SEND */
#define RECEIVING       0x08    /* process blocked trying to RECEIVE */
#define SIGNALED        0x10    /* set when new kernel signal arrives */
#define SIG_PENDING     0x20    /* unready while signal being processed */
#define P_STOP          0x40    /* set when process is being traced */
#define NO_PRIV         0x80    /* keep forked system process from running */

/* Scheduling priorities for p_priority. Values must start at zero (highest
 * priority) and increment.  Priorities of the processes in the boot image
 * can be set in table.c. IDLE must have a queue for itself, to prevent low
 * priority user processes to run round-robin with IDLE.
 */
#define NR_SCHED_QUEUES   16    /* MUST equal minimum priority + 1 */
#define TASK_Q             0    /* highest, used for kernel tasks */
#define MAX_USER_Q         0    /* highest priority for user processes */
#define USER_Q             7    /* default (should correspond to nice 0) */
#define MIN_USER_Q        14    /* minimum priority for user processes */
#define IDLE_Q            15    /* lowest, only IDLE process goes here */

/* Magic process table addresses. */
```

```
#define BEG_PROC_ADDR (&proc[0])
#define BEG_USER_ADDR (&proc[NR_TASKS])
#define END_PROC_ADDR (&proc[NR_TASKS + NR_PROCS])

#define NIL_PROC          ((struct proc *) 0)
#define NIL_SYS_PROC      ((struct proc *) 1)
#define cproc_addr(n)     (&(proc + NR_TASKS)[(n)])
#define proc_addr(n)      (pproc_addr + NR_TASKS)[(n)]
#define proc_nr(p)        ((p)->p_nr)

#define isokprocn(n)      ((unsigned) ((n) + NR_TASKS) < NR_PROCS + NR_TASKS)
#define isemptyn(n)       isemptyp(proc_addr(n))
#define isemptyp(p)       ((p)->p_rts_flags == SLOT_FREE)
#define iskernelp(p)      iskerneln((p)->p_nr)
#define iskerneln(n)      ((n) < 0)
#define isuserp(p)        isusern((p)->p_nr)
#define isusern(n)        ((n) >= 0)

/* The process table and pointers to process table slots. The pointers allow
 * faster access because now a process entry can be found by indexing the
 * pproc_addr array, while accessing an element i requires a multiplication
 * with sizeof(struct proc) to determine the address.
 */
EXTERN struct proc proc[NR_TASKS + NR_PROCS];   /* process table */
EXTERN struct proc *pproc_addr[NR_TASKS + NR_PROCS];
EXTERN struct proc *rdy_head[NR_SCHED_QUEUES]; /* ptrs to ready list headers */
EXTERN struct proc *rdy_tail[NR_SCHED_QUEUES]; /* ptrs to ready list tails */

#endif /* PROC_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                    kernel/sconst.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
! Miscellaneous constants used in assembler code.
W               =       _WORD_SIZE      ! Machine word size.

! Offsets in struct proc. They MUST match proc.h.
P_STACKBASE     =       0
GSREG           =       P_STACKBASE
FSREG           =       GSREG + 2       ! 386 introduces FS and GS segments
ESREG           =       FSREG + 2
DSREG           =       ESREG + 2
DIREG           =       DSREG + 2
SIREG           =       DIREG + W
BPREG           =       SIREG + W
STREG           =       BPREG + W       ! hole for another SP
BXREG           =       STREG + W
DXREG           =       BXREG + W
CXREG           =       DXREG + W
AXREG           =       CXREG + W
RETADR          =       AXREG + W       ! return address for save() call
PCREG           =       RETADR + W
CSREG           =       PCREG + W
PSWREG          =       CSREG + W
SPREG           =       PSWREG + W
SSREG           =       SPREG + W
P_STACKTOP      =       SSREG + W
P_LDT_SEL       =       P_STACKTOP
```

```
P_LDT           =       P_LDT_SEL + W

Msize           =       9               ! size of a message in 32-bit words
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                      kernel/priv.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
#ifndef PRIV_H
#define PRIV_H

/* Declaration of the system privileges structure. It defines flags, system
 * call masks, an synchronous alarm timer, I/O privileges, pending hardware
 * interrupts and notifications, and so on.
 * System processes each get their own structure with properties, whereas all
 * user processes share one structure. This setup provides a clear separation
 * between common and privileged process fields and is very space efficient.
 *
 * Changes:
 *   Jul 01, 2005       Created.  (Jorrit N. Herder)
 */
#include <minix/com.h>
#include "protect.h"
#include "const.h"
#include "type.h"

struct priv {
  proc_nr_t s_proc_nr;          /* number of associated process */
  sys_id_t s_id;                /* index of this system structure */
  short s_flags;                /* PREEMTIBLE, BILLABLE, etc. */

  short s_trap_mask;            /* allowed system call traps */
  sys_map_t s_ipc_from;         /* allowed callers to receive from */
  sys_map_t s_ipc_to;           /* allowed destination processes */
  long s_call_mask;             /* allowed kernel calls */

  sys_map_t s_notify_pending;   /* bit map with pending notifications */
  irq_id_t s_int_pending;       /* pending hardware interrupts */
  sigset_t s_sig_pending;       /* pending signals */

  timer_t s_alarm_timer;        /* synchronous alarm timer */
  struct far_mem s_farmem[NR_REMOTE_SEGS];  /* remote memory map */
  reg_t *s_stack_guard;         /* stack guard word for kernel tasks */
};

/* Guard word for task stacks. */
#define STACK_GUARD     ((reg_t) (sizeof(reg_t) == 2 ? 0xBEEF : 0xDEADBEEF))

/* Bits for the system property flags. */
#define PREEMPTIBLE     0x01    /* kernel tasks are not preemptible */
#define BILLABLE        0x04    /* some processes are not billable */
#define SYS_PROC        0x10    /* system processes are privileged */
#define SENDREC_BUSY    0x20    /* sendrec() in progress */

/* Magic system structure table addresses. */
#define BEG_PRIV_ADDR (&priv[0])
#define END_PRIV_ADDR (&priv[NR_SYS_PROCS])

```

```
#define priv_addr(i)      (ppriv_addr)[(i)]
#define priv_id(rp)       ((rp)->p_priv->s_id)
#define priv(rp)          ((rp)->p_priv)

#define id_to_nr(id)    priv_addr(id)->s_proc_nr
#define nr_to_id(nr)    priv(proc_addr(nr))->s_id

/* The system structures table and pointers to individual table slots. The
 * pointers allow faster access because now a process entry can be found by
 * indexing the psys_addr array, while accessing an element i requires a
 * multiplication with sizeof(struct sys) to determine the address.
 */
EXTERN struct priv priv[NR_SYS_PROCS];          /* system properties table */
EXTERN struct priv *ppriv_addr[NR_SYS_PROCS];   /* direct slot pointers */

/* Unprivileged user processes all share the same privilege structure.
 * This id must be fixed because it is used to check send mask entries.
 */
#define USER_PRIV_ID    0

/* Make sure the system can boot. The following sanity check verifies that
 * the system privileges table is large enough for the number of processes
 * in the boot image.
 */
#if (NR_BOOT_PROCS > NR_SYS_PROCS)
#error NR_SYS_PROCS must be larger than NR_BOOT_PROCS
#endif

#endif /* PRIV_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                    kernel/protect.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/* Constants for protected mode. */

/* Table sizes. */
#define GDT_SIZE (FIRST_LDT_INDEX + NR_TASKS + NR_PROCS)
                                        /* spec. and LDT's */
#define IDT_SIZE (IRQ8_VECTOR + 8)      /* only up to the highest vector */
#define LDT_SIZE (2 + NR_REMOTE_SEGS)   /* CS, DS and remote segments */

/* Fixed global descriptors.  1 to 7 are prescribed by the BIOS. */
#define GDT_INDEX            1  /* GDT descriptor */
#define IDT_INDEX            2  /* IDT descriptor */
#define DS_INDEX             3  /* kernel DS */
#define ES_INDEX             4  /* kernel ES (386: flag 4 Gb at startup) */
#define SS_INDEX             5  /* kernel SS (386: monitor SS at startup) */
#define CS_INDEX             6  /* kernel CS */
#define MON_CS_INDEX         7  /* temp for BIOS (386: monitor CS at startup) */
#define TSS_INDEX            8  /* kernel TSS */
#define DS_286_INDEX         9  /* scratch 16-bit source segment */
#define ES_286_INDEX        10  /* scratch 16-bit destination segment */
#define A_INDEX             11  /* 64K memory segment at A0000 */
#define B_INDEX             12  /* 64K memory segment at B0000 */
#define C_INDEX             13  /* 64K memory segment at C0000 */
#define D_INDEX             14  /* 64K memory segment at D0000 */
#define FIRST_LDT_INDEX     15  /* rest of descriptors are LDT's */

```

```
#define GDT_SELECTOR      0x08  /* (GDT_INDEX * DESC_SIZE) bad for asld */
#define IDT_SELECTOR      0x10  /* (IDT_INDEX * DESC_SIZE) */
#define DS_SELECTOR       0x18  /* (DS_INDEX * DESC_SIZE) */
#define ES_SELECTOR       0x20  /* (ES_INDEX * DESC_SIZE) */
#define FLAT_DS_SELECTOR  0x21  /* less privileged ES */
#define SS_SELECTOR       0x28  /* (SS_INDEX * DESC_SIZE) */
#define CS_SELECTOR       0x30  /* (CS_INDEX * DESC_SIZE) */
#define MON_CS_SELECTOR   0x38  /* (MON_CS_INDEX * DESC_SIZE) */
#define TSS_SELECTOR      0x40  /* (TSS_INDEX * DESC_SIZE) */
#define DS_286_SELECTOR   0x49  /* (DS_286_INDEX*DESC_SIZE+TASK_PRIVILEGE) */
#define ES_286_SELECTOR   0x51  /* (ES_286_INDEX*DESC_SIZE+TASK_PRIVILEGE) */

/* Fixed local descriptors. */
#define CS_LDT_INDEX         0  /* process CS */
#define DS_LDT_INDEX         1  /* process DS=ES=FS=GS=SS */
#define EXTRA_LDT_INDEX      2  /* first of the extra LDT entries */

/* Privileges. */
#define INTR_PRIVILEGE       0  /* kernel and interrupt handlers */
#define TASK_PRIVILEGE       1  /* kernel tasks */
#define USER_PRIVILEGE       3  /* servers and user processes */

/* 286 hardware constants. */

/* Exception vector numbers. */
#define BOUNDS_VECTOR        5  /* bounds check failed */
#define INVAL_OP_VECTOR      6  /* invalid opcode */
#define COPROC_NOT_VECTOR    7  /* coprocessor not available */
#define DOUBLE_FAULT_VECTOR  8
#define COPROC_SEG_VECTOR    9  /* coprocessor segment overrun */
#define INVAL_TSS_VECTOR    10  /* invalid TSS */
#define SEG_NOT_VECTOR      11  /* segment not present */
#define STACK_FAULT_VECTOR  12  /* stack exception */
#define PROTECTION_VECTOR   13  /* general protection */

/* Selector bits. */
#define TI                0x04  /* table indicator */
#define RPL               0x03  /* requester privilege level */

/* Descriptor structure offsets. */
#define DESC_BASE            2  /* to base_low */
#define DESC_BASE_MIDDLE     4  /* to base_middle */
#define DESC_ACCESS          5  /* to access byte */
#define DESC_SIZE            8  /* sizeof (struct segdesc_s) */

/* Base and limit sizes and shifts. */
#define BASE_MIDDLE_SHIFT   16  /* shift for base --> base_middle */

/* Access-byte and type-byte bits. */
#define PRESENT           0x80  /* set for descriptor present */
#define DPL               0x60  /* descriptor privilege level mask */
#define DPL_SHIFT            5
#define SEGMENT           0x10  /* set for segment-type descriptors */

/* Access-byte bits. */
#define EXECUTABLE        0x08  /* set for executable segment */
#define CONFORMING        0x04  /* set for conforming segment if executable */
#define EXPAND_DOWN       0x04  /* set for expand-down segment if !executable*/
#define READABLE          0x02  /* set for readable segment if executable */
#define WRITEABLE         0x02  /* set for writeable segment if !executable */
```

```
#define TSS_BUSY          0x02  /* set if TSS descriptor is busy */
#define ACCESSED          0x01  /* set if segment accessed */

/* Special descriptor types. */
#define AVL_286_TSS          1  /* available 286 TSS */
#define LDT                  2  /* local descriptor table */
#define BUSY_286_TSS         3  /* set transparently to the software */
#define CALL_286_GATE        4  /* not used */
#define TASK_GATE            5  /* only used by debugger */
#define INT_286_GATE         6  /* interrupt gate, used for all vectors */
#define TRAP_286_GATE        7  /* not used */

/* Extra 386 hardware constants. */

/* Exception vector numbers. */
#define PAGE_FAULT_VECTOR   14
#define COPROC_ERR_VECTOR   16  /* coprocessor error */

/* Descriptor structure offsets. */
#define DESC_GRANULARITY     6  /* to granularity byte */
#define DESC_BASE_HIGH       7  /* to base_high */

/* Base and limit sizes and shifts. */
#define BASE_HIGH_SHIFT     24  /* shift for base --> base_high */
#define BYTE_GRAN_MAX   0xFFFFFL   /* maximum size for byte granular segment */
#define GRANULARITY_SHIFT   16  /* shift for limit --> granularity */
#define OFFSET_HIGH_SHIFT   16  /* shift for (gate) offset --> offset_high */
#define PAGE_GRAN_SHIFT     12  /* extra shift for page granular limits */

/* Type-byte bits. */
#define DESC_386_BIT  0x08 /* 386 types are obtained by ORing with this */
                            /* LDT's and TASK_GATE's don't need it */

/* Granularity byte. */
#define GRANULAR          0x80  /* set for 4K granularilty */
#define DEFAULT           0x40  /* set for 32-bit defaults (executable seg) */
#define BIG               0x40  /* set for "BIG" (expand-down seg) */
#define AVL               0x10  /* 0 for available */
#define LIMIT_HIGH        0x0F  /* mask for high bits of limit */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                  kernel/table.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/* The object file of "table.c" contains most kernel data. Variables that
 * are declared in the *.h files appear with EXTERN in front of them, as in
 *
 *    EXTERN int x;
 *
 * Normally EXTERN is defined as extern, so when they are included in another
 * file, no storage is allocated.  If EXTERN were not present, but just say,
 *
 *    int x;
 *
 * then including this file in several source files would cause 'x' to be
 * declared several times.  While some linkers accept this, others do not,
 * so they are declared extern when included normally.  However, it must be
 * declared for real somewhere.  That is done here, by redefining EXTERN as
 * the null string, so that inclusion of all *.h files in table.c actually
 * generates storage for them.
 *
 * Various variables could not be declared EXTERN, but are declared PUBLIC
 * or PRIVATE. The reason for this is that extern variables cannot have a
 * default initialization. If such variables are shared, they must also be
 * declared in one of the *.h files without the initialization.  Examples
 * include 'boot_image' (this file) and 'idt' and 'gdt' (protect.c).
 *
 * Changes:
 *    Aug 02, 2005   set privileges and minimal boot image (Jorrit N. Herder)
 *    Oct 17, 2004   updated above and tasktab comments  (Jorrit N. Herder)
 *    May 01, 2004   changed struct for system image  (Jorrit N. Herder)
 */
#define _TABLE

#include "kernel.h"
#include "proc.h"
#include "ipc.h"
#include <minix/com.h>
#include <ibm/int86.h>

/* Define stack sizes for the kernel tasks included in the system image. */
#define NO_STACK        0
#define SMALL_STACK     (128 * sizeof(char *))
#define IDL_S   SMALL_STACK     /* 3 intr, 3 temps, 4 db for Intel */
#define HRD_S   NO_STACK        /* dummy task, uses kernel stack */
#define TSK_S   SMALL_STACK     /* system and clock task */

/* Stack space for all the task stacks.  Declared as (char *) to align it. */
#define TOT_STACK_SPACE (IDL_S + HRD_S + (2 * TSK_S))
PUBLIC char *t_stack[TOT_STACK_SPACE / sizeof(char *)];

/* Define flags for the various process types. */
#define IDL_F   (SYS_PROC | PREEMPTIBLE | BILLABLE)     /* idle task */
#define TSK_F   (SYS_PROC)                              /* kernel tasks */
#define SRV_F   (SYS_PROC | PREEMPTIBLE)                /* system services */
#define USR_F   (BILLABLE | PREEMPTIBLE)                /* user processes */

/* Define system call traps for the various process types. These call masks
 * determine what system call traps a process is allowed to make.
 */
#define TSK_T   (1 << RECEIVE)                  /* clock and system */
#define SRV_T   (~0)                            /* system services */
#define USR_T   ((1 << SENDREC) | (1 << ECHO))  /* user processes */

/* Send masks determine to whom processes can send messages or notifications.
 * The values here are used for the processes in the boot image. We rely on
 * the initialization code in main() to match the s_nr_to_id() mapping for the
 * processes in the boot image, so that the send mask that is defined here
 * can be directly copied onto map[0] of the actual send mask. Privilege
 * structure 0 is shared by user processes.
 */
#define s(n)            (1 << s_nr_to_id(n))
#define SRV_M (~0)
#define SYS_M (~0)
#define USR_M (s(PM_PROC_NR) | s(FS_PROC_NR) | s(RS_PROC_NR))
#define DRV_M (USR_M | s(SYSTEM) | s(CLOCK) | s(LOG_PROC_NR) | s(TTY_PROC_NR))

/* Define kernel calls that processes are allowed to make. This is not looking
 * very nice, but we need to define the access rights on a per call basis.
 * Note that the reincarnation server has all bits on, because it should
 * be allowed to distribute rights to services that it starts.
 */
#define c(n)    (1 << ((n)-KERNEL_CALL))
#define RS_C    ~0
#define PM_C    ~(c(SYS_DEVIO) | c(SYS_SDEVIO) | c(SYS_VDEVIO) \
    | c(SYS_IRQCTL) | c(SYS_INT86))
#define FS_C    (c(SYS_KILL) | c(SYS_VIRCOPY) | c(SYS_VIRVCOPY) | c(SYS_UMAP) \
    | c(SYS_GETINFO) | c(SYS_EXIT) | c(SYS_TIMES) | c(SYS_SETALARM))
#define DRV_C   (FS_C | c(SYS_SEGCTL) | c(SYS_IRQCTL) | c(SYS_INT86) \
    | c(SYS_DEVIO) | c(SYS_VDEVIO) | c(SYS_SDEVIO))
#define MEM_C   (DRV_C | c(SYS_PHYSCOPY) | c(SYS_PHYSVCOPY))

/* The system image table lists all programs that are part of the boot image.
 * The order of the entries here MUST agree with the order of the programs
 * in the boot image and all kernel tasks must come first.
 * Each entry provides the process number, flags, quantum size (qs), scheduling
 * queue, allowed traps, ipc mask, and a name for the process table. The
 * initial program counter and stack size is also provided for kernel tasks.
 */
PUBLIC struct boot_image image[] = {
/* process nr,   pc, flags, qs,  queue, stack, traps, ipcto, call,  name */
 { IDLE,  idle_task, IDL_F,  8, IDLE_Q, IDL_S,     0,     0,     0, "IDLE"  },
 { CLOCK,clock_task, TSK_F, 64, TASK_Q, TSK_S, TSK_T,     0,     0, "CLOCK" },
 { SYSTEM, sys_task, TSK_F, 64, TASK_Q, TSK_S, TSK_T,     0,     0, "SYSTEM"},
 { HARDWARE,      0, TSK_F, 64, TASK_Q, HRD_S,     0,     0,     0, "KERNEL"},
 { PM_PROC_NR,    0, SRV_F, 32,      3, 0,     SRV_T, SRV_M,  PM_C, "pm"    },
 { FS_PROC_NR,    0, SRV_F, 32,      4, 0,     SRV_T, SRV_M,  FS_C, "fs"    },
 { RS_PROC_NR,    0, SRV_F,  4,      3, 0,     SRV_T, SYS_M,  RS_C, "rs"    },
 { TTY_PROC_NR,   0, SRV_F,  4,      1, 0,     SRV_T, SYS_M, DRV_C, "tty"   },
 { MEM_PROC_NR,   0, SRV_F,  4,      2, 0,     SRV_T, DRV_M, MEM_C, "memory"},
 { LOG_PROC_NR,   0, SRV_F,  4,      2, 0,     SRV_T, SYS_M, DRV_C, "log"   },
 { DRVR_PROC_NR,  0, SRV_F,  4,      2, 0,     SRV_T, SYS_M, DRV_C, "driver"},
 { INIT_PROC_NR,  0, USR_F,  8, USER_Q, 0,     USR_T, USR_M,     0, "init"  },
};

/* Verify the size of the system image table at compile time. Also verify that
 * the first chunk of the ipc mask has enough bits to accommodate the processes
 * in the image.
 * If a problem is detected, the size of the 'dummy' array will be negative,
 * causing a compile time error. Note that no space is actually allocated
 * because 'dummy' is declared extern.
 */
extern int dummy[(NR_BOOT_PROCS==sizeof(image)/
        sizeof(struct boot_image))?1:-1];
extern int dummy[(BITCHUNK_BITS > NR_BOOT_PROCS - 1) ? 1 : -1];

```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                  kernel/mpx.s
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
#
! Chooses between the 8086 and 386 versions of the Minix startup code.

#include <minix/config.h>
#if _WORD_SIZE == 2
```

```
#include "mpx88.s"
#else
#include "mpx386.s"
#endif
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                    kernel/mpx386.s
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
#
! This file, mpx386.s, is included by mpx.s when Minix is compiled for
! 32-bit Intel CPUs. The alternative mpx88.s is compiled for 16-bit CPUs.

! This file is part of the lowest layer of the MINIX kernel.  (The other part
! is "proc.c".)  The lowest layer does process switching and message handling.
! Furthermore it contains the assembler startup code for Minix and the 32-bit
! interrupt handlers.  It cooperates with the code in "start.c" to set up a
! good environment for main().

! Every transition to the kernel goes through this file.  Transitions to the
! kernel may be nested.  The initial entry may be with a system call (i.e.,
! send or receive a message), an exception or a hardware interrupt;  kernel
! reentries may only be made by hardware interrupts.  The count of reentries
! is kept in "k_reenter". It is important for deciding whether to switch to
! the kernel stack and for protecting the message passing code in "proc.c".

! For the message passing trap, most of the machine state is saved in the
! proc table.  (Some of the registers need not be saved.)  Then the stack is
! switched to "k_stack", and interrupts are reenabled.  Finally, the system
! call handler (in C) is called.  When it returns, interrupts are disabled
! again and the code falls into the restart routine, to finish off held-up
! interrupts and run the process or task whose pointer is in "proc_ptr".

! Hardware interrupt handlers do the same, except  (1) The entire state must
! be saved.  (2) There are too many handlers to do this inline, so the save
! routine is called.  A few cycles are saved by pushing the address of the
! appropiate restart routine for a return later.  (3) A stack switch is
! avoided when the stack is already switched.  (4) The (master) 8259 interrupt
! controller is reenabled centrally in save().  (5) Each interrupt handler
! masks its interrupt line using the 8259 before enabling (other unmasked)
! interrupts, and unmasks it after servicing the interrupt.  This limits the
! nest level to the number of lines and protects the handler from itself.

! For communication with the boot monitor at startup time some constant
! data are compiled into the beginning of the text segment. This facilitates
! reading the data at the start of the boot process, since only the first
! sector of the file needs to be read.

! Some data storage is also allocated at the end of this file. This data
! will be at the start of the data segment of the kernel and will be read
! and modified by the boot monitor before the kernel starts.

! sections

.sect .text
begtext:
.sect .rom
begrom:
.sect .data
```

```
begdata:
.sect .bss
begbss:

#include <minix/config.h>
#include <minix/const.h>
#include <minix/com.h>
#include <ibm/interrupt.h>
#include "const.h"
#include "protect.h"
#include "sconst.h"

/* Selected 386 tss offsets. */
#define TSS3_S_SP0      4

! Exported functions
! Note: in assembly language the .define statement applied to a function name
! is loosely equivalent to a prototype in C code -- it makes it possible to
! link to an entity declared in the assembly code but does not create
! the entity.

.define _restart
.define save

.define _divide_error
.define _single_step_exception
.define _nmi
.define _breakpoint_exception
.define _overflow
.define _bounds_check
.define _inval_opcode
.define _copr_not_available
.define _double_fault
.define _copr_seg_overrun
.define _inval_tss
.define _segment_not_present
.define _stack_exception
.define _general_protection
.define _page_fault
.define _copr_error

.define _hwint00        ! handlers for hardware interrupts
.define _hwint01
.define _hwint02
.define _hwint03
.define _hwint04
.define _hwint05
.define _hwint06
.define _hwint07
.define _hwint08
.define _hwint09
.define _hwint10
.define _hwint11
.define _hwint12
.define _hwint13
.define _hwint14
.define _hwint15

.define _s_call
.define _p_s_call
```

```
.define _level0_call

! Exported variables.
.define begbss
.define begdata

.sect .text
!*============================================================================*
!*                              MINIX                                         *
!*============================================================================*
MINIX:                          ! this is the entry point for the MINIX kernel
        jmp     over_flags      ! skip over the next few bytes
        .data2  CLICK_SHIFT     ! for the monitor: memory granularity
flags:
        .data2  0x01FD          ! boot monitor flags:
                                !       call in 386 mode, make bss, make stack,
                                !       load high, don't patch, will return,
                                !       uses generic INT, memory vector,
                                !       new boot code return
        nop                     ! extra byte to sync up disassembler
over_flags:

! Set up a C stack frame on the monitor stack.  (The monitor sets cs and ds
! right.  The ss descriptor still references the monitor data segment.)
        movzx   esp, sp         ! monitor stack is a 16 bit stack
        push    ebp
        mov     ebp, esp
        push    esi
        push    edi
        cmp     4(ebp), 0       ! monitor return vector is
        jz      noret           ! nonzero if return possible
        inc     (_mon_return)
noret:  mov     (_mon_sp), esp  ! save stack pointer for later return

! Copy the monitor global descriptor table to the address space of kernel and
! switch over to it.  Prot_init() can then update it with immediate effect.

        sgdt    (_gdt+GDT_SELECTOR)             ! get the monitor gdtr
        mov     esi, (_gdt+GDT_SELECTOR+2)      ! absolute address of GDT
        mov     ebx, _gdt                       ! address of kernel GDT
        mov     ecx, 8*8                        ! copying eight descriptors
copygdt:
 eseg   movb    al, (esi)
        movb    (ebx), al
        inc     esi
        inc     ebx
        loop    copygdt
        mov     eax, (_gdt+DS_SELECTOR+2)       ! base of kernel data
        and     eax, 0x00FFFFFF                 ! only 24 bits
        add     eax, _gdt                       ! eax = vir2phys(gdt)
        mov     (_gdt+GDT_SELECTOR+2), eax      ! set base of GDT
        lgdt    (_gdt+GDT_SELECTOR)             ! switch over to kernel GDT

! Locate boot parameters, set up kernel segment registers and stack.
        mov     ebx, 8(ebp)     ! boot parameters offset
        mov     edx, 12(ebp)    ! boot parameters length
        mov     eax, 16(ebp)    ! address of a.out headers
        mov     (_aout), eax
        mov     ax, ds          ! kernel data
        mov     es, ax
        mov     fs, ax
        mov     gs, ax
        mov     ss, ax
        mov     esp, k_stktop   ! set sp to point to the top of kernel stack

! Call C startup code to set up a proper environment to run main().
        push    edx
        push    ebx
        push    SS_SELECTOR
        push    DS_SELECTOR
        push    CS_SELECTOR
        call    _cstart         ! cstart(cs, ds, mds, parmoff, parmlen)
        add     esp, 5*4

! Reload gdtr, idtr and the segment registers to global descriptor table set
! up by prot_init().

        lgdt    (_gdt+GDT_SELECTOR)
        lidt    (_gdt+IDT_SELECTOR)

        jmpf    CS_SELECTOR:csinit
csinit:
    o16 mov     ax, DS_SELECTOR
        mov     ds, ax
        mov     es, ax
        mov     fs, ax
        mov     gs, ax
        mov     ss, ax
    o16 mov     ax, TSS_SELECTOR        ! no other TSS is used
        ltr     ax
        push    0                       ! set flags to known good state
        popf                            ! esp, clear nested task and int enable

        jmp     _main                   ! main()


!*===========================================================================*
!*                              interrupt handlers                           *
!*              interrupt handlers for 386 32-bit protected mode             *
!*===========================================================================*

!*===========================================================================*
!*                              hwint00 - 07                                 *
!*===========================================================================*
! Note this is a macro, it just looks like a subroutine.
#define hwint_master(irq)       \
        call    save                    /* save interrupted process state */;\
        push    (_irq_handlers+4*irq)   /* irq_handlers[irq]              */;\
        call    _intr_handle            /* intr_handle(irq_handlers[irq]) */;\
        pop     ecx                                                         ;\
        cmp     (_irq_actids+4*irq), 0  /* interrupt still active?        */;\
        jz      0f                                                          ;\
        inb     INT_CTLMASK             /* get current mask */              ;\
        orb     al, [1<<irq]            /* mask irq */                      ;\
        outb    INT_CTLMASK             /* disable the irq                */;\
0:      movb    al, END_OF_INT                                              ;\
        outb    INT_CTL                 /* reenable master 8259           */;\
        ret                             /* restart (another) process      */

! Each of these entry points is an expansion of the hwint_master macro
```

```
        .align  16
_hwint00:               ! Interrupt routine for irq 0 (the clock).
        hwint_master(0)

        .align  16
_hwint01:               ! Interrupt routine for irq 1 (keyboard)
        hwint_master(1)

        .align  16
_hwint02:               ! Interrupt routine for irq 2 (cascade!)
        hwint_master(2)

        .align  16
_hwint03:               ! Interrupt routine for irq 3 (second serial)
        hwint_master(3)

        .align  16
_hwint04:               ! Interrupt routine for irq 4 (first serial)
        hwint_master(4)

        .align  16
_hwint05:               ! Interrupt routine for irq 5 (XT winchester)
        hwint_master(5)

        .align  16
_hwint06:               ! Interrupt routine for irq 6 (floppy)
        hwint_master(6)

        .align  16
_hwint07:               ! Interrupt routine for irq 7 (printer)
        hwint_master(7)

!*===========================================================================*
!*                              hwint08 - 15                                 *
!*===========================================================================*
! Note this is a macro, it just looks like a subroutine.
#define hwint_slave(irq)        \
        call    save                    /* save interrupted process state */;\
        push    (_irq_handlers+4*irq)   /* irq_handlers[irq]              */;\
        call    _intr_handle            /* intr_handle(irq_handlers[irq]) */;\
        pop     ecx                                                         ;\
        cmp     (_irq_actids+4*irq), 0  /* interrupt still active?        */;\
        jz      0f                                                          ;\
        inb     INT2_CTLMASK                                                ;\
        orb     al, [1<<[irq-8]]                                            ;\
        outb    INT2_CTLMASK            /* disable the irq                */;\
0:      movb    al, END_OF_INT                                              ;\
        outb    INT_CTL                 /* reenable master 8259           */;\
        outb    INT2_CTL                /* reenable slave 8259            */;\
        ret                             /* restart (another) process      */

! Each of these entry points is an expansion of the hwint_slave macro
        .align  16
_hwint08:               ! Interrupt routine for irq 8 (realtime clock)
        hwint_slave(8)

        .align  16
_hwint09:               ! Interrupt routine for irq 9 (irq 2 redirected)
        hwint_slave(9)

        .align  16
_hwint10:               ! Interrupt routine for irq 10
        hwint_slave(10)

        .align  16
_hwint11:               ! Interrupt routine for irq 11
        hwint_slave(11)

        .align  16
_hwint12:               ! Interrupt routine for irq 12
        hwint_slave(12)

        .align  16
_hwint13:               ! Interrupt routine for irq 13 (FPU exception)
        hwint_slave(13)

        .align  16
_hwint14:               ! Interrupt routine for irq 14 (AT winchester)
        hwint_slave(14)

        .align  16
_hwint15:               ! Interrupt routine for irq 15
        hwint_slave(15)

!*===========================================================================*
!*                              save                                         *
!*===========================================================================*
! Save for protected mode.
! This is much simpler than for 8086 mode, because the stack already points
! into the process table, or has already been switched to the kernel stack.

        .align  16
save:
        cld                     ! set direction flag to a known value
        pushad                  ! save "general" registers
    o16 push    ds              ! save ds
    o16 push    es              ! save es
    o16 push    fs              ! save fs
    o16 push    gs              ! save gs
        mov     dx, ss          ! ss is kernel data segment
        mov     ds, dx          ! load rest of kernel segments
        mov     es, dx          ! kernel does not use fs, gs
        mov     eax, esp        ! prepare to return
        incb    (_k_reenter)    ! from -1 if not reentering
        jnz     set_restart1    ! stack is already kernel stack
        mov     esp, k_stktop
        push    _restart        ! build return address for int handler
        xor     ebp, ebp        ! for stacktrace
        jmp     RETADR-P_STACKBASE(eax)

        .align  4
set_restart1:
        push    restart1
        jmp     RETADR-P_STACKBASE(eax)

!*===========================================================================*
!*                              _s_call                                      *
!*===========================================================================*
        .align  16
_s_call:
```

```
_p_s_call:
        cld                     ! set direction flag to a known value
        sub     esp, 6*4        ! skip RETADR, eax, ecx, edx, ebx, est
        push    ebp             ! stack already points into proc table
        push    esi
        push    edi
    o16 push    ds
    o16 push    es
    o16 push    fs
    o16 push    gs
        mov     dx, ss
        mov     ds, dx
        mov     es, dx
        incb    (_k_reenter)
        mov     esi, esp        ! assumes P_STACKBASE == 0
        mov     esp, k_stktop
        xor     ebp, ebp        ! for stacktrace
                                ! end of inline save
                                ! now set up parameters for sys_call()
        push    ebx             ! pointer to user message
        push    eax             ! src/dest
        push    ecx             ! SEND/RECEIVE/BOTH
        call    _sys_call       ! sys_call(function, src_dest, m_ptr)
                                ! caller is now explicitly in proc_ptr
        mov     AXREG(esi), eax ! sys_call MUST PRESERVE si

! Fall into code to restart proc/task running.

!*===========================================================================*
!*                              restart                                      *
!*===========================================================================*
_restart:

! Restart the current process or the next process if it is set.

        cmp     (_next_ptr), 0          ! see if another process is scheduled
        jz      0f
        mov     eax, (_next_ptr)
        mov     (_proc_ptr), eax        ! schedule new process
        mov     (_next_ptr), 0
0:      mov     esp, (_proc_ptr)        ! will assume P_STACKBASE == 0
        lldt    P_LDT_SEL(esp)          ! enable process' segment descriptors
        lea     eax, P_STACKTOP(esp)    ! arrange for next interrupt
        mov     (_tss+TSS3_S_SP0), eax  ! to save state in process table
restart1:
        decb    (_k_reenter)
    o16 pop     gs
    o16 pop     fs
    o16 pop     es
    o16 pop     ds
        popad
        add     esp, 4          ! skip return adr
        iretd                   ! continue process

!*===========================================================================*
!*                              exception handlers                           *
!*===========================================================================*
_divide_error:
        push    DIVIDE_VECTOR
        jmp     exception
```

```

_single_step_exception:
        push    DEBUG_VECTOR
        jmp     exception

_nmi:
        push    NMI_VECTOR
        jmp     exception

_breakpoint_exception:
        push    BREAKPOINT_VECTOR
        jmp     exception

_overflow:
        push    OVERFLOW_VECTOR
        jmp     exception

_bounds_check:
        push    BOUNDS_VECTOR
        jmp     exception

_inval_opcode:
        push    INVAL_OP_VECTOR
        jmp     exception

_copr_not_available:
        push    COPROC_NOT_VECTOR
        jmp     exception

_double_fault:
        push    DOUBLE_FAULT_VECTOR
        jmp     errexception

_copr_seg_overrun:
        push    COPROC_SEG_VECTOR
        jmp     exception

_inval_tss:
        push    INVAL_TSS_VECTOR
        jmp     errexception

_segment_not_present:
        push    SEG_NOT_VECTOR
        jmp     errexception

_stack_exception:
        push    STACK_FAULT_VECTOR
        jmp     errexception

_general_protection:
        push    PROTECTION_VECTOR
        jmp     errexception

_page_fault:
        push    PAGE_FAULT_VECTOR
        jmp     errexception

_copr_error:
        push    COPROC_ERR_VECTOR
        jmp     exception
```

```

!*===========================================================================*
!*                              exception                                    *
!*===========================================================================*
! This is called for all exceptions which do not push an error code.

        .align  16
exception:
 sseg   mov     (trap_errno), 0         ! clear trap_errno
 sseg   pop     (ex_number)
        jmp     exception1

!*===========================================================================*
!*                              errexception                                 *
!*===========================================================================*
! This is called for all exceptions which push an error code.

        .align  16
errexception:
 sseg   pop     (ex_number)
 sseg   pop     (trap_errno)
exception1:                             ! Common for all exceptions.
        push    eax                     ! eax is scratch register
        mov     eax, 0+4(esp)           ! old eip
 sseg   mov     (old_eip), eax
        movzx   eax, 4+4(esp)           ! old cs
 sseg   mov     (old_cs), eax
        mov     eax, 8+4(esp)           ! old eflags
 sseg   mov     (old_eflags), eax
        pop     eax
        call    save
        push    (old_eflags)
        push    (old_cs)
        push    (old_eip)
        push    (trap_errno)
        push    (ex_number)
        call    _exception              ! (ex_number, trap_errno, old_eip,
                                        !       old_cs, old_eflags)
        add     esp, 5*4
        ret

!*===========================================================================*
!*                              level0_call                                  *
!*===========================================================================*
_level0_call:
        call    save
        jmp     (_level0_func)

!*===========================================================================*
!*                              data                                         *
!*===========================================================================*

.sect .rom      ! Before the string table please
        .data2  0x526F          ! this must be the first data entry (magic #)

.sect .bss
k_stack:
        .space  K_STACK_BYTES   ! kernel stack
k_stktop:                       ! top of kernel stack
        .comm   ex_number, 4
        .comm   trap_errno, 4
        .comm   old_eip, 4
        .comm   old_cs, 4
        .comm   old_eflags, 4
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                kernel/start.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/* This file contains the C startup code for Minix on Intel processors.
 * It cooperates with mpx.s to set up a good environment for main().
 *
 * This code runs in real mode for a 16 bit kernel and may have to switch
 * to protected mode for a 286.
 * For a 32 bit kernel this already runs in protected mode, but the selectors
 * are still those given by the BIOS with interrupts disabled, so the
 * descriptors need to be reloaded and interrupt descriptors made.
 */

#include "kernel.h"
#include "protect.h"
#include "proc.h"
#include <stdlib.h>
#include <string.h>

FORWARD _PROTOTYPE( char *get_value, (_CONST char *params, _CONST char *key));
/*===========================================================================*
 *                              cstart                                       *
 *===========================================================================*/
PUBLIC void cstart(cs, ds, mds, parmoff, parmsize)
U16_t cs, ds;                   /* kernel code and data segment */
U16_t mds;                      /* monitor data segment */
U16_t parmoff, parmsize;        /* boot parameters offset and length */
{
/* Perform system initializations prior to calling main(). Most settings are
 * determined with help of the environment strings passed by MINIX' loader.
 */
  char params[128*sizeof(char *)];              /* boot monitor parameters */
  register char *value;                         /* value in key=value pair */
  extern int etext, end;

  /* Decide if mode is protected; 386 or higher implies protected mode.
   * This must be done first, because it is needed for, e.g., seg2phys().
   * For 286 machines we cannot decide on protected mode, yet. This is
   * done below.
   */
#if _WORD_SIZE != 2
  machine.protected = 1;
#endif

  /* Record where the kernel and the monitor are. */
  kinfo.code_base = seg2phys(cs);
  kinfo.code_size = (phys_bytes) &etext;        /* size of code segment */
  kinfo.data_base = seg2phys(ds);
  kinfo.data_size = (phys_bytes) &end;          /* size of data segment */

  /* Initialize protected mode descriptors. */
  prot_init();

  /* Copy the boot parameters to the local buffer. */
  kinfo.params_base = seg2phys(mds) + parmoff;
  kinfo.params_size = MIN(parmsize,sizeof(params)-2);
  phys_copy(kinfo.params_base, vir2phys(params), kinfo.params_size);

  /* Record miscellaneous information for user-space servers. */
  kinfo.nr_procs = NR_PROCS;
  kinfo.nr_tasks = NR_TASKS;
  strncpy(kinfo.release, OS_RELEASE, sizeof(kinfo.release));
  kinfo.release[sizeof(kinfo.release)-1] = '\0';
  strncpy(kinfo.version, OS_VERSION, sizeof(kinfo.version));
  kinfo.version[sizeof(kinfo.version)-1] = '\0';
  kinfo.proc_addr = (vir_bytes) proc;
  kinfo.kmem_base = vir2phys(0);
  kinfo.kmem_size = (phys_bytes) &end;

  /* Processor?  86, 186, 286, 386, ...
   * Decide if mode is protected for older machines.
   */
  machine.processor=atoi(get_value(params, "processor"));
#if _WORD_SIZE == 2
  machine.protected = machine.processor >= 286;
#endif
  if (! machine.protected) mon_return = 0;

  /* XT, AT or MCA bus? */
  value = get_value(params, "bus");
  if (value == NIL_PTR || strcmp(value, "at") == 0) {
      machine.pc_at = TRUE;                        /* PC-AT compatible hardware */
  } else if (strcmp(value, "mca") == 0) {
      machine.pc_at = machine.ps_mca = TRUE;     /* PS/2 with micro channel */
  }

  /* Type of VDU: */
  value = get_value(params, "video");            /* EGA or VGA video unit */
  if (strcmp(value, "ega") == 0) machine.vdu_ega = TRUE;
  if (strcmp(value, "vga") == 0) machine.vdu_vga = machine.vdu_ega = TRUE;

  /* Return to assembler code to switch to protected mode (if 286),
   * reload selectors and call main().
   */
}

/*===========================================================================*
 *                              get_value                                    *
 *===========================================================================*/

PRIVATE char *get_value(params, name)
_CONST char *params;                              /* boot monitor parameters */
_CONST char *name;                                /* key to look up */
{
/* Get environment value - kernel version of getenv to avoid setting up the
 * usual environment array.
 */
  register _CONST char *namep;
  register char *envp;

  for (envp = (char *) params; *envp != 0;) {
        for (namep = name; *namep != 0 && *namep == *envp; namep++, envp++)
                ;
        if (*namep == '\0' && *envp == '=') return(envp + 1);
        while (*envp++ != 0)
                ;
  }
  return(NIL_PTR);
}
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                      kernel/main.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/* This file contains the main program of MINIX as well as its shutdown code.
 * The routine main() initializes the system and starts the ball rolling by
 * setting up the process table, interrupt vectors, and scheduling each task
 * to run to initialize itself.
 * The routine shutdown() does the opposite and brings down MINIX.
 *
 * The entries into this file are:
 *   main:              MINIX main program
 *   prepare_shutdown:  prepare to take MINIX down
 *
 * Changes:
 *   Nov 24, 2004   simplified main() with system image  (Jorrit N. Herder)
 *   Aug 20, 2004   new prepare_shutdown() and shutdown()  (Jorrit N. Herder)
 */
#include "kernel.h"
#include <signal.h>
#include <string.h>
#include <unistd.h>
#include <a.out.h>
#include <minix/callnr.h>
#include <minix/com.h>
#include "proc.h"

/* Prototype declarations for PRIVATE functions. */
FORWARD _PROTOTYPE( void announce, (void));
FORWARD _PROTOTYPE( void shutdown, (timer_t *tp));

/*===========================================================================*
 *                              main                                         *
 *===========================================================================*/
PUBLIC void main()
{
/* Start the ball rolling. */
  struct boot_image *ip;        /* boot image pointer */
  register struct proc *rp;     /* process pointer */
  register struct priv *sp;     /* privilege structure pointer */
  register int i, s;
  int hdrindex;                 /* index to array of a.out headers */
  phys_clicks text_base;
  vir_clicks text_clicks, data_clicks;
  reg_t ktsb;                   /* kernel task stack base */
  struct exec e_hdr;            /* for a copy of an a.out header */

  /* Initialize the interrupt controller. */
  intr_init(1);

  /* Clear the process table. Anounce each slot as empty and set up mappings
   * for proc_addr() and proc_nr() macros. Do the same for the table with
   * privilege structures for the system processes.
   */
  for (rp = BEG_PROC_ADDR, i = -NR_TASKS; rp < END_PROC_ADDR; ++rp, ++i) {
        rp->p_rts_flags = SLOT_FREE;            /* initialize free slot */
        rp->p_nr = i;                           /* proc number from ptr */
        (pproc_addr + NR_TASKS)[i] = rp;        /* proc ptr from number */
  }
  for (sp = BEG_PRIV_ADDR, i = 0; sp < END_PRIV_ADDR; ++sp, ++i) {
        sp->s_proc_nr = NONE;                   /* initialize as free */
        sp->s_id = i;                           /* priv structure index */
        ppriv_addr[i] = sp;                     /* priv ptr from number */
  }

  /* Set up proc table entries for tasks and servers.  The stacks of the
   * kernel tasks are initialized to an array in data space.  The stacks
   * of the servers have been added to the data segment by the monitor, so
   * the stack pointer is set to the end of the data segment.  All the
   * processes are in low memory on the 8086.  On the 386 only the kernel
   * is in low memory, the rest is loaded in extended memory.
   */

  /* Task stacks. */
  ktsb = (reg_t) t_stack;

  for (i=0; i < NR_BOOT_PROCS; ++i) {
        ip = &image[i];                         /* process' attributes */
        rp = proc_addr(ip->proc_nr);            /* get process pointer */
        rp->p_max_priority = ip->priority;      /* max scheduling priority */
        rp->p_priority = ip->priority;          /* current priority */
        rp->p_quantum_size = ip->quantum;       /* quantum size in ticks */
        rp->p_ticks_left = ip->quantum;         /* current credit */
        strncpy(rp->p_name, ip->proc_name, P_NAME_LEN); /* set process name */
        (void) get_priv(rp, (ip->flags & SYS_PROC));    /* assign structure */
        priv(rp)->s_flags = ip->flags;                  /* process flags */
        priv(rp)->s_trap_mask = ip->trap_mask;          /* allowed traps */
        priv(rp)->s_call_mask = ip->call_mask;          /* kernel call mask */
        priv(rp)->s_ipc_to.chunk[0] = ip->ipc_to;       /* restrict targets */
        if (iskerneln(proc_nr(rp))) {           /* part of the kernel? */
                if (ip->stksize > 0) {          /* HARDWARE stack size is 0 */
                        rp->p_priv->s_stack_guard = (reg_t *) ktsb;
                        *rp->p_priv->s_stack_guard = STACK_GUARD;
                }
                ktsb += ip->stksize;    /* point to high end of stack */
                rp->p_reg.sp = ktsb;    /* this task's initial stack ptr */
                text_base = kinfo.code_base >> CLICK_SHIFT;
                                        /* processes that are in the kernel */
                hdrindex = 0;           /* all use the first a.out header */
        } else {
                hdrindex = 1 + i-NR_TASKS;      /* servers, drivers, INIT */
        }

        /* The bootstrap loader created an array of the a.out headers at
         * absolute address 'aout'. Get one element to e_hdr.
         */
        phys_copy(aout + hdrindex * A_MINHDR, vir2phys(&e_hdr),
                                                (phys_bytes) A_MINHDR);
        /* Convert addresses to clicks and build process memory map */
        text_base = e_hdr.a_syms >> CLICK_SHIFT;
        text_clicks = (e_hdr.a_text + CLICK_SIZE-1) >> CLICK_SHIFT;
        if (!(e_hdr.a_flags & A_SEP)) text_clicks = 0;     /* common I&D */
        data_clicks = (e_hdr.a_total + CLICK_SIZE-1) >> CLICK_SHIFT;
        rp->p_memmap[T].mem_phys = text_base;
        rp->p_memmap[T].mem_len  = text_clicks;
        rp->p_memmap[D].mem_phys = text_base + text_clicks;
        rp->p_memmap[D].mem_len  = data_clicks;
        rp->p_memmap[S].mem_phys = text_base + text_clicks + data_clicks;
        rp->p_memmap[S].mem_vir  = data_clicks; /* empty - stack is in data */

        /* Set initial register values.  The processor status word for tasks
         * is different from that of other processes because tasks can
         * access I/O; this is not allowed to less-privileged processes
         */
        rp->p_reg.pc = (reg_t) ip->initial_pc;
        rp->p_reg.psw = (iskernelp(rp)) ? INIT_TASK_PSW : INIT_PSW;

        /* Initialize the server stack pointer. Take it down one word
         * to give crtso.s something to use as "argc".
         */
        if (isusern(proc_nr(rp))) {             /* user-space process? */
                rp->p_reg.sp = (rp->p_memmap[S].mem_vir +
                                rp->p_memmap[S].mem_len) << CLICK_SHIFT;
                rp->p_reg.sp -= sizeof(reg_t);
        }

        /* Set ready. The HARDWARE task is never ready. */
        if (rp->p_nr != HARDWARE) {
                rp->p_rts_flags = 0;            /* runnable if no flags */
                lock_enqueue(rp);               /* add to scheduling queues */
        } else {
                rp->p_rts_flags = NO_MAP;       /* prevent from running */
        }

        /* Code and data segments must be allocated in protected mode. */
        alloc_segments(rp);
  }

  /* We're definitely not shutting down. */
  shutdown_started = 0;

  /* MINIX is now ready. All boot image processes are on the ready queue.
   * Return to the assembly code to start running the current process.
   */
  bill_ptr = proc_addr(IDLE);           /* it has to point somewhere */
  announce();                           /* print MINIX startup banner */
  restart();
}

/*===========================================================================*
 *                              announce                                     *
 *===========================================================================*/
PRIVATE void announce(void)
{
  /* Display the MINIX startup banner. */
  kprintf("MINIX %s.%s."
      "Copyright 2006, Vrije Universiteit, Amsterdam, The Netherlands\n",
      OS_RELEASE, OS_VERSION);

  /* Real mode, or 16/32-bit protected mode? */
  kprintf("Executing in %s mode.\n\n",
      machine.protected ? "32-bit protected" : "real");
}

/*===========================================================================*
 *                              prepare_shutdown                             *
 *===========================================================================*/
PUBLIC void prepare_shutdown(how)
int how;
{
/* This function prepares to shutdown MINIX. */
  static timer_t shutdown_timer;
  register struct proc *rp;
  message m;

  /* Show debugging dumps on panics. Make sure that the TTY task is still
   * available to handle them. This is done with help of a non-blocking send.
   * We rely on TTY to call sys_abort() when it is done with the dumps.
   */
  if (how == RBT_PANIC) {
      m.m_type = PANIC_DUMPS;
      if (nb_send(TTY_PROC_NR,&m)==OK)  /* don't block if TTY isn't ready */
          return;                       /* await sys_abort() from TTY */
  }

  /* Send a signal to all system processes that are still alive to inform
   * them that the MINIX kernel is shutting down. A proper shutdown sequence
   * should be implemented by a user-space server. This mechanism is useful
   * as a backup in case of system panics, so that system processes can still
   * run their shutdown code, e.g, to synchronize the FS or to let the TTY
   * switch to the first console.
   */
  kprintf("Sending SIGKSTOP to system processes ...\n");
  for (rp=BEG_PROC_ADDR; rp<END_PROC_ADDR; rp++) {
      if (!isemptyp(rp) && (priv(rp)->s_flags & SYS_PROC) && !iskernelp(rp))
          send_sig(proc_nr(rp), SIGKSTOP);
  }

  /* We're shutting down. Diagnostics may behave differently now. */
  shutdown_started = 1;

  /* Notify system processes of the upcoming shutdown and allow them to be
   * scheduled by setting a watchog timer that calls shutdown(). The timer
   * argument passes the shutdown status.
   */
  kprintf("MINIX will now be shut down ...\n");
  tmr_arg(&shutdown_timer)->ta_int = how;

  /* Continue after 1 second, to give processes a chance to get
   * scheduled to do shutdown work.
   */
  set_timer(&shutdown_timer, get_uptime() + HZ, shutdown);
}

/*===========================================================================*
 *                              shutdown                                     *
 *===========================================================================*/
PRIVATE void shutdown(tp)
timer_t *tp;
```

```
{
/* This function is called from prepare_shutdown or stop_sequence to bring
 * down MINIX. How to shutdown is in the argument: RBT_HALT (return to the
 * monitor), RBT_MONITOR (execute given code), RBT_RESET (hard reset).
 */
  int how = tmr_arg(tp)->ta_int;
  u16_t magic;

  /* Now mask all interrupts, including the clock, and stop the clock. */
  outb(INT_CTLMASK, ~0);
  clock_stop();

  if (mon_return && how != RBT_RESET) {
        /* Reinitialize the interrupt controllers to the BIOS defaults. */
        intr_init(0);
        outb(INT_CTLMASK, 0);
        outb(INT2_CTLMASK, 0);

        /* Return to the boot monitor. Set the program if not already done. */
        if (how != RBT_MONITOR) phys_copy(vir2phys(""), kinfo.params_base, 1);
        level0(monitor);
  }

  /* Reset the system by jumping to the reset address (real mode), or by
   * forcing a processor shutdown (protected mode). First stop the BIOS
   * memory test by setting a soft reset flag.
   */
  magic = STOP_MEM_CHECK;
  phys_copy(vir2phys(&magic), SOFT_RESET_FLAG_ADDR, SOFT_RESET_FLAG_SIZE);
  level0(reset);
}
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                    kernel/proc.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/* This file contains essentially all of the process and message handling.
 * Together with "mpx.s" it forms the lowest layer of the MINIX kernel.
 * There is one entry point from the outside:
 *
 *   sys_call:        a system call, i.e., the kernel is trapped with an INT
 *
 * As well as several entry points used from the interrupt and task level:
 *
 *   lock_notify:     notify a process of a system event
 *   lock_send:       send a message to a process
 *   lock_enqueue:    put a process on one of the scheduling queues
 *   lock_dequeue:    remove a process from the scheduling queues
 *
 * Changes:
 *   Aug 19, 2005     rewrote scheduling code  (Jorrit N. Herder)
 *   Jul 25, 2005     rewrote system call handling  (Jorrit N. Herder)
 *   May 26, 2005     rewrote message passing functions  (Jorrit N. Herder)
 *   May 24, 2005     new notification system call  (Jorrit N. Herder)
 *   Oct 28, 2004     nonblocking send and receive calls  (Jorrit N. Herder)
 *
 * The code here is critical to make everything work and is important for the
 * overall performance of the system. A large fraction of the code deals with
 * list manipulation. To make this both easy to understand and fast to execute
 * pointer pointers are used throughout the code. Pointer pointers prevent
 * exceptions for the head or tail of a linked list.
 *
 *  node_t *queue, *new_node;   // assume these as global variables
 *  node_t **xpp = &queue;      // get pointer pointer to head of queue
 *  while (*xpp != NULL)        // find last pointer of the linked list
 *      xpp = &(*xpp)->next;    // get pointer to next pointer
 *  *xpp = new_node;            // now replace the end (the NULL pointer)
 *  new_node->next = NULL;      // and mark the new end of the list
 *
 * For example, when adding a new node to the end of the list, one normally
 * makes an exception for an empty list and looks up the end of the list for
 * nonempty lists. As shown above, this is not required with pointer pointers.
 */

#include <minix/com.h>
#include <minix/callnr.h>
#include "kernel.h"
#include "proc.h"

/* Scheduling and message passing functions. The functions are available to
 * other parts of the kernel through lock_...(). The lock temporarily disables
 * interrupts to prevent race conditions.
 */
FORWARD _PROTOTYPE( int mini_send, (struct proc *caller_ptr, int dst,
                message *m_ptr, unsigned flags) );
FORWARD _PROTOTYPE( int mini_receive, (struct proc *caller_ptr, int src,
                message *m_ptr, unsigned flags) );
FORWARD _PROTOTYPE( int mini_notify, (struct proc *caller_ptr, int dst) );

FORWARD _PROTOTYPE( void enqueue, (struct proc *rp) );
FORWARD _PROTOTYPE( void dequeue, (struct proc *rp) );
FORWARD _PROTOTYPE( void sched, (struct proc *rp, int *queue, int *front) );
FORWARD _PROTOTYPE( void pick_proc, (void) );

#define BuildMess(m_ptr, src, dst_ptr) \
        (m_ptr)->m_source = (src);                                      \
        (m_ptr)->m_type = NOTIFY_FROM(src);                             \
        (m_ptr)->NOTIFY_TIMESTAMP = get_uptime();                       \
        switch (src) {                                                  \
        case HARDWARE:                                                  \
                (m_ptr)->NOTIFY_ARG = priv(dst_ptr)->s_int_pending;     \
                priv(dst_ptr)->s_int_pending = 0;                       \
                break;                                                  \
        case SYSTEM:                                                    \
                (m_ptr)->NOTIFY_ARG = priv(dst_ptr)->s_sig_pending;     \
                priv(dst_ptr)->s_sig_pending = 0;                       \
                break;                                                  \
        }

#define CopyMess(s,sp,sm,dp,dm) \
        cp_mess(s, (sp)->p_memmap[D].mem_phys,  \
                 (vir_bytes)sm, (dp)->p_memmap[D].mem_phys, (vir_bytes)dm)

```

```
/*===========================================================================*
 *                              sys_call                                     *
 *===========================================================================*/
PUBLIC int sys_call(call_nr, src_dst, m_ptr)
int call_nr;                    /* system call number and flags */
int src_dst;                    /* src to receive from or dst to send to */
message *m_ptr;                 /* pointer to message in the caller's space */
{
/* System calls are done by trapping to the kernel with an INT instruction.
 * The trap is caught and sys_call() is called to send or receive a message
 * (or both). The caller is always given by 'proc_ptr'.
 */
  register struct proc *caller_ptr = proc_ptr;  /* get pointer to caller */
  int function = call_nr & SYSCALL_FUNC;        /* get system call function */
  unsigned flags = call_nr & SYSCALL_FLAGS;     /* get flags */
  int mask_entry;                               /* bit to check in send mask */
  int result;                                   /* the system call's result */
  vir_clicks vlo, vhi;          /* virtual clicks containing message to send */

  /* Check if the process has privileges for the requested call. Calls to the
   * kernel may only be SENDREC, because tasks always reply and may not block
   * if the caller doesn't do receive().
   */
  if (! (priv(caller_ptr)->s_trap_mask & (1 << function)) ||
          (iskerneln(src_dst) && function != SENDREC
           && function != RECEIVE)) {
      kprintf("sys_call: trap %d not allowed, caller %d, src_dst %d\n",
          function, proc_nr(caller_ptr), src_dst);
      return(ECALLDENIED);              /* trap denied by mask or kernel */
  }

  /* Require a valid source and/ or destination process, unless echoing. */
  if (! (isokprocn(src_dst) || src_dst == ANY || function == ECHO)) {
      kprintf("sys_call: invalid src_dst, src_dst %d, caller %d\n",
          src_dst, proc_nr(caller_ptr));
      return(EBADSRCDST);               /* invalid process number */
  }

  /* If the call involves a message buffer, i.e., for SEND, RECEIVE, SENDREC,
   * or ECHO, check the message pointer. This check allows a message to be
   * anywhere in data or stack or gap. It will have to be made more elaborate
   * for machines which don't have the gap mapped.
   */
  if (function & CHECK_PTR) {
      vlo = (vir_bytes) m_ptr >> CLICK_SHIFT;
      vhi = ((vir_bytes) m_ptr + MESS_SIZE - 1) >> CLICK_SHIFT;
      if (vlo < caller_ptr->p_memmap[D].mem_vir || vlo > vhi ||
              vhi >= caller_ptr->p_memmap[S].mem_vir +
              caller_ptr->p_memmap[S].mem_len) {
          kprintf("sys_call: invalid message pointer, trap %d, caller %d\n",
                function, proc_nr(caller_ptr));
          return(EFAULT);               /* invalid message pointer */
      }
  }

  /* If the call is to send to a process, i.e., for SEND, SENDREC or NOTIFY,
   * verify that the caller is allowed to send to the given destination and
   * that the destination is still alive.
   */
  if (function & CHECK_DST) {
      if (! get_sys_bit(priv(caller_ptr)->s_ipc_to, nr_to_id(src_dst))) {
          kprintf("sys_call: ipc mask denied %d sending to %d\n",
                proc_nr(caller_ptr), src_dst);
          return(ECALLDENIED);          /* call denied by ipc mask */
      }

      if (isemptyn(src_dst) && !shutdown_started) {
          kprintf("sys_call: dead dest; %d, %d, %d\n",
              function, proc_nr(caller_ptr), src_dst);
          return(EDEADDST);             /* cannot send to the dead */
      }
  }

  /* Now check if the call is known and try to perform the request. The only
   * system calls that exist in MINIX are sending and receiving messages.
   *   - SENDREC: combines SEND and RECEIVE in a single system call
   *   - SEND:    sender blocks until its message has been delivered
   *   - RECEIVE: receiver blocks until an acceptable message has arrived
   *   - NOTIFY:  nonblocking call; deliver notification or mark pending
   *   - ECHO:    nonblocking call; directly echo back the message
   */
  switch(function) {
  case SENDREC:
      /* A flag is set so that notifications cannot interrupt SENDREC. */
      priv(caller_ptr)->s_flags |= SENDREC_BUSY;
      /* fall through */
  case SEND:
      result = mini_send(caller_ptr, src_dst, m_ptr, flags);
      if (function == SEND || result != OK) {
          break;                                /* done, or SEND failed */
      }                                         /* fall through for SENDREC */
  case RECEIVE:
      if (function == RECEIVE)
          priv(caller_ptr)->s_flags &= ~SENDREC_BUSY;
      result = mini_receive(caller_ptr, src_dst, m_ptr, flags);
      break;
  case NOTIFY:
      result = mini_notify(caller_ptr, src_dst);
      break;
  case ECHO:
      CopyMess(caller_ptr->p_nr, caller_ptr, m_ptr, caller_ptr, m_ptr);
      result = OK;
      break;
  default:
      result = EBADCALL;                        /* illegal system call */
  }

  /* Now, return the result of the system call to the caller. */
  return(result);
}

/*===========================================================================*
 *                              mini_send                                    *
 *===========================================================================*/
PRIVATE int mini_send(caller_ptr, dst, m_ptr, flags)
register struct proc *caller_ptr;       /* who is trying to send a message? */
int dst;                                /* to whom is message being sent? */
message *m_ptr;                         /* pointer to message buffer */
unsigned flags;                         /* system call flags */
{
```

```
/* Send a message from 'caller_ptr' to 'dst'. If 'dst' is blocked waiting
 * for this message, copy the message to it and unblock 'dst'. If 'dst' is
 * not waiting at all, or is waiting for another source, queue 'caller_ptr'.
 */
  register struct proc *dst_ptr = proc_addr(dst);
  register struct proc **xpp;
  register struct proc *xp;

  /* Check for deadlock by 'caller_ptr' and 'dst' sending to each other. */
  xp = dst_ptr;
  while (xp->p_rts_flags & SENDING) {           /* check while sending */
        xp = proc_addr(xp->p_sendto);           /* get xp's destination */
        if (xp == caller_ptr) return(ELOCKED);  /* deadlock if cyclic */
  }

  /* Check if 'dst' is blocked waiting for this message. The destination's
   * SENDING flag may be set when its SENDREC call blocked while sending.
   */
  if ( (dst_ptr->p_rts_flags & (RECEIVING | SENDING)) == RECEIVING &&
       (dst_ptr->p_getfrom == ANY || dst_ptr->p_getfrom == caller_ptr->p_nr)) {
        /* Destination is indeed waiting for this message. */
        CopyMess(caller_ptr->p_nr, caller_ptr, m_ptr, dst_ptr,
                 dst_ptr->p_messbuf);
        if ((dst_ptr->p_rts_flags &= ~RECEIVING) == 0) enqueue(dst_ptr);
  } else if ( ! (flags & NON_BLOCKING)) {
        /* Destination is not waiting.  Block and dequeue caller. */
        caller_ptr->p_messbuf = m_ptr;
        if (caller_ptr->p_rts_flags == 0) dequeue(caller_ptr);
        caller_ptr->p_rts_flags |= SENDING;
        caller_ptr->p_sendto = dst;

        /* Process is now blocked.  Put in on the destination's queue. */
        xpp = &dst_ptr->p_caller_q;             /* find end of list */
        while (*xpp != NIL_PROC) xpp = &(*xpp)->p_q_link;
        *xpp = caller_ptr;                      /* add caller to end */
        caller_ptr->p_q_link = NIL_PROC;        /* mark new end of list */
  } else {
        return(ENOTREADY);
  }
  return(OK);
}

/*===========================================================================*
 *                              mini_receive                                 *
 *===========================================================================*/
PRIVATE int mini_receive(caller_ptr, src, m_ptr, flags)
register struct proc *caller_ptr;       /* process trying to get message */
int src;                                /* which message source is wanted */
message *m_ptr;                         /* pointer to message buffer */
unsigned flags;                         /* system call flags */
{
/* A process or task wants to get a message.  If a message is already queued,
 * acquire it and deblock the sender.  If no message from the desired source
 * is available block the caller, unless the flags don't allow blocking.
 */
  register struct proc **xpp;
  register struct notification **ntf_q_pp;
  message m;
  int bit_nr;
  sys_map_t *map;
```

```
   bitchunk_t *chunk;
   int i, src_id, src_proc_nr;

   /* Check to see if a message from desired source is already available.
    * The caller's SENDING flag may be set if SENDREC couldn't send. If it is
    * set, the process should be blocked.
    */
   if (!(caller_ptr->p_rts_flags & SENDING)) {

     /* Check if there are pending notifications, except for SENDREC. */
     if (! (priv(caller_ptr)->s_flags & SENDREC_BUSY)) {

         map = &priv(caller_ptr)->s_notify_pending;
         for (chunk=&map->chunk[0]; chunk<&map->chunk[NR_SYS_CHUNKS]; chunk++) {

             /* Find a pending notification from the requested source. */
             if (! *chunk) continue;                     /* no bits in chunk */
             for (i=0; ! (*chunk & (1<<i)); ++i) {}      /* look up the bit */
             src_id = (chunk - &map->chunk[0]) * BITCHUNK_BITS + i;
             if (src_id >= NR_SYS_PROCS) break;          /* out of range */
             src_proc_nr = id_to_nr(src_id);             /* get source proc */
             if (src!=ANY && src!=src_proc_nr) continue; /* source not ok */
             *chunk &= ~(1 << i);                        /* no longer pending */

             /* Found a suitable source, deliver the notification message. */
             BuildMess(&m, src_proc_nr, caller_ptr);     /* assemble message */
             CopyMess(src_proc_nr, proc_addr(HARDWARE), &m, caller_ptr, m_ptr);
             return(OK);                                 /* report success */
         }
     }

     /* Check caller queue. Use pointer pointers to keep code simple. */
     xpp = &caller_ptr->p_caller_q;
     while (*xpp != NIL_PROC) {
         if (src == ANY || src == proc_nr(*xpp)) {
             /* Found acceptable message. Copy it and update status. */
             CopyMess((*xpp)->p_nr, *xpp, (*xpp)->p_messbuf, caller_ptr, m_ptr);
             if (((*xpp)->p_rts_flags &= ~SENDING) == 0) enqueue(*xpp);
             *xpp = (*xpp)->p_q_link;            /* remove from queue */
             return(OK);                         /* report success */
         }
         xpp = &(*xpp)->p_q_link;                /* proceed to next */
     }
   }

   /* No suitable message is available or the caller couldn't send in SENDREC.
    * Block the process trying to receive, unless the flags tell otherwise.
    */
   if ( ! (flags & NON_BLOCKING)) {
       caller_ptr->p_getfrom = src;
       caller_ptr->p_messbuf = m_ptr;
       if (caller_ptr->p_rts_flags == 0) dequeue(caller_ptr);
       caller_ptr->p_rts_flags |= RECEIVING;
       return(OK);
   } else {
       return(ENOTREADY);
   }
 }
```

```
/*===========================================================================*
 *                              mini_notify                                  *
 *===========================================================================*/
PRIVATE int mini_notify(caller_ptr, dst)
register struct proc *caller_ptr;       /* sender of the notification */
int dst;                                /* which process to notify */
{
  register struct proc *dst_ptr = proc_addr(dst);
  int src_id;                           /* source id for late delivery */
  message m;                            /* the notification message */

  /* Check to see if target is blocked waiting for this message. A process
   * can be both sending and receiving during a SENDREC system call.
   */
  if ((dst_ptr->p_rts_flags & (RECEIVING|SENDING)) == RECEIVING &&
      ! (priv(dst_ptr)->s_flags & SENDREC_BUSY) &&
      (dst_ptr->p_getfrom == ANY || dst_ptr->p_getfrom == caller_ptr->p_nr)) {

      /* Destination is indeed waiting for a message. Assemble a notification
       * message and deliver it. Copy from pseudo-source HARDWARE, since the
       * message is in the kernel's address space.
       */
      BuildMess(&m, proc_nr(caller_ptr), dst_ptr);
      CopyMess(proc_nr(caller_ptr), proc_addr(HARDWARE), &m,
          dst_ptr, dst_ptr->p_messbuf);
      dst_ptr->p_rts_flags &= ~RECEIVING;       /* deblock destination */
      if (dst_ptr->p_rts_flags == 0) enqueue(dst_ptr);
      return(OK);
  }

  /* Destination is not ready to receive the notification. Add it to the
   * bit map with pending notifications. Note the indirectness: the system id
   * instead of the process number is used in the pending bit map.
   */
  src_id = priv(caller_ptr)->s_id;
  set_sys_bit(priv(dst_ptr)->s_notify_pending, src_id);
  return(OK);
}

/*===========================================================================*
 *                              lock_notify                                  *
 *===========================================================================*/
PUBLIC int lock_notify(src, dst)
int src;                        /* sender of the notification */
int dst;                        /* who is to be notified */
{
/* Safe gateway to mini_notify() for tasks and interrupt handlers. The sender
 * is explicitly given to prevent confusion where the call comes from. MINIX
 * kernel is not reentrant, which means to interrupts are disabled after
 * the first kernel entry (hardware interrupt, trap, or exception). Locking
 * is done by temporarily disabling interrupts.
 */
  int result;

  /* Exception or interrupt occurred, thus already locked. */
  if (k_reenter >= 0) {
      result = mini_notify(proc_addr(src), dst);
  }

  /* Call from task level, locking is required. */
  else {
      lock(0, "notify");
      result = mini_notify(proc_addr(src), dst);
      unlock(0);
  }
  return(result);
}

/*===========================================================================*
 *                              enqueue                                      *
 *===========================================================================*/
PRIVATE void enqueue(rp)
register struct proc *rp;       /* this process is now runnable */
{
/* Add 'rp' to one of the queues of runnable processes.  This function is
 * responsible for inserting a process into one of the scheduling queues.
 * The mechanism is implemented here.   The actual scheduling policy is
 * defined in sched() and pick_proc().
 */
  int q;                                        /* scheduling queue to use */
  int front;                                    /* add to front or back */

  /* Determine where to insert to process. */
  sched(rp, &q, &front);

  /* Now add the process to the queue. */
  if (rdy_head[q] == NIL_PROC) {                /* add to empty queue */
      rdy_head[q] = rdy_tail[q] = rp;           /* create a new queue */
      rp->p_nextready = NIL_PROC;               /* mark new end */
  }
  else if (front) {                             /* add to head of queue */
      rp->p_nextready = rdy_head[q];            /* chain head of queue */
      rdy_head[q] = rp;                         /* set new queue head */
  }
  else {                                        /* add to tail of queue */
      rdy_tail[q]->p_nextready = rp;            /* chain tail of queue */
      rdy_tail[q] = rp;                         /* set new queue tail */
      rp->p_nextready = NIL_PROC;               /* mark new end */
  }

  /* Now select the next process to run. */
  pick_proc();
}

/*===========================================================================*
 *                              dequeue                                      *
 *===========================================================================*/
PRIVATE void dequeue(rp)
register struct proc *rp;       /* this process is no longer runnable */
{
/* A process must be removed from the scheduling queues, for example, because
 * it has blocked.  If the currently active process is removed, a new process
 * is picked to run by calling pick_proc().
 */
  register int q = rp->p_priority;              /* queue to use */
  register struct proc **xpp;                   /* iterate over queue */
  register struct proc *prev_xp;

  /* Side-effect for kernel: check if the task's stack still is ok? */
  if (iskernelp(rp)) {
        if (*priv(rp)->s_stack_guard != STACK_GUARD)
                panic("stack overrun by task", proc_nr(rp));
  }

  /* Now make sure that the process is not in its ready queue. Remove the
   * process if it is found. A process can be made unready even if it is not
   * running by being sent a signal that kills it.
   */
  prev_xp = NIL_PROC;
  for (xpp = &rdy_head[q]; *xpp != NIL_PROC; xpp = &(*xpp)->p_nextready) {

      if (*xpp == rp) {                         /* found process to remove */
          *xpp = (*xpp)->p_nextready;           /* replace with next chain */
          if (rp == rdy_tail[q])                /* queue tail removed */
              rdy_tail[q] = prev_xp;            /* set new tail */
          if (rp == proc_ptr || rp == next_ptr) /* active process removed */
              pick_proc();                      /* pick new process to run */
          break;
      }
      prev_xp = *xpp;                           /* save previous in chain */
  }
}

/*===========================================================================*
 *                              sched                                        *
 *===========================================================================*/
PRIVATE void sched(rp, queue, front)
register struct proc *rp;                       /* process to be scheduled */
int *queue;                                     /* return: queue to use */
int *front;                                     /* return: front or back */
{
/* This function determines the scheduling policy.  It is called whenever a
 * process must be added to one of the scheduling queues to decide where to
 * insert it.  As a side-effect the process' priority may be updated.
 */
  static struct proc *prev_ptr = NIL_PROC;      /* previous without time */
  int time_left = (rp->p_ticks_left > 0);       /* quantum fully consumed */
  int penalty = 0;                              /* change in priority */

  /* Check whether the process has time left. Otherwise give a new quantum
   * and possibly raise the priority.  Processes using multiple quantums
   * in a row get a lower priority to catch infinite loops in high priority
   * processes (system servers and drivers).
   */
  if ( ! time_left) {                           /* quantum consumed ? */
      rp->p_ticks_left = rp->p_quantum_size;    /* give new quantum */
      if (prev_ptr == rp) penalty ++;           /* catch infinite loops */
      else penalty --;                          /* give slow way back */
      prev_ptr = rp;                            /* store ptr for next */
  }

  /* Determine the new priority of this process. The bounds are determined
   * by IDLE's queue and the maximum priority of this process. Kernel tasks
   * and the idle process are never changed in priority.
   */
  if (penalty != 0 && ! iskernelp(rp)) {
      rp->p_priority += penalty;                /* update with penalty */
      if (rp->p_priority < rp->p_max_priority)  /* check upper bound */
          rp->p_priority=rp->p_max_priority;
      else if (rp->p_priority > IDLE_Q-1)       /* check lower bound */
```

```
          rp->p_priority = IDLE_Q-1;
  }

  /* If there is time left, the process is added to the front of its queue,
   * so that it can immediately run. The queue to use simply is always the
   * process' current priority.
   */
  *queue = rp->p_priority;
  *front = time_left;
}

/*===========================================================================*
 *                              pick_proc                                    *
 *===========================================================================*/
PRIVATE void pick_proc()
{
/* Decide who to run now.  A new process is selected by setting 'next_ptr'.
 * When a billable process is selected, record it in 'bill_ptr', so that the
 * clock task can tell who to bill for system time.
 */
  register struct proc *rp;                     /* process to run */
  int q;                                        /* iterate over queues */

  /* Check each of the scheduling queues for ready processes. The number of
   * queues is defined in proc.h, and priorities are set in the image table.
   * The lowest queue contains IDLE, which is always ready.
   */
  for (q=0; q < NR_SCHED_QUEUES; q++) {
      if ( (rp = rdy_head[q]) != NIL_PROC) {
          next_ptr = rp;                        /* run process 'rp' next */
          if (priv(rp)->s_flags & BILLABLE)
              bill_ptr = rp;                    /* bill for system time */
          return;
      }
  }
}

/*===========================================================================*
 *                              lock_send                                    *
 *===========================================================================*/
PUBLIC int lock_send(dst, m_ptr)
int dst;                        /* to whom is message being sent? */
message *m_ptr;                 /* pointer to message buffer */
{
/* Safe gateway to mini_send() for tasks. */
  int result;
  lock(2, "send");
  result = mini_send(proc_ptr, dst, m_ptr, NON_BLOCKING);
  unlock(2);
  return(result);
}

/*===========================================================================*
 *                              lock_enqueue                                 *
 *===========================================================================*/
PUBLIC void lock_enqueue(rp)
struct proc *rp;                /* this process is now runnable */
{
/* Safe gateway to enqueue() for tasks. */
  lock(3, "enqueue");
  enqueue(rp);
  unlock(3);
}

/*===========================================================================*
 *                              lock_dequeue                                 *
 *===========================================================================*/
PUBLIC void lock_dequeue(rp)
struct proc *rp;                /* this process is no longer runnable */
{
/* Safe gateway to dequeue() for tasks. */
  lock(4, "dequeue");
  dequeue(rp);
  unlock(4);
}
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                      kernel/exception.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/* This file contains a simple exception handler.  Exceptions in user
 * processes are converted to signals. Exceptions in a kernel task cause
 * a panic.
 */

#include "kernel.h"
#include <signal.h>
#include "proc.h"

/*===========================================================================*
 *                              exception                                    *
 *===========================================================================*/
PUBLIC void exception(vec_nr)
unsigned vec_nr;
{
/* An exception or unexpected interrupt has occurred. */

  struct ex_s {
        char *msg;
        int signum;
        int minprocessor;
  };
  static struct ex_s ex_data[] = {
        { "Divide error", SIGFPE, 86 },
        { "Debug exception", SIGTRAP, 86 },
        { "Nonmaskable interrupt", SIGBUS, 86 },
        { "Breakpoint", SIGEMT, 86 },
        { "Overflow", SIGFPE, 86 },
        { "Bounds check", SIGFPE, 186 },
        { "Invalid opcode", SIGILL, 186 },
        { "Coprocessor not available", SIGFPE, 186 },
        { "Double fault", SIGBUS, 286 },
        { "Copressor segment overrun", SIGSEGV, 286 },
        { "Invalid TSS", SIGSEGV, 286 },
        { "Segment not present", SIGSEGV, 286 },
        { "Stack exception", SIGSEGV, 286 },    /* STACK_FAULT already used */
        { "General protection", SIGSEGV, 286 },
        { "Page fault", SIGSEGV, 386 },         /* not close */
        { NIL_PTR, SIGILL, 0 },                 /* probably software trap */
        { "Coprocessor error", SIGFPE, 386 },
  };
  register struct ex_s *ep;
  struct proc *saved_proc;

  /* Save proc_ptr, because it may be changed by debug statements. */
  saved_proc = proc_ptr;

  ep = &ex_data[vec_nr];

  if (vec_nr == 2) {            /* spurious NMI on some machines */
        kprintf("got spurious NMI\n");
        return;
  }

  /* If an exception occurs while running a process, the k_reenter variable
   * will be zero. Exceptions in interrupt handlers or system traps will make
   * k_reenter larger than zero.
   */
  if (k_reenter == 0 && ! iskernelp(saved_proc)) {
        cause_sig(proc_nr(saved_proc), ep->signum);
        return;
  }

  /* Exception in system code. This is not supposed to happen. */
  if (ep->msg == NIL_PTR || machine.processor < ep->minprocessor)
        kprintf("\nIntel-reserved exception %d\n", vec_nr);
  else
        kprintf("\n%s\n", ep->msg);
  kprintf("k_reenter = %d ", k_reenter);
  kprintf("process %d (%s), ", proc_nr(saved_proc), saved_proc->p_name);
  kprintf("pc = %u:0x%x", (unsigned) saved_proc->p_reg.cs,
  (unsigned) saved_proc->p_reg.pc);

  panic("exception in a kernel task", NO_NUM);
}
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                 kernel/i8259.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/* This file contains routines for initializing the 8259 interrupt controller:
 *      put_irq_handler: register an interrupt handler
 *      rm_irq_handler: deregister an interrupt handler
 *      intr_handle:    handle a hardware interrupt
 *      intr_init:      initialize the interrupt controller(s)
 */

#include "kernel.h"
#include "proc.h"
#include <minix/com.h>
```

```

#define ICW1_AT         0x11    /* edge triggered, cascade, need ICW4 */
#define ICW1_PC         0x13    /* edge triggered, no cascade, need ICW4 */
#define ICW1_PS         0x19    /* level triggered, cascade, need ICW4 */
#define ICW4_AT_SLAVE   0x01    /* not SFNM, not buffered, normal EOI, 8086 */
#define ICW4_AT_MASTER  0x05    /* not SFNM, not buffered, normal EOI, 8086 */
#define ICW4_PC_SLAVE   0x09    /* not SFNM, buffered, normal EOI, 8086 */
#define ICW4_PC_MASTER  0x0D    /* not SFNM, buffered, normal EOI, 8086 */

#define set_vec(nr, addr)       ((void)0)

/*===========================================================================*
 *                              intr_init                                    *
 *===========================================================================*/
PUBLIC void intr_init(mine)
int mine;
{
/* Initialize the 8259s, finishing with all interrupts disabled.  This is
 * only done in protected mode, in real mode we don't touch the 8259s, but
 * use the BIOS locations instead.  The flag "mine" is set if the 8259s are
 * to be programmed for MINIX, or to be reset to what the BIOS expects.
 */
  int i;

  intr_disable();

      /* The AT and newer PS/2 have two interrupt controllers, one master,
       * one slaved at IRQ 2.  (We don't have to deal with the PC that
       * has just one controller, because it must run in real mode.)
       */
      outb(INT_CTL, machine.ps_mca ? ICW1_PS : ICW1_AT);
      outb(INT_CTLMASK, mine ? IRQ0_VECTOR : BIOS_IRQ0_VEC);
                                                      /* ICW2 for master */
      outb(INT_CTLMASK, (1 << CASCADE_IRQ));          /* ICW3 tells slaves */
      outb(INT_CTLMASK, ICW4_AT_MASTER);
      outb(INT_CTLMASK, ~(1 << CASCADE_IRQ));         /* IRQ 0-7 mask */
      outb(INT2_CTL, machine.ps_mca ? ICW1_PS : ICW1_AT);
      outb(INT2_CTLMASK, mine ? IRQ8_VECTOR : BIOS_IRQ8_VEC);
                                                      /* ICW2 for slave */
      outb(INT2_CTLMASK, CASCADE_IRQ);        /* ICW3 is slave nr */
      outb(INT2_CTLMASK, ICW4_AT_SLAVE);
      outb(INT2_CTLMASK, ~0);                         /* IRQ 8-15 mask */

      /* Copy the BIOS vectors from the BIOS to the Minix location, so we
       * can still make BIOS calls without reprogramming the i8259s.
       */
      phys_copy(BIOS_VECTOR(0) * 4L, VECTOR(0) * 4L, 8 * 4L);
}

/*===========================================================================*
 *                              put_irq_handler                              *
 *===========================================================================*/
PUBLIC void put_irq_handler(hook, irq, handler)
irq_hook_t *hook;
int irq;
irq_handler_t handler;
{
/* Register an interrupt handler. */
  int id;
  irq_hook_t **line;

  if (irq < 0 || irq >= NR_IRQ_VECTORS)
      panic("invalid call to put_irq_handler", irq);

  line = &irq_handlers[irq];
  id = 1;
  while (*line != NULL) {
      if (hook == *line) return;        /* extra initialization */
      line = &(*line)->next;
      id <<= 1;
  }
  if (id == 0) panic("Too many handlers for irq", irq);

  hook->next = NULL;
  hook->handler = handler;
  hook->irq = irq;
  hook->id = id;
  *line = hook;

  irq_use |= 1 << irq;
}

/*===========================================================================*
 *                              rm_irq_handler                               *
 *===========================================================================*/
PUBLIC void rm_irq_handler(hook)
irq_hook_t *hook;
{
/* Unregister an interrupt handler. */
  int irq = hook->irq;
  int id = hook->id;
  irq_hook_t **line;

  if (irq < 0 || irq >= NR_IRQ_VECTORS)
      panic("invalid call to rm_irq_handler", irq);

  line = &irq_handlers[irq];
  while (*line != NULL) {
      if ((*line)->id == id) {
          (*line) = (*line)->next;
          if (! irq_handlers[irq]) irq_use &= ~(1 << irq);
          return;
      }
      line = &(*line)->next;
  }
  /* When the handler is not found, normally return here. */
}

/*===========================================================================*
 *                              intr_handle                                  *
 *===========================================================================*/
PUBLIC void intr_handle(hook)
irq_hook_t *hook;
{
/* Call the interrupt handlers for an interrupt with the given hook list.
 * The assembly part of the handler has already masked the IRQ, reenabled the
 * controller(s) and enabled interrupts.
 */

  /* Call list of handlers for an IRQ. */
  while (hook != NULL) {
      /* For each handler in the list, mark it active by setting its ID bit,
       * call the function, and unmark it if the function returns true.
       */
      irq_actids[hook->irq] |= hook->id;
      if ((*hook->handler)(hook)) irq_actids[hook->irq] &= ~hook->id;
      hook = hook->next;
  }

  /* The assembly code will now disable interrupts, unmask the IRQ if and only
   * if all active ID bits are cleared, and restart a process.
   */
}
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                kernel/protect.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/* This file contains code for initialization of protected mode, to initialize
 * code and data segment descriptors, and to initialize global descriptors
 * for local descriptors in the process table.
 */

#include "kernel.h"
#include "proc.h"
#include "protect.h"

#define INT_GATE_TYPE   (INT_286_GATE | DESC_386_BIT)
#define TSS_TYPE        (AVL_286_TSS  | DESC_386_BIT)

struct desctableptr_s {
  char limit[sizeof(u16_t)];
  char base[sizeof(u32_t)];             /* really u24_t + pad for 286 */
};

struct gatedesc_s {
  u16_t offset_low;
  u16_t selector;
  u8_t pad;                     /* |000|XXXXX| ig & trpg, |XXXXXXXX| task g */
  u8_t p_dpl_type;              /* |P|DL|0|TYPE| */
  u16_t offset_high;
};

struct tss_s {
  reg_t backlink;
  reg_t sp0;                    /* stack pointer to use during interrupt */
  reg_t ss0;                    /*   "   segment  "  "    "        "     */
  reg_t sp1;
  reg_t ss1;
  reg_t sp2;
  reg_t ss2;
  reg_t cr3;
  reg_t ip;
  reg_t flags;
  reg_t ax;
  reg_t cx;
  reg_t dx;
  reg_t bx;
  reg_t sp;
  reg_t bp;
  reg_t si;
  reg_t di;
  reg_t es;
  reg_t cs;
  reg_t ss;
  reg_t ds;
  reg_t fs;
  reg_t gs;
  reg_t ldt;
  u16_t trap;
  u16_t iobase;
/* u8_t iomap[0]; */
};

PUBLIC struct segdesc_s gdt[GDT_SIZE];          /* used in klib.s and mpx.s */
PRIVATE struct gatedesc_s idt[IDT_SIZE];        /* zero-init so none present */
PUBLIC struct tss_s tss;                        /* zero init */

FORWARD _PROTOTYPE( void int_gate, (unsigned vec_nr, vir_bytes offset,
                unsigned dpl_type) );
FORWARD _PROTOTYPE( void sdesc, (struct segdesc_s *segdp, phys_bytes base,
                vir_bytes size) );

/*===========================================================================*
 *                              prot_init                                    *
 *===========================================================================*/
PUBLIC void prot_init()
{
/* Set up tables for protected mode.
 * All GDT slots are allocated at compile time.
 */
  struct gate_table_s *gtp;
  struct desctableptr_s *dtp;
  unsigned ldt_index;
  register struct proc *rp;

  static struct gate_table_s {
        _PROTOTYPE( void (*gate), (void) );
        unsigned char vec_nr;
        unsigned char privilege;
  }
  gate_table[] = {
        { divide_error, DIVIDE_VECTOR, INTR_PRIVILEGE },
        { single_step_exception, DEBUG_VECTOR, INTR_PRIVILEGE },
        { nmi, NMI_VECTOR, INTR_PRIVILEGE },
        { breakpoint_exception, BREAKPOINT_VECTOR, USER_PRIVILEGE },
        { overflow, OVERFLOW_VECTOR, USER_PRIVILEGE },
        { bounds_check, BOUNDS_VECTOR, INTR_PRIVILEGE },
        { inval_opcode, INVAL_OP_VECTOR, INTR_PRIVILEGE },
        { copr_not_available, COPROC_NOT_VECTOR, INTR_PRIVILEGE },
        { double_fault, DOUBLE_FAULT_VECTOR, INTR_PRIVILEGE },
        { copr_seg_overrun, COPROC_SEG_VECTOR, INTR_PRIVILEGE },
        { inval_tss, INVAL_TSS_VECTOR, INTR_PRIVILEGE },
        { segment_not_present, SEG_NOT_VECTOR, INTR_PRIVILEGE },
        { stack_exception, STACK_FAULT_VECTOR, INTR_PRIVILEGE },
        { general_protection, PROTECTION_VECTOR, INTR_PRIVILEGE },
        { page_fault, PAGE_FAULT_VECTOR, INTR_PRIVILEGE },
        { copr_error, COPROC_ERR_VECTOR, INTR_PRIVILEGE },
        { hwint00, VECTOR( 0), INTR_PRIVILEGE },
        { hwint01, VECTOR( 1), INTR_PRIVILEGE },
        { hwint02, VECTOR( 2), INTR_PRIVILEGE },
        { hwint03, VECTOR( 3), INTR_PRIVILEGE },
        { hwint04, VECTOR( 4), INTR_PRIVILEGE },
        { hwint05, VECTOR( 5), INTR_PRIVILEGE },
        { hwint06, VECTOR( 6), INTR_PRIVILEGE },
        { hwint07, VECTOR( 7), INTR_PRIVILEGE },
        { hwint08, VECTOR( 8), INTR_PRIVILEGE },
        { hwint09, VECTOR( 9), INTR_PRIVILEGE },
        { hwint10, VECTOR(10), INTR_PRIVILEGE },
        { hwint11, VECTOR(11), INTR_PRIVILEGE },
        { hwint12, VECTOR(12), INTR_PRIVILEGE },
        { hwint13, VECTOR(13), INTR_PRIVILEGE },
        { hwint14, VECTOR(14), INTR_PRIVILEGE },
        { hwint15, VECTOR(15), INTR_PRIVILEGE },
        { s_call, SYS386_VECTOR, USER_PRIVILEGE },      /* 386 system call */
        { level0_call, LEVEL0_VECTOR, TASK_PRIVILEGE },
  };

  /* Build gdt and idt pointers in GDT where the BIOS expects them. */
  dtp= (struct desctableptr_s *) &gdt[GDT_INDEX];
  * (u16_t *) dtp->limit = (sizeof gdt) - 1;
  * (u32_t *) dtp->base = vir2phys(gdt);

  dtp= (struct desctableptr_s *) &gdt[IDT_INDEX];
  * (u16_t *) dtp->limit = (sizeof idt) - 1;
  * (u32_t *) dtp->base = vir2phys(idt);

  /* Build segment descriptors for tasks and interrupt handlers. */
  init_codeseg(&gdt[CS_INDEX],
         kinfo.code_base, kinfo.code_size, INTR_PRIVILEGE);
  init_dataseg(&gdt[DS_INDEX],
         kinfo.data_base, kinfo.data_size, INTR_PRIVILEGE);
  init_dataseg(&gdt[ES_INDEX], 0L, 0, TASK_PRIVILEGE);

  /* Build scratch descriptors for functions in klib88. */
  init_dataseg(&gdt[DS_286_INDEX], 0L, 0, TASK_PRIVILEGE);
  init_dataseg(&gdt[ES_286_INDEX], 0L, 0, TASK_PRIVILEGE);

  /* Build local descriptors in GDT for LDT's in process table.
   * The LDT's are allocated at compile time in the process table, and
   * initialized whenever a process' map is initialized or changed.
   */
  for (rp = BEG_PROC_ADDR, ldt_index = FIRST_LDT_INDEX;
       rp < END_PROC_ADDR; ++rp, ldt_index++) {
        init_dataseg(&gdt[ldt_index], vir2phys(rp->p_ldt),
                                     sizeof(rp->p_ldt), INTR_PRIVILEGE);
        gdt[ldt_index].access = PRESENT | LDT;
        rp->p_ldt_sel = ldt_index * DESC_SIZE;
  }

  /* Build main TSS.
   * This is used only to record the stack pointer to be used after an
   * interrupt.
   * The pointer is set up so that an interrupt automatically saves the
   * current process's registers ip:cs:f:sp:ss in the correct slots in the
   * process table.
   */
  tss.ss0 = DS_SELECTOR;
  init_dataseg(&gdt[TSS_INDEX], vir2phys(&tss), sizeof(tss), INTR_PRIVILEGE);
  gdt[TSS_INDEX].access = PRESENT | (INTR_PRIVILEGE << DPL_SHIFT) | TSS_TYPE;

  /* Build descriptors for interrupt gates in IDT. */
  for (gtp = &gate_table[0];
       gtp < &gate_table[sizeof gate_table / sizeof gate_table[0]]; ++gtp) {
        int_gate(gtp->vec_nr, (vir_bytes) gtp->gate,
                 PRESENT | INT_GATE_TYPE | (gtp->privilege << DPL_SHIFT));
  }

  /* Complete building of main TSS. */
  tss.iobase = sizeof tss;      /* empty i/o permissions map */
}

/*===========================================================================*
 *                              init_codeseg                                 *
 *===========================================================================*/
PUBLIC void init_codeseg(segdp, base, size, privilege)
register struct segdesc_s *segdp;
phys_bytes base;
vir_bytes size;
int privilege;
{
/* Build descriptor for a code segment. */
  sdesc(segdp, base, size);
  segdp->access = (privilege << DPL_SHIFT)
                | (PRESENT | SEGMENT | EXECUTABLE | READABLE);
                /* CONFORMING = 0, ACCESSED = 0 */
}

/*===========================================================================*
 *                              init_dataseg                                 *
 *===========================================================================*/
PUBLIC void init_dataseg(segdp, base, size, privilege)
register struct segdesc_s *segdp;
phys_bytes base;
vir_bytes size;
int privilege;
{
/* Build descriptor for a data segment. */
  sdesc(segdp, base, size);
  segdp->access = (privilege << DPL_SHIFT) | (PRESENT | SEGMENT | WRITEABLE);
                /* EXECUTABLE = 0, EXPAND_DOWN = 0, ACCESSED = 0 */
}

/*===========================================================================*
 *                              sdesc                                        *
 *===========================================================================*/
PRIVATE void sdesc(segdp, base, size)
register struct segdesc_s *segdp;
phys_bytes base;
vir_bytes size;
{
/* Fill in the size fields (base, limit and granularity) of a descriptor. */
  segdp->base_low = base;
  segdp->base_middle = base >> BASE_MIDDLE_SHIFT;
  segdp->base_high = base >> BASE_HIGH_SHIFT;

  --size;                       /* convert to a limit, 0 size means 4G */
  if (size > BYTE_GRAN_MAX) {
        segdp->limit_low = size >> PAGE_GRAN_SHIFT;
        segdp->granularity = GRANULAR | (size >>
                                    (PAGE_GRAN_SHIFT + GRANULARITY_SHIFT));
  } else {
        segdp->limit_low = size;
        segdp->granularity = size >> GRANULARITY_SHIFT;
  }
  segdp->granularity |= DEFAULT;        /* means BIG for data seg */
}

/*===========================================================================*
 *                              seg2phys                                     *
 *===========================================================================*/
PUBLIC phys_bytes seg2phys(seg)
U16_t seg;
{
/* Return the base address of a segment, with seg being either a 8086 segment
 * register, or a 286/386 segment selector.
 */
  phys_bytes base;
  struct segdesc_s *segdp;

  if (! machine.protected) {
        base = hclick_to_physb(seg);
  } else {
        segdp = &gdt[seg >> 3];
        base =    ((u32_t) segdp->base_low << 0)
                | ((u32_t) segdp->base_middle << 16)
                | ((u32_t) segdp->base_high << 24);
  }
  return base;
}

/*===========================================================================*
 *                              phys2seg                                     *
 *===========================================================================*/
PUBLIC void phys2seg(seg, off, phys)
u16_t *seg;
vir_bytes *off;
phys_bytes phys;
{
/* Return a segment selector and offset that can be used to reach a physical
 * address, for use by a driver doing memory I/O in the A0000 - DFFFF range.
 */
  *seg = FLAT_DS_SELECTOR;
  *off = phys;
}

/*===========================================================================*
 *                              int_gate                                     *
 *===========================================================================*/
PRIVATE void int_gate(vec_nr, offset, dpl_type)
unsigned vec_nr;
vir_bytes offset;
unsigned dpl_type;
{
/* Build descriptor for an interrupt gate. */
  register struct gatedesc_s *idp;

  idp = &idt[vec_nr];
  idp->offset_low = offset;
  idp->selector = CS_SELECTOR;
  idp->p_dpl_type = dpl_type;
  idp->offset_high = offset >> OFFSET_HIGH_SHIFT;
}

/*===========================================================================*
 *                              enable_iop                                   *
 *===========================================================================*/
PUBLIC void enable_iop(pp)
struct proc *pp;
{
/* Allow a user process to use I/O instructions.  Change the I/O Permission
 * Level bits in the psw. These specify least-privileged Current Permission
 * Level allowed to execute I/O instructions. Users and servers have CPL 3.
 * You can't have less privilege than that. Kernel has CPL 0, tasks CPL 1.
 */
  pp->p_reg.psw |= 0x3000;
}

/*===========================================================================*
 *                              alloc_segments                               *
 *===========================================================================*/
PUBLIC void alloc_segments(rp)
register struct proc *rp;
{
/* This is called at system initialization from main() and by do_newmap().
 * The code has a separate function because of all hardware-dependencies.
 * Note that IDLE is part of the kernel and gets TASK_PRIVILEGE here.
 */
  phys_bytes code_bytes;
  phys_bytes data_bytes;
  int privilege;

  if (machine.protected) {
      data_bytes = (phys_bytes) (rp->p_memmap[S].mem_vir +
          rp->p_memmap[S].mem_len) << CLICK_SHIFT;
      if (rp->p_memmap[T].mem_len == 0)
          code_bytes = data_bytes;      /* common I&D, poor protect */
      else
          code_bytes = (phys_bytes) rp->p_memmap[T].mem_len << CLICK_SHIFT;
      privilege = (iskernelp(rp)) ? TASK_PRIVILEGE : USER_PRIVILEGE;
      init_codeseg(&rp->p_ldt[CS_LDT_INDEX],
          (phys_bytes) rp->p_memmap[T].mem_phys << CLICK_SHIFT,
          code_bytes, privilege);
      init_dataseg(&rp->p_ldt[DS_LDT_INDEX],
          (phys_bytes) rp->p_memmap[D].mem_phys << CLICK_SHIFT,
          data_bytes, privilege);
      rp->p_reg.cs = (CS_LDT_INDEX * DESC_SIZE) | TI | privilege;
      rp->p_reg.gs =
      rp->p_reg.fs =
      rp->p_reg.ss =
      rp->p_reg.es =
      rp->p_reg.ds = (DS_LDT_INDEX*DESC_SIZE) | TI | privilege;
  } else {
      rp->p_reg.cs = click_to_hclick(rp->p_memmap[T].mem_phys);
      rp->p_reg.ss =
      rp->p_reg.es =
      rp->p_reg.ds = click_to_hclick(rp->p_memmap[D].mem_phys);
  }
```

```
}
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                      kernel/klib.s
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
#
! Chooses between the 8086 and 386 versions of the low level kernel code.

#include <minix/config.h>
#if _WORD_SIZE == 2
#include "klib88.s"
#else
#include "klib386.s"
#endif
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                      kernel/klib386.s
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
#
! sections

.sect .text; .sect .rom; .sect .data; .sect .bss

#include <minix/config.h>
#include <minix/const.h>
#include "const.h"
#include "sconst.h"
#include "protect.h"

! This file contains a number of assembly code utility routines needed by the
! kernel.  They are:

.define _monitor        ! exit Minix and return to the monitor
.define _int86          ! let the monitor make an 8086 interrupt call
.define _cp_mess        ! copies messages from source to destination
.define _exit           ! dummy for library routines
.define __exit          ! dummy for library routines
.define ___exit         ! dummy for library routines
.define ___main         ! dummy for GCC
.define _phys_insw      ! transfer data from (disk controller) port to memory
.define _phys_insb      ! likewise byte by byte
.define _phys_outsw     ! transfer data from memory to (disk controller) port
.define _phys_outsb     ! likewise byte by byte
.define _enable_irq     ! enable an irq at the 8259 controller
.define _disable_irq    ! disable an irq
.define _phys_copy      ! copy data from anywhere to anywhere in memory
.define _phys_memset    ! write pattern anywhere in memory
.define _mem_rdw        ! copy one word from [segment:offset]
.define _reset          ! reset the system
.define _idle_task      ! task executed when there is no work
.define _level0         ! call a function at level 0
.define _read_tsc       ! read the cycle counter (Pentium and up)
.define _read_cpu_flags ! read the cpu flags
```

```

! The routines only guarantee to preserve the registers the C compiler
! expects to be preserved (ebx, esi, edi, ebp, esp, segment registers, and
! direction bit in the flags).

.sect .text
!*===========================================================================*
!*                              monitor                                      *
!*===========================================================================*
! PUBLIC void monitor();
! Return to the monitor.

_monitor:
        mov     esp, (_mon_sp)          ! restore monitor stack pointer
    o16 mov     dx, SS_SELECTOR         ! monitor data segment
        mov     ds, dx
        mov     es, dx
        mov     fs, dx
        mov     gs, dx
        mov     ss, dx
        pop     edi
        pop     esi
        pop     ebp
    o16 retf                            ! return to the monitor


!*===========================================================================*
!*                              int86                                        *
!*===========================================================================*
! PUBLIC void int86();
_int86:
        cmpb    (_mon_return), 0        ! is the monitor there?
        jnz     0f
        movb    ah, 0x01                ! an int 13 error seems appropriate
        movb    (_reg86+ 0), ah         ! reg86.w.f = 1 (set carry flag)
        movb    (_reg86+13), ah         ! reg86.b.ah = 0x01 = "invalid command"
        ret
0:      push    ebp                     ! save C registers
        push    esi
        push    edi
        push    ebx
        pushf                           ! save flags
        cli                             ! no interruptions

        inb     INT2_CTLMASK
        movb    ah, al
        inb     INT_CTLMASK
        push    eax                     ! save interrupt masks
        mov     eax, (_irq_use)         ! map of in-use IRQ's
        and     eax, ~[1<<CLOCK_IRQ]    ! keep the clock ticking
        outb    INT_CTLMASK             ! enable all unused IRQ's and vv.
        movb    al, ah
        outb    INT2_CTLMASK

        mov     eax, SS_SELECTOR        ! monitor data segment
        mov     ss, ax
        xchg    esp, (_mon_sp)          ! switch stacks
        push    (_reg86+36)             ! parameters used in INT call
        push    (_reg86+32)
        push    (_reg86+28)
        push    (_reg86+24)
        push    (_reg86+20)
        push    (_reg86+16)
        push    (_reg86+12)
        push    (_reg86+ 8)
        push    (_reg86+ 4)
        push    (_reg86+ 0)
        mov     ds, ax                  ! remaining data selectors
        mov     es, ax
        mov     fs, ax
        mov     gs, ax
        push    cs
        push    return                  ! kernel return address and selector
    o16 jmpf    20+2*4+10*4+2*4(esp)    ! make the call
return:
        pop     (_reg86+ 0)
        pop     (_reg86+ 4)
        pop     (_reg86+ 8)
        pop     (_reg86+12)
        pop     (_reg86+16)
        pop     (_reg86+20)
        pop     (_reg86+24)
        pop     (_reg86+28)
        pop     (_reg86+32)
        pop     (_reg86+36)
        lgdt    (_gdt+GDT_SELECTOR)     ! reload global descriptor table
        jmpf    CS_SELECTOR:csinit      ! restore everything
csinit: mov     eax, DS_SELECTOR
        mov     ds, ax
        mov     es, ax
        mov     fs, ax
        mov     gs, ax
        mov     ss, ax
        xchg    esp, (_mon_sp)          ! unswitch stacks
        lidt    (_gdt+IDT_SELECTOR)     ! reload interrupt descriptor table
        andb    (_gdt+TSS_SELECTOR+DESC_ACCESS), ~0x02  ! clear TSS busy bit
        mov     eax, TSS_SELECTOR
        ltr     ax                      ! set TSS register

        pop     eax
        outb    INT_CTLMASK             ! restore interrupt masks
        movb    al, ah
        outb    INT2_CTLMASK

        add     (_lost_ticks), ecx      ! record lost clock ticks

        popf                            ! restore flags
        pop     ebx                     ! restore C registers
        pop     edi
        pop     esi
        pop     ebp
        ret


!*===========================================================================*
!*                              cp_mess                                      *
!*===========================================================================*
! PUBLIC void cp_mess(int src, phys_clicks src_clicks, vir_bytes src_offset,
!                     phys_clicks dst_clicks, vir_bytes dst_offset);
! This routine makes a fast copy of a message from anywhere in the address
! space to anywhere else.  It also copies the source address provided as a
! parameter to the call into the first word of the destination message.
!
! Note that the message size, "Msize" is in DWORDS (not bytes) and must be set
! correctly.  Changing the definition of message in the type file and not
! changing it here will lead to total disaster.

CM_ARGS =       4 + 4 + 4 + 4 + 4       ! 4 + 4 + 4 + 4 + 4
!               es  ds edi esi eip      proc scl sof dcl dof

        .align  16
_cp_mess:
        cld
        push    esi
        push    edi
        push    ds
        push    es

        mov     eax, FLAT_DS_SELECTOR
        mov     ds, ax
        mov     es, ax

        mov     esi, CM_ARGS+4(esp)             ! src clicks
        shl     esi, CLICK_SHIFT
        add     esi, CM_ARGS+4+4(esp)           ! src offset
        mov     edi, CM_ARGS+4+4+4(esp)         ! dst clicks
        shl     edi, CLICK_SHIFT
        add     edi, CM_ARGS+4+4+4+4(esp)       ! dst offset

        mov     eax, CM_ARGS(esp)       ! process number of sender
        stos                            ! copy number of sender to dest message
        add     esi, 4                  ! do not copy first word
        mov     ecx, Msize - 1          ! remember, first word does not count
        rep
        movs                            ! copy the message

        pop     es
        pop     ds
        pop     edi
        pop     esi
        ret                             ! that is all folks!


!*===========================================================================*
!*                              exit                                         *
!*===========================================================================*
! PUBLIC void exit();
! Some library routines use exit, so provide a dummy version.
! Actual calls to exit cannot occur in the kernel.
! GNU CC likes to call ___main from main() for nonobvious reasons.

_exit:
__exit:
___exit:
        sti
        jmp     ___exit

___main:
        ret

```

```

!*===========================================================================*
!*                              phys_insw                                    *
!*===========================================================================*
! PUBLIC void phys_insw(Port_t port, phys_bytes buf, size_t count);
! Input an array from an I/O port.  Absolute address version of insw().

_phys_insw:
        push    ebp
        mov     ebp, esp
        cld
        push    edi
        push    es
        mov     ecx, FLAT_DS_SELECTOR
        mov     es, cx
        mov     edx, 8(ebp)             ! port to read from
        mov     edi, 12(ebp)            ! destination addr
        mov     ecx, 16(ebp)            ! byte count
        shr     ecx, 1                  ! word count
rep o16 ins                             ! input many words
        pop     es
        pop     edi
        pop     ebp
        ret


!*===========================================================================*
!*                              phys_insb                                    *
!*===========================================================================*
! PUBLIC void phys_insb(Port_t port, phys_bytes buf, size_t count);
! Input an array from an I/O port.  Absolute address version of insb().

_phys_insb:
        push    ebp
        mov     ebp, esp
        cld
        push    edi
        push    es
        mov     ecx, FLAT_DS_SELECTOR
        mov     es, cx
        mov     edx, 8(ebp)             ! port to read from
        mov     edi, 12(ebp)            ! destination addr
        mov     ecx, 16(ebp)            ! byte count
!       shr     ecx, 1                  ! word count
   rep  insb                            ! input many bytes
        pop     es
        pop     edi
        pop     ebp
        ret


!*===========================================================================*
!*                              phys_outsw                                   *
!*===========================================================================*
! PUBLIC void phys_outsw(Port_t port, phys_bytes buf, size_t count);
! Output an array to an I/O port.  Absolute address version of outsw().

        .align  16
_phys_outsw:
        push    ebp
        mov     ebp, esp
        cld
        push    esi
        push    ds
        mov     ecx, FLAT_DS_SELECTOR
        mov     ds, cx
        mov     edx, 8(ebp)             ! port to write to
        mov     esi, 12(ebp)            ! source addr
        mov     ecx, 16(ebp)            ! byte count
        shr     ecx, 1                  ! word count
rep o16 outs                            ! output many words
        pop     ds
        pop     esi
        pop     ebp
        ret


!*===========================================================================*
!*                              phys_outsb                                   *
!*===========================================================================*
! PUBLIC void phys_outsb(Port_t port, phys_bytes buf, size_t count);
! Output an array to an I/O port.  Absolute address version of outsb().

        .align  16
_phys_outsb:
        push    ebp
        mov     ebp, esp
        cld
        push    esi
        push    ds
        mov     ecx, FLAT_DS_SELECTOR
        mov     ds, cx
        mov     edx, 8(ebp)             ! port to write to
        mov     esi, 12(ebp)            ! source addr
        mov     ecx, 16(ebp)            ! byte count
   rep  outsb                           ! output many bytes
        pop     ds
        pop     esi
        pop     ebp
        ret


!*==========================================================================*
!*                              enable_irq                                  *
!*==========================================================================*/
! PUBLIC void enable_irq(irq_hook_t *hook)
! Enable an interrupt request line by clearing an 8259 bit.
! Equivalent C code for hook->irq < 8:
!   if ((irq_actids[hook->irq] &= ~hook->id) == 0)
!       outb(INT_CTLMASK, inb(INT_CTLMASK) & ~(1 << irq));

        .align  16
_enable_irq:
        push    ebp
        mov     ebp, esp
        pushf
        cli
        mov     eax, 8(ebp)             ! hook
        mov     ecx, 8(eax)             ! irq
        mov     eax, 12(eax)            ! id bit
        not     eax
        and     _irq_actids(ecx*4), eax ! clear this id bit
        jnz     en_done                 ! still masked by other handlers?
        movb    ah, ~1
        rolb    ah, cl                  ! ah = ~(1 << (irq % 8))
        mov     edx, INT_CTLMASK        ! enable irq < 8 at the master 8259
        cmpb    cl, 8
        jb      0f
        mov     edx, INT2_CTLMASK       ! enable irq >= 8 at the slave 8259
0:      inb     dx
        andb    al, ah
        outb    dx                      ! clear bit at the 8259
en_done:popf
        leave
        ret


!*===========================================================================*
!*                              disable_irq                                  *
!*===========================================================================*/
! PUBLIC int disable_irq(irq_hook_t *hook)
! Disable an interrupt request line by setting an 8259 bit.
! Equivalent C code for irq < 8:
!   irq_actids[hook->irq] |= hook->id;
!   outb(INT_CTLMASK, inb(INT_CTLMASK) | (1 << irq));
! Returns true iff the interrupt was not already disabled.

        .align  16
_disable_irq:
        push    ebp
        mov     ebp, esp
        pushf
        cli
        mov     eax, 8(ebp)             ! hook
        mov     ecx, 8(eax)             ! irq
        mov     eax, 12(eax)            ! id bit
        or      _irq_actids(ecx*4), eax ! set this id bit
        movb    ah, 1
        rolb    ah, cl                  ! ah = (1 << (irq % 8))
        mov     edx, INT_CTLMASK        ! disable irq < 8 at the master 8259
        cmpb    cl, 8
        jb      0f
        mov     edx, INT2_CTLMASK       ! disable irq >= 8 at the slave 8259
0:      inb     dx
        testb   al, ah
        jnz     dis_already             ! already disabled?
        orb     al, ah
        outb    dx                      ! set bit at the 8259
        mov     eax, 1                  ! disabled by this function
        popf
        leave
        ret
dis_already:
        xor     eax, eax                ! already disabled
        popf
        leave
        ret
```

```
!*===========================================================================*
!*                              phys_copy                                    *
!*===========================================================================*
! PUBLIC void phys_copy(phys_bytes source, phys_bytes destination,
!                       phys_bytes bytecount);
! Copy a block of physical memory.

PC_ARGS =       4 + 4 + 4 + 4   ! 4 + 4 + 4
!               es edi esi eip   src dst len

        .align  16
_phys_copy:
        cld
        push    esi
        push    edi
        push    es

        mov     eax, FLAT_DS_SELECTOR
        mov     es, ax

        mov     esi, PC_ARGS(esp)
        mov     edi, PC_ARGS+4(esp)
        mov     eax, PC_ARGS+4+4(esp)

        cmp     eax, 10                 ! avoid align overhead for small counts
        jb      pc_small
        mov     ecx, esi                ! align source, hope target is too
        neg     ecx
        and     ecx, 3                  ! count for alignment
        sub     eax, ecx
        rep
   eseg movsb
        mov     ecx, eax
        shr     ecx, 2                  ! count of dwords
        rep
   eseg movs
        and     eax, 3
pc_small:
        xchg    ecx, eax                ! remainder
        rep
   eseg movsb

        pop     es
        pop     edi
        pop     esi
        ret

!*===========================================================================*
!*                              phys_memset                                  *
!*===========================================================================*
! PUBLIC void phys_memset(phys_bytes source, unsigned long pattern,
!       phys_bytes bytecount);
! Fill a block of physical memory with pattern.

        .align  16
_phys_memset:
        push    ebp
        mov     ebp, esp
        push    esi
        push    ebx
        push    ds
        mov     esi, 8(ebp)
        mov     eax, 16(ebp)
        mov     ebx, FLAT_DS_SELECTOR
        mov     ds, bx
        mov     ebx, 12(ebp)
        shr     eax, 2
fill_start:
        mov     (esi), ebx
        add     esi, 4
        dec     eax
        jnz     fill_start
        ! Any remaining bytes?
        mov     eax, 16(ebp)
        and     eax, 3
remain_fill:
        cmp     eax, 0
        jz      fill_done
        movb    bl, 12(ebp)
        movb    (esi), bl
        add     esi, 1
        inc     ebp
        dec     eax
        jmp     remain_fill
fill_done:
        pop     ds
        pop     ebx
        pop     esi
        pop     ebp
        ret

!*===========================================================================*
!*                              mem_rdw                                      *
!*===========================================================================*
! PUBLIC u16_t mem_rdw(U16_t segment, u16_t *offset);
! Load and return word at far pointer segment:offset.

        .align  16
_mem_rdw:
        mov     cx, ds
        mov     ds, 4(esp)              ! segment
        mov     eax, 4+4(esp)           ! offset
        movzx   eax, (eax)              ! word to return
        mov     ds, cx
        ret


!*===========================================================================*
!*                              reset                                        *
!*===========================================================================*
! PUBLIC void reset();
! Reset the system by loading IDT with offset 0 and interrupting.

_reset:
        lidt    (idt_zero)
        int     3               ! anything goes, the 386 will not like it
.sect .data
idt_zero:       .data4  0, 0
.sect .text

```

```

!*===========================================================================*
!*                              idle_task                                    *
!*===========================================================================*
_idle_task:
! This task is called when the system has nothing else to do.  The HLT
! instruction puts the processor in a state where it draws minimum power.
        push    halt
        call    _level0         ! level0(halt)
        pop     eax
        jmp     _idle_task
halt:
        sti
        hlt
        cli
        ret

!*===========================================================================*
!*                              level0                                       *
!*===========================================================================*
! PUBLIC void level0(void (*func)(void))
! Call a function at permission level 0.  This allows kernel tasks to do
! things that are only possible at the most privileged CPU level.
!
_level0:
        mov     eax, 4(esp)
        mov     (_level0_func), eax
        int     LEVEL0_VECTOR
        ret


!*===========================================================================*
!*                              read_tsc                                     *
!*===========================================================================*
! PUBLIC void read_tsc(unsigned long *high, unsigned long *low);
! Read the cycle counter of the CPU. Pentium and up.
.align 16
_read_tsc:
.data1 0x0f             ! this is the RDTSC instruction
.data1 0x31             ! it places the TSC in EDX:EAX
        push ebp
        mov ebp, 8(esp)
        mov (ebp), edx
        mov ebp, 12(esp)
        mov (ebp), eax
        pop ebp
        ret

!*===========================================================================*
!*                              read_flags                                           *
!*===========================================================================*
! PUBLIC unsigned long read_cpu_flags(void);
! Read CPU status flags from C.
.align 16
_read_cpu_flags:
        pushf
        mov eax, (esp)
        popf
        ret

```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                kernel/utility.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/* This file contains a collection of miscellaneous procedures:
 *   panic:         abort MINIX due to a fatal error
 *   kprintf:       diagnostic output for the kernel
 *
 * Changes:
 *   Dec 10, 2004   kernel printing to circular buffer  (Jorrit N. Herder)
 *
 * This file contains the routines that take care of kernel messages, i.e.,
 * diagnostic output within the kernel. Kernel messages are not directly
 * displayed on the console, because this must be done by the output driver.
 * Instead, the kernel accumulates characters in a buffer and notifies the
 * output driver when a new message is ready.
 */

#include <minix/com.h>
#include "kernel.h"
#include <stdarg.h>
#include <unistd.h>
#include <stddef.h>
#include <stdlib.h>
#include <signal.h>
#include "proc.h"

#define END_OF_KMESS    -1
FORWARD _PROTOTYPE(void kputc, (int c));

/*===========================================================================*
 *                              panic                                        *
 *===========================================================================*/
PUBLIC void panic(mess,nr)
_CONST char *mess;
int nr;
{
/* The system has run aground of a fatal kernel error. Terminate execution. */
  static int panicking = 0;
  if (panicking ++) return;             /* prevent recursive panics */

  if (mess != NULL) {
        kprintf("\nKernel panic: %s", mess);
        if (nr != NO_NUM) kprintf(" %d", nr);
        kprintf("\n",NO_NUM);
  }

  /* Abort MINIX. */
  prepare_shutdown(RBT_PANIC);
}

/*===========================================================================*
 *                              kprintf                                      *
 *===========================================================================*/
PUBLIC void kprintf(const char *fmt, ...)       /* format to be printed */
{
  int c;                                        /* next character in fmt */
  int d;
  unsigned long u;                              /* hold number argument */
  int base;                                 /* base of number arg */
  int negative = 0;                         /* print minus sign */
  static char x2c[] = "0123456789ABCDEF";   /* nr conversion table */
  char ascii[8 * sizeof(long) / 3 + 2];     /* string for ascii number */
  char *s = NULL;                           /* string to be printed */
  va_list argp;                             /* optional arguments */

  va_start(argp, fmt);                      /* init variable arguments */

  while((c=*fmt++) != 0) {

      if (c == '%') {                       /* expect format '%key' */
          switch(c = *fmt++) {              /* determine what to do */

          /* Known keys are %d, %u, %x, %s, and %%. This is easily extended
           * with number types like %b and %o by providing a different base.
           * Number type keys don't set a string to 's', but use the general
           * conversion after the switch statement.
           */
          case 'd':                         /* output decimal */
              d = va_arg(argp, signed int);
              if (d < 0) { negative = 1; u = -d; }  else { u = d; }
              base = 10;
              break;
          case 'u':                         /* output unsigned long */
              u = va_arg(argp, unsigned long);
              base = 10;
              break;
          case 'x':                         /* output hexadecimal */
              u = va_arg(argp, unsigned long);
              base = 0x10;
              break;
          case 's':                         /* output string */
              s = va_arg(argp, char *);
              if (s == NULL) s = "(null)";
              break;
          case '%':                         /* output percent */
              s = "%";
              break;

          /* Unrecognized key. */
          default:                          /* echo back %key */
              s = "%?";
              s[1] = c;                     /* set unknown key */
          }

          /* Assume a number if no string is set. Convert to ascii. */
          if (s == NULL) {
              s = ascii + sizeof(ascii)-1;
              *s = 0;
              do {  *--s = x2c[(u % base)]; }   /* work backwards */
              while ((u /= base) > 0);
          }

          /* This is where the actual output for format "%key" is done. */
          if (negative) kputc('-');         /* print sign if negative */
          while(*s != 0) { kputc(*s++); }   /* print string/ number */
          s = NULL;                         /* reset for next round */
      }
      else {
          kputc(c);                           /* print and continue */
      }
  }
  kputc(END_OF_KMESS);                        /* terminate output */
  va_end(argp);                               /* end variable arguments */
}

/*===========================================================================*
 *                              kputc                                        *
 *===========================================================================*/
PRIVATE void kputc(c)
int c;                                /* character to append */
{
/* Accumulate a single character for a kernel message. Send a notification
 * to the output driver if an END_OF_KMESS is encountered.
 */
  if (c != END_OF_KMESS) {
      kmess.km_buf[kmess.km_next] = c;  /* put normal char in buffer */
      if (kmess.km_size < KMESS_BUF_SIZE)
          kmess.km_size += 1;
      kmess.km_next = (kmess.km_next + 1) % KMESS_BUF_SIZE;
  } else {
      send_sig(OUTPUT_PROC_NR, SIGKMESS);
  }
}
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                kernel/system.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
  /* Function prototypes for the system library.
   * The implementation is contained in src/kernel/system/.
   *
   * The system library allows access to system services by doing a kernel call.
   * Kernel calls are transformed into request messages to the SYS task that is
   * responsible for handling the call. By convention, sys_call() is transformed
   * into a message with type SYS_CALL that is handled in a function do_call().
   */

  #ifndef SYSTEM_H
  #define SYSTEM_H

  /* Common includes for the system library. */
  #include "kernel.h"
  #include "proto.h"
  #include "proc.h"

  /* Default handler for unused kernel calls. */
  _PROTOTYPE( int do_unused, (message *m_ptr) );
  _PROTOTYPE( int do_exec, (message *m_ptr) );
  _PROTOTYPE( int do_fork, (message *m_ptr) );
  _PROTOTYPE( int do_newmap, (message *m_ptr) );
  _PROTOTYPE( int do_exit, (message *m_ptr) );
  _PROTOTYPE( int do_trace, (message *m_ptr) );
  _PROTOTYPE( int do_nice, (message *m_ptr) );
```

```
_PROTOTYPE( int do_copy, (message *m_ptr) );
#define do_vircopy      do_copy
#define do_physcopy     do_copy
_PROTOTYPE( int do_vcopy, (message *m_ptr) );
#define do_virvcopy     do_vcopy
#define do_physvcopy    do_vcopy
_PROTOTYPE( int do_umap, (message *m_ptr) );
_PROTOTYPE( int do_memset, (message *m_ptr) );
_PROTOTYPE( int do_abort, (message *m_ptr) );
_PROTOTYPE( int do_getinfo, (message *m_ptr) );
_PROTOTYPE( int do_privctl, (message *m_ptr) );
_PROTOTYPE( int do_segctl, (message *m_ptr) );
_PROTOTYPE( int do_irqctl, (message *m_ptr) );
_PROTOTYPE( int do_devio, (message *m_ptr) );
_PROTOTYPE( int do_vdevio, (message *m_ptr) );
_PROTOTYPE( int do_int86, (message *m_ptr) );
_PROTOTYPE( int do_sdevio, (message *m_ptr) );
_PROTOTYPE( int do_kill, (message *m_ptr) );
_PROTOTYPE( int do_getksig, (message *m_ptr) );
_PROTOTYPE( int do_endksig, (message *m_ptr) );
_PROTOTYPE( int do_sigsend, (message *m_ptr) );
_PROTOTYPE( int do_sigreturn, (message *m_ptr) );
_PROTOTYPE( int do_times, (message *m_ptr) );
_PROTOTYPE( int do_setalarm, (message *m_ptr) );

#endif  /* SYSTEM_H */



```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                   kernel/system.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/* This task provides an interface between the kernel and user-space system
 * processes. System services can be accessed by doing a kernel call. Kernel
 * calls are  transformed into request messages, which are handled by this
 * task. By convention, a sys_call() is transformed in a SYS_CALL request
 * message that is handled in a function named do_call().
 *
 * A private call vector is used to map all kernel calls to the functions that
 * handle them. The actual handler functions are contained in separate files
 * to keep this file clean. The call vector is used in the system task's main
 * loop to handle all incoming requests.
 *
 * In addition to the main sys_task() entry point, which starts the main loop,
 * there are several other minor entry points:
 *   get_priv:          assign privilege structure to user or system process
 *   send_sig:          send a signal directly to a system process
 *   cause_sig:         take action to cause a signal to occur via PM
 *   umap_local:        map virtual address in LOCAL_SEG to physical
 *   umap_remote:       map virtual address in REMOTE_SEG to physical
 *   umap_bios:         map virtual address in BIOS_SEG to physical
 *   virtual_copy:      copy bytes from one virtual address to another
 *   get_randomness:    accumulate randomness in a buffer
 *
 * Changes:
 *   Aug 04, 2005   check if kernel call is allowed  (Jorrit N. Herder)
 *   Jul 20, 2005   send signal to services with message  (Jorrit N. Herder)
 *   Jan 15, 2005   new, generalized virtual copy function  (Jorrit N. Herder)
 *   Oct 10, 2004   dispatch system calls from call vector  (Jorrit N. Herder)
 *   Sep 30, 2004   source code documentation updated  (Jorrit N. Herder)
 */

#include "kernel.h"
#include "system.h"
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>
#include <sys/sigcontext.h>
#include <ibm/memory.h>
#include "protect.h"

/* Declaration of the call vector that defines the mapping of kernel calls
 * to handler functions. The vector is initialized in sys_init() with map(),
 * which makes sure the kernel call numbers are ok. No space is allocated,
 * because the dummy is declared extern. If an illegal call is given, the
 * array size will be negative and this won't compile.
 */
PUBLIC int (*call_vec[NR_SYS_CALLS])(message *m_ptr);

#define map(call_nr, handler) \
    {extern int dummy[NR_SYS_CALLS>(unsigned)(call_nr-KERNEL_CALL) ? 1:-1];} \
    call_vec[(call_nr-KERNEL_CALL)] = (handler)

FORWARD _PROTOTYPE( void initialize, (void));

/*===========================================================================*
 *                              sys_task                                     *
 *===========================================================================*/
PUBLIC void sys_task()
{
/* Main entry point of sys_task.  Get the message and dispatch on type. */
  static message m;
  register int result;
  register struct proc *caller_ptr;
  unsigned int call_nr;
  int s;

  /* Initialize the system task. */
  initialize();

  while (TRUE) {
      /* Get work. Block and wait until a request message arrives. */
      receive(ANY, &m);
      call_nr = (unsigned) m.m_type - KERNEL_CALL;
      caller_ptr = proc_addr(m.m_source);

      /* See if the caller made a valid request and try to handle it. */
      if (! (priv(caller_ptr)->s_call_mask & (1<<call_nr))) {
          kprintf("SYSTEM: request %d from %d denied.\n", call_nr,m.m_source);
          result = ECALLDENIED;                 /* illegal message type */
      } else if (call_nr >= NR_SYS_CALLS) {             /* check call number */
          kprintf("SYSTEM: illegal request %d from %d.\n", call_nr,m.m_source);
          result = EBADREQUEST;                 /* illegal message type */
      }
      else {
          result = (*call_vec[call_nr])(&m);    /* handle the kernel call */
      }

      /* Send a reply, unless inhibited by a handler function. Use the kernel
       * function lock_send() to prevent a system call trap. The destination
       * is known to be blocked waiting for a message.
       */
      if (result != EDONTREPLY) {
          m.m_type = result;                   /* report status of call */
          if (OK != (s=lock_send(m.m_source, &m))) {
              kprintf("SYSTEM, reply to %d failed: %d\n", m.m_source, s);
          }
      }
  }
}

/*===========================================================================*
 *                              initialize                                   *
 *===========================================================================*/
PRIVATE void initialize(void)
{
  register struct priv *sp;
  int i;

  /* Initialize IRQ handler hooks. Mark all hooks available. */
  for (i=0; i<NR_IRQ_HOOKS; i++) {
      irq_hooks[i].proc_nr = NONE;
  }

  /* Initialize all alarm timers for all processes. */
  for (sp=BEG_PRIV_ADDR; sp < END_PRIV_ADDR; sp++) {
    tmr_inittimer(&(sp->s_alarm_timer));
  }

  /* Initialize the call vector to a safe default handler. Some kernel calls
   * may be disabled or nonexistant. Then explicitly map known calls to their
   * handler functions. This is done with a macro that gives a compile error
   * if an illegal call number is used. The ordering is not important here.
   */
  for (i=0; i<NR_SYS_CALLS; i++) {
      call_vec[i] = do_unused;
  }

  /* Process management. */
  map(SYS_FORK, do_fork);              /* a process forked a new process */
  map(SYS_EXEC, do_exec);              /* update process after execute */
  map(SYS_EXIT, do_exit);              /* clean up after process exit */
  map(SYS_NICE, do_nice);              /* set scheduling priority */
  map(SYS_PRIVCTL, do_privctl);        /* system privileges control */
  map(SYS_TRACE, do_trace);            /* request a trace operation */

  /* Signal handling. */
  map(SYS_KILL, do_kill);              /* cause a process to be signaled */
  map(SYS_GETKSIG, do_getksig);        /* PM checks for pending signals */
  map(SYS_ENDKSIG, do_endksig);        /* PM finished processing signal */
  map(SYS_SIGSEND, do_sigsend);        /* start POSIX-style signal */
  map(SYS_SIGRETURN, do_sigreturn);    /* return from POSIX-style signal */

  /* Device I/O. */
  map(SYS_IRQCTL, do_irqctl);          /* interrupt control operations */
  map(SYS_DEVIO, do_devio);            /* inb, inw, inl, outb, outw, outl */
  map(SYS_SDEVIO, do_sdevio);          /* phys_insb, _insw, _outsb, _outsw */
  map(SYS_VDEVIO, do_vdevio);           /* vector with devio requests */
  map(SYS_INT86, do_int86);             /* real-mode BIOS calls */

  /* Memory management. */
  map(SYS_NEWMAP, do_newmap);           /* set up a process memory map */
  map(SYS_SEGCTL, do_segctl);           /* add segment and get selector */
  map(SYS_MEMSET, do_memset);           /* write char to memory area */

  /* Copying. */
  map(SYS_UMAP, do_umap);               /* map virtual to physical address */
  map(SYS_VIRCOPY, do_vircopy);         /* use pure virtual addressing */
  map(SYS_PHYSCOPY, do_physcopy);       /* use physical addressing */
  map(SYS_VIRVCOPY, do_virvcopy);       /* vector with copy requests */
  map(SYS_PHYSVCOPY, do_physvcopy);     /* vector with copy requests */

  /* Clock functionality. */
  map(SYS_TIMES, do_times);             /* get uptime and process times */
  map(SYS_SETALARM, do_setalarm);       /* schedule a synchronous alarm */

  /* System control. */
  map(SYS_ABORT, do_abort);             /* abort MINIX */
  map(SYS_GETINFO, do_getinfo);         /* request system information */
}

/*===========================================================================*
 *                              get_priv                                     *
 *===========================================================================*/
PUBLIC int get_priv(rc, proc_type)
register struct proc *rc;               /* new (child) process pointer */
int proc_type;                          /* system or user process flag */
{
/* Get a privilege structure. All user processes share the same privilege
 * structure. System processes get their own privilege structure.
 */
  register struct priv *sp;                     /* privilege structure */

  if (proc_type == SYS_PROC) {                  /* find a new slot */
      for (sp = BEG_PRIV_ADDR; sp < END_PRIV_ADDR; ++sp)
          if (sp->s_proc_nr == NONE && sp->s_id != USER_PRIV_ID) break;
      if (sp->s_proc_nr != NONE) return(ENOSPC);
      rc->p_priv = sp;                          /* assign new slot */
      rc->p_priv->s_proc_nr = proc_nr(rc);      /* set association */
      rc->p_priv->s_flags = SYS_PROC;           /* mark as privileged */
  } else {
      rc->p_priv = &priv[USER_PRIV_ID];         /* use shared slot */
      rc->p_priv->s_proc_nr = INIT_PROC_NR;     /* set association */
      rc->p_priv->s_flags = 0;                  /* no initial flags */
  }
  return(OK);
}

/*===========================================================================*
 *                              get_randomness                               *
 *===========================================================================*/
PUBLIC void get_randomness(source)
int source;
{
/* On machines with the RDTSC (cycle counter read instruction - pentium
 * and up), use that for high-resolution raw entropy gathering. Otherwise,
 * use the realtime clock (tick resolution).
 *
 * Unfortunately this test is run-time - we don't want to bother with
 * compiling different kernels for different machines.
 *
 * On machines without RDTSC, we use read_clock().
 */
  int r_next;
  unsigned long tsc_high, tsc_low;

  source %= RANDOM_SOURCES;
  r_next= krandom.bin[source].r_next;
  if (machine.processor > 486) {
      read_tsc(&tsc_high, &tsc_low);
      krandom.bin[source].r_buf[r_next] = tsc_low;
  } else {
      krandom.bin[source].r_buf[r_next] = read_clock();
  }
  if (krandom.bin[source].r_size < RANDOM_ELEMENTS) {
        krandom.bin[source].r_size ++;
  }
  krandom.bin[source].r_next = (r_next + 1 ) % RANDOM_ELEMENTS;
}

/*===========================================================================*
 *                              send_sig                                     *
 *===========================================================================*/
PUBLIC void send_sig(proc_nr, sig_nr)
int proc_nr;                    /* system process to be signalled */
int sig_nr;                     /* signal to be sent, 1 to _NSIG */
{
/* Notify a system process about a signal. This is straightforward. Simply
 * set the signal that is to be delivered in the pending signals map and
 * send a notification with source SYSTEM.
 */
  register struct proc *rp;

  rp = proc_addr(proc_nr);
  sigaddset(&priv(rp)->s_sig_pending, sig_nr);
  lock_notify(SYSTEM, proc_nr);
}

/*===========================================================================*
 *                              cause_sig                                    *
 *===========================================================================*/
PUBLIC void cause_sig(proc_nr, sig_nr)
int proc_nr;                    /* process to be signalled */
int sig_nr;                     /* signal to be sent, 1 to _NSIG */
{
/* A system process wants to send a signal to a process.  Examples are:
 *  - HARDWARE wanting to cause a SIGSEGV after a CPU exception
 *  - TTY wanting to cause SIGINT upon getting a DEL
 *  - FS wanting to cause SIGPIPE for a broken pipe
 * Signals are handled by sending a message to PM.  This function handles the
 * signals and makes sure the PM gets them by sending a notification. The
 * process being signaled is blocked while PM has not finished all signals
 * for it.
 * Race conditions between calls to this function and the system calls that
 * process pending kernel signals cannot exist. Signal related functions are
 * only called when a user process causes a CPU exception and from the kernel
 * process level, which runs to completion.
 */
  register struct proc *rp;

  /* Check if the signal is already pending. Process it otherwise. */
  rp = proc_addr(proc_nr);
  if (! sigismember(&rp->p_pending, sig_nr)) {
      sigaddset(&rp->p_pending, sig_nr);
      if (! (rp->p_rts_flags & SIGNALED)) {             /* other pending */
          if (rp->p_rts_flags == 0) lock_dequeue(rp);   /* make not ready */
          rp->p_rts_flags |= SIGNALED | SIG_PENDING;    /* update flags */
          send_sig(PM_PROC_NR, SIGKSIG);
      }
  }
}

/*===========================================================================*
 *                              umap_local                                   *
 *===========================================================================*/
PUBLIC phys_bytes umap_local(rp, seg, vir_addr, bytes)
register struct proc *rp;       /* pointer to proc table entry for process */
int seg;                        /* T, D, or S segment */
vir_bytes vir_addr;             /* virtual address in bytes within the seg */
vir_bytes bytes;                /* # of bytes to be copied */
{
/* Calculate the physical memory address for a given virtual address. */
  vir_clicks vc;                /* the virtual address in clicks */
  phys_bytes pa;                /* intermediate variables as phys_bytes */
  phys_bytes seg_base;

  /* If 'seg' is D it could really be S and vice versa.  T really means T.
   * If the virtual address falls in the gap,  it causes a problem. On the
   * 8088 it is probably a legal stack reference, since "stackfaults" are
   * not detected by the hardware.  On 8088s, the gap is called S and
   * accepted, but on other machines it is called D and rejected.
   * The Atari ST behaves like the 8088 in this respect.
   */

  if (bytes <= 0) return( (phys_bytes) 0);
  if (vir_addr + bytes <= vir_addr) return 0;   /* overflow */
  vc = (vir_addr + bytes - 1) >> CLICK_SHIFT;   /* last click of data */

  if (seg != T)
        seg = (vc < rp->p_memmap[D].mem_vir + rp->p_memmap[D].mem_len ? D : S);

  if ((vir_addr>>CLICK_SHIFT) >= rp->p_memmap[seg].mem_vir +
        rp->p_memmap[seg].mem_len) return( (phys_bytes) 0 );

  if (vc >= rp->p_memmap[seg].mem_vir +
        rp->p_memmap[seg].mem_len) return( (phys_bytes) 0 );

  seg_base = (phys_bytes) rp->p_memmap[seg].mem_phys;
  seg_base = seg_base << CLICK_SHIFT;   /* segment origin in bytes */
  pa = (phys_bytes) vir_addr;
  pa -= rp->p_memmap[seg].mem_vir << CLICK_SHIFT;
  return(seg_base + pa);
}
```

```
/*===========================================================================*
 *                              umap_remote                                  *
 *===========================================================================*/
PUBLIC phys_bytes umap_remote(rp, seg, vir_addr, bytes)
register struct proc *rp;       /* pointer to proc table entry for process */
int seg;                        /* index of remote segment */
vir_bytes vir_addr;             /* virtual address in bytes within the seg */
vir_bytes bytes;                /* # of bytes to be copied */
{
/* Calculate the physical memory address for a given virtual address. */
  struct far_mem *fm;

  if (bytes <= 0) return( (phys_bytes) 0);
  if (seg < 0 || seg >= NR_REMOTE_SEGS) return( (phys_bytes) 0);

  fm = &rp->p_priv->s_farmem[seg];
  if (! fm->in_use) return( (phys_bytes) 0);
  if (vir_addr + bytes > fm->mem_len) return( (phys_bytes) 0);

  return(fm->mem_phys + (phys_bytes) vir_addr);
}

/*===========================================================================*
 *                              umap_bios                                    *
 *===========================================================================*/
PUBLIC phys_bytes umap_bios(rp, vir_addr, bytes)
register struct proc *rp;       /* pointer to proc table entry for process */
vir_bytes vir_addr;             /* virtual address in BIOS segment */
vir_bytes bytes;                /* # of bytes to be copied */
{
/* Calculate the physical memory address at the BIOS. Note: currently, BIOS
 * address zero (the first BIOS interrupt vector) is not considered as an
 * error here, but since the physical address will be zero as well, the
 * calling function will think an error occurred. This is not a problem,
 * since no one uses the first BIOS interrupt vector.
 */

  /* Check all acceptable ranges. */
  if (vir_addr >= BIOS_MEM_BEGIN && vir_addr + bytes <= BIOS_MEM_END)
        return (phys_bytes) vir_addr;
  else if (vir_addr >= BASE_MEM_TOP && vir_addr + bytes <= UPPER_MEM_END)
        return (phys_bytes) vir_addr;
  kprintf("Warning, error in umap_bios, virtual address 0x%x\n", vir_addr);
  return 0;
}

/*===========================================================================*
 *                              virtual_copy                                 *
 *===========================================================================*/
PUBLIC int virtual_copy(src_addr, dst_addr, bytes)
struct vir_addr *src_addr;      /* source virtual address */
struct vir_addr *dst_addr;      /* destination virtual address */
vir_bytes bytes;                /* # of bytes to copy  */
{
/* Copy bytes from virtual address src_addr to virtual address dst_addr.
 * Virtual addresses can be in ABS, LOCAL_SEG, REMOTE_SEG, or BIOS_SEG.
 */
  struct vir_addr *vir_addr[2]; /* virtual source and destination address */
  phys_bytes phys_addr[2];      /* absolute source and destination */
  int seg_index;
  int i;

  /* Check copy count. */
  if (bytes <= 0) return(EDOM);

  /* Do some more checks and map virtual addresses to physical addresses. */
  vir_addr[_SRC_] = src_addr;
  vir_addr[_DST_] = dst_addr;
  for (i=_SRC_; i<=_DST_; i++) {

      /* Get physical address. */
      switch((vir_addr[i]->segment & SEGMENT_TYPE)) {
      case LOCAL_SEG:
          seg_index = vir_addr[i]->segment & SEGMENT_INDEX;
          phys_addr[i] = umap_local( proc_addr(vir_addr[i]->proc_nr),
              seg_index, vir_addr[i]->offset, bytes );
          break;
      case REMOTE_SEG:
          seg_index = vir_addr[i]->segment & SEGMENT_INDEX;
          phys_addr[i] = umap_remote( proc_addr(vir_addr[i]->proc_nr),
              seg_index, vir_addr[i]->offset, bytes );
          break;
      case BIOS_SEG:
          phys_addr[i] = umap_bios( proc_addr(vir_addr[i]->proc_nr),
              vir_addr[i]->offset, bytes );
          break;
      case PHYS_SEG:
          phys_addr[i] = vir_addr[i]->offset;
          break;
      default:
          return(EINVAL);
      }

      /* Check if mapping succeeded. */
      if (phys_addr[i] <= 0 && vir_addr[i]->segment != PHYS_SEG)
          return(EFAULT);
  }

  /* Now copy bytes between physical addresseses. */
  phys_copy(phys_addr[_SRC_], phys_addr[_DST_], (phys_bytes) bytes);
  return(OK);
}
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                            kernel/system/do_setalarm.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/* The kernel call implemented in this file:
 *   m_type:    SYS_SETALARM
 *
 * The parameters for this kernel call are:
 *    m2_l1:    ALRM_EXP_TIME           (alarm's expiration time)
 *    m2_i2:    ALRM_ABS_TIME           (expiration time is absolute?)
 *    m2_l1:    ALRM_TIME_LEFT          (return seconds left of previous)
 */

#include "../system.h"
```

```

#if USE_SETALARM

FORWARD _PROTOTYPE( void cause_alarm, (timer_t *tp) );

/*===========================================================================*
 *                              do_setalarm                                  *
 *===========================================================================*/
PUBLIC int do_setalarm(m_ptr)
message *m_ptr;                 /* pointer to request message */
{
/* A process requests a synchronous alarm, or wants to cancel its alarm. */
  register struct proc *rp;     /* pointer to requesting process */
  int proc_nr;                  /* which process wants the alarm */
  long exp_time;                /* expiration time for this alarm */
  int use_abs_time;             /* use absolute or relative time */
  timer_t *tp;                  /* the process' timer structure */
  clock_t uptime;               /* placeholder for current uptime */

  /* Extract shared parameters from the request message. */
  exp_time = m_ptr->ALRM_EXP_TIME;      /* alarm's expiration time */
  use_abs_time = m_ptr->ALRM_ABS_TIME;  /* flag for absolute time */
  proc_nr = m_ptr->m_source;            /* process to interrupt later */
  rp = proc_addr(proc_nr);
  if (! (priv(rp)->s_flags & SYS_PROC)) return(EPERM);

  /* Get the timer structure and set the parameters for this alarm. */
  tp = &(priv(rp)->s_alarm_timer);
  tmr_arg(tp)->ta_int = proc_nr;
  tp->tmr_func = cause_alarm;

  /* Return the ticks left on the previous alarm. */
  uptime = get_uptime();
  if ((tp->tmr_exp_time != TMR_NEVER) && (uptime < tp->tmr_exp_time) ) {
      m_ptr->ALRM_TIME_LEFT = (tp->tmr_exp_time - uptime);
  } else {
      m_ptr->ALRM_TIME_LEFT = 0;
  }

  /* Finally, (re)set the timer depending on the expiration time. */
  if (exp_time == 0) {
      reset_timer(tp);
  } else {
      tp->tmr_exp_time = (use_abs_time) ? exp_time : exp_time + get_uptime();
      set_timer(tp, tp->tmr_exp_time, tp->tmr_func);
  }
  return(OK);
}

/*===========================================================================*
 *                              cause_alarm                                  *
 *===========================================================================*/
PRIVATE void cause_alarm(tp)
timer_t *tp;
{
/* Routine called if a timer goes off and the process requested a synchronous
 * alarm. The process number is stored in timer argument 'ta_int'. Notify that
 * process with a notification message from CLOCK.
 */
  int proc_nr = tmr_arg(tp)->ta_int;            /* get process number */
  lock_notify(CLOCK, proc_nr);                  /* notify process */
}

#endif /* USE_SETALARM */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                      kernel/system/do_exec.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/* The kernel call implemented in this file:
 *   m_type:    SYS_EXEC
 *
 * The parameters for this kernel call are:
 *    m1_i1:    PR_PROC_NR              (process that did exec call)
 *    m1_p1:    PR_STACK_PTR            (new stack pointer)
 *    m1_p2:    PR_NAME_PTR             (pointer to program name)
 *    m1_p3:    PR_IP_PTR               (new instruction pointer)
 */
#include "../system.h"
#include <string.h>
#include <signal.h>

#if USE_EXEC

/*===========================================================================*
 *                              do_exec                                      *
 *===========================================================================*/
PUBLIC int do_exec(m_ptr)
register message *m_ptr;        /* pointer to request message */
{
/* Handle sys_exec().  A process has done a successful EXEC. Patch it up. */
  register struct proc *rp;
  reg_t sp;                     /* new sp */
  phys_bytes phys_name;
  char *np;

  rp = proc_addr(m_ptr->PR_PROC_NR);
  sp = (reg_t) m_ptr->PR_STACK_PTR;
  rp->p_reg.sp = sp;            /* set the stack pointer */
  phys_memset(vir2phys(&rp->p_ldt[EXTRA_LDT_INDEX]), 0,
        (LDT_SIZE - EXTRA_LDT_INDEX) * sizeof(rp->p_ldt[0]));
  rp->p_reg.pc = (reg_t) m_ptr->PR_IP_PTR;      /* set pc */
  rp->p_rts_flags &= ~RECEIVING;        /* PM does not reply to EXEC call */
  if (rp->p_rts_flags == 0) lock_enqueue(rp);

  /* Save command name for debugging, ps(1) output, etc. */
  phys_name = numap_local(m_ptr->m_source, (vir_bytes) m_ptr->PR_NAME_PTR,
                                        (vir_bytes) P_NAME_LEN - 1);
  if (phys_name != 0) {
        phys_copy(phys_name, vir2phys(rp->p_name), (phys_bytes) P_NAME_LEN - 1);
        for (np = rp->p_name; (*np & BYTE) >= ' '; np++) {}
        *np = 0;                                        /* mark end */
  } else {
        strncpy(rp->p_name, "<unset>", P_NAME_LEN);
  }
  return(OK);
}
#endif /* USE_EXEC */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                kernel/clock.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

/* This file contains the clock task, which handles time related functions.
 * Important events that are handled by the CLOCK include setting and
 * monitoring alarm timers and deciding when to (re)schedule processes.
 * The CLOCK offers a direct interface to kernel processes. System services
 * can access its services through system calls, such as sys_setalarm(). The
 * CLOCK task thus is hidden from the outside world.
 *
 * Changes:
 *   Oct 08, 2005   reordering and comment editing (A. S. Woodhull)
 *   Mar 18, 2004   clock interface moved to SYSTEM task (Jorrit N. Herder)
 *   Sep 30, 2004   source code documentation updated  (Jorrit N. Herder)
 *   Sep 24, 2004   redesigned alarm timers  (Jorrit N. Herder)
 *
 * The function do_clocktick() is triggered by the clock's interrupt
 * handler when a watchdog timer has expired or a process must be scheduled.
 *
 * In addition to the main clock_task() entry point, which starts the main
 * loop, there are several other minor entry points:
 *   clock_stop:        called just before MINIX shutdown
 *   get_uptime:        get realtime since boot in clock ticks
 *   set_timer:         set a watchdog timer (+)
 *   reset_timer:       reset a watchdog timer (+)
 *   read_clock:        read the counter of channel 0 of the 8253A timer
 *
 * (+) The CLOCK task keeps tracks of watchdog timers for the entire kernel.
 * The watchdog functions of expired timers are executed in do_clocktick().
 * It is crucial that watchdog functions not block, or the CLOCK task may
 * be blocked. Do not send() a message when the receiver is not expecting it.
 * Instead, notify(), which always returns, should be used.
 */

#include "kernel.h"
#include "proc.h"
#include <signal.h>
#include <minix/com.h>

/* Function prototype for PRIVATE functions. */
FORWARD _PROTOTYPE( void init_clock, (void) );
FORWARD _PROTOTYPE( int clock_handler, (irq_hook_t *hook) );
FORWARD _PROTOTYPE( int do_clocktick, (message *m_ptr) );

/* Clock parameters. */
#define COUNTER_FREQ (2*TIMER_FREQ) /* counter frequency using square wave */
#define LATCH_COUNT     0x00    /* cc00xxxx, c = channel, x = any */
#define SQUARE_WAVE     0x36    /* ccaammmb, a = access, m = mode, b = BCD */
                                /*   11x11, 11 = LSB then MSB, x11 = sq wave */
#define TIMER_COUNT ((unsigned) (TIMER_FREQ/HZ)) /* initial value for counter*/
#define TIMER_FREQ  1193182L    /* clock frequency for timer in PC and AT */

#define CLOCK_ACK_BIT   0x80    /* PS/2 clock interrupt acknowledge bit */

/* The CLOCK's timers queue. The functions in <timers.h> operate on this.
 * Each system process possesses a single synchronous alarm timer. If other
 * kernel parts want to use additional timers, they must declare their own
 * persistent (static) timer structure, which can be passed to the clock
 * via (re)set_timer().
 * When a timer expires its watchdog function is run by the CLOCK task.
 */
PRIVATE timer_t *clock_timers;          /* queue of CLOCK timers */
PRIVATE clock_t next_timeout;           /* realtime that next timer expires */

/* The time is incremented by the interrupt handler on each clock tick. */
PRIVATE clock_t realtime;               /* real time clock */
PRIVATE irq_hook_t clock_hook;          /* interrupt handler hook */

/*===========================================================================*
 *                              clock_task                                   *
 *===========================================================================*/
PUBLIC void clock_task()
{
/* Main program of clock task. If the call is not HARD_INT it is an error.
 */
  message m;                    /* message buffer for both input and output */
  int result;                   /* result returned by the handler */

  init_clock();                 /* initialize clock task */

  /* Main loop of the clock task.  Get work, process it. Never reply. */
  while (TRUE) {

      /* Go get a message. */
      receive(ANY, &m);

      /* Handle the request. Only clock ticks are expected. */
      switch (m.m_type) {
      case HARD_INT:
          result = do_clocktick(&m);    /* handle clock tick */
          break;
      default:                          /* illegal request type */
          kprintf("CLOCK: illegal request %d from %d.\n", m.m_type,m.m_source);
      }
  }
}

/*===========================================================================*
 *                              do_clocktick                                 *
 *===========================================================================*/
PRIVATE int do_clocktick(m_ptr)
message *m_ptr;                         /* pointer to request message */
{
/* Despite its name, this routine is not called on every clock tick. It
 * is called on those clock ticks when a lot of work needs to be done.
 */

  /* A process used up a full quantum. The interrupt handler stored this
   * process in 'prev_ptr'.  First make sure that the process is not on the
   * scheduling queues.  Then announce the process ready again. Since it has
   * no more time left, it gets a new quantum and is inserted at the right
   * place in the queues.  As a side-effect a new process will be scheduled.
   */
  if (prev_ptr->p_ticks_left <= 0 && priv(prev_ptr)->s_flags & PREEMPTIBLE) {
      lock_dequeue(prev_ptr);           /* take it off the queues */
      lock_enqueue(prev_ptr);           /* and reinsert it again */
  }

  /* Check if a clock timer expired and run its watchdog function. */
  if (next_timeout <= realtime) {
        tmrs_exptimers(&clock_timers, realtime, NULL);
        next_timeout = clock_timers == NULL ?
                TMR_NEVER : clock_timers->tmr_exp_time;
  }

  /* Inhibit sending a reply. */
  return(EDONTREPLY);
}

/*===========================================================================*
 *                              init_clock                                   *
 *===========================================================================*/
PRIVATE void init_clock()
{
  /* Initialize the CLOCK's interrupt hook. */
  clock_hook.proc_nr = CLOCK;

  /* Initialize channel 0 of the 8253A timer to, e.g., 60 Hz. */
  outb(TIMER_MODE, SQUARE_WAVE);        /* set timer to run continuously */
  outb(TIMER0, TIMER_COUNT);            /* load timer low byte */
  outb(TIMER0, TIMER_COUNT >> 8);       /* load timer high byte */
  put_irq_handler(&clock_hook, CLOCK_IRQ, clock_handler);/* register handler */
  enable_irq(&clock_hook);              /* ready for clock interrupts */
}

/*===========================================================================*
 *                              clock_stop                                   *
 *===========================================================================*/
PUBLIC void clock_stop()
{
/* Reset the clock to the BIOS rate. (For rebooting) */
  outb(TIMER_MODE, 0x36);
  outb(TIMER0, 0);
  outb(TIMER0, 0);
}

/*===========================================================================*
 *                              clock_handler                                *
 *===========================================================================*/
PRIVATE int clock_handler(hook)
irq_hook_t *hook;
{
/* This executes on each clock tick (i.e., every time the timer chip generates
 * an interrupt). It does a little bit of work so the clock task does not have
 * to be called on every tick.  The clock task is called when:
 *
 *      (1) the scheduling quantum of the running process has expired, or
 *      (2) a timer has expired and the watchdog function should be run.
 *
 * Many global global and static variables are accessed here.  The safety of
 * this must be justified. All scheduling and message passing code acquires a
 * lock by temporarily disabling interrupts, so no conflicts with calls from
 * the task level can occur. Furthermore, interrupts are not reentrant, the
 * interrupt handler cannot be bothered by other interrupts.
 *
 * Variables that are updated in the clock's interrupt handler:
 *      lost_ticks:
 *              Clock ticks counted outside the clock task. This for example
 *              is used when the boot monitor processes a real mode interrupt.
 *      realtime:
 *              The current uptime is incremented with all outstanding ticks.
 *      proc_ptr, bill_ptr:
 *              These are used for accounting.  It does not matter if proc.c
 *              is changing them, provided they are always valid pointers,
 *              since at worst the previous process would be billed.
 */
  register unsigned ticks;

  /* Acknowledge the PS/2 clock interrupt. */
  if (machine.ps_mca) outb(PORT_B, inb(PORT_B) | CLOCK_ACK_BIT);

  /* Get number of ticks and update realtime. */
  ticks = lost_ticks + 1;
  lost_ticks = 0;
  realtime += ticks;

  /* Update user and system accounting times. Charge the current process for
   * user time. If the current process is not billable, that is, if a non-user
   * process is running, charge the billable process for system time as well.
   * Thus the unbillable process' user time is the billable user's system time.
   */
  proc_ptr->p_user_time += ticks;
  if (priv(proc_ptr)->s_flags & PREEMPTIBLE) {
      proc_ptr->p_ticks_left -= ticks;
  }
  if (! (priv(proc_ptr)->s_flags & BILLABLE)) {
      bill_ptr->p_sys_time += ticks;
      bill_ptr->p_ticks_left -= ticks;
  }

  /* Check if do_clocktick() must be called. Done for alarms and scheduling.
   * Some processes, such as the kernel tasks, cannot be preempted.
   */
  if ((next_timeout <= realtime) || (proc_ptr->p_ticks_left <= 0)) {
      prev_ptr = proc_ptr;                      /* store running process */
      lock_notify(HARDWARE, CLOCK);             /* send notification */
  }
  return(1);                                    /* reenable interrupts */
}

/*===========================================================================*
 *                              get_uptime                                   *
 *===========================================================================*/
PUBLIC clock_t get_uptime()
{
/* Get and return the current clock uptime in ticks. */
  return(realtime);
}

/*===========================================================================*
 *                              set_timer                                    *
 *===========================================================================*/
PUBLIC void set_timer(tp, exp_time, watchdog)
struct timer *tp;               /* pointer to timer structure */
clock_t exp_time;               /* expiration realtime */
tmr_func_t watchdog;            /* watchdog to be called */
{
/* Insert the new timer in the active timers list. Always update the
```

```
 * next timeout time by setting it to the front of the active list.
 */
  tmrs_settimer(&clock_timers, tp, exp_time, watchdog, NULL);
  next_timeout = clock_timers->tmr_exp_time;
}

/*===========================================================================*
 *                              reset_timer                                  *
 *===========================================================================*/
PUBLIC void reset_timer(tp)
struct timer *tp;               /* pointer to timer structure */
{
/* The timer pointed to by 'tp' is no longer needed. Remove it from both the
 * active and expired lists. Always update the next timeout time by setting
 * it to the front of the active list.
 */
  tmrs_clrtimer(&clock_timers, tp, NULL);
  next_timeout = (clock_timers == NULL) ?
        TMR_NEVER : clock_timers->tmr_exp_time;
}

/*===========================================================================*
 *                              read_clock                                   *
 *===========================================================================*/
PUBLIC unsigned long read_clock()
{
/* Read the counter of channel 0 of the 8253A timer.  This counter counts
 * down at a rate of TIMER_FREQ and restarts at TIMER_COUNT-1 when it
 * reaches zero. A hardware interrupt (clock tick) occurs when the counter
 * gets to zero and restarts its cycle.
 */
  unsigned count;

  outb(TIMER_MODE, LATCH_COUNT);
  count = inb(TIMER0);
  count |= (inb(TIMER0) << 8);

  return count;
}
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                      drivers/drivers.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/* This is the master header for all device drivers. It includes some other
 * files and defines the principal constants.
 */
#define _POSIX_SOURCE      1    /* tell headers to include POSIX stuff */
#define _MINIX             1    /* tell headers to include MINIX stuff */
#define _SYSTEM            1    /* get negative error number in <errno.h> */

/* The following are so basic, all the *.c files get them automatically. */
#include <minix/config.h>       /* MUST be first */
#include <ansi.h>               /* MUST be second */
#include <minix/type.h>
#include <minix/com.h>
#include <minix/dmap.h>
#include <minix/callnr.h>
#include <sys/types.h>
```

```
#include <minix/const.h>
#include <minix/devio.h>
#include <minix/syslib.h>
#include <minix/sysutil.h>
#include <minix/bitmap.h>

#include <ibm/interrupt.h>      /* IRQ vectors and miscellaneous ports */
#include <ibm/bios.h>           /* BIOS index numbers */
#include <ibm/ports.h>          /* Well-known ports */

#include <string.h>
#include <signal.h>
#include <stdlib.h>
#include <limits.h>
#include <stddef.h>
#include <errno.h>
#include <unistd.h>

```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                      drivers/libdriver/driver.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/* Types and constants shared between the generic and device dependent
 * device driver code.
 */

#define _POSIX_SOURCE      1    /* tell headers to include POSIX stuff */
#define _MINIX             1    /* tell headers to include MINIX stuff */
#define _SYSTEM            1    /* get negative error number in <errno.h> */

/* The following are so basic, all the *.c files get them automatically. */
#include <minix/config.h>       /* MUST be first */
#include <ansi.h>               /* MUST be second */
#include <minix/type.h>
#include <minix/ipc.h>
#include <minix/com.h>
#include <minix/callnr.h>
#include <sys/types.h>
#include <minix/const.h>
#include <minix/syslib.h>
#include <minix/sysutil.h>

#include <string.h>
#include <limits.h>
#include <stddef.h>
#include <errno.h>

#include <minix/partition.h>
#include <minix/u64.h>

/* Info about and entry points into the device dependent code. */
struct driver {
  _PROTOTYPE( char *(*dr_name), (void) );
  _PROTOTYPE( int (*dr_open), (struct driver *dp, message *m_ptr) );
  _PROTOTYPE( int (*dr_close), (struct driver *dp, message *m_ptr) );
  _PROTOTYPE( int (*dr_ioctl), (struct driver *dp, message *m_ptr) );
  _PROTOTYPE( struct device *(*dr_prepare), (int device) );
  _PROTOTYPE( int (*dr_transfer), (int proc_nr, int opcode, off_t position,
                                        iovec_t *iov, unsigned nr_req) );
  _PROTOTYPE( void (*dr_cleanup), (void) );
  _PROTOTYPE( void (*dr_geometry), (struct partition *entry) );
  _PROTOTYPE( void (*dr_signal), (struct driver *dp, message *m_ptr) );
  _PROTOTYPE( void (*dr_alarm), (struct driver *dp, message *m_ptr) );
  _PROTOTYPE( int (*dr_cancel), (struct driver *dp, message *m_ptr) );
  _PROTOTYPE( int (*dr_select), (struct driver *dp, message *m_ptr) );
  _PROTOTYPE( int (*dr_other), (struct driver *dp, message *m_ptr) );
  _PROTOTYPE( int (*dr_hw_int), (struct driver *dp, message *m_ptr) );
};

#if (CHIP == INTEL)

/* Number of bytes you can DMA before hitting a 64K boundary: */
#define dma_bytes_left(phys)    \
   ((unsigned) (sizeof(int) == 2 ? 0 : 0x10000) - (unsigned) ((phys) & 0xFFFF))

#endif /* CHIP == INTEL */

/* Base and size of a partition in bytes. */
struct device {
  u64_t dv_base;
  u64_t dv_size;
};

#define NIL_DEV         ((struct device *) 0)

/* Functions defined by driver.c: */
_PROTOTYPE( void driver_task, (struct driver *dr) );
_PROTOTYPE( char *no_name, (void) );
_PROTOTYPE( int do_nop, (struct driver *dp, message *m_ptr) );
_PROTOTYPE( struct device *nop_prepare, (int device) );
_PROTOTYPE( void nop_cleanup, (void) );
_PROTOTYPE( void nop_task, (void) );
_PROTOTYPE( void nop_signal, (struct driver *dp, message *m_ptr) );
_PROTOTYPE( void nop_alarm, (struct driver *dp, message *m_ptr) );
_PROTOTYPE( int nop_cancel, (struct driver *dp, message *m_ptr) );
_PROTOTYPE( int nop_select, (struct driver *dp, message *m_ptr) );
_PROTOTYPE( int do_diocntl, (struct driver *dp, message *m_ptr) );

/* Parameters for the disk drive. */
#define SECTOR_SIZE      512    /* physical sector size in bytes */
#define SECTOR_SHIFT       9    /* for division */
#define SECTOR_MASK      511    /* and remainder */

/* Size of the DMA buffer buffer in bytes. */
#define USE_EXTRA_DMA_BUF  0    /* usually not needed */
#define DMA_BUF_SIZE    (DMA_SECTORS * SECTOR_SIZE)

#if (CHIP == INTEL)
extern u8_t *tmp_buf;                   /* the DMA buffer */
#else
extern u8_t tmp_buf[];                  /* the DMA buffer */
#endif
extern phys_bytes tmp_phys;             /* phys address of DMA buffer */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                      drivers/libdriver/drvlib.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

/* IBM device driver definitions                      Author: Kees J. Bot
 *                                                            7 Dec 1995
 */

#include <ibm/partition.h>

_PROTOTYPE( void partition, (struct driver *dr, int device, int style, int atapi) );

/* BIOS parameter table layout. */
#define bp_cylinders(t)         (* (u16_t *) (&(t)[0]))
#define bp_heads(t)             (* (u8_t *)  (&(t)[2]))
#define bp_reduced_wr(t)        (* (u16_t *) (&(t)[3]))
#define bp_precomp(t)           (* (u16_t *) (&(t)[5]))
#define bp_max_ecc(t)           (* (u8_t *)  (&(t)[7]))
#define bp_ctlbyte(t)           (* (u8_t *)  (&(t)[8]))
#define bp_landingzone(t)       (* (u16_t *) (&(t)[12]))
#define bp_sectors(t)           (* (u8_t *)  (&(t)[14]))

/* Miscellaneous. */
#define DEV_PER_DRIVE   (1 + NR_PARTITIONS)
#define MINOR_t0        64
#define MINOR_r0        120
#define MINOR_d0p0s0    128
#define MINOR_fd0p0     (28<<2)
#define P_FLOPPY        0
#define P_PRIMARY       1
#define P_SUB           2
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                      drivers/libdriver/driver.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

/* This file contains device independent device driver interface.
 *
 * Changes:
 *   Jul 25, 2005   added SYS_SIG type for signals  (Jorrit N. Herder)
 *   Sep 15, 2004   added SYN_ALARM type for timeouts  (Jorrit N. Herder)
 *   Jul 23, 2004   removed kernel dependencies  (Jorrit N. Herder)
 *   Apr 02, 1992   constructed from AT wini and floppy driver  (Kees J. Bot)
 *
 *
 * The drivers support the following operations (using message format m2):
 *
 *    m_type      DEVICE    PROC_NR     COUNT    POSITION  ADRRESS
 * ----------------------------------------------------------------
 * |  DEV_OPEN  | device  | proc nr |         |         |         |
 * |------------+---------+---------+---------+---------+---------|
 * |  DEV_CLOSE | device  | proc nr |         |         |         |
 * |------------+---------+---------+---------+---------+---------|
 * |  DEV_READ  | device  | proc nr |  bytes  |  offset | buf ptr |
 * |------------+---------+---------+---------+---------+---------|
 * |  DEV_WRITE | device  | proc nr |  bytes  |  offset | buf ptr |
 * |------------+---------+---------+---------+---------+---------|
 * | DEV_GATHER | device  | proc nr | iov len |  offset | iov ptr |
 * |------------+---------+---------+---------+---------+---------|
 * | DEV_SCATTER| device  | proc nr | iov len |  offset | iov ptr |
 * |------------+---------+---------+---------+---------+---------|
 * |  DEV_IOCTL | device  | proc nr |func code|         | buf ptr |
 * |------------+---------+---------+---------+---------+---------|
 * |  CANCEL    | device  | proc nr | r/w     |         |         |
 * |------------+---------+---------+---------+---------+---------|
 * |  HARD_STOP |         |         |         |         |         |
 * ----------------------------------------------------------------
 *
 * The file contains one entry point:
 *
 *   driver_task:       called by the device dependent task entry
 */

#include "../drivers.h"
#include <sys/ioc_disk.h>
#include "driver.h"

#define BUF_EXTRA       0

/* Claim space for variables. */
PRIVATE u8_t buffer[(unsigned) 2 * DMA_BUF_SIZE + BUF_EXTRA];
u8_t *tmp_buf;                  /* the DMA buffer eventually */
phys_bytes tmp_phys;            /* phys address of DMA buffer */

FORWARD _PROTOTYPE( void init_buffer, (void) );
FORWARD _PROTOTYPE( int do_rdwt, (struct driver *dr, message *mp) );
FORWARD _PROTOTYPE( int do_vrdwt, (struct driver *dr, message *mp) );

int device_caller;

/*===========================================================================*
 *                              driver_task                                  *
 *===========================================================================*/
PUBLIC void driver_task(dp)
struct driver *dp;      /* Device dependent entry points. */
{
/* Main program of any device driver task. */

  int r, proc_nr;
  message mess;

  /* Get a DMA buffer. */
  init_buffer();

  /* Here is the main loop of the disk task.  It waits for a message, carries
   * it out, and sends a reply.
   */
  while (TRUE) {

        /* Wait for a request to read or write a disk block. */
        if(receive(ANY, &mess) != OK) continue;

        device_caller = mess.m_source;
        proc_nr = mess.PROC_NR;

        /* Now carry out the work. */
        switch(mess.m_type) {
        case DEV_OPEN:          r = (*dp->dr_open)(dp, &mess);  break;
        case DEV_CLOSE:         r = (*dp->dr_close)(dp, &mess); break;
        case DEV_IOCTL:         r = (*dp->dr_ioctl)(dp, &mess); break;
        case CANCEL:            r = (*dp->dr_cancel)(dp, &mess);break;
        case DEV_SELECT:        r = (*dp->dr_select)(dp, &mess);break;

        case DEV_READ:
        case DEV_WRITE:   r = do_rdwt(dp, &mess);       break;
        case DEV_GATHER:
        case DEV_SCATTER: r = do_vrdwt(dp, &mess);      break;

        case HARD_INT:          /* leftover interrupt or expired timer. */
                                if(dp->dr_hw_int) {
                                        (*dp->dr_hw_int)(dp, &mess);
                                }
                                continue;
        case SYS_SIG:           (*dp->dr_signal)(dp, &mess);
                                continue;       /* don't reply */
        case SYN_ALARM:         (*dp->dr_alarm)(dp, &mess);
                                continue;       /* don't reply */
        default:
                if(dp->dr_other)
                        r = (*dp->dr_other)(dp, &mess);
                else
                        r = EINVAL;
                break;
        }

        /* Clean up leftover state. */
        (*dp->dr_cleanup)();

        /* Finally, prepare and send the reply message. */
        if (r != EDONTREPLY) {
                mess.m_type = TASK_REPLY;
                mess.REP_PROC_NR = proc_nr;
                /* Status is # of bytes transferred or error code. */
                mess.REP_STATUS = r;
                send(device_caller, &mess);
        }
  }
}

/*===========================================================================*
 *                              init_buffer                                  *
 *===========================================================================*/
PRIVATE void init_buffer()
{
/* Select a buffer that can safely be used for DMA transfers.  It may also
 * be used to read partition tables and such.  Its absolute address is
 * 'tmp_phys', the normal address is 'tmp_buf'.
 */

  unsigned left;

  tmp_buf = buffer;
  sys_umap(SELF, D, (vir_bytes)buffer, (phys_bytes)sizeof(buffer), &tmp_phys);

  if ((left = dma_bytes_left(tmp_phys)) < DMA_BUF_SIZE) {
        /* First half of buffer crosses a 64K boundary, can't DMA into that */
        tmp_buf += left;
        tmp_phys += left;
  }
}

/*===========================================================================*
 *                              do_rdwt                                      *
 *===========================================================================*/
PRIVATE int do_rdwt(dp, mp)
struct driver *dp;              /* device dependent entry points */
message *mp;                    /* pointer to read or write message */
{
/* Carry out a single read or write request. */
  iovec_t iovec1;
  int r, opcode;
  phys_bytes phys_addr;

  /* Disk address?  Address and length of the user buffer? */
  if (mp->COUNT < 0) return(EINVAL);

  /* Check the user buffer. */
  sys_umap(mp->PROC_NR, D, (vir_bytes) mp->ADDRESS, mp->COUNT, &phys_addr);
  if (phys_addr == 0) return(EFAULT);

  /* Prepare for I/O. */
  if ((*dp->dr_prepare)(mp->DEVICE) == NIL_DEV) return(ENXIO);

  /* Create a one element scatter/gather vector for the buffer. */
  opcode = mp->m_type == DEV_READ ? DEV_GATHER : DEV_SCATTER;
  iovec1.iov_addr = (vir_bytes) mp->ADDRESS;
  iovec1.iov_size = mp->COUNT;

  /* Transfer bytes from/to the device. */
  r = (*dp->dr_transfer)(mp->PROC_NR, opcode, mp->POSITION, &iovec1, 1);

  /* Return the number of bytes transferred or an error code. */
  return(r == OK ? (mp->COUNT - iovec1.iov_size) : r);
}

/*==========================================================================*
 *                              do_vrdwt                                    *
 *==========================================================================*/
PRIVATE int do_vrdwt(dp, mp)
struct driver *dp;      /* device dependent entry points */
message *mp;            /* pointer to read or write message */
{
/* Carry out an device read or write to/from a vector of user addresses.
 * The "user addresses" are assumed to be safe, i.e. FS transferring to/from
 * its own buffers, so they are not checked.
 */
  static iovec_t iovec[NR_IOREQS];
  iovec_t *iov;
  phys_bytes iovec_size;
  unsigned nr_req;
  int r;

  nr_req = mp->COUNT;   /* Length of I/O vector */

  if (mp->m_source < 0) {
    /* Called by a task, no need to copy vector. */
    iov = (iovec_t *) mp->ADDRESS;
  } else {
    /* Copy the vector from the caller to kernel space. */
    if (nr_req > NR_IOREQS) nr_req = NR_IOREQS;
    iovec_size = (phys_bytes) (nr_req * sizeof(iovec[0]));

    if (OK != sys_datacopy(mp->m_source, (vir_bytes) mp->ADDRESS,
                SELF, (vir_bytes) iovec, iovec_size))
        panic((*dp->dr_name)(),"bad I/O vector by", mp->m_source);
    iov = iovec;
  }

  /* Prepare for I/O. */
  if ((*dp->dr_prepare)(mp->DEVICE) == NIL_DEV) return(ENXIO);

  /* Transfer bytes from/to the device. */
  r = (*dp->dr_transfer)(mp->PROC_NR, mp->m_type, mp->POSITION, iov, nr_req);

  /* Copy the I/O vector back to the caller. */
  if (mp->m_source >= 0) {
    sys_datacopy(SELF, (vir_bytes) iovec,
        mp->m_source, (vir_bytes) mp->ADDRESS, iovec_size);
  }
  return(r);
}

/*===========================================================================*
 *                              no_name                                      *
 *===========================================================================*/
PUBLIC char *no_name()
{
/* Use this default name if there is no specific name for the device. This was
 * originally done by fetching the name from the task table for this process:
 * "return(tasktab[proc_number(proc_ptr) + NR_TASKS].name);", but currently a
 * real "noname" is returned. Perhaps, some system information service can be
 * queried for a name at a later time.
 */
  static char name[] = "noname";
  return name;
}

/*============================================================================*
 *                              do_nop                                        *
 *============================================================================*/
PUBLIC int do_nop(dp, mp)
struct driver *dp;
message *mp;
{
/* Nothing there, or nothing to do. */

  switch (mp->m_type) {
  case DEV_OPEN:        return(ENODEV);
  case DEV_CLOSE:       return(OK);
  case DEV_IOCTL:       return(ENOTTY);
  default:              return(EIO);
  }
}
```

```
/*===========================================================================*
 *                              nop_signal                                   *
 *===========================================================================*/
PUBLIC void nop_signal(dp, mp)
struct driver *dp;
message *mp;
{
/* Default action for signal is to ignore. */
}

/*===========================================================================*
 *                              nop_alarm                                    *
 *===========================================================================*/
PUBLIC void nop_alarm(dp, mp)
struct driver *dp;
message *mp;
{
/* Ignore the leftover alarm. */
}

/*==========================================================================*
 *                              nop_prepare                                 *
 *==========================================================================*/
PUBLIC struct device *nop_prepare(device)
{
/* Nothing to prepare for. */
  return(NIL_DEV);
}

/*==========================================================================*
 *                              nop_cleanup                                 *
 *==========================================================================*/
PUBLIC void nop_cleanup()
{
/* Nothing to clean up. */
}

/*==========================================================================*
 *                              nop_cancel                                  *
 *==========================================================================*/
PUBLIC int nop_cancel(struct driver *dr, message *m)
{
/* Nothing to do for cancel. */
   return(OK);
}

/*==========================================================================*
 *                              nop_select                                  *
 *==========================================================================*/
PUBLIC int nop_select(struct driver *dr, message *m)
{
/* Nothing to do for select. */
   return(OK);
}

/*===========================================================================*
 *                              do_diocntl                                   *
 *===========================================================================*/
PUBLIC int do_diocntl(dp, mp)
struct driver *dp;
```

```
message *mp;                    /* pointer to ioctl request */
{
/* Carry out a partition setting/getting request. */
  struct device *dv;
  struct partition entry;
  int s;

  if (mp->REQUEST != DIOCSETP && mp->REQUEST != DIOCGETP) {
        if(dp->dr_other) {
                return dp->dr_other(dp, mp);
        } else return(ENOTTY);
  }

  /* Decode the message parameters. */
  if ((dv = (*dp->dr_prepare)(mp->DEVICE)) == NIL_DEV) return(ENXIO);

  if (mp->REQUEST == DIOCSETP) {
        /* Copy just this one partition table entry. */
        if (OK != (s=sys_datacopy(mp->PROC_NR, (vir_bytes) mp->ADDRESS,
                SELF, (vir_bytes) &entry, sizeof(entry))))
            return s;
        dv->dv_base = entry.base;
        dv->dv_size = entry.size;
  } else {
        /* Return a partition table entry and the geometry of the drive. */
        entry.base = dv->dv_base;
        entry.size = dv->dv_size;
        (*dp->dr_geometry)(&entry);
        if (OK != (s=sys_datacopy(SELF, (vir_bytes) &entry,
                mp->PROC_NR, (vir_bytes) mp->ADDRESS, sizeof(entry))))
            return s;
  }
  return(OK);
}
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                              drivers/libdriver/drvlib.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/* IBM device driver utility functions.                 Author: Kees J. Bot
 *                                                              7 Dec 1995
 * Entry point:
 *   partition: partition a disk to the partition table(s) on it.
 */

#include "driver.h"
#include "drvlib.h"
#include <unistd.h>

/* Extended partition? */
#define ext_part(s)     ((s) == 0x05 || (s) == 0x0F)

FORWARD _PROTOTYPE( void extpartition, (struct driver *dp, int extdev,
                                                unsigned long extbase) );
FORWARD _PROTOTYPE( int get_part_table, (struct driver *dp, int device,
                        unsigned long offset, struct part_entry *table));
FORWARD _PROTOTYPE( void sort, (struct part_entry *table) );

#ifndef CD_SECTOR_SIZE
```

```
#define CD_SECTOR_SIZE 2048
#endif

/*===========================================================================*
 *                              partition                                    *
 *===========================================================================*/
PUBLIC void partition(dp, device, style, atapi)
struct driver *dp;      /* device dependent entry points */
int device;             /* device to partition */
int style;              /* partitioning style: floppy, primary, sub. */
int atapi;              /* atapi device */
{
/* This routine is called on first open to initialize the partition tables
 * of a device.  It makes sure that each partition falls safely within the
 * device's limits.  Depending on the partition style we are either making
 * floppy partitions, primary partitions or subpartitions.  Only primary
 * partitions are sorted, because they are shared with other operating
 * systems that expect this.
 */
  struct part_entry table[NR_PARTITIONS], *pe;
  int disk, par;
  struct device *dv;
  unsigned long base, limit, part_limit;

  /* Get the geometry of the device to partition */
  if ((dv = (*dp->dr_prepare)(device)) == NIL_DEV
                                || cmp64u(dv->dv_size, 0) == 0) return;
  base = div64u(dv->dv_base, SECTOR_SIZE);
  limit = base + div64u(dv->dv_size, SECTOR_SIZE);

  /* Read the partition table for the device. */
  if(!get_part_table(dp, device, 0L, table)) {
          return;
  }

  /* Compute the device number of the first partition. */
  switch (style) {
  case P_FLOPPY:
        device += MINOR_fd0p0;
        break;
  case P_PRIMARY:
        sort(table);            /* sort a primary partition table */
        device += 1;
        break;
  case P_SUB:
        disk = device / DEV_PER_DRIVE;
        par = device % DEV_PER_DRIVE - 1;
        device = MINOR_d0p0s0 + (disk * NR_PARTITIONS + par) * NR_PARTITIONS;
  }

  /* Find an array of devices. */
  if ((dv = (*dp->dr_prepare)(device)) == NIL_DEV) return;

  /* Set the geometry of the partitions from the partition table. */
  for (par = 0; par < NR_PARTITIONS; par++, dv++) {
        /* Shrink the partition to fit within the device. */
        pe = &table[par];
        part_limit = pe->lowsec + pe->size;
        if (part_limit < pe->lowsec) part_limit = limit;
        if (part_limit > limit) part_limit = limit;
        if (pe->lowsec < base) pe->lowsec = base;
        if (part_limit < pe->lowsec) part_limit = pe->lowsec;

        dv->dv_base = mul64u(pe->lowsec, SECTOR_SIZE);
        dv->dv_size = mul64u(part_limit - pe->lowsec, SECTOR_SIZE);

        if (style == P_PRIMARY) {
                /* Each Minix primary partition can be subpartitioned. */
                if (pe->sysind == MINIX_PART)
                        partition(dp, device + par, P_SUB, atapi);

                /* An extended partition has logical partitions. */
                if (ext_part(pe->sysind))
                        extpartition(dp, device + par, pe->lowsec);
        }
  }
}

/*===========================================================================*
 *                              extpartition                                 *
 *===========================================================================*/
PRIVATE void extpartition(dp, extdev, extbase)
struct driver *dp;      /* device dependent entry points */
int extdev;             /* extended partition to scan */
unsigned long extbase;  /* sector offset of the base extended partition */
{
/* Extended partitions cannot be ignored alas, because people like to move
 * files to and from DOS partitions.  Avoid reading this code, it's no fun.
 */
  struct part_entry table[NR_PARTITIONS], *pe;
  int subdev, disk, par;
  struct device *dv;
  unsigned long offset, nextoffset;

  disk = extdev / DEV_PER_DRIVE;
  par = extdev % DEV_PER_DRIVE - 1;
  subdev = MINOR_d0p0s0 + (disk * NR_PARTITIONS + par) * NR_PARTITIONS;

  offset = 0;
  do {
        if (!get_part_table(dp, extdev, offset, table)) return;
        sort(table);

        /* The table should contain one logical partition and optionally
         * another extended partition.  (It's a linked list.)
         */
        nextoffset = 0;
        for (par = 0; par < NR_PARTITIONS; par++) {
                pe = &table[par];
                if (ext_part(pe->sysind)) {
                        nextoffset = pe->lowsec;
                } else
                if (pe->sysind != NO_PART) {
                        if ((dv = (*dp->dr_prepare)(subdev)) == NIL_DEV) return;

                        dv->dv_base = mul64u(extbase + offset + pe->lowsec,
                                                                SECTOR_SIZE);
                        dv->dv_size = mul64u(pe->size, SECTOR_SIZE);

                        /* Out of devices? */
                        if (++subdev % NR_PARTITIONS == 0) return;
                }
        }
  } while ((offset = nextoffset) != 0);
}

/*===========================================================================*
 *                              get_part_table                               *
 *===========================================================================*/
PRIVATE int get_part_table(dp, device, offset, table)
struct driver *dp;
int device;
unsigned long offset;           /* sector offset to the table */
struct part_entry *table;       /* four entries */
{
/* Read the partition table for the device, return true iff there were no
 * errors.
 */
  iovec_t iovec1;
  off_t position;
  static unsigned char partbuf[CD_SECTOR_SIZE];

  position = offset << SECTOR_SHIFT;
  iovec1.iov_addr = (vir_bytes) partbuf;
  iovec1.iov_size = CD_SECTOR_SIZE;
  if ((*dp->dr_prepare)(device) != NIL_DEV) {
        (void) (*dp->dr_transfer)(SELF, DEV_GATHER, position, &iovec1, 1);
  }
  if (iovec1.iov_size != 0) {
        return 0;
  }
  if (partbuf[510] != 0x55 || partbuf[511] != 0xAA) {
        /* Invalid partition table. */
        return 0;
  }
  memcpy(table, (partbuf + PART_TABLE_OFF), NR_PARTITIONS * sizeof(table[0]));
  return 1;
}

/*===========================================================================*
 *                              sort                                         *
 *===========================================================================*/
PRIVATE void sort(table)
struct part_entry *table;
{
/* Sort a partition table. */
  struct part_entry *pe, tmp;
  int n = NR_PARTITIONS;

  do {
        for (pe = table; pe < table + NR_PARTITIONS-1; pe++) {
                if (pe[0].sysind == NO_PART
                        || (pe[0].lowsec > pe[1].lowsec
                                        && pe[1].sysind != NO_PART)) {
                        tmp = pe[0]; pe[0] = pe[1]; pe[1] = tmp;
                }
        }
  } while (--n > 0);
}
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                      drivers/memory/memory.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

/* This file contains the device dependent part of the drivers for the
 * following special files:
 *     /dev/ram         - RAM disk
 *     /dev/mem         - absolute memory
 *     /dev/kmem        - kernel virtual memory
 *     /dev/null        - null device (data sink)
 *     /dev/boot        - boot device loaded from boot image
 *     /dev/zero        - null byte stream generator
 *
 *  Changes:
 *      Apr 29, 2005    added null byte generator  (Jorrit N. Herder)
 *      Apr 09, 2005    added support for boot device  (Jorrit N. Herder)
 *      Jul 26, 2004    moved RAM driver to user-space  (Jorrit N. Herder)
 *      Apr 20, 1992    device dependent/independent split  (Kees J. Bot)
 */

#include "../drivers.h"
#include "../libdriver/driver.h"
#include <sys/ioc_memory.h>
#include "../../kernel/const.h"
#include "../../kernel/config.h"
#include "../../kernel/type.h"

#include "assert.h"

#define NR_DEVS            6            /* number of minor devices */

PRIVATE struct device m_geom[NR_DEVS];  /* base and size of each device */
PRIVATE int m_seg[NR_DEVS];             /* segment index of each device */
PRIVATE int m_device;                   /* current device */
PRIVATE struct kinfo kinfo;             /* kernel information */
PRIVATE struct machine machine;         /* machine information */

extern int errno;                       /* error number for PM calls */

FORWARD _PROTOTYPE( char *m_name, (void)                                );
FORWARD _PROTOTYPE( struct device *m_prepare, (int device)              );
FORWARD _PROTOTYPE( int m_transfer, (int proc_nr, int opcode, off_t position,
                                        iovec_t *iov, unsigned nr_req)  );
FORWARD _PROTOTYPE( int m_do_open, (struct driver *dp, message *m_ptr)  );
FORWARD _PROTOTYPE( void m_init, (void) );
FORWARD _PROTOTYPE( int m_ioctl, (struct driver *dp, message *m_ptr)    );
FORWARD _PROTOTYPE( void m_geometry, (struct partition *entry)          );

/* Entry points to this driver. */
PRIVATE struct driver m_dtab = {
  m_name,       /* current device's name */
  m_do_open,    /* open or mount */
  do_nop,       /* nothing on a close */
  m_ioctl,      /* specify ram disk geometry */
  m_prepare,    /* prepare for I/O on a given minor device */
  m_transfer,   /* do the I/O */
  nop_cleanup,  /* no need to clean up */
  m_geometry,   /* memory device "geometry" */
  nop_signal,   /* system signals */
  nop_alarm,
  nop_cancel,
  nop_select,
  NULL,
  NULL
};

/* Buffer for the /dev/zero null byte feed. */
#define ZERO_BUF_SIZE                   1024
PRIVATE char dev_zero[ZERO_BUF_SIZE];

#define click_to_round_k(n) \
        ((unsigned) ((((unsigned long) (n) << CLICK_SHIFT) + 512) / 1024))

/*===========================================================================*
 *                              main                                         *
 *===========================================================================*/
PUBLIC int main(void)
{
/* Main program. Initialize the memory driver and start the main loop. */
  m_init();
  driver_task(&m_dtab);
  return(OK);
}

/*===========================================================================*
 *                              m_name                                       *
 *===========================================================================*/
PRIVATE char *m_name()
{
/* Return a name for the current device. */
  static char name[] = "memory";
  return name;
}

/*===========================================================================*
 *                              m_prepare                                    *
 *===========================================================================*/
PRIVATE struct device *m_prepare(device)
int device;
{
/* Prepare for I/O on a device: check if the minor device number is ok. */
  if (device < 0 || device >= NR_DEVS) return(NIL_DEV);
  m_device = device;

  return(&m_geom[device]);
}

/*===========================================================================*
 *                              m_transfer                                   *
 *===========================================================================*/
PRIVATE int m_transfer(proc_nr, opcode, position, iov, nr_req)
int proc_nr;                    /* process doing the request */
int opcode;                     /* DEV_GATHER or DEV_SCATTER */
off_t position;                 /* offset on device to read or write */
iovec_t *iov;                   /* pointer to read or write request vector */
unsigned nr_req;                /* length of request vector */
{
/* Read or write one the driver's minor devices. */
  phys_bytes mem_phys;
  int seg;
  unsigned count, left, chunk;
  vir_bytes user_vir;
  struct device *dv;
  unsigned long dv_size;
  int s;

  /* Get minor device number and check for /dev/null. */
  dv = &m_geom[m_device];
  dv_size = cv64ul(dv->dv_size);

  while (nr_req > 0) {

        /* How much to transfer and where to / from. */
        count = iov->iov_size;
        user_vir = iov->iov_addr;

        switch (m_device) {

        /* No copying; ignore request. */
        case NULL_DEV:
            if (opcode == DEV_GATHER) return(OK);       /* always at EOF */
            break;

        /* Virtual copying. For RAM disk, kernel memory and boot device. */
        case RAM_DEV:
        case KMEM_DEV:
        case BOOT_DEV:
            if (position >= dv_size) return(OK);        /* check for EOF */
            if (position + count > dv_size) count = dv_size - position;
            seg = m_seg[m_device];

            if (opcode == DEV_GATHER) {                 /* copy actual data */
                sys_vircopy(SELF,seg,position, proc_nr,D,user_vir, count);
            } else {
                sys_vircopy(proc_nr,D,user_vir, SELF,seg,position, count);
            }
            break;

        /* Physical copying. Only used to access entire memory. */
        case MEM_DEV:
            if (position >= dv_size) return(OK);        /* check for EOF */
            if (position + count > dv_size) count = dv_size - position;
            mem_phys = cv64ul(dv->dv_base) + position;

            if (opcode == DEV_GATHER) {                 /* copy data */
                sys_physcopy(NONE, PHYS_SEG, mem_phys,
                        proc_nr, D, user_vir, count);
            } else {
                sys_physcopy(proc_nr, D, user_vir,
                        NONE, PHYS_SEG, mem_phys, count);
            }
            break;

        /* Null byte stream generator. */
        case ZERO_DEV:
            if (opcode == DEV_GATHER) {
                left = count;
                while (left > 0) {
                    chunk = (left > ZERO_BUF_SIZE) ? ZERO_BUF_SIZE : left;
                    if (OK != (s=sys_vircopy(SELF, D, (vir_bytes) dev_zero,
                            proc_nr, D, user_vir, chunk)))
                        report("MEM","sys_vircopy failed", s);
                    left -= chunk;
                    user_vir += chunk;
                }
            }
            break;

        /* Unknown (illegal) minor device. */
        default:
            return(EINVAL);
        }

        /* Book the number of bytes transferred. */
        position += count;
        iov->iov_addr += count;
        if ((iov->iov_size -= count) == 0) { iov++; nr_req--; }

  }
  return(OK);
}

/*===========================================================================*
 *                              m_do_open                                    *
 *===========================================================================*/
PRIVATE int m_do_open(dp, m_ptr)
struct driver *dp;
message *m_ptr;
{
/* Check device number on open.  (This used to give I/O privileges to a
 * process opening /dev/mem or /dev/kmem. This may be needed in case of
 * memory mapped I/O. With system calls to do I/O this is no longer needed.)
 */
  if (m_prepare(m_ptr->DEVICE) == NIL_DEV) return(ENXIO);

  return(OK);
}

/*===========================================================================*
 *                              m_init                                       *
 *===========================================================================*/
PRIVATE void m_init()
{
  /* Initialize this task. All minor devices are initialized one by one. */
  int i, s;

  if (OK != (s=sys_getkinfo(&kinfo))) {
      panic("MEM","Couldn't get kernel information.",s);
  }

  /* Install remote segment for /dev/kmem memory. */
  m_geom[KMEM_DEV].dv_base = cvul64(kinfo.kmem_base);
  m_geom[KMEM_DEV].dv_size = cvul64(kinfo.kmem_size);
  if (OK != (s=sys_segctl(&m_seg[KMEM_DEV], (u16_t *) &s, (vir_bytes *) &s,
              kinfo.kmem_base, kinfo.kmem_size))) {
      panic("MEM","Couldn't install remote segment.",s);
  }

  /* Install remote segment for /dev/boot memory, if enabled. */
  m_geom[BOOT_DEV].dv_base = cvul64(kinfo.bootdev_base);
  m_geom[BOOT_DEV].dv_size = cvul64(kinfo.bootdev_size);
  if (kinfo.bootdev_base > 0) {
      if (OK != (s=sys_segctl(&m_seg[BOOT_DEV], (u16_t *) &s, (vir_bytes *) &s,
              kinfo.bootdev_base, kinfo.bootdev_size))) {
          panic("MEM","Couldn't install remote segment.",s);
      }
  }

  /* Initialize /dev/zero. Simply write zeros into the buffer. */
  for (i=0; i<ZERO_BUF_SIZE; i++) {
       dev_zero[i] = '\0';
  }

  /* Set up memory ranges for /dev/mem. */
  if (OK != (s=sys_getmachine(&machine))) {
      panic("MEM","Couldn't get machine information.",s);
  }
  if (! machine.protected) {
        m_geom[MEM_DEV].dv_size =   cvul64(0x100000); /* 1M for 8086 systems */
  } else {
        m_geom[MEM_DEV].dv_size = cvul64(0xFFFFFFFF); /* 4G-1 for 386 systems */
  }
}

/*===========================================================================*
 *                              m_ioctl                                      *
 *===========================================================================*/
PRIVATE int m_ioctl(dp, m_ptr)
struct driver *dp;                      /* pointer to driver structure */
message *m_ptr;                         /* pointer to control message */
{
/* I/O controls for the memory driver. Currently there is one I/O control:
 * - MIOCRAMSIZE: to set the size of the RAM disk.
 */
  struct device *dv;
  if ((dv = m_prepare(m_ptr->DEVICE)) == NIL_DEV) return(ENXIO);

  switch (m_ptr->REQUEST) {
    case MIOCRAMSIZE: {
        /* FS wants to create a new RAM disk with the given size. */
        phys_bytes ramdev_size;
        phys_bytes ramdev_base;
        int s;

        if (m_ptr->PROC_NR != FS_PROC_NR) {
            report("MEM", "warning, MIOCRAMSIZE called by", m_ptr->PROC_NR);
            return(EPERM);
        }

        /* Try to allocate a piece of memory for the RAM disk. */
        ramdev_size = m_ptr->POSITION;
        if (allocmem(ramdev_size, &ramdev_base) < 0) {
            report("MEM", "warning, allocmem failed", errno);
            return(ENOMEM);
        }
        dv->dv_base = cvul64(ramdev_base);
        dv->dv_size = cvul64(ramdev_size);

        if (OK != (s=sys_segctl(&m_seg[RAM_DEV], (u16_t *) &s, (vir_bytes *) &s,
                ramdev_base, ramdev_size))) {
                panic("MEM","Couldn't install remote segment.",s);
        }
        break;
    }

    default:
        return(do_diocntl(&m_dtab, m_ptr));
  }
  return(OK);
}

/*===========================================================================*
 *                              m_geometry                                   *
 *===========================================================================*/
PRIVATE void m_geometry(entry)
struct partition *entry;
{
  /* Memory devices don't have a geometry, but the outside world insists. */
  entry->cylinders = div64u(m_geom[m_device].dv_size, SECTOR_SIZE) / (64 * 32);
  entry->heads = 64;
  entry->sectors = 32;
}
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                    drivers/at_wini/at_wini.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
#include "../drivers.h"
#include "../libdriver/driver.h"
#include "../libdriver/drvlib.h"

_PROTOTYPE(int main, (void));

#define VERBOSE           0     /* display identify messages during boot */
#define ENABLE_ATAPI    0       /* add ATAPI cd-rom support to driver */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                    drivers/at_wini/at_wini.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/* This file contains the device dependent part of a driver for the IBM-AT
 * winchester controller.  Written by Adri Koppes.
 *
 * The file contains one entry point:
 *
 *   at_winchester_task:        main entry when system is brought up
 *
 * Changes:
 *   Aug 19, 2005   ata pci support, supports SATA  (Ben Gras)
 *   Nov 18, 2004   moved AT disk driver to user-space  (Jorrit N. Herder)
 *   Aug 20, 2004   watchdogs replaced by sync alarms  (Jorrit N. Herder)
 *   Mar 23, 2000   added ATAPI CDROM support  (Michael Temari)
 *   May 14, 2000   d-d/i rewrite  (Kees J. Bot)
 *   Apr 13, 1992   device dependent/independent split  (Kees J. Bot)
 */
```

```

#include "at_wini.h"
#include "../libpci/pci.h"

#include <minix/sysutil.h>
#include <minix/keymap.h>
#include <sys/ioc_disk.h>

#define ATAPI_DEBUG         0   /* To debug ATAPI code. */

/* I/O Ports used by winchester disk controllers. */

/* Read and write registers */
#define REG_CMD_BASE0   0x1F0   /* command base register of controller 0 */
#define REG_CMD_BASE1   0x170   /* command base register of controller 1 */
#define REG_CTL_BASE0   0x3F6   /* control base register of controller 0 */
#define REG_CTL_BASE1   0x376   /* control base register of controller 1 */

#define REG_DATA            0   /* data register (offset from the base reg.) */
#define REG_PRECOMP         1   /* start of write precompensation */
#define REG_COUNT           2   /* sectors to transfer */
#define REG_SECTOR          3   /* sector number */
#define REG_CYL_LO          4   /* low byte of cylinder number */
#define REG_CYL_HI          5   /* high byte of cylinder number */
#define REG_LDH             6   /* lba, drive and head */
#define   LDH_DEFAULT           0xA0    /* ECC enable, 512 bytes per sector */
#define   LDH_LBA               0x40    /* Use LBA addressing */
#define   ldh_init(drive)       (LDH_DEFAULT | ((drive) << 4))

/* Read only registers */
#define REG_STATUS          7   /* status */
#define   STATUS_BSY            0x80    /* controller busy */
#define   STATUS_RDY            0x40    /* drive ready */
#define   STATUS_WF             0x20    /* write fault */
#define   STATUS_SC             0x10    /* seek complete (obsolete) */
#define   STATUS_DRQ            0x08    /* data transfer request */
#define   STATUS_CRD            0x04    /* corrected data */
#define   STATUS_IDX            0x02    /* index pulse */
#define   STATUS_ERR            0x01    /* error */
#define   STATUS_ADMBSY        0x100    /* administratively busy (software) */
#define REG_ERROR           1   /* error code */
#define   ERROR_BB              0x80    /* bad block */
#define   ERROR_ECC             0x40    /* bad ecc bytes */
#define   ERROR_ID              0x10    /* id not found */
#define   ERROR_AC              0x04    /* aborted command */
#define   ERROR_TK              0x02    /* track zero error */
#define   ERROR_DM              0x01    /* no data address mark */

/* Write only registers */
#define REG_COMMAND         7   /* command */
#define   CMD_IDLE              0x00    /* for w_command: drive idle */
#define   CMD_RECALIBRATE       0x10    /* recalibrate drive */
#define   CMD_READ              0x20    /* read data */
#define   CMD_READ_EXT          0x24    /* read data (LBA48 addressed) */
#define   CMD_WRITE             0x30    /* write data */
#define   CMD_WRITE_EXT         0x34    /* write data (LBA48 addressed) */
#define   CMD_READVERIFY        0x40    /* read verify */
#define   CMD_FORMAT            0x50    /* format track */
#define   CMD_SEEK              0x70    /* seek cylinder */
#define   CMD_DIAG              0x90    /* execute device diagnostics */
```

```
#define   CMD_SPECIFY           0x91    /* specify parameters */
#define   ATA_IDENTIFY          0xEC    /* identify drive */
/* #define REG_CTL              0x206   */ /* control register */
#define REG_CTL         0       /* control register */
#define   CTL_NORETRY           0x80    /* disable access retry */
#define   CTL_NOECC             0x40    /* disable ecc retry */
#define   CTL_EIGHTHEADS        0x08    /* more than eight heads */
#define   CTL_RESET             0x04    /* reset controller */
#define   CTL_INTDISABLE        0x02    /* disable interrupts */

#define REG_STATUS          7   /* status */
#define   STATUS_BSY            0x80    /* controller busy */
#define   STATUS_DRDY           0x40    /* drive ready */
#define   STATUS_DMADF          0x20    /* dma ready/drive fault */
#define   STATUS_SRVCDSC        0x10    /* service or dsc */
#define   STATUS_DRQ            0x08    /* data transfer request */
#define   STATUS_CORR           0x04    /* correctable error occurred */
#define   STATUS_CHECK          0x01    /* check error */

/* Interrupt request lines. */
#define NO_IRQ           0      /* no IRQ set yet */

#define ATAPI_PACKETSIZE        12
#define SENSE_PACKETSIZE        18

/* Common command block */
struct command {
  u8_t  precomp;        /* REG_PRECOMP, etc. */
  u8_t  count;
  u8_t  sector;
  u8_t  cyl_lo;
  u8_t  cyl_hi;
  u8_t  ldh;
  u8_t  command;
};

/* Error codes */
#define ERR              (-1)   /* general error */
#define ERR_BAD_SECTOR   (-2)   /* block marked bad detected */

/* Some controllers don't interrupt, the clock will wake us up. */
#define WAKEUP          (32*HZ) /* drive may be out for 31 seconds max */

/* Miscellaneous. */
#define MAX_DRIVES         8
#define COMPAT_DRIVES      4
#define MAX_SECS         256    /* controller can transfer this many sectors */
#define MAX_ERRORS         4    /* how often to try rd/wt before quitting */
#define NR_MINORS       (MAX_DRIVES * DEV_PER_DRIVE)
#define SUB_PER_DRIVE   (NR_PARTITIONS * NR_PARTITIONS)
#define NR_SUBDEVS      (MAX_DRIVES * SUB_PER_DRIVE)
#define DELAY_USECS     1000    /* controller timeout in microseconds */
#define DELAY_TICKS        1    /* controller timeout in ticks */
#define DEF_TIMEOUT_TICKS       300     /* controller timeout in ticks */
#define RECOVERY_USECS 500000   /* controller recovery time in microseconds */
#define RECOVERY_TICKS    30    /* controller recovery time in ticks */
#define INITIALIZED     0x01    /* drive is initialized */
#define DEAF            0x02    /* controller must be reset */
#define SMART           0x04    /* drive supports ATA commands */
#define ATAPI              0    /* don't bother with ATAPI; optimise out */
```

```
#define IDENTIFIED      0x10    /* w_identify done successfully */
#define IGNORING        0x20    /* w_identify failed once */

/* Timeouts and max retries. */
int timeout_ticks = DEF_TIMEOUT_TICKS, max_errors = MAX_ERRORS;
int wakeup_ticks = WAKEUP;
long w_standard_timeouts = 0, w_pci_debug = 0, w_instance = 0,
 w_lba48 = 0, atapi_debug = 0;

int w_testing = 0, w_silent = 0;

int w_next_drive = 0;

/* Variables. */

/* wini is indexed by controller first, then drive (0-3).
 * controller 0 is always the 'compatability' ide controller, at
 * the fixed locations, whether present or not.
 */
PRIVATE struct wini {           /* main drive struct, one entry per drive */
  unsigned state;               /* drive state: deaf, initialized, dead */
  unsigned w_status;            /* device status register */
  unsigned base_cmd;            /* command base register */
  unsigned base_ctl;            /* control base register */
  unsigned irq;                 /* interrupt request line */
  unsigned irq_mask;            /* 1 << irq */
  unsigned irq_need_ack;        /* irq needs to be acknowledged */
  int irq_hook_id;              /* id of irq hook at the kernel */
  int lba48;                    /* supports lba48 */
  unsigned lcylinders;          /* logical number of cylinders (BIOS) */
  unsigned lheads;              /* logical number of heads */
  unsigned lsectors;            /* logical number of sectors per track */
  unsigned pcylinders;          /* physical number of cylinders (translated) */
  unsigned pheads;              /* physical number of heads */
  unsigned psectors;            /* physical number of sectors per track */
  unsigned ldhpref;             /* top four bytes of the LDH (head) register */
  unsigned precomp;             /* write precompensation cylinder / 4 */
  unsigned max_count;           /* max request for this drive */
  unsigned open_ct;             /* in-use count */
  struct device part[DEV_PER_DRIVE];    /* disks and partitions */
  struct device subpart[SUB_PER_DRIVE]; /* subpartitions */
} wini[MAX_DRIVES], *w_wn;

PRIVATE int w_device = -1;
PRIVATE int w_controller = -1;
PRIVATE int w_major = -1;
PRIVATE char w_id_string[40];

PRIVATE int win_tasknr;                 /* my task number */
PRIVATE int w_command;                  /* current command in execution */
PRIVATE u8_t w_byteval;                 /* used for SYS_IRQCTL */
PRIVATE int w_drive;                    /* selected drive */
PRIVATE int w_controller;               /* selected controller */
PRIVATE struct device *w_dv;            /* device's base and size */

FORWARD _PROTOTYPE( void init_params, (void)                         );
FORWARD _PROTOTYPE( void init_drive, (struct wini *, int, int, int, int, int, int));
FORWARD _PROTOTYPE( void init_params_pci, (int)                      );
FORWARD _PROTOTYPE( int w_do_open, (struct driver *dp, message *m_ptr)  );
FORWARD _PROTOTYPE( struct device *w_prepare, (int dev)              );
FORWARD _PROTOTYPE( int w_identify, (void)                              );
FORWARD _PROTOTYPE( char *w_name, (void)                                );
FORWARD _PROTOTYPE( int w_specify, (void)                               );
FORWARD _PROTOTYPE( int w_io_test, (void)                               );
FORWARD _PROTOTYPE( int w_transfer, (int proc_nr, int opcode, off_t position,
                                        iovec_t *iov, unsigned nr_req)  );
FORWARD _PROTOTYPE( int com_out, (struct command *cmd)                  );
FORWARD _PROTOTYPE( void w_need_reset, (void)                           );
FORWARD _PROTOTYPE( void ack_irqs, (unsigned int)                       );
FORWARD _PROTOTYPE( int w_do_close, (struct driver *dp, message *m_ptr) );
FORWARD _PROTOTYPE( int w_other, (struct driver *dp, message *m_ptr)    );
FORWARD _PROTOTYPE( int w_hw_int, (struct driver *dp, message *m_ptr)   );
FORWARD _PROTOTYPE( int com_simple, (struct command *cmd)               );
FORWARD _PROTOTYPE( void w_timeout, (void)                              );
FORWARD _PROTOTYPE( int w_reset, (void)                                 );
FORWARD _PROTOTYPE( void w_intr_wait, (void)                            );
FORWARD _PROTOTYPE( int at_intr_wait, (void)                            );
FORWARD _PROTOTYPE( int w_waitfor, (int mask, int value)                );
FORWARD _PROTOTYPE( void w_geometry, (struct partition *entry)          );

/* Entry points to this driver. */
PRIVATE struct driver w_dtab = {
  w_name,               /* current device's name */
  w_do_open,            /* open or mount request, initialize device */
  w_do_close,           /* release device */
  do_diocntl,           /* get or set a partition's geometry */
  w_prepare,            /* prepare for I/O on a given minor device */
  w_transfer,           /* do the I/O */
  nop_cleanup,          /* nothing to clean up */
  w_geometry,           /* tell the geometry of the disk */
  nop_signal,           /* no cleanup needed on shutdown */
  nop_alarm,            /* ignore leftover alarms */
  nop_cancel,           /* ignore CANCELs */
  nop_select,           /* ignore selects */
  w_other,              /* catch-all for unrecognized commands and ioctls */
  w_hw_int              /* leftover hardware interrupts */
};

/*===========================================================================*
 *                              at_winchester_task                           *
 *===========================================================================*/
PUBLIC int main()
{
/* Set special disk parameters then call the generic main loop. */
  init_params();
  driver_task(&w_dtab);
  return(OK);
}

/*===========================================================================*
 *                              init_params                                  *
 *===========================================================================*/
PRIVATE void init_params()
{
/* This routine is called at startup to initialize the drive parameters. */

  u16_t parv[2];
  unsigned int vector, size;
  int drive, nr_drives;
  struct wini *wn;
  u8_t params[16];
  int s;

  /* Boot variables. */
  env_parse("ata_std_timeout", "d", 0, &w_standard_timeouts, 0, 1);
  env_parse("ata_pci_debug", "d", 0, &w_pci_debug, 0, 1);
  env_parse("ata_instance", "d", 0, &w_instance, 0, 8);
  env_parse("ata_lba48", "d", 0, &w_lba48, 0, 1);
  env_parse("atapi_debug", "d", 0, &atapi_debug, 0, 1);

  if (w_instance == 0) {
          /* Get the number of drives from the BIOS data area */
          if ((s=sys_vircopy(SELF, BIOS_SEG, NR_HD_DRIVES_ADDR,
                          SELF, D, (vir_bytes) params, NR_HD_DRIVES_SIZE)) != OK)
                panic(w_name(), "Couldn't read BIOS", s);
          if ((nr_drives = params[0]) > 2) nr_drives = 2;

          for (drive = 0, wn = wini; drive < COMPAT_DRIVES; drive++, wn++) {
                if (drive < nr_drives) {
                    /* Copy the BIOS parameter vector */
                    vector = (drive == 0) ? BIOS_HD0_PARAMS_ADDR:BIOS_HD1_PARAMS_ADDR;
                    size = (drive == 0) ? BIOS_HD0_PARAMS_SIZE:BIOS_HD1_PARAMS_SIZE;
                    if ((s=sys_vircopy(SELF, BIOS_SEG, vector,
                                        SELF, D, (vir_bytes) parv, size)) != OK)
                                panic(w_name(), "Couldn't read BIOS", s);

                        /* Calculate the address of the parameters and copy them */
                        if ((s=sys_vircopy(
                                SELF, BIOS_SEG, hclick_to_physb(parv[1]) + parv[0],
                                SELF, D, (phys_bytes) params, 16L))!=OK)
                            panic(w_name(),"Couldn't copy parameters", s);

                        /* Copy the parameters to the structures of the drive */
                        wn->lcylinders = bp_cylinders(params);
                        wn->lheads = bp_heads(params);
                        wn->lsectors = bp_sectors(params);
                        wn->precomp = bp_precomp(params) >> 2;
                }

                /* Fill in non-BIOS parameters. */
                init_drive(wn,
                        drive < 2 ? REG_CMD_BASE0 : REG_CMD_BASE1,
                        drive < 2 ? REG_CTL_BASE0 : REG_CTL_BASE1,
                        NO_IRQ, 0, 0, drive);
                w_next_drive++;
        }
  }

  /* Look for controllers on the pci bus. Skip none the first instance,
   * skip one and then 2 for every instance, for every next instance.
   */
  if (w_instance == 0)
        init_params_pci(0);
  else
        init_params_pci(w_instance*2-1);

}

#define ATA_IF_NOTCOMPAT1 (1L << 0)
#define ATA_IF_NOTCOMPAT2 (1L << 2)
```

```

/*===========================================================================*
 *                              init_drive                                   *
 *===========================================================================*/
PRIVATE void init_drive(struct wini *w, int base_cmd, int base_ctl, int irq, int ack, int hook, int drive)
{
        w->state = 0;
        w->w_status = 0;
        w->base_cmd = base_cmd;
        w->base_ctl = base_ctl;
        w->irq = irq;
        w->irq_mask = 1 << irq;
        w->irq_need_ack = ack;
        w->irq_hook_id = hook;
        w->ldhpref = ldh_init(drive);
        w->max_count = MAX_SECS << SECTOR_SHIFT;
        w->lba48 = 0;
}

/*===========================================================================*
 *                              init_params_pci                              *
 *===========================================================================*/
PRIVATE void init_params_pci(int skip)
{
  int r, devind, drive;
  u16_t vid, did;
  pci_init();
  for(drive = w_next_drive; drive < MAX_DRIVES; drive++)
        wini[drive].state = IGNORING;
  for(r = pci_first_dev(&devind, &vid, &did);
        r != 0 && w_next_drive < MAX_DRIVES; r = pci_next_dev(&devind, &vid, &did)) {
        int interface, irq, irq_hook;
        /* Base class must be 01h (mass storage), subclass must
         * be 01h (ATA).
         */
        if (pci_attr_r8(devind, PCI_BCR) != 0x01 ||
           pci_attr_r8(devind, PCI_SCR) != 0x01) {
           continue;
        }
        /* Found a controller.
         * Programming interface register tells us more.
         */
        interface = pci_attr_r8(devind, PCI_PIFR);
        irq = pci_attr_r8(devind, PCI_ILR);

        /* Any non-compat drives? */
        if (interface & (ATA_IF_NOTCOMPAT1 | ATA_IF_NOTCOMPAT2)) {
                int s;
                irq_hook = irq;
                if (skip > 0) {
                        if (w_pci_debug) printf("atapci skipping controller (remain %d)\n", skip);
                        skip--;
                        continue;
                }
                if ((s=sys_irqsetpolicy(irq, 0, &irq_hook)) != OK) {
                        printf("atapci: couldn't set IRQ policy %d\n", irq);
                        continue;
                }
                if ((s=sys_irqenable(&irq_hook)) != OK) {
                        printf("atapci: couldn't enable IRQ line %d\n", irq);
                        continue;
                }
        } else {
                /* If not.. this is not the ata-pci controller we're
                 * looking for.
                 */
                if (w_pci_debug) printf("atapci skipping compatability controller\n");
                continue;
        }

        /* Primary channel not in compatability mode? */
        if (interface & ATA_IF_NOTCOMPAT1) {
                u32_t base_cmd, base_ctl;
                base_cmd = pci_attr_r32(devind, PCI_BAR) & 0xffffffe0;
                base_ctl = pci_attr_r32(devind, PCI_BAR_2) & 0xffffffe0;
                if (base_cmd != REG_CMD_BASE0 && base_cmd != REG_CMD_BASE1) {
                        init_drive(&wini[w_next_drive],
                                base_cmd, base_ctl, irq, 1, irq_hook, 0);
                        init_drive(&wini[w_next_drive+1],
                                base_cmd, base_ctl, irq, 1, irq_hook, 1);
                        if (w_pci_debug)
                                printf("atapci %d: 0x%x 0x%x irq %d\n", devind, base_cmd, base_ctl, irq);
                } else printf("atapci: ignored drives on primary channel, base %x\n", base_cmd);
        }

        /* Secondary channel not in compatability mode? */
        if (interface & ATA_IF_NOTCOMPAT2) {
                u32_t base_cmd, base_ctl;
                base_cmd = pci_attr_r32(devind, PCI_BAR_3) & 0xffffffe0;
                base_ctl = pci_attr_r32(devind, PCI_BAR_4) & 0xffffffe0;
                if (base_cmd != REG_CMD_BASE0 && base_cmd != REG_CMD_BASE1) {
                        init_drive(&wini[w_next_drive+2],
                                base_cmd, base_ctl, irq, 1, irq_hook, 2);
                        init_drive(&wini[w_next_drive+3],
                                base_cmd, base_ctl, irq, 1, irq_hook, 3);
                        if (w_pci_debug)
                                printf("atapci %d: 0x%x 0x%x irq %d\n", devind, base_cmd, base_ctl, irq);
                } else printf("atapci: ignored drives on secondary channel, base %x\n", base_cmd);
        }
        w_next_drive += 4;
  }
}

/*===========================================================================*
 *                              w_do_open                                    *
 *===========================================================================*/
PRIVATE int w_do_open(dp, m_ptr)
struct driver *dp;
message *m_ptr;
{
/* Device open: Initialize the controller and read the partition table. */

  struct wini *wn;

  if (w_prepare(m_ptr->DEVICE) == NIL_DEV) return(ENXIO);

  wn = w_wn;

  /* If we've probed it before and it failed, don't probe it again. */
  if (wn->state & IGNORING) return ENXIO;

  /* If we haven't identified it yet, or it's gone deaf,
   * (re-)identify it.
   */
  if (!(wn->state & IDENTIFIED) || (wn->state & DEAF)) {
        /* Try to identify the device. */
        if (w_identify() != OK) {
                if (wn->state & DEAF) w_reset();
                wn->state = IGNORING;
                return(ENXIO);
        }
          /* Do a test transaction unless it's a CD drive (then
           * we can believe the controller, and a test may fail
           * due to no CD being in the drive). If it fails, ignore
           * the device forever.
           */
          if (!(wn->state & ATAPI) && w_io_test() != OK) {
                wn->state |= IGNORING;
                return(ENXIO);
          }
  }

   /* If it's not an ATAPI device, then don't open with RO_BIT. */
   if (!(wn->state & ATAPI) && (m_ptr->COUNT & RO_BIT)) return EACCES;

  /* Partition the drive if it's being opened for the first time,
   * or being opened after being closed.
   */
  if (wn->open_ct == 0) {

        /* Partition the disk. */
        memset(wn->part, sizeof(wn->part), 0);
        memset(wn->subpart, sizeof(wn->subpart), 0);
        partition(&w_dtab, w_drive * DEV_PER_DRIVE, P_PRIMARY, wn->state & ATAPI);
  }
  wn->open_ct++;
  return(OK);
}

/*===========================================================================*
 *                              w_prepare                                    *
 *===========================================================================*/
PRIVATE struct device *w_prepare(int device)
{
/* Prepare for I/O on a device. */
struct wini *prev_wn;
prev_wn = w_wn;
  w_device = device;

  if (device < NR_MINORS) {                     /* d0, d0p[0-3], d1, ... */
        w_drive = device / DEV_PER_DRIVE;       /* save drive number */
        w_wn = &wini[w_drive];
        w_dv = &w_wn->part[device % DEV_PER_DRIVE];
  } else
  if ((unsigned) (device -= MINOR_d0p0s0) < NR_SUBDEVS) {/*d[0-7]p[0-3]s[0-3]*/
        w_drive = device / SUB_PER_DRIVE;
        w_wn = &wini[w_drive];
        w_dv = &w_wn->subpart[device % SUB_PER_DRIVE];
  } else {
        w_device = -1;
```

```
                return(NIL_DEV);
        }
        return(w_dv);
}

/*===========================================================================*
 *                              w_identify                                   *
 *===========================================================================*/
PRIVATE int w_identify()
{
/* Find out if a device exists, if it is an old AT disk, or a newer ATA
 * drive, a removable media device, etc.
 */

  struct wini *wn = w_wn;
  struct command cmd;
  int i, s;
  unsigned long size;
#define id_byte(n)      (&tmp_buf[2 * (n)])
#define id_word(n)      (((u16_t) id_byte(n)[0] <<  0) \
                        |((u16_t) id_byte(n)[1] <<  8))
#define id_longword(n)  (((u32_t) id_byte(n)[0] <<  0) \
                        |((u32_t) id_byte(n)[1] <<  8) \
                        |((u32_t) id_byte(n)[2] << 16) \
                        |((u32_t) id_byte(n)[3] << 24))

  /* Try to identify the device. */
  cmd.ldh     = wn->ldhpref;
  cmd.command = ATA_IDENTIFY;
  if (com_simple(&cmd) == OK) {
        /* This is an ATA device. */
        wn->state |= SMART;

        /* Device information. */
        if ((s=sys_insw(wn->base_cmd + REG_DATA, SELF, tmp_buf, SECTOR_SIZE)) != OK)
                panic(w_name(),"Call to sys_insw() failed", s);

        /* Why are the strings byte swapped??? */
        for (i = 0; i < 40; i++) w_id_string[i] = id_byte(27)[i^1];

        /* Preferred CHS translation mode. */
        wn->pcylinders = id_word(1);
        wn->pheads = id_word(3);
        wn->psectors = id_word(6);
        size = (u32_t) wn->pcylinders * wn->pheads * wn->psectors;

        if ((id_byte(49)[1] & 0x02) && size > 512L*1024*2) {
                /* Drive is LBA capable and is big enough to trust it to
                 * not make a mess of it.
                 */
                wn->ldhpref |= LDH_LBA;
                size = id_longword(60);

                if (w_lba48 && ((id_word(83)) & (1L << 10))) {
                        /* Drive is LBA48 capable (and LBA48 is turned on). */
                        if (id_word(102) || id_word(103)) {
                                /* If no. of sectors doesn't fit in 32 bits,
                                 * trunacte to this. So it's LBA32 for now.
                                 * This can still address devices up to 2TB
                                 * though.
                                 */
                                size = ULONG_MAX;
                        } else {
                                /* Actual number of sectors fits in 32 bits. */
                                size = id_longword(100);
                        }

                        wn->lba48 = 1;
                }
        }

        if (wn->lcylinders == 0) {
                /* No BIOS parameters?  Then make some up. */
                wn->lcylinders = wn->pcylinders;
                wn->lheads = wn->pheads;
                wn->lsectors = wn->psectors;
                while (wn->lcylinders > 1024) {
                        wn->lheads *= 2;
                        wn->lcylinders /= 2;
                }
        }
  } else {
        /* Not an ATA device; no translations, no special features.  Don't
         * touch it unless the BIOS knows about it.
         */
        if (wn->lcylinders == 0) { return(ERR); }       /* no BIOS parameters */
        wn->pcylinders = wn->lcylinders;
        wn->pheads = wn->lheads;
        wn->psectors = wn->lsectors;
        size = (u32_t) wn->pcylinders * wn->pheads * wn->psectors;
  }

  /* Size of the whole drive */
  wn->part[0].dv_size = mul64u(size, SECTOR_SIZE);

  /* Reset/calibrate (where necessary) */
  if (w_specify() != OK && w_specify() != OK) {
        return(ERR);
  }

  if (wn->irq == NO_IRQ) {
          /* Everything looks OK; register IRQ so we can stop polling. */
          wn->irq = w_drive < 2 ? AT_WINI_0_IRQ : AT_WINI_1_IRQ;
          wn->irq_hook_id = wn->irq;    /* id to be returned if interrupt occurs */
          if ((s=sys_irqsetpolicy(wn->irq, IRQ_REENABLE, &wn->irq_hook_id)) != OK)
                panic(w_name(), "couldn't set IRQ policy", s);
          if ((s=sys_irqenable(&wn->irq_hook_id)) != OK)
                panic(w_name(), "couldn't enable IRQ line", s);
  }
  wn->state |= IDENTIFIED;
  return(OK);
}

/*===========================================================================*
 *                              w_name                                       *
 *===========================================================================*/
PRIVATE char *w_name()
{
/* Return a name for the current device. */
  static char name[] = "AT-D0";

  name[4] = '0' + w_drive;
  return name;
}

/*===========================================================================*
 *                              w_io_test                                    *
 *===========================================================================*/
PRIVATE int w_io_test(void)
{
        int r, save_dev;
        int save_timeout, save_errors, save_wakeup;
        iovec_t iov;
        static char buf[SECTOR_SIZE];
        iov.iov_addr = (vir_bytes) buf;
        iov.iov_size = sizeof(buf);
        save_dev = w_device;

        /* Reduce timeout values for this test transaction. */
        save_timeout = timeout_ticks;
        save_errors = max_errors;
        save_wakeup = wakeup_ticks;

        if (!w_standard_timeouts) {
                timeout_ticks = HZ * 4;
                wakeup_ticks = HZ * 6;
                max_errors = 3;
        }

        w_testing = 1;

        /* Try I/O on the actual drive (not any (sub)partition). */
        if (w_prepare(w_drive * DEV_PER_DRIVE) == NIL_DEV)
                panic(w_name(), "Couldn't switch devices", NO_NUM);

        r = w_transfer(SELF, DEV_GATHER, 0, &iov, 1);

        /* Switch back. */
        if (w_prepare(save_dev) == NIL_DEV)
                panic(w_name(), "Couldn't switch back devices", NO_NUM);

        /* Restore parameters. */
        timeout_ticks = save_timeout;
        max_errors = save_errors;
        wakeup_ticks = save_wakeup;
        w_testing = 0;

        /* Test if everything worked. */
        if (r != OK || iov.iov_size != 0) {
                return ERR;
        }

        /* Everything worked. */

        return OK;
}
```

```
/*===========================================================================*
 *                              w_specify                                    *
 *===========================================================================*/
PRIVATE int w_specify()
{
/* Routine to initialize the drive after boot or when a reset is needed. */

  struct wini *wn = w_wn;
  struct command cmd;

  if ((wn->state & DEAF) && w_reset() != OK) {
        return(ERR);
  }

  if (!(wn->state & ATAPI)) {
        /* Specify parameters: precompensation, number of heads and sectors. */
        cmd.precomp = wn->precomp;
        cmd.count   = wn->psectors;
        cmd.ldh     = w_wn->ldhpref | (wn->pheads - 1);
        cmd.command = CMD_SPECIFY;              /* Specify some parameters */

        /* Output command block and see if controller accepts the parameters. */
        if (com_simple(&cmd) != OK) return(ERR);

        if (!(wn->state & SMART)) {
                /* Calibrate an old disk. */
                cmd.sector  = 0;
                cmd.cyl_lo  = 0;
                cmd.cyl_hi  = 0;
                cmd.ldh     = w_wn->ldhpref;
                cmd.command = CMD_RECALIBRATE;

                if (com_simple(&cmd) != OK) return(ERR);
        }
  }
  wn->state |= INITIALIZED;
  return(OK);
}

/*===========================================================================*
 *                              do_transfer                                  *
 *===========================================================================*/
PRIVATE int do_transfer(struct wini *wn, unsigned int precomp, unsigned int count,
        unsigned int sector, unsigned int opcode)
{
        struct command cmd;
        unsigned secspcyl = wn->pheads * wn->psectors;

        cmd.precomp = precomp;
        cmd.count   = count;
        cmd.command = opcode == DEV_SCATTER ? CMD_WRITE : CMD_READ;
        /*
        if (w_lba48 && wn->lba48) {
        } else  */
        if (wn->ldhpref & LDH_LBA) {
                cmd.sector  = (sector >>  0) & 0xFF;
                cmd.cyl_lo  = (sector >>  8) & 0xFF;
                cmd.cyl_hi  = (sector >> 16) & 0xFF;
                cmd.ldh     = wn->ldhpref | ((sector >> 24) & 0xF);
        } else {
                int cylinder, head, sec;
                cylinder = sector / secspcyl;
                head = (sector % secspcyl) / wn->psectors;
                sec = sector % wn->psectors;
                cmd.sector  = sec + 1;
                cmd.cyl_lo  = cylinder & BYTE;
                cmd.cyl_hi  = (cylinder >> 8) & BYTE;
                cmd.ldh     = wn->ldhpref | head;
        }

        return com_out(&cmd);
}

/*===========================================================================*
 *                              w_transfer                                   *
 *===========================================================================*/
PRIVATE int w_transfer(proc_nr, opcode, position, iov, nr_req)
int proc_nr;                    /* process doing the request */
int opcode;                     /* DEV_GATHER or DEV_SCATTER */
off_t position;                 /* offset on device to read or write */
iovec_t *iov;                   /* pointer to read or write request vector */
unsigned nr_req;                /* length of request vector */
{
  struct wini *wn = w_wn;
  iovec_t *iop, *iov_end = iov + nr_req;
  int r, s, errors;
  unsigned long block;
  unsigned long dv_size = cv64ul(w_dv->dv_size);
  unsigned cylinder, head, sector, nbytes;

  /* Check disk address. */
  if ((position & SECTOR_MASK) != 0) return(EINVAL);

  errors = 0;

  while (nr_req > 0) {
        /* How many bytes to transfer? */
        nbytes = 0;
        for (iop = iov; iop < iov_end; iop++) nbytes += iop->iov_size;
        if ((nbytes & SECTOR_MASK) != 0) return(EINVAL);

        /* Which block on disk and how close to EOF? */
        if (position >= dv_size) return(OK);            /* At EOF */
        if (position + nbytes > dv_size) nbytes = dv_size - position;
        block = div64u(add64ul(w_dv->dv_base, position), SECTOR_SIZE);

        if (nbytes >= wn->max_count) {
                /* The drive can't do more then max_count at once. */
                nbytes = wn->max_count;
        }

        /* First check to see if a reinitialization is needed. */
        if (!(wn->state & INITIALIZED) && w_specify() != OK) return(EIO);

        /* Tell the controller to transfer nbytes bytes. */
        r = do_transfer(wn, wn->precomp, ((nbytes >> SECTOR_SHIFT) & BYTE),
                block, opcode);

        while (r == OK && nbytes > 0) {
                /* For each sector, wait for an interrupt and fetch the data
```

```
                 * (read), or supply data to the controller and wait for an
                 * interrupt (write).
                 */

                if (opcode == DEV_GATHER) {
                        /* First an interrupt, then data. */
                        if ((r = at_intr_wait()) != OK) {
                                /* An error, send data to the bit bucket. */
                                if (w_wn->w_status & STATUS_DRQ) {
        if ((s=sys_insw(wn->base_cmd + REG_DATA, SELF, tmp_buf, SECTOR_SIZE)) != OK)
                panic(w_name(),"Call to sys_insw() failed", s);
                                }
                                break;
                        }
                }

                /* Wait for data transfer requested. */
                if (!w_waitfor(STATUS_DRQ, STATUS_DRQ)) { r = ERR; break; }

                /* Copy bytes to or from the device's buffer. */
                if (opcode == DEV_GATHER) {
        if ((s=sys_insw(wn->base_cmd + REG_DATA, proc_nr, (void *) iov->iov_addr, SECTOR_SIZE)) != OK)
                panic(w_name(),"Call to sys_insw() failed", s);
                } else {
        if ((s=sys_outsw(wn->base_cmd + REG_DATA, proc_nr, (void *) iov->iov_addr, SECTOR_SIZE)) != OK)
                panic(w_name(),"Call to sys_insw() failed", s);

                        /* Data sent, wait for an interrupt. */
                        if ((r = at_intr_wait()) != OK) break;
                }

                /* Book the bytes successfully transferred. */
                nbytes -= SECTOR_SIZE;
                position += SECTOR_SIZE;
                iov->iov_addr += SECTOR_SIZE;
                if ((iov->iov_size -= SECTOR_SIZE) == 0) { iov++; nr_req--; }
        }

        /* Any errors? */
        if (r != OK) {
                /* Don't retry if sector marked bad or too many errors. */
                if (r == ERR_BAD_SECTOR || ++errors == max_errors) {
                        w_command = CMD_IDLE;
                        return(EIO);
                }
        }
  }

  w_command = CMD_IDLE;
  return(OK);
}

/*===========================================================================*
 *                              com_out                                      *
 *===========================================================================*/
PRIVATE int com_out(cmd)
struct command *cmd;            /* Command block */
{
/* Output the command block to the winchester controller and return status */

```

```
  struct wini *wn = w_wn;
  unsigned base_cmd = wn->base_cmd;
  unsigned base_ctl = wn->base_ctl;
  pvb_pair_t outbyte[7];               /* vector for sys_voutb() */
  int s;                               /* status for sys_(v)outb() */

  if (w_wn->state & IGNORING) return ERR;

  if (!w_waitfor(STATUS_BSY, 0)) {
        printf("%s: controller not ready\n", w_name());
        return(ERR);
  }

  /* Select drive. */
  if ((s=sys_outb(base_cmd + REG_LDH, cmd->ldh)) != OK)
        panic(w_name(),"Couldn't write register to select drive",s);

  if (!w_waitfor(STATUS_BSY, 0)) {
        printf("%s: com_out: drive not ready\n", w_name());
        return(ERR);
  }

  /* Schedule a wakeup call, some controllers are flaky. This is done with
   * a synchronous alarm. If a timeout occurs a SYN_ALARM message is sent
   * from HARDWARE, so that w_intr_wait() can call w_timeout() in case the
   * controller was not able to execute the command. Leftover timeouts are
   * simply ignored by the main loop.
   */
  sys_setalarm(wakeup_ticks, 0);

  wn->w_status = STATUS_ADMBSY;
  w_command = cmd->command;
  pv_set(outbyte[0], base_ctl + REG_CTL, wn->pheads >= 8 ? CTL_EIGHTHEADS : 0);
  pv_set(outbyte[1], base_cmd + REG_PRECOMP, cmd->precomp);
  pv_set(outbyte[2], base_cmd + REG_COUNT, cmd->count);
  pv_set(outbyte[3], base_cmd + REG_SECTOR, cmd->sector);
  pv_set(outbyte[4], base_cmd + REG_CYL_LO, cmd->cyl_lo);
  pv_set(outbyte[5], base_cmd + REG_CYL_HI, cmd->cyl_hi);
  pv_set(outbyte[6], base_cmd + REG_COMMAND, cmd->command);
  if ((s=sys_voutb(outbyte,7)) != OK)
        panic(w_name(),"Couldn't write registers with sys_voutb()",s);
  return(OK);
}

/*===========================================================================*
 *                              w_need_reset                                 *
 *===========================================================================*/
PRIVATE void w_need_reset()
{
/* The controller needs to be reset. */
  struct wini *wn;
  int dr = 0;

  for (wn = wini; wn < &wini[MAX_DRIVES]; wn++, dr++) {
        if (wn->base_cmd == w_wn->base_cmd) {
                wn->state |= DEAF;
                wn->state &= ~INITIALIZED;
        }
  }
}
```

```
/*===========================================================================*
 *                              w_do_close                                   *
 *===========================================================================*/
PRIVATE int w_do_close(dp, m_ptr)
struct driver *dp;
message *m_ptr;
{
/* Device close: Release a device. */
  if (w_prepare(m_ptr->DEVICE) == NIL_DEV)
        return(ENXIO);
  w_wn->open_ct--;
  return(OK);
}

/*===========================================================================*
 *                              com_simple                                   *
 *===========================================================================*/
PRIVATE int com_simple(cmd)
struct command *cmd;            /* Command block */
{
/* A simple controller command, only one interrupt and no data-out phase. */
  int r;

  if (w_wn->state & IGNORING) return ERR;

  if ((r = com_out(cmd)) == OK) r = at_intr_wait();
  w_command = CMD_IDLE;
  return(r);
}

/*===========================================================================*
 *                              w_timeout                                    *
 *===========================================================================*/
PRIVATE void w_timeout(void)
{
  struct wini *wn = w_wn;

  switch (w_command) {
  case CMD_IDLE:
        break;          /* fine */
  case CMD_READ:
  case CMD_WRITE:
        /* Impossible, but not on PC's:  The controller does not respond. */

        /* Limiting multisector I/O seems to help. */
        if (wn->max_count > 8 * SECTOR_SIZE) {
                wn->max_count = 8 * SECTOR_SIZE;
        } else {
                wn->max_count = SECTOR_SIZE;
        }
        /*FALL THROUGH*/
  default:
        /* Some other command. */
        if (w_testing)  wn->state |= IGNORING;  /* Kick out this drive. */
        else if (!w_silent) printf("%s: timeout on command %02x\n", w_name(), w_command);
        w_need_reset();
        wn->w_status = 0;
  }
}
```

```
/*===========================================================================*
 *                              w_reset                                      *
 *===========================================================================*/
PRIVATE int w_reset()
{
/* Issue a reset to the controller.  This is done after any catastrophe,
 * like the controller refusing to respond.
 */
  int s;
  struct wini *wn = w_wn;

  /* Don't bother if this drive is forgotten. */
  if (w_wn->state & IGNORING) return ERR;

  /* Wait for any internal drive recovery. */
  tickdelay(RECOVERY_TICKS);

  /* Strobe reset bit */
  if ((s=sys_outb(wn->base_ctl + REG_CTL, CTL_RESET)) != OK)
        panic(w_name(),"Couldn't strobe reset bit",s);
  tickdelay(DELAY_TICKS);
  if ((s=sys_outb(wn->base_ctl + REG_CTL, 0)) != OK)
        panic(w_name(),"Couldn't strobe reset bit",s);
  tickdelay(DELAY_TICKS);

  /* Wait for controller ready */
  if (!w_waitfor(STATUS_BSY, 0)) {
        printf("%s: reset failed, drive busy\n", w_name());
        return(ERR);
  }

  /* The error register should be checked now, but some drives mess it up. */

  for (wn = wini; wn < &wini[MAX_DRIVES]; wn++) {
        if (wn->base_cmd == w_wn->base_cmd) {
                wn->state &= ~DEAF;
                if (w_wn->irq_need_ack) {
                        /* Make sure irq is actually enabled.. */
                        sys_irqenable(&w_wn->irq_hook_id);
                }
        }
  }


  return(OK);
}

/*===========================================================================*
 *                              w_intr_wait                                  *
 *===========================================================================*/
PRIVATE void w_intr_wait()
{
/* Wait for a task completion interrupt. */

  message m;

  if (w_wn->irq != NO_IRQ) {
        /* Wait for an interrupt that sets w_status to "not busy". */
        while (w_wn->w_status & (STATUS_ADMBSY|STATUS_BSY)) {
                receive(ANY, &m);               /* expect HARD_INT message */
                if (m.m_type == SYN_ALARM) {    /* but check for timeout */
                    w_timeout();                /* a.o. set w_status */
                } else if (m.m_type == HARD_INT) {
                    sys_inb(w_wn->base_cmd + REG_STATUS, &w_wn->w_status);
                    ack_irqs(m.NOTIFY_ARG);
                } else {
                        printf("AT_WINI got unexpected message %d from %d\n",
                                m.m_type, m.m_source);
                }
        }
  } else {
        /* Interrupt not yet allocated; use polling. */
        (void) w_waitfor(STATUS_BSY, 0);
  }
}

/*===========================================================================*
 *                              at_intr_wait                                 *
 *===========================================================================*/
PRIVATE int at_intr_wait()
{
/* Wait for an interrupt, study the status bits and return error/success. */
  int r;
  int s,inbval;         /* read value with sys_inb */

  w_intr_wait();
  if ((w_wn->w_status & (STATUS_BSY | STATUS_WF | STATUS_ERR)) == 0) {
        r = OK;
  } else {
        if ((s=sys_inb(w_wn->base_cmd + REG_ERROR, &inbval)) != OK)
                panic(w_name(),"Couldn't read register",s);
        if ((w_wn->w_status & STATUS_ERR) && (inbval & ERROR_BB)) {
                r = ERR_BAD_SECTOR;     /* sector marked bad, retries won't help */
        } else {
                r = ERR;                /* any other error */
        }
  }
  w_wn->w_status |= STATUS_ADMBSY;      /* assume still busy with I/O */
  return(r);
}

/*===========================================================================*
 *                              w_waitfor                                    *
 *===========================================================================*/
PRIVATE int w_waitfor(mask, value)
int mask;                       /* status mask */
int value;                      /* required status */
{
/* Wait until controller is in the required state.  Return zero on timeout.
 * An alarm that set a timeout flag is used. TIMEOUT is in micros, we need
 * ticks. Disabling the alarm is not needed, because a static flag is used
 * and a leftover timeout cannot do any harm.
 */
  clock_t t0, t1;
  int s;
  getuptime(&t0);
  do {
        if ((s=sys_inb(w_wn->base_cmd + REG_STATUS, &w_wn->w_status)) != OK)
                panic(w_name(),"Couldn't read register",s);
        if ((w_wn->w_status & mask) == value) {
                return 1;
        }
  } while ((s=getuptime(&t1)) == OK && (t1-t0) < timeout_ticks );
  if (OK != s) printf("AT_WINI: warning, get_uptime failed: %d\n",s);

  w_need_reset();                       /* controller gone deaf */
  return(0);
}

/*===========================================================================*
 *                              w_geometry                                   *
 *===========================================================================*/
PRIVATE void w_geometry(entry)
struct partition *entry;
{
  struct wini *wn = w_wn;

  if (wn->state & ATAPI) {              /* Make up some numbers. */
        entry->cylinders = div64u(wn->part[0].dv_size, SECTOR_SIZE) / (64*32);
        entry->heads = 64;
        entry->sectors = 32;
  } else {                              /* Return logical geometry. */
        entry->cylinders = wn->lcylinders;
        entry->heads = wn->lheads;
        entry->sectors = wn->lsectors;
  }
}

/*===========================================================================*
 *                              w_other                                      *
 *===========================================================================*/
PRIVATE int w_other(dr, m)
struct driver *dr;
message *m;
{
        int r, timeout, prev;

        if (m->m_type != DEV_IOCTL ) {
                return EINVAL;
        }

        if (m->REQUEST == DIOCTIMEOUT) {
                if ((r=sys_datacopy(m->PROC_NR, (vir_bytes)m->ADDRESS,
                        SELF, (vir_bytes)&timeout, sizeof(timeout))) != OK)
                        return r;

                if (timeout == 0) {
                        /* Restore defaults. */
                        timeout_ticks = DEF_TIMEOUT_TICKS;
                        max_errors = MAX_ERRORS;
                        wakeup_ticks = WAKEUP;
                        w_silent = 0;
                } else if (timeout < 0) {
                        return EINVAL;
                } else  {
                        prev = wakeup_ticks;

                        if (!w_standard_timeouts) {
                                /* Set (lower) timeout, lower error
```

```
                                 * tolerance and set silent mode.
                                 */
                                wakeup_ticks = timeout;
                                max_errors = 3;
                                w_silent = 1;

                                if (timeout_ticks > timeout)
                                        timeout_ticks = timeout;
                        }

                        if ((r=sys_datacopy(SELF, (vir_bytes)&prev,
                                m->PROC_NR, (vir_bytes)m->ADDRESS, sizeof(prev))) != OK)
                                return r;
                }

                return OK;
        } else  if (m->REQUEST == DIOCOPENCT) {
                int count;
                if (w_prepare(m->DEVICE) == NIL_DEV) return ENXIO;
                count = w_wn->open_ct;
                if ((r=sys_datacopy(SELF, (vir_bytes)&count,
                        m->PROC_NR, (vir_bytes)m->ADDRESS, sizeof(count))) != OK)
                        return r;
                return OK;
        }
        return EINVAL;
}

/*===========================================================================*
 *                              w_hw_int                                     *
 *===========================================================================*/
PRIVATE int w_hw_int(dr, m)
struct driver *dr;
message *m;
{
  /* Leftover interrupt(s) received; ack it/them. */
  ack_irqs(m->NOTIFY_ARG);

  return OK;
}


/*===========================================================================*
 *                              ack_irqs                                     *
 *===========================================================================*/
PRIVATE void ack_irqs(unsigned int irqs)
{
  unsigned int drive;
  for (drive = 0; drive < MAX_DRIVES && irqs; drive++) {
        if (!(wini[drive].state & IGNORING) && wini[drive].irq_need_ack &&
                (wini[drive].irq_mask & irqs)) {
                if (sys_inb((wini[drive].base_cmd + REG_STATUS), &wini[drive].w_status) != OK)
                        printf("couldn't ack irq on drive %d\n", drive);
                if (sys_irqenable(&wini[drive].irq_hook_id) != OK)
                        printf("couldn't re-enable drive %d\n", drive);
                irqs &= ~wini[drive].irq_mask;
        }
  }
}
```

```
#define STSTR(a) if (status & STATUS_ ## a) { strcat(str, #a); strcat(str, " "); }
#define ERRSTR(a) if (e & ERROR_ ## a) { strcat(str, #a); strcat(str, " "); }
char *strstatus(int status)
{
        static char str[200];
        str[0] = '\0';

        STSTR(BSY);
        STSTR(DRDY);
        STSTR(DMADF);
        STSTR(SRVCDSC);
        STSTR(DRQ);
        STSTR(CORR);
        STSTR(CHECK);
        return str;
}

char *strerr(int e)
{
        static char str[200];
        str[0] = '\0';

        ERRSTR(BB);
        ERRSTR(ECC);
        ERRSTR(ID);
        ERRSTR(AC);
        ERRSTR(TK);
        ERRSTR(DM);

        return str;
}
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                    drivers/tty/tty.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/*      tty.h - Terminals       */

#include <timers.h>

/* First minor numbers for the various classes of TTY devices. */
#define CONS_MINOR         0
#define LOG_MINOR         15
#define RS232_MINOR       16
#define TTYPX_MINOR      128
#define PTYPX_MINOR      192

#define LINEWRAP           1    /* console.c - wrap lines at column 80 */

#define TTY_IN_BYTES     256    /* tty input queue size */
#define TAB_SIZE           8    /* distance between tab stops */
#define TAB_MASK           7    /* mask to compute a tab stop position */

#define ESC             '\33'   /* escape */

#define O_NOCTTY       00400    /* from <fcntl.h>, or cc will choke */
```

```
#define O_NONBLOCK     04000

struct tty;
typedef _PROTOTYPE( int (*devfun_t), (struct tty *tp, int try_only) );
typedef _PROTOTYPE( void (*devfunarg_t), (struct tty *tp, int c) );

typedef struct tty {
  int tty_events;               /* set when TTY should inspect this line */
  int tty_index;                /* index into TTY table */
  int tty_minor;                /* device minor number */

  /* Input queue.  Typed characters are stored here until read by a program. */
  u16_t *tty_inhead;            /* pointer to place where next char goes */
  u16_t *tty_intail;            /* pointer to next char to be given to prog */
  int tty_incount;              /* # chars in the input queue */
  int tty_eotct;                /* number of "line breaks" in input queue */
  devfun_t tty_devread;         /* routine to read from low level buffers */
  devfun_t tty_icancel;         /* cancel any device input */
  int tty_min;                  /* minimum requested #chars in input queue */
  timer_t tty_tmr;              /* the timer for this tty */

  /* Output section. */
  devfun_t tty_devwrite;        /* routine to start actual device output */
  devfunarg_t tty_echo;         /* routine to echo characters input */
  devfun_t tty_ocancel;         /* cancel any ongoing device output */
  devfun_t tty_break;           /* let the device send a break */

  /* Terminal parameters and status. */
  int tty_position;             /* current position on the screen for echoing */
  char tty_reprint;             /* 1 when echoed input messed up, else 0 */
  char tty_escaped;             /* 1 when LNEXT (^V) just seen, else 0 */
  char tty_inhibited;           /* 1 when STOP (^S) just seen (stops output) */
  char tty_pgrp;                /* slot number of controlling process */
  char tty_openct;              /* count of number of opens of this tty */

  /* Information about incomplete I/O requests is stored here. */
  char tty_inrepcode;           /* reply code, TASK_REPLY or REVIVE */
  char tty_inrevived;           /* set to 1 if revive callback is pending */
  char tty_incaller;            /* process that made the call (usually FS) */
  char tty_inproc;              /* process that wants to read from tty */
  vir_bytes tty_in_vir;         /* virtual address where data is to go */
  int tty_inleft;               /* how many chars are still needed */
  int tty_incum;                /* # chars input so far */
  char tty_outrepcode;          /* reply code, TASK_REPLY or REVIVE */
  char tty_outrevived;          /* set to 1 if revive callback is pending */
  char tty_outcaller;           /* process that made the call (usually FS) */
  char tty_outproc;             /* process that wants to write to tty */
  vir_bytes tty_out_vir;        /* virtual address where data comes from */
  int tty_outleft;              /* # chars yet to be output */
  int tty_outcum;               /* # chars output so far */
  char tty_iocaller;            /* process that made the call (usually FS) */
  char tty_ioproc;              /* process that wants to do an ioctl */
  int tty_ioreq;                /* ioctl request code */
  vir_bytes tty_iovir;          /* virtual address of ioctl buffer */

  /* select() data */
  int tty_select_ops;           /* which operations are interesting */
  int tty_select_proc;          /* which process wants notification */

  /* Miscellaneous. */
  devfun_t tty_ioctl;           /* set line speed, etc. at the device level */
  devfun_t tty_close;           /* tell the device that the tty is closed */
  void *tty_priv;               /* pointer to per device private data */
  struct termios tty_termios;   /* terminal attributes */
  struct winsize tty_winsize;   /* window size (#lines and #columns) */

  u16_t tty_inbuf[TTY_IN_BYTES];/* tty input buffer */

} tty_t;

/* Memory allocated in tty.c, so extern here. */
extern tty_t tty_table[NR_CONS+NR_RS_LINES+NR_PTYS];
extern int ccurrent;            /* currently visible console */
extern int irq_hook_id;         /* hook id for keyboard irq */

extern unsigned long kbd_irq_set;
extern unsigned long rs_irq_set;

/* Values for the fields. */
#define NOT_ESCAPED        0    /* previous character is not LNEXT (^V) */
#define ESCAPED            1    /* previous character was LNEXT (^V) */
#define RUNNING            0    /* no STOP (^S) has been typed to stop output */
#define STOPPED            1    /* STOP (^S) has been typed to stop output */

/* Fields and flags on characters in the input queue. */
#define IN_CHAR       0x00FF    /* low 8 bits are the character itself */
#define IN_LEN        0x0F00    /* length of char if it has been echoed */
#define IN_LSHIFT          8    /* length = (c & IN_LEN) >> IN_LSHIFT */
#define IN_EOT        0x1000    /* char is a line break (^D, LF) */
#define IN_EOF        0x2000    /* char is EOF (^D), do not return to user */
#define IN_ESC        0x4000    /* escaped by LNEXT (^V), no interpretation */

/* Times and timeouts. */
#define force_timeout() ((void) (0))

/* Memory allocated in tty.c, so extern here. */
extern timer_t *tty_timers;             /* queue of TTY timers */
extern clock_t tty_next_timeout;        /* next TTY timeout */

/* Number of elements and limit of a buffer. */
#define buflen(buf)     (sizeof(buf) / sizeof((buf)[0]))
#define bufend(buf)     ((buf) + buflen(buf))

/* Memory allocated in tty.c, so extern here. */
extern struct machine machine;  /* machine information (a.o.: pc_at, ega) */

/* Function prototypes for TTY driver. */
/* tty.c */
_PROTOTYPE( void handle_events, (struct tty *tp)                        );
_PROTOTYPE( void sigchar, (struct tty *tp, int sig)                     );
_PROTOTYPE( void tty_task, (void)                                       );
_PROTOTYPE( int in_process, (struct tty *tp, char *buf, int count)      );
_PROTOTYPE( void out_process, (struct tty *tp, char *bstart, char *bpos,
                               char *bend, int *icount, int *ocount)   );
_PROTOTYPE( void tty_wakeup, (clock_t now)                              );
_PROTOTYPE( void tty_reply, (int code, int replyee, int proc_nr,
                                                   int status)     );
_PROTOTYPE( int tty_devnop, (struct tty *tp, int try)                   );
_PROTOTYPE( int select_try, (struct tty *tp, int ops)                   );
_PROTOTYPE( int select_retry, (struct tty *tp)                          );
```

```

/* console.c */
_PROTOTYPE( void kputc, (int c)                                         );
_PROTOTYPE( void cons_stop, (void)                                      );
_PROTOTYPE( void do_new_kmess, (message *m)                             );
_PROTOTYPE( void do_diagnostics, (message *m)                           );
_PROTOTYPE( void scr_init, (struct tty *tp)                             );
_PROTOTYPE( void toggle_scroll, (void)                                  );
_PROTOTYPE( int con_loadfont, (message *m)                              );
_PROTOTYPE( void select_console, (int cons_line)                        );

/* keyboard.c */
_PROTOTYPE( void kb_init, (struct tty *tp)                              );
_PROTOTYPE( void kb_init_once, (void)                                   );
_PROTOTYPE( int kbd_loadmap, (message *m)                               );
_PROTOTYPE( void do_panic_dumps, (message *m)                           );
_PROTOTYPE( void do_fkey_ctl, (message *m)                              );
_PROTOTYPE( void kbd_interrupt, (message *m)                            );

/* vidcopy.s */
_PROTOTYPE( void vid_vid_copy, (unsigned src, unsigned dst, unsigned count));
_PROTOTYPE( void mem_vid_copy, (u16_t *src, unsigned dst, unsigned count));
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                        drivers/tty/tty.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/* This file contains the terminal driver, both for the IBM console and regular
 * ASCII terminals.  It handles only the device-independent part of a TTY, the
 * device dependent parts are in console.c, rs232.c, etc.  This file contains
 * two main entry points, tty_task() and tty_wakeup(), and several minor entry
 * points for use by the device-dependent code.
 *
 * The device-independent part accepts "keyboard" input from the device-
 * dependent part, performs input processing (special key interpretation),
 * and sends the input to a process reading from the TTY.  Output to a TTY
 * is sent to the device-dependent code for output processing and "screen"
 * display.  Input processing is done by the device by calling 'in_process'
 * on the input characters, output processing may be done by the device itself
 * or by calling 'out_process'.  The TTY takes care of input queuing, the
 * device does the output queuing.  If a device receives an external signal,
 * like an interrupt, then it causes tty_wakeup() to be run by the CLOCK task
 * to, you guessed it, wake up the TTY to check if input or output can
 * continue.
 *
 * The valid messages and their parameters are:
 *
 *   HARD_INT:       output has been completed or input has arrived
 *   SYS_SIG:      e.g., MINIX wants to shutdown; run code to cleanly stop
 *   DEV_READ:       a process wants to read from a terminal
 *   DEV_WRITE:      a process wants to write on a terminal
 *   DEV_IOCTL:      a process wants to change a terminal's parameters
 *   DEV_OPEN:       a tty line has been opened
 *   DEV_CLOSE:      a tty line has been closed
 *   DEV_SELECT:     start select notification request
 *   DEV_STATUS:     FS wants to know status for SELECT or REVIVE
 *   CANCEL:         terminate a previous incomplete system call immediately
 *
 *    m_type      TTY_LINE   PROC_NR    COUNT   TTY_SPEK  TTY_FLAGS  ADDRESS
 * ---------------------------------------------------------------------------
 * | HARD_INT    |         |         |         |         |         |         |
 * |-------------+---------+---------+---------+---------+---------+---------|
 * | SYS_SIG     | sig set |         |         |         |         |         |
 * |-------------+---------+---------+---------+---------+---------+---------|
 * | DEV_READ    |minor dev| proc nr |  count  |         O_NONBLOCK| buf ptr |
 * |-------------+---------+---------+---------+---------+---------+---------|
 * | DEV_WRITE   |minor dev| proc nr |  count  |         |         | buf ptr |
 * |-------------+---------+---------+---------+---------+---------+---------|
 * | DEV_IOCTL   |minor dev| proc nr |func code|erase etc|  flags  |         |
 * |-------------+---------+---------+---------+---------+---------+---------|
 * | DEV_OPEN    |minor dev| proc nr | O_NOCTTY|         |         |         |
 * |-------------+---------+---------+---------+---------+---------+---------|
 * | DEV_CLOSE   |minor dev| proc nr |         |         |         |         |
 * |-------------+---------+---------+---------+---------+---------+---------|
 * | DEV_STATUS  |         |         |         |         |         |         |
 * |-------------+---------+---------+---------+---------+---------+---------|
 * | CANCEL      |minor dev| proc nr |         |         |         |         |
 * ---------------------------------------------------------------------------
 *
 * Changes:
 *   Jan 20, 2004   moved TTY driver to user-space  (Jorrit N. Herder)
 *   Sep 20, 2004   local timer management/ sync alarms  (Jorrit N. Herder)
 *   Jul 13, 2004   support for function key observers  (Jorrit N. Herder)
 */

#include "../drivers.h"
#include "../drivers.h"
#include <termios.h>
#include <sys/ioc_tty.h>
#include <signal.h>
#include <minix/callnr.h>
#include <minix/keymap.h>
#include "tty.h"

#include <sys/time.h>
#include <sys/select.h>

extern int irq_hook_id;

unsigned long kbd_irq_set = 0;
unsigned long rs_irq_set = 0;

/* Address of a tty structure. */
#define tty_addr(line)  (&tty_table[line])

/* Macros for magic tty types. */
#define isconsole(tp)   ((tp) < tty_addr(NR_CONS))
#define ispty(tp)       ((tp) >= tty_addr(NR_CONS+NR_RS_LINES))

/* Macros for magic tty structure pointers. */
#define FIRST_TTY       tty_addr(0)
#define END_TTY         tty_addr(sizeof(tty_table) / sizeof(tty_table[0]))

/* A device exists if at least its 'devread' function is defined. */
#define tty_active(tp)  ((tp)->tty_devread != NULL)

/* RS232 lines or pseudo terminals can be completely configured out. */
```

```
#if NR_RS_LINES == 0
#define rs_init(tp)     ((void) 0)
#endif
#if NR_PTYS == 0
#define pty_init(tp)    ((void) 0)
#define do_pty(tp, mp)  ((void) 0)
#endif

FORWARD _PROTOTYPE( void tty_timed_out, (timer_t *tp)                   );
FORWARD _PROTOTYPE( void expire_timers, (void)                          );
FORWARD _PROTOTYPE( void settimer, (tty_t *tty_ptr, int enable)         );
FORWARD _PROTOTYPE( void do_cancel, (tty_t *tp, message *m_ptr)         );
FORWARD _PROTOTYPE( void do_ioctl, (tty_t *tp, message *m_ptr)          );
FORWARD _PROTOTYPE( void do_open, (tty_t *tp, message *m_ptr)           );
FORWARD _PROTOTYPE( void do_close, (tty_t *tp, message *m_ptr)          );
FORWARD _PROTOTYPE( void do_read, (tty_t *tp, message *m_ptr)           );
FORWARD _PROTOTYPE( void do_write, (tty_t *tp, message *m_ptr)          );
FORWARD _PROTOTYPE( void do_select, (tty_t *tp, message *m_ptr)         );
FORWARD _PROTOTYPE( void do_status, (message *m_ptr)                    );
FORWARD _PROTOTYPE( void in_transfer, (tty_t *tp)                       );
FORWARD _PROTOTYPE( int tty_echo, (tty_t *tp, int ch)                   );
FORWARD _PROTOTYPE( void rawecho, (tty_t *tp, int ch)                   );
FORWARD _PROTOTYPE( int back_over, (tty_t *tp)                          );
FORWARD _PROTOTYPE( void reprint, (tty_t *tp)                           );
FORWARD _PROTOTYPE( void dev_ioctl, (tty_t *tp)                         );
FORWARD _PROTOTYPE( void setattr, (tty_t *tp)                           );
FORWARD _PROTOTYPE( void tty_icancel, (tty_t *tp)                       );
FORWARD _PROTOTYPE( void tty_init, (void)                               );

/* Default attributes. */
PRIVATE struct termios termios_defaults = {
  TINPUT_DEF, TOUTPUT_DEF, TCTRL_DEF, TLOCAL_DEF, TSPEED_DEF, TSPEED_DEF,
  {
        TEOF_DEF, TEOL_DEF, TERASE_DEF, TINTR_DEF, TKILL_DEF, TMIN_DEF,
        TQUIT_DEF, TTIME_DEF, TSUSP_DEF, TSTART_DEF, TSTOP_DEF,
        TREPRINT_DEF, TLNEXT_DEF, TDISCARD_DEF,
  },
};
PRIVATE struct winsize winsize_defaults;        /* = all zeroes */

/* Global variables for the TTY task (declared extern in tty.h). */
PUBLIC tty_t tty_table[NR_CONS+NR_RS_LINES+NR_PTYS];
PUBLIC int ccurrent;                    /* currently active console */
PUBLIC timer_t *tty_timers;             /* queue of TTY timers */
PUBLIC clock_t tty_next_timeout;        /* time that the next alarm is due */
PUBLIC struct machine machine;          /* kernel environment variables */

/*===========================================================================*
 *                              tty_task                                     *
 *===========================================================================*/
PUBLIC void main(void)
{
/* Main routine of the terminal task. */

  message tty_mess;             /* buffer for all incoming messages */
  unsigned line;
  int s;
  char *types[] = {"task","driver","server", "user"};
  register struct proc *rp;
  register tty_t *tp;

  /* Initialize the TTY driver. */
  tty_init();

  /* Get kernel environment (protected_mode, pc_at and ega are needed). */
  if (OK != (s=sys_getmachine(&machine))) {
    panic("TTY","Couldn’t obtain kernel environment.", s);
  }

  /* Final one-time keyboard initialization. */
  kb_init_once();

  printf("\n");

  while (TRUE) {

        /* Check for and handle any events on any of the ttys. */
        for (tp = FIRST_TTY; tp < END_TTY; tp++) {
                if (tp->tty_events) handle_events(tp);
        }

        /* Get a request message. */
        receive(ANY, &tty_mess);

        /* First handle all kernel notification types that the TTY supports.
         *  - An alarm went off, expire all timers and handle the events.
         *  - A hardware interrupt also is an invitation to check for events.
         *  - A new kernel message is available for printing.
         *  - Reset the console on system shutdown.
         * Then see if this message is different from a normal device driver
         * request and should be handled separately. These extra functions
         * do not operate on a device, in constrast to the driver requests.
         */
        switch (tty_mess.m_type) {
        case SYN_ALARM:                 /* fall through */
                expire_timers();        /* run watchdogs of expired timers */
                continue;               /* contine to check for events */
        case HARD_INT: {                /* hardware interrupt notification */
                if (tty_mess.NOTIFY_ARG & kbd_irq_set)
                        kbd_interrupt(&tty_mess);/* fetch chars from keyboard */
#if NR_RS_LINES > 0
                if (tty_mess.NOTIFY_ARG & rs_irq_set)
                        rs_interrupt(&tty_mess);/* serial I/O */
#endif
                expire_timers();        /* run watchdogs of expired timers */
                continue;               /* contine to check for events */
        }
        case SYS_SIG: {                 /* system signal */
                sigset_t sigset = (sigset_t) tty_mess.NOTIFY_ARG;

                if (sigismember(&sigset, SIGKSTOP)) {
                        cons_stop();            /* switch to primary console */
                        if (irq_hook_id != -1) {
                                sys_irqdisable(&irq_hook_id);
                                sys_irqrmpolicy(KEYBOARD_IRQ, &irq_hook_id);
                        }
                }
                if (sigismember(&sigset, SIGTERM)) cons_stop();
                if (sigismember(&sigset, SIGKMESS)) do_new_kmess(&tty_mess);
                continue;
        }
        case PANIC_DUMPS:               /* allow panic dumps */
                cons_stop();            /* switch to primary console */
                do_panic_dumps(&tty_mess);
                continue;
        case DIAGNOSTICS:               /* a server wants to print some */
                do_diagnostics(&tty_mess);
                continue;
        case FKEY_CONTROL:              /* (un)register a fkey observer */
                do_fkey_ctl(&tty_mess);
                continue;
        default:                        /* should be a driver request */
                ;                       /* do nothing; end switch */
        }

        /* Only device requests should get to this point. All requests,
         * except DEV_STATUS, have a minor device number. Check this
         * exception and get the minor device number otherwise.
         */
        if (tty_mess.m_type == DEV_STATUS) {
                do_status(&tty_mess);
                continue;
        }
        line = tty_mess.TTY_LINE;
        if ((line - CONS_MINOR) < NR_CONS) {
                tp = tty_addr(line - CONS_MINOR);
        } else if (line == LOG_MINOR) {
                tp = tty_addr(0);
        } else if ((line - RS232_MINOR) < NR_RS_LINES) {
                tp = tty_addr(line - RS232_MINOR + NR_CONS);
        } else if ((line - TTYPX_MINOR) < NR_PTYS) {
                tp = tty_addr(line - TTYPX_MINOR + NR_CONS + NR_RS_LINES);
        } else if ((line - PTYPX_MINOR) < NR_PTYS) {
                tp = tty_addr(line - PTYPX_MINOR + NR_CONS + NR_RS_LINES);
                if (tty_mess.m_type != DEV_IOCTL) {
                        do_pty(tp, &tty_mess);
                        continue;
                }
        } else {
                tp = NULL;
        }

        /* If the device doesn't exist or is not configured return ENXIO. */
        if (tp == NULL || ! tty_active(tp)) {
                printf("Warning, TTY got illegal request %d from %d\n",
                        tty_mess.m_type, tty_mess.m_source);
                tty_reply(TASK_REPLY, tty_mess.m_source,
                                                tty_mess.PROC_NR, ENXIO);
                continue;
        }

        /* Execute the requested device driver function. */
        switch (tty_mess.m_type) {
            case DEV_READ:       do_read(tp, &tty_mess);          break;
            case DEV_WRITE:      do_write(tp, &tty_mess);         break;
            case DEV_IOCTL:      do_ioctl(tp, &tty_mess);         break;
            case DEV_OPEN:       do_open(tp, &tty_mess);          break;
            case DEV_CLOSE:      do_close(tp, &tty_mess);         break;
            case DEV_SELECT:     do_select(tp, &tty_mess);        break;
            case CANCEL:         do_cancel(tp, &tty_mess);        break;
            default:
                printf("Warning, TTY got unexpected request %d from %d\n",
                        tty_mess.m_type, tty_mess.m_source);
            tty_reply(TASK_REPLY, tty_mess.m_source,
                                                tty_mess.PROC_NR, EINVAL);
        }
  }
}

/*===========================================================================*
 *                              do_status                                    *
 *===========================================================================*/
PRIVATE void do_status(m_ptr)
message *m_ptr;
{
  register struct tty *tp;
  int event_found;
  int status;
  int ops;

  /* Check for select or revive events on any of the ttys. If we found an,
   * event return a single status message for it. The FS will make another
   * call to see if there is more.
   */
  event_found = 0;
  for (tp = FIRST_TTY; tp < END_TTY; tp++) {
        if ((ops = select_try(tp, tp->tty_select_ops)) &&
                        tp->tty_select_proc == m_ptr->m_source) {

                /* I/O for a selected minor device is ready. */
                m_ptr->m_type = DEV_IO_READY;
                m_ptr->DEV_MINOR = tp->tty_index;
                m_ptr->DEV_SEL_OPS = ops;

                tp->tty_select_ops &= ~ops;     /* unmark select event */
                event_found = 1;
                break;
        }
        else if (tp->tty_inrevived && tp->tty_incaller == m_ptr->m_source) {

                /* Suspended request finished. Send a REVIVE. */
                m_ptr->m_type = DEV_REVIVE;
                m_ptr->REP_PROC_NR = tp->tty_inproc;
                m_ptr->REP_STATUS = tp->tty_incum;

                tp->tty_inleft = tp->tty_incum = 0;
                tp->tty_inrevived = 0;          /* unmark revive event */
                event_found = 1;
                break;
        }
        else if (tp->tty_outrevived && tp->tty_outcaller == m_ptr->m_source) {

                /* Suspended request finished. Send a REVIVE. */
                m_ptr->m_type = DEV_REVIVE;
                m_ptr->REP_PROC_NR = tp->tty_outproc;
                m_ptr->REP_STATUS = tp->tty_outcum;

                tp->tty_outcum = 0;
                tp->tty_outrevived = 0;         /* unmark revive event */
                event_found = 1;
                break;
        }
  }

#if NR_PTYS > 0
  if (!event_found)
        event_found = pty_status(m_ptr);
#endif

  if (! event_found) {
        /* No events of interest were found. Return an empty message. */
        m_ptr->m_type = DEV_NO_STATUS;
  }

  /* Almost done. Send back the reply message to the caller. */
  if ((status = send(m_ptr->m_source, m_ptr)) != OK) {
        panic("TTY","send in do_status failed, status\n", status);
  }
}

/*===========================================================================*
 *                              do_read                                      *
 *===========================================================================*/
PRIVATE void do_read(tp, m_ptr)
register tty_t *tp;             /* pointer to tty struct */
register message *m_ptr;        /* pointer to message sent to the task */
{
/* A process wants to read from a terminal. */
  int r, status;
  phys_bytes phys_addr;

  /* Check if there is already a process hanging in a read, check if the
   * parameters are correct, do I/O.
   */
  if (tp->tty_inleft > 0) {
        r = EIO;
  } else
  if (m_ptr->COUNT <= 0) {
        r = EINVAL;
  } else
  if (sys_umap(m_ptr->PROC_NR, D, (vir_bytes) m_ptr->ADDRESS, m_ptr->COUNT,
                &phys_addr) != OK) {
        r = EFAULT;
  } else {
        /* Copy information from the message to the tty struct. */
        tp->tty_inrepcode = TASK_REPLY;
        tp->tty_incaller = m_ptr->m_source;
        tp->tty_inproc = m_ptr->PROC_NR;
        tp->tty_in_vir = (vir_bytes) m_ptr->ADDRESS;
        tp->tty_inleft = m_ptr->COUNT;

        if (!(tp->tty_termios.c_lflag & ICANON)
                                        && tp->tty_termios.c_cc[VTIME] > 0) {
                if (tp->tty_termios.c_cc[VMIN] == 0) {
                        /* MIN & TIME specify a read timer that finishes the
                         * read in TIME/10 seconds if no bytes are available.
                         */
                        settimer(tp, TRUE);
                        tp->tty_min = 1;
                } else {
                        /* MIN & TIME specify an inter-byte timer that may
                         * have to be cancelled if there are no bytes yet.
                         */
                        if (tp->tty_eotct == 0) {
                                settimer(tp, FALSE);
                                tp->tty_min = tp->tty_termios.c_cc[VMIN];
                        }
                }
        }

        /* Anything waiting in the input buffer? Clear it out... */
        in_transfer(tp);
        /* ...then go back for more. */
        handle_events(tp);
        if (tp->tty_inleft == 0)  {
                if (tp->tty_select_ops)
                        select_retry(tp);
                return;                 /* already done */
        }

        /* There were no bytes in the input queue available, so either suspend
         * the caller or break off the read if nonblocking.
         */
        if (m_ptr->TTY_FLAGS & O_NONBLOCK) {
                r = EAGAIN;                             /* cancel the read */
                tp->tty_inleft = tp->tty_incum = 0;
        } else {
                r = SUSPEND;                            /* suspend the caller */
                tp->tty_inrepcode = REVIVE;
        }
  }
  tty_reply(TASK_REPLY, m_ptr->m_source, m_ptr->PROC_NR, r);
  if (tp->tty_select_ops)
        select_retry(tp);
}

/*===========================================================================*
 *                              do_write                                     *
 *===========================================================================*/
PRIVATE void do_write(tp, m_ptr)
register tty_t *tp;
register message *m_ptr;        /* pointer to message sent to the task */
{
/* A process wants to write on a terminal. */
  int r;
  phys_bytes phys_addr;

  /* Check if there is already a process hanging in a write, check if the
   * parameters are correct, do I/O.
   */
  if (tp->tty_outleft > 0) {
        r = EIO;
  } else
  if (m_ptr->COUNT <= 0) {
        r = EINVAL;
  } else
  if (sys_umap(m_ptr->PROC_NR, D, (vir_bytes) m_ptr->ADDRESS, m_ptr->COUNT,
                &phys_addr) != OK) {
        r = EFAULT;
  } else {
        /* Copy message parameters to the tty structure. */
        tp->tty_outrepcode = TASK_REPLY;
        tp->tty_outcaller = m_ptr->m_source;
        tp->tty_outproc = m_ptr->PROC_NR;
        tp->tty_out_vir = (vir_bytes) m_ptr->ADDRESS;
        tp->tty_outleft = m_ptr->COUNT;

        /* Try to write. */
        handle_events(tp);
        if (tp->tty_outleft == 0)
                return; /* already done */

        /* None or not all the bytes could be written, so either suspend the
         * caller or break off the write if nonblocking.
         */
        if (m_ptr->TTY_FLAGS & O_NONBLOCK) {            /* cancel the write */
                r = tp->tty_outcum > 0 ? tp->tty_outcum : EAGAIN;
                tp->tty_outleft = tp->tty_outcum = 0;
        } else {
                r = SUSPEND;                            /* suspend the caller */
                tp->tty_outrepcode = REVIVE;
        }
  }
  tty_reply(TASK_REPLY, m_ptr->m_source, m_ptr->PROC_NR, r);
}

/*===========================================================================*
 *                              do_ioctl                                     *
 *===========================================================================*/
PRIVATE void do_ioctl(tp, m_ptr)
register tty_t *tp;
message *m_ptr;                 /* pointer to message sent to task */
{
/* Perform an IOCTL on this terminal. Posix termios calls are handled
 * by the IOCTL system call
 */

  int r;
  union {
        int i;
  } param;
  size_t size;

  /* Size of the ioctl parameter. */
  switch (m_ptr->TTY_REQUEST) {
    case TCGETS:        /* Posix tcgetattr function */
    case TCSETS:        /* Posix tcsetattr function, TCSANOW option */
    case TCSETSW:       /* Posix tcsetattr function, TCSADRAIN option */
    case TCSETSF:       /* Posix tcsetattr function, TCSAFLUSH option */
        size = sizeof(struct termios);
        break;

    case TCSBRK:        /* Posix tcsendbreak function */
    case TCFLOW:        /* Posix tcflow function */
    case TCFLSH:        /* Posix tcflush function */
    case TIOCGPGRP:     /* Posix tcgetpgrp function */
    case TIOCSPGRP:     /* Posix tcsetpgrp function */
        size = sizeof(int);
        break;

```

```
      case TIOCGWINSZ:    /* get window size (not Posix) */
      case TIOCSWINSZ:    /* set window size (not Posix) */
          size = sizeof(struct winsize);
          break;

      case KIOCSMAP:      /* load keymap (Minix extension) */
          size = sizeof(keymap_t);
          break;

      case TIOCSFON:      /* load font (Minix extension) */
          size = sizeof(u8_t [8192]);
          break;

      case TCDRAIN:       /* Posix tcdrain function -- no parameter */
      default:            size = 0;
    }

    r = OK;
    switch (m_ptr->TTY_REQUEST) {
      case TCGETS:
          /* Get the termios attributes. */
          r = sys_vircopy(SELF, D, (vir_bytes) &tp->tty_termios,
                  m_ptr->PROC_NR, D, (vir_bytes) m_ptr->ADDRESS,
                  (vir_bytes) size);
          break;

      case TCSETSW:
      case TCSETSF:
      case TCDRAIN:
          if (tp->tty_outleft > 0) {
                  /* Wait for all ongoing output processing to finish. */
                  tp->tty_iocaller = m_ptr->m_source;
                  tp->tty_ioproc = m_ptr->PROC_NR;
                  tp->tty_ioreq = m_ptr->REQUEST;
                  tp->tty_iovir = (vir_bytes) m_ptr->ADDRESS;
                  r = SUSPEND;
                  break;
          }
          if (m_ptr->TTY_REQUEST == TCDRAIN) break;
          if (m_ptr->TTY_REQUEST == TCSETSF) tty_icancel(tp);
          /*FALL THROUGH*/
      case TCSETS:
          /* Set the termios attributes. */
          r = sys_vircopy( m_ptr->PROC_NR, D, (vir_bytes) m_ptr->ADDRESS,
                  SELF, D, (vir_bytes) &tp->tty_termios, (vir_bytes) size);
          if (r != OK) break;
          setattr(tp);
          break;

      case TCFLSH:
          r = sys_vircopy( m_ptr->PROC_NR, D, (vir_bytes) m_ptr->ADDRESS,
                  SELF, D, (vir_bytes) &param.i, (vir_bytes) size);
          if (r != OK) break;
          switch (param.i) {
              case TCIFLUSH:      tty_icancel(tp);                            break;
              case TCOFLUSH:      (*tp->tty_ocancel)(tp, 0);                  break;
              case TCIOFLUSH:     tty_icancel(tp); (*tp->tty_ocancel)(tp, 0); break;
              default:            r = EINVAL;
          }
          break;

      case TCFLOW:
          r = sys_vircopy( m_ptr->PROC_NR, D, (vir_bytes) m_ptr->ADDRESS,
                  SELF, D, (vir_bytes) &param.i, (vir_bytes) size);
          if (r != OK) break;
          switch (param.i) {
              case TCOOFF:
              case TCOON:
                  tp->tty_inhibited = (param.i == TCOOFF);
                  tp->tty_events = 1;
                  break;
              case TCIOFF:
                  (*tp->tty_echo)(tp, tp->tty_termios.c_cc[VSTOP]);
                  break;
              case TCION:
                  (*tp->tty_echo)(tp, tp->tty_termios.c_cc[VSTART]);
                  break;
              default:
                  r = EINVAL;
          }
          break;

      case TCSBRK:
          if (tp->tty_break != NULL) (*tp->tty_break)(tp,0);
          break;

      case TIOCGWINSZ:
          r = sys_vircopy(SELF, D, (vir_bytes) &tp->tty_winsize,
                  m_ptr->PROC_NR, D, (vir_bytes) m_ptr->ADDRESS,
                  (vir_bytes) size);
          break;

      case TIOCSWINSZ:
          r = sys_vircopy( m_ptr->PROC_NR, D, (vir_bytes) m_ptr->ADDRESS,
                  SELF, D, (vir_bytes) &tp->tty_winsize, (vir_bytes) size);
          /* SIGWINCH... */
          break;

      case KIOCSMAP:
          /* Load a new keymap (only /dev/console). */
          if (isconsole(tp)) r = kbd_loadmap(m_ptr);
          break;

      case TIOCSFON:
          /* Load a font into an EGA or VGA card (hs@hck.hr) */
          if (isconsole(tp)) r = con_loadfont(m_ptr);
          break;

  /* These Posix functions are allowed to fail if _POSIX_JOB_CONTROL is
   * not defined.
   */
      case TIOCGPGRP:
      case TIOCSPGRP:
      default:
          r = ENOTTY;
    }

    /* Send the reply. */
    tty_reply(TASK_REPLY, m_ptr->m_source, m_ptr->PROC_NR, r);
  }
```

```
/*===========================================================================*
 *                              do_open                                      *
 *===========================================================================*/
PRIVATE void do_open(tp, m_ptr)
register tty_t *tp;
message *m_ptr;                 /* pointer to message sent to task */
{
/* A tty line has been opened.  Make it the callers controlling tty if
 * O_NOCTTY is *not* set and it is not the log device.  1 is returned if
 * the tty is made the controlling tty, otherwise OK or an error code.
 */
  int r = OK;

  if (m_ptr->TTY_LINE == LOG_MINOR) {
        /* The log device is a write-only diagnostics device. */
        if (m_ptr->COUNT & R_BIT) r = EACCES;
  } else {
        if (!(m_ptr->COUNT & O_NOCTTY)) {
                tp->tty_pgrp = m_ptr->PROC_NR;
                r = 1;
        }
        tp->tty_openct++;
  }
  tty_reply(TASK_REPLY, m_ptr->m_source, m_ptr->PROC_NR, r);
}

/*===========================================================================*
 *                              do_close                                     *
 *===========================================================================*/
PRIVATE void do_close(tp, m_ptr)
register tty_t *tp;
message *m_ptr;                 /* pointer to message sent to task */
{
/* A tty line has been closed.  Clean up the line if it is the last close. */

  if (m_ptr->TTY_LINE != LOG_MINOR && --tp->tty_openct == 0) {
        tp->tty_pgrp = 0;
        tty_icancel(tp);
        (*tp->tty_ocancel)(tp, 0);
        (*tp->tty_close)(tp, 0);
        tp->tty_termios = termios_defaults;
        tp->tty_winsize = winsize_defaults;
        setattr(tp);
  }
  tty_reply(TASK_REPLY, m_ptr->m_source, m_ptr->PROC_NR, OK);
}

/*===========================================================================*
 *                              do_cancel                                    *
 *===========================================================================*/
PRIVATE void do_cancel(tp, m_ptr)
register tty_t *tp;
message *m_ptr;                 /* pointer to message sent to task */
{
/* A signal has been sent to a process that is hanging trying to read or write.
 * The pending read or write must be finished off immediately.
 */

  int proc_nr;
  int mode;

  /* Check the parameters carefully, to avoid cancelling twice. */
  proc_nr = m_ptr->PROC_NR;
  mode = m_ptr->COUNT;
  if ((mode & R_BIT) && tp->tty_inleft != 0 && proc_nr == tp->tty_inproc) {
        /* Process was reading when killed.  Clean up input. */
        tty_icancel(tp);
        tp->tty_inleft = tp->tty_incum = 0;
  }
  if ((mode & W_BIT) && tp->tty_outleft != 0 && proc_nr == tp->tty_outproc) {
        /* Process was writing when killed.  Clean up output. */
        (*tp->tty_ocancel)(tp, 0);
        tp->tty_outleft = tp->tty_outcum = 0;
  }
  if (tp->tty_ioreq != 0 && proc_nr == tp->tty_ioproc) {
        /* Process was waiting for output to drain. */
        tp->tty_ioreq = 0;
  }
  tp->tty_events = 1;
  tty_reply(TASK_REPLY, m_ptr->m_source, proc_nr, EINTR);
}

PUBLIC int select_try(struct tty *tp, int ops)
{
        int ready_ops = 0;

        /* Special case. If line is hung up, no operations will block.
         * (and it can be seen as an exceptional condition.)
         */
        if (tp->tty_termios.c_ospeed == B0) {
                ready_ops |= ops;
        }

        if (ops & SEL_RD) {
                /* will i/o not block on read? */
                if (tp->tty_inleft > 0) {
                        ready_ops |= SEL_RD;    /* EIO - no blocking */
                } else if (tp->tty_incount > 0) {
                        /* Is a regular read possible? tty_incount
                         * says there is data. But a read will only succeed
                         * in canonical mode if a newline has been seen.
                         */
                        if (!(tp->tty_termios.c_lflag & ICANON) ||
                                tp->tty_eotct > 0) {
                                ready_ops |= SEL_RD;
                        }
                }
        }

        if (ops & SEL_WR)  {
                if (tp->tty_outleft > 0)  ready_ops |= SEL_WR;
                else if ((*tp->tty_devwrite)(tp, 1)) ready_ops |= SEL_WR;
        }

        return ready_ops;
}

PUBLIC int select_retry(struct tty *tp)
{
        if (select_try(tp, tp->tty_select_ops))
                notify(tp->tty_select_proc);
        return OK;
}

/*===========================================================================*
 *                              handle_events                                *
 *===========================================================================*/
PUBLIC void handle_events(tp)
tty_t *tp;                      /* TTY to check for events. */
{
/* Handle any events pending on a TTY.  These events are usually device
 * interrupts.
 *
 * Two kinds of events are prominent:
 *      - a character has been received from the console or an RS232 line.
 *      - an RS232 line has completed a write request (on behalf of a user).
 * The interrupt handler may delay the interrupt message at its discretion
 * to avoid swamping the TTY task.  Messages may be overwritten when the
 * lines are fast or when there are races between different lines, input
 * and output, because MINIX only provides single buffering for interrupt
 * messages (in proc.c).  This is handled by explicitly checking each line
 * for fresh input and completed output on each interrupt.
 */
  char *buf;
  unsigned count;
  int status;

  do {
        tp->tty_events = 0;

        /* Read input and perform input processing. */
        (*tp->tty_devread)(tp, 0);

        /* Perform output processing and write output. */
        (*tp->tty_devwrite)(tp, 0);

        /* Ioctl waiting for some event? */
        if (tp->tty_ioreq != 0) dev_ioctl(tp);
  } while (tp->tty_events);

  /* Transfer characters from the input queue to a waiting process. */
  in_transfer(tp);

  /* Reply if enough bytes are available. */
  if (tp->tty_incum >= tp->tty_min && tp->tty_inleft > 0) {
        if (tp->tty_inrepcode == REVIVE) {
                notify(tp->tty_incaller);
                tp->tty_inrevived = 1;
        } else {
                tty_reply(tp->tty_inrepcode, tp->tty_incaller,
                        tp->tty_inproc, tp->tty_incum);
                tp->tty_inleft = tp->tty_incum = 0;
        }
  }
  if (tp->tty_select_ops)
        select_retry(tp);
#if NR_PTYS > 0
  if (ispty(tp))
        select_retry_pty(tp);
```

```
#endif
}

/*===========================================================================*
 *                              in_transfer                                  *
 *===========================================================================*/
PRIVATE void in_transfer(tp)
register tty_t *tp;             /* pointer to terminal to read from */
{
/* Transfer bytes from the input queue to a process reading from a terminal. */

  int ch;
  int count;
  char buf[64], *bp;

  /* Force read to succeed if the line is hung up, looks like EOF to reader. */
  if (tp->tty_termios.c_ospeed == B0) tp->tty_min = 0;

  /* Anything to do? */
  if (tp->tty_inleft == 0 || tp->tty_eotct < tp->tty_min) return;

  bp = buf;
  while (tp->tty_inleft > 0 && tp->tty_eotct > 0) {
        ch = *tp->tty_intail;

        if (!(ch & IN_EOF)) {
                /* One character to be delivered to the user. */
                *bp = ch & IN_CHAR;
                tp->tty_inleft--;
                if (++bp == bufend(buf)) {
                        /* Temp buffer full, copy to user space. */
                        sys_vircopy(SELF, D, (vir_bytes) buf,
                                tp->tty_inproc, D, tp->tty_in_vir,
                                (vir_bytes) buflen(buf));
                        tp->tty_in_vir += buflen(buf);
                        tp->tty_incum += buflen(buf);
                        bp = buf;
                }
        }

        /* Remove the character from the input queue. */
        if (++tp->tty_intail == bufend(tp->tty_inbuf))
                tp->tty_intail = tp->tty_inbuf;
        tp->tty_incount--;
        if (ch & IN_EOT) {
                tp->tty_eotct--;
                /* Don't read past a line break in canonical mode. */
                if (tp->tty_termios.c_lflag & ICANON) tp->tty_inleft = 0;
        }
  }

  if (bp > buf) {
        /* Leftover characters in the buffer. */
        count = bp - buf;
        sys_vircopy(SELF, D, (vir_bytes) buf,
                tp->tty_inproc, D, tp->tty_in_vir, (vir_bytes) count);
        tp->tty_in_vir += count;
        tp->tty_incum += count;
  }

  /* Usually reply to the reader, possibly even if incum == 0 (EOF). */
  if (tp->tty_inleft == 0) {
        if (tp->tty_inrepcode == REVIVE) {
                notify(tp->tty_incaller);
                tp->tty_inrevived = 1;
        } else {
                tty_reply(tp->tty_inrepcode, tp->tty_incaller,
                        tp->tty_inproc, tp->tty_incum);
                tp->tty_inleft = tp->tty_incum = 0;
        }
  }
}

/*===========================================================================*
 *                              in_process                                   *
 *===========================================================================*/
PUBLIC int in_process(tp, buf, count)
register tty_t *tp;             /* terminal on which character has arrived */
char *buf;                      /* buffer with input characters */
int count;                      /* number of input characters */
{
/* Characters have just been typed in.  Process, save, and echo them.  Return
 * the number of characters processed.
 */

  int ch, sig, ct;
  int timeset = FALSE;
  static unsigned char csize_mask[] = { 0x1F, 0x3F, 0x7F, 0xFF };

  for (ct = 0; ct < count; ct++) {
        /* Take one character. */
        ch = *buf++ & BYTE;

        /* Strip to seven bits? */
        if (tp->tty_termios.c_iflag & ISTRIP) ch &= 0x7F;

        /* Input extensions? */
        if (tp->tty_termios.c_lflag & IEXTEN) {

                /* Previous character was a character escape? */
                if (tp->tty_escaped) {
                        tp->tty_escaped = NOT_ESCAPED;
                        ch |= IN_ESC;   /* protect character */
                }

                /* LNEXT (^V) to escape the next character? */
                if (ch == tp->tty_termios.c_cc[VLNEXT]) {
                        tp->tty_escaped = ESCAPED;
                        rawecho(tp, '^');
                        rawecho(tp, '\b');
                        continue;       /* do not store the escape */
                }

                /* REPRINT (^R) to reprint echoed characters? */
                if (ch == tp->tty_termios.c_cc[VREPRINT]) {
                        reprint(tp);
                        continue;
                }
        }

        /* _POSIX_VDISABLE is a normal character value, so better escape it. */
        if (ch == _POSIX_VDISABLE) ch |= IN_ESC;

        /* Map CR to LF, ignore CR, or map LF to CR. */
        if (ch == '\r') {
                if (tp->tty_termios.c_iflag & IGNCR) continue;
                if (tp->tty_termios.c_iflag & ICRNL) ch = '\n';
        } else
        if (ch == '\n') {
                if (tp->tty_termios.c_iflag & INLCR) ch = '\r';
        }

        /* Canonical mode? */
        if (tp->tty_termios.c_lflag & ICANON) {

                /* Erase processing (rub out of last character). */
                if (ch == tp->tty_termios.c_cc[VERASE]) {
                        (void) back_over(tp);
                        if (!(tp->tty_termios.c_lflag & ECHOE)) {
                                (void) tty_echo(tp, ch);
                        }
                        continue;
                }

                /* Kill processing (remove current line). */
                if (ch == tp->tty_termios.c_cc[VKILL]) {
                        while (back_over(tp)) {}
                        if (!(tp->tty_termios.c_lflag & ECHOE)) {
                                (void) tty_echo(tp, ch);
                                if (tp->tty_termios.c_lflag & ECHOK)
                                        rawecho(tp, '\n');
                        }
                        continue;
                }

                /* EOF (^D) means end-of-file, an invisible "line break". */
                if (ch == tp->tty_termios.c_cc[VEOF]) ch |= IN_EOT | IN_EOF;

                /* The line may be returned to the user after an LF. */
                if (ch == '\n') ch |= IN_EOT;

                /* Same thing with EOL, whatever it may be. */
                if (ch == tp->tty_termios.c_cc[VEOL]) ch |= IN_EOT;
        }

        /* Start/stop input control? */
        if (tp->tty_termios.c_iflag & IXON) {

                /* Output stops on STOP (^S). */
                if (ch == tp->tty_termios.c_cc[VSTOP]) {
                        tp->tty_inhibited = STOPPED;
                        tp->tty_events = 1;
                        continue;
                }

                /* Output restarts on START (^Q) or any character if IXANY. */
                if (tp->tty_inhibited) {
                        if (ch == tp->tty_termios.c_cc[VSTART]
                                        || (tp->tty_termios.c_iflag & IXANY)) {
                                tp->tty_inhibited = RUNNING;
                                tp->tty_events = 1;
                                if (ch == tp->tty_termios.c_cc[VSTART])
                                        continue;
                        }
                }
        }

        if (tp->tty_termios.c_lflag & ISIG) {
                /* Check for INTR (^?) and QUIT (^\) characters. */
                if (ch == tp->tty_termios.c_cc[VINTR]
                                        || ch == tp->tty_termios.c_cc[VQUIT]) {
                        sig = SIGINT;
                        if (ch == tp->tty_termios.c_cc[VQUIT]) sig = SIGQUIT;
                        sigchar(tp, sig);
                        (void) tty_echo(tp, ch);
                        continue;
                }
        }

        /* Is there space in the input buffer? */
        if (tp->tty_incount == buflen(tp->tty_inbuf)) {
                /* No space; discard in canonical mode, keep in raw mode. */
                if (tp->tty_termios.c_lflag & ICANON) continue;
                break;
        }

        if (!(tp->tty_termios.c_lflag & ICANON)) {
                /* In raw mode all characters are "line breaks". */
                ch |= IN_EOT;

                /* Start an inter-byte timer? */
                if (!timeset && tp->tty_termios.c_cc[VMIN] > 0
                                && tp->tty_termios.c_cc[VTIME] > 0) {
                        settimer(tp, TRUE);
                        timeset = TRUE;
                }
        }

        /* Perform the intricate function of echoing. */
        if (tp->tty_termios.c_lflag & (ECHO|ECHONL)) ch = tty_echo(tp, ch);

        /* Save the character in the input queue. */
        *tp->tty_inhead++ = ch;
        if (tp->tty_inhead == bufend(tp->tty_inbuf))
                tp->tty_inhead = tp->tty_inbuf;
        tp->tty_incount++;
        if (ch & IN_EOT) tp->tty_eotct++;

        /* Try to finish input if the queue threatens to overflow. */
        if (tp->tty_incount == buflen(tp->tty_inbuf)) in_transfer(tp);
  }
  return ct;
}

/*===========================================================================*
 *                              echo                                         *
 *===========================================================================*/
PRIVATE int tty_echo(tp, ch)
register tty_t *tp;             /* terminal on which to echo */
register int ch;                /* pointer to character to echo */
```

```
{
/* Echo the character if echoing is on.  Some control characters are echoed
 * with their normal effect, other control characters are echoed as "^X",
 * normal characters are echoed normally.  EOF (^D) is echoed, but immediately
 * backspaced over.  Return the character with the echoed length added to its
 * attributes.
 */
  int len, rp;

  ch &= ~IN_LEN;
  if (!(tp->tty_termios.c_lflag & ECHO)) {
        if (ch == ('\n' | IN_EOT) && (tp->tty_termios.c_lflag
                                        & (ICANON|ECHONL)) == (ICANON|ECHONL))
                (*tp->tty_echo)(tp, '\n');
        return(ch);
  }

  /* "Reprint" tells if the echo output has been messed up by other output. */
  rp = tp->tty_incount == 0 ? FALSE : tp->tty_reprint;

  if ((ch & IN_CHAR) < ' ') {
        switch (ch & (IN_ESC|IN_EOF|IN_EOT|IN_CHAR)) {
            case '\t':
                len = 0;
                do {
                        (*tp->tty_echo)(tp, ' ');
                        len++;
                } while (len < TAB_SIZE && (tp->tty_position & TAB_MASK) != 0);
                break;
            case '\r' | IN_EOT:
            case '\n' | IN_EOT:
                (*tp->tty_echo)(tp, ch & IN_CHAR);
                len = 0;
                break;
            default:
                (*tp->tty_echo)(tp, '^');
                (*tp->tty_echo)(tp, '@' + (ch & IN_CHAR));
                len = 2;
        }
  } else
  if ((ch & IN_CHAR) == '\177') {
        /* A DEL prints as "^?". */
        (*tp->tty_echo)(tp, '^');
        (*tp->tty_echo)(tp, '?');
        len = 2;
  } else {
        (*tp->tty_echo)(tp, ch & IN_CHAR);
        len = 1;
  }
  if (ch & IN_EOF) while (len > 0) { (*tp->tty_echo)(tp, '\b'); len--; }

  tp->tty_reprint = rp;
  return(ch | (len << IN_LSHIFT));
}

/*===========================================================================*
 *                              rawecho                                      *
 *===========================================================================*/
PRIVATE void rawecho(tp, ch)
register tty_t *tp;
```

```
int ch;
{
/* Echo without interpretation if ECHO is set. */
  int rp = tp->tty_reprint;
  if (tp->tty_termios.c_lflag & ECHO) (*tp->tty_echo)(tp, ch);
  tp->tty_reprint = rp;
}

/*===========================================================================*
 *                              back_over                                    *
 *===========================================================================*/
PRIVATE int back_over(tp)
register tty_t *tp;
{
/* Backspace to previous character on screen and erase it. */
  u16_t *head;
  int len;

  if (tp->tty_incount == 0) return(0);  /* queue empty */
  head = tp->tty_inhead;
  if (head == tp->tty_inbuf) head = bufend(tp->tty_inbuf);
  if (*--head & IN_EOT) return(0);              /* can't erase "line breaks" */
  if (tp->tty_reprint) reprint(tp);             /* reprint if messed up */
  tp->tty_inhead = head;
  tp->tty_incount--;
  if (tp->tty_termios.c_lflag & ECHOE) {
        len = (*head & IN_LEN) >> IN_LSHIFT;
        while (len > 0) {
                rawecho(tp, '\b');
                rawecho(tp, ' ');
                rawecho(tp, '\b');
                len--;
        }
  }
  return(1);                            /* one character erased */
}

/*===========================================================================*
 *                              reprint                                      *
 *===========================================================================*/
PRIVATE void reprint(tp)
register tty_t *tp;             /* pointer to tty struct */
{
/* Restore what has been echoed to screen before if the user input has been
 * messed up by output, or if REPRINT (^R) is typed.
 */
  int count;
  u16_t *head;

  tp->tty_reprint = FALSE;

  /* Find the last line break in the input. */
  head = tp->tty_inhead;
  count = tp->tty_incount;
  while (count > 0) {
        if (head == tp->tty_inbuf) head = bufend(tp->tty_inbuf);
        if (head[-1] & IN_EOT) break;
        head--;
        count--;
  }
  if (count == tp->tty_incount) return;         /* no reason to reprint */

  /* Show REPRINT (^R) and move to a new line. */
  (void) tty_echo(tp, tp->tty_termios.c_cc[VREPRINT] | IN_ESC);
  rawecho(tp, '\r');
  rawecho(tp, '\n');

  /* Reprint from the last break onwards. */
  do {
        if (head == bufend(tp->tty_inbuf)) head = tp->tty_inbuf;
        *head = tty_echo(tp, *head);
        head++;
        count++;
  } while (count < tp->tty_incount);
}

/*===========================================================================*
 *                              out_process                                  *
 *===========================================================================*/
PUBLIC void out_process(tp, bstart, bpos, bend, icount, ocount)
tty_t *tp;
char *bstart, *bpos, *bend;     /* start/pos/end of circular buffer */
int *icount;                    /* # input chars / input chars used */
int *ocount;                    /* max output chars / output chars used */
{
/* Perform output processing on a circular buffer.  *icount is the number of
 * bytes to process, and the number of bytes actually processed on return.
 * *ocount is the space available on input and the space used on output.
 * (Naturally *icount < *ocount.)  The column position is updated modulo
 * the TAB size, because we really only need it for tabs.
 */

  int tablen;
  int ict = *icount;
  int oct = *ocount;
  int pos = tp->tty_position;

  while (ict > 0) {
        switch (*bpos) {
        case '\7':
                break;
        case '\b':
                pos--;
                break;
        case '\r':
                pos = 0;
                break;
        case '\n':
                if ((tp->tty_termios.c_oflag & (OPOST|ONLCR))
                                                        == (OPOST|ONLCR)) {
                        /* Map LF to CR+LF if there is space.  Note that the
                         * next character in the buffer is overwritten, so
                         * we stop at this point.
                         */
                        if (oct >= 2) {
                                *bpos = '\r';
                                if (++bpos == bend) bpos = bstart;
                                *bpos = '\n';
                                pos = 0;
                                ict--;
                                oct -= 2;
                        }
                        goto out_done;  /* no space or buffer got changed */
                }
                break;
        case '\t':
                /* Best guess for the tab length. */
                tablen = TAB_SIZE - (pos & TAB_MASK);

                if ((tp->tty_termios.c_oflag & (OPOST|XTABS))
                                                        == (OPOST|XTABS)) {
                        /* Tabs must be expanded. */
                        if (oct >= tablen) {
                                pos += tablen;
                                ict--;
                                oct -= tablen;
                                do {
                                        *bpos = ' ';
                                        if (++bpos == bend) bpos = bstart;
                                } while (--tablen != 0);
                        }
                        goto out_done;
                }
                /* Tabs are output directly. */
                pos += tablen;
                break;
        default:
                /* Assume any other character prints as one character. */
                pos++;
        }
        if (++bpos == bend) bpos = bstart;
        ict--;
        oct--;
  }
out_done:
  tp->tty_position = pos & TAB_MASK;

  *icount -= ict;       /* [io]ct are the number of chars not used */
  *ocount -= oct;       /* *[io]count are the number of chars that are used */
}

/*===========================================================================*
 *                              dev_ioctl                                    *
 *===========================================================================*/
PRIVATE void dev_ioctl(tp)
tty_t *tp;
{
/* The ioctl's TCSETSW, TCSETSF and TCDRAIN wait for output to finish to make
 * sure that an attribute change doesn't affect the processing of current
 * output.  Once output finishes the ioctl is executed as in do_ioctl().
 */
  int result;

  if (tp->tty_outleft > 0) return;              /* output not finished */

  if (tp->tty_ioreq != TCDRAIN) {
        if (tp->tty_ioreq == TCSETSF) tty_icancel(tp);
        result = sys_vircopy(tp->tty_ioproc, D, tp->tty_iovir,
                        SELF, D, (vir_bytes) &tp->tty_termios,
                        (vir_bytes) sizeof(tp->tty_termios));
        setattr(tp);
  }
  tp->tty_ioreq = 0;
  tty_reply(REVIVE, tp->tty_iocaller, tp->tty_ioproc, result);
}

/*===========================================================================*
 *                              setattr                                      *
 *===========================================================================*/
PRIVATE void setattr(tp)
tty_t *tp;
{
/* Apply the new line attributes (raw/canonical, line speed, etc.) */
  u16_t *inp;
  int count;

  if (!(tp->tty_termios.c_lflag & ICANON)) {
        /* Raw mode; put a "line break" on all characters in the input queue.
         * It is undefined what happens to the input queue when ICANON is
         * switched off, a process should use TCSAFLUSH to flush the queue.
         * Keeping the queue to preserve typeahead is the Right Thing, however
         * when a process does use TCSANOW to switch to raw mode.
         */
        count = tp->tty_eotct = tp->tty_incount;
        inp = tp->tty_intail;
        while (count > 0) {
                *inp |= IN_EOT;
                if (++inp == bufend(tp->tty_inbuf)) inp = tp->tty_inbuf;
                --count;
        }
  }

  /* Inspect MIN and TIME. */
  settimer(tp, FALSE);
  if (tp->tty_termios.c_lflag & ICANON) {
        /* No MIN & TIME in canonical mode. */
        tp->tty_min = 1;
  } else {
        /* In raw mode MIN is the number of chars wanted, and TIME how long
         * to wait for them.  With interesting exceptions if either is zero.
         */
        tp->tty_min = tp->tty_termios.c_cc[VMIN];
        if (tp->tty_min == 0 && tp->tty_termios.c_cc[VTIME] > 0)
                tp->tty_min = 1;
  }

  if (!(tp->tty_termios.c_iflag & IXON)) {
        /* No start/stop output control, so don't leave output inhibited. */
        tp->tty_inhibited = RUNNING;
        tp->tty_events = 1;
  }

  /* Setting the output speed to zero hangs up the phone. */
  if (tp->tty_termios.c_ospeed == B0) sigchar(tp, SIGHUP);

  /* Set new line speed, character size, etc at the device level. */
  (*tp->tty_ioctl)(tp, 0);
}
```

```
/*===========================================================================*
 *                              tty_reply                                    *
 *===========================================================================*/
PUBLIC void tty_reply(code, replyee, proc_nr, status)
int code;                       /* TASK_REPLY or REVIVE */
int replyee;                    /* destination address for the reply */
int proc_nr;                    /* to whom should the reply go? */
int status;                     /* reply code */
{
/* Send a reply to a process that wanted to read or write data. */
  message tty_mess;

  tty_mess.m_type = code;
  tty_mess.REP_PROC_NR = proc_nr;
  tty_mess.REP_STATUS = status;

  if ((status = send(replyee, &tty_mess)) != OK) {
        panic("TTY","tty_reply failed, status\n", status);
  }
}

/*===========================================================================*
 *                              sigchar                                      *
 *===========================================================================*/
PUBLIC void sigchar(tp, sig)
register tty_t *tp;
int sig;                        /* SIGINT, SIGQUIT, SIGKILL or SIGHUP */
{
/* Process a SIGINT, SIGQUIT or SIGKILL char from the keyboard or SIGHUP from
 * a tty close, "stty 0", or a real RS-232 hangup.  MM will send the signal to
 * the process group (INT, QUIT), all processes (KILL), or the session leader
 * (HUP).
 */
  int status;

  if (tp->tty_pgrp != 0)
      if (OK != (status = sys_kill(tp->tty_pgrp, sig)))
        panic("TTY","Error, call to sys_kill failed", status);

  if (!(tp->tty_termios.c_lflag & NOFLSH)) {
        tp->tty_incount = tp->tty_eotct = 0;    /* kill earlier input */
        tp->tty_intail = tp->tty_inhead;
        (*tp->tty_ocancel)(tp, 0);                      /* kill all output */
        tp->tty_inhibited = RUNNING;
        tp->tty_events = 1;
  }
}

/*===========================================================================*
 *                              tty_icancel                                  *
 *===========================================================================*/
PRIVATE void tty_icancel(tp)
register tty_t *tp;
{
/* Discard all pending input, tty buffer or device. */

  tp->tty_incount = tp->tty_eotct = 0;
  tp->tty_intail = tp->tty_inhead;
  (*tp->tty_icancel)(tp, 0);
}
```

```
/*===========================================================================*
 *                              tty_init                                     *
 *===========================================================================*/
PRIVATE void tty_init()
{
/* Initialize tty structure and call device initialization routines. */

  register tty_t *tp;
  int s;
  struct sigaction sigact;

  /* Initialize the terminal lines. */
  for (tp = FIRST_TTY,s=0; tp < END_TTY; tp++,s++) {

        tp->tty_index = s;

        tmr_inittimer(&tp->tty_tmr);

        tp->tty_intail = tp->tty_inhead = tp->tty_inbuf;
        tp->tty_min = 1;
        tp->tty_termios = termios_defaults;
        tp->tty_icancel = tp->tty_ocancel = tp->tty_ioctl = tp->tty_close =
                                                                tty_devnop;
        if (tp < tty_addr(NR_CONS)) {
                scr_init(tp);
                tp->tty_minor = CONS_MINOR + s;
        } else
        if (tp < tty_addr(NR_CONS+NR_RS_LINES)) {
                rs_init(tp);
                tp->tty_minor = RS232_MINOR + s-NR_CONS;
        } else {
                pty_init(tp);
                tp->tty_minor = s - (NR_CONS+NR_RS_LINES) + TTYPX_MINOR;
        }
  }
}

/*===========================================================================*
 *                              tty_timed_out                                *
 *===========================================================================*/
PRIVATE void tty_timed_out(timer_t *tp)
{
/* This timer has expired. Set the events flag, to force processing. */
  tty_t *tty_ptr;
  tty_ptr = &tty_table[tmr_arg(tp)->ta_int];
  tty_ptr->tty_min = 0;                 /* force read to succeed */
  tty_ptr->tty_events = 1;
}

/*===========================================================================*
 *                              expire_timers                                *
 *===========================================================================*/
PRIVATE void expire_timers(void)
{
/* A synchronous alarm message was received. Check if there are any expired
 * timers. Possibly set the event flag and reschedule another alarm.
 */
  clock_t now;                          /* current time */
  int s;

  /* Get the current time to compare the timers against. */
  if ((s=getuptime(&now)) != OK)
        panic("TTY","Couldn't get uptime from clock.", s);

  /* Scan the queue of timers for expired timers. This dispatch the watchdog
   * functions of expired timers. Possibly a new alarm call must be scheduled.
   */
  tmrs_exptimers(&tty_timers, now, NULL);
  if (tty_timers == NULL) tty_next_timeout = TMR_NEVER;
  else {                                          /* set new sync alarm */
        tty_next_timeout = tty_timers->tmr_exp_time;
        if ((s=sys_setalarm(tty_next_timeout, 1)) != OK)
                panic("TTY","Couldn't set synchronous alarm.", s);
  }
}

/*===========================================================================*
 *                              settimer                                     *
 *===========================================================================*/
PRIVATE void settimer(tty_ptr, enable)
tty_t *tty_ptr;                 /* line to set or unset a timer on */
int enable;                     /* set timer if true, otherwise unset */
{
  clock_t now;                          /* current time */
  clock_t exp_time;
  int s;

  /* Get the current time to calculate the timeout time. */
  if ((s=getuptime(&now)) != OK)
        panic("TTY","Couldn't get uptime from clock.", s);
  if (enable) {
        exp_time = now + tty_ptr->tty_termios.c_cc[VTIME] * (HZ/10);
        /* Set a new timer for enabling the TTY events flags. */
        tmrs_settimer(&tty_timers, &tty_ptr->tty_tmr,
                exp_time, tty_timed_out, NULL);
  } else {
        /* Remove the timer from the active and expired lists. */
        tmrs_clrtimer(&tty_timers, &tty_ptr->tty_tmr, NULL);
  }

  /* Now check if a new alarm must be scheduled. This happens when the front
   * of the timers queue was disabled or reinserted at another position, or
   * when a new timer was added to the front.
   */
  if (tty_timers == NULL) tty_next_timeout = TMR_NEVER;
  else if (tty_timers->tmr_exp_time != tty_next_timeout) {
        tty_next_timeout = tty_timers->tmr_exp_time;
        if ((s=sys_setalarm(tty_next_timeout, 1)) != OK)
                panic("TTY","Couldn't set synchronous alarm.", s);
  }
}

/*===========================================================================*
 *                              tty_devnop                                   *
 *===========================================================================*/
PUBLIC int tty_devnop(tp, try)
tty_t *tp;
int try;
{
  /* Some functions need not be implemented at the device level. */
}

/*===========================================================================*
 *                              do_select                                    *
 *===========================================================================*/
PRIVATE void do_select(tp, m_ptr)
register tty_t *tp;             /* pointer to tty struct */
register message *m_ptr;        /* pointer to message sent to the task */
{
        int ops, ready_ops = 0, watch;

        ops = m_ptr->PROC_NR & (SEL_RD|SEL_WR|SEL_ERR);
        watch = (m_ptr->PROC_NR & SEL_NOTIFY) ? 1 : 0;

        ready_ops = select_try(tp, ops);

        if (!ready_ops && ops && watch) {
                tp->tty_select_ops |= ops;
                tp->tty_select_proc = m_ptr->m_source;
        }

        tty_reply(TASK_REPLY, m_ptr->m_source, m_ptr->PROC_NR, ready_ops);

        return;
}
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                    drivers/tty/keyboard.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/* Keyboard driver for PC's and AT's.
 *
 * Changes:
 *   Jul 13, 2004   processes can observe function keys  (Jorrit N. Herder)
 *   Jun 15, 2004   removed wreboot(), except panic dumps (Jorrit N. Herder)
 *   Feb 04, 1994   loadable keymaps  (Marcus Hampel)
 */

#include "../drivers.h"
#include <sys/time.h>
#include <sys/select.h>
#include <termios.h>
#include <signal.h>
#include <unistd.h>
#include <minix/callnr.h>
#include <minix/com.h>
#include <minix/keymap.h>
#include "tty.h"
#include "keymaps/us-std.src"
#include "../../kernel/const.h"
#include "../../kernel/config.h"
#include "../../kernel/type.h"
#include "../../kernel/proc.h"

int irq_hook_id = -1;
```

```

/* Standard and AT keyboard.  (PS/2 MCA implies AT throughout.) */
#define KEYBD           0x60    /* I/O port for keyboard data */

/* AT keyboard. */
#define KB_COMMAND      0x64    /* I/O port for commands on AT */
#define KB_STATUS       0x64    /* I/O port for status on AT */
#define KB_ACK          0xFA    /* keyboard ack response */
#define KB_OUT_FULL     0x01    /* status bit set when keypress char pending */
#define KB_IN_FULL      0x02    /* status bit set when not ready to receive */
#define LED_CODE        0xED    /* command to keyboard to set LEDs */
#define MAX_KB_ACK_RETRIES 0x1000       /* max #times to wait for kb ack */
#define MAX_KB_BUSY_RETRIES 0x1000      /* max #times to loop while kb busy */
#define KBIT            0x80    /* bit used to ack characters to keyboard */

/* Miscellaneous. */
#define ESC_SCAN        0x01    /* reboot key when panicking */
#define SLASH_SCAN      0x35    /* to recognize numeric slash */
#define RSHIFT_SCAN     0x36    /* to distinguish left and right shift */
#define HOME_SCAN       0x47    /* first key on the numeric keypad */
#define INS_SCAN        0x52    /* INS for use in CTRL-ALT-INS reboot */
#define DEL_SCAN        0x53    /* DEL for use in CTRL-ALT-DEL reboot */

#define CONSOLE            0    /* line number for console */
#define KB_IN_BYTES       32    /* size of keyboard input buffer */
PRIVATE char ibuf[KB_IN_BYTES]; /* input buffer */
PRIVATE char *ihead = ibuf;     /* next free spot in input buffer */
PRIVATE char *itail = ibuf;     /* scan code to return to TTY */
PRIVATE int icount;             /* # codes in buffer */

PRIVATE int esc;                /* escape scan code detected? */
PRIVATE int alt_l;              /* left alt key state */
PRIVATE int alt_r;              /* right alt key state */
PRIVATE int alt;                /* either alt key */
PRIVATE int ctrl_l;             /* left control key state */
PRIVATE int ctrl_r;             /* right control key state */
PRIVATE int ctrl;               /* either control key */
PRIVATE int shift_l;            /* left shift key state */
PRIVATE int shift_r;            /* right shift key state */
PRIVATE int shift;              /* either shift key */
PRIVATE int num_down;           /* num lock key depressed */
PRIVATE int caps_down;          /* caps lock key depressed */
PRIVATE int scroll_down;        /* scroll lock key depressed */
PRIVATE int locks[NR_CONS];     /* per console lock keys state */

/* Lock key active bits.  Chosen to be equal to the keyboard LED bits. */
#define SCROLL_LOCK     0x01
#define NUM_LOCK        0x02
#define CAPS_LOCK       0x04

PRIVATE char numpad_map[] =
                {'H', 'Y', 'A', 'B', 'D', 'C', 'V', 'U', 'G', 'S', 'T', '@'};

/* Variables and definition for observed function keys. */
typedef struct observer { int proc_nr; int events; } obs_t;
PRIVATE obs_t  fkey_obs[12];    /* observers for F1-F12 */
PRIVATE obs_t sfkey_obs[12];    /* observers for SHIFT F1-F12 */

FORWARD _PROTOTYPE( int kb_ack, (void)                                  );
FORWARD _PROTOTYPE( int kb_wait, (void)                                 );
FORWARD _PROTOTYPE( int func_key, (int scode)                          );
FORWARD _PROTOTYPE( int scan_keyboard, (void)                          );
FORWARD _PROTOTYPE( unsigned make_break, (int scode)                   );
FORWARD _PROTOTYPE( void set_leds, (void)                              );
FORWARD _PROTOTYPE( void show_key_mappings, (void)                     );
FORWARD _PROTOTYPE( int kb_read, (struct tty *tp, int try)             );
FORWARD _PROTOTYPE( unsigned map_key, (int scode)                      );

/*===========================================================================*
 *                              map_key0                                     *
 *===========================================================================*/
/* Map a scan code to an ASCII code ignoring modifiers. */
#define map_key0(scode)  \
        ((unsigned) keymap[(scode) * MAP_COLS])

/*===========================================================================*
 *                              map_key                                      *
 *===========================================================================*/
PRIVATE unsigned map_key(scode)
int scode;
{
/* Map a scan code to an ASCII code. */

  int caps, column, lk;
  u16_t *keyrow;

  if (scode == SLASH_SCAN && esc) return '/';   /* don't map numeric slash */

  keyrow = &keymap[scode * MAP_COLS];

  caps = shift;
  lk = locks[ccurrent];
  if ((lk & NUM_LOCK) && HOME_SCAN <= scode && scode <= DEL_SCAN) caps = !caps;
  if ((lk & CAPS_LOCK) && (keyrow[0] & HASCAPS)) caps = !caps;

  if (alt) {
        column = 2;
        if (ctrl || alt_r) column = 3;  /* Ctrl + Alt == AltGr */
        if (caps) column = 4;
  } else {
        column = 0;
        if (caps) column = 1;
        if (ctrl) column = 5;
  }
  return keyrow[column] & ~HASCAPS;
}

/*===========================================================================*
 *                              kbd_interrupt                                *
 *===========================================================================*/
PUBLIC void kbd_interrupt(m_ptr)
message *m_ptr;
{
/* A keyboard interrupt has occurred.  Process it. */
  int scode;
  static timer_t timer;         /* timer must be static! */

  /* Fetch the character from the keyboard hardware and acknowledge it. */
  scode = scan_keyboard();

  /* Store the scancode in memory so the task can get at it later. */
  if (icount < KB_IN_BYTES) {
        *ihead++ = scode;
        if (ihead == ibuf + KB_IN_BYTES) ihead = ibuf;
        icount++;
        tty_table[ccurrent].tty_events = 1;
        if (tty_table[ccurrent].tty_select_ops & SEL_RD) {
                select_retry(&tty_table[ccurrent]);
        }
  }
}

/*===========================================================================*
 *                              kb_read                                      *
 *===========================================================================*/
PRIVATE int kb_read(tp, try)
tty_t *tp;
int try;
{
/* Process characters from the circular keyboard buffer. */
  char buf[3];
  int scode;
  unsigned ch;

  tp = &tty_table[ccurrent];            /* always use the current console */

  if (try) {
        if (icount > 0) return 1;
        return 0;
  }

  while (icount > 0) {
        scode = *itail++;                       /* take one key scan code */
        if (itail == ibuf + KB_IN_BYTES) itail = ibuf;
        icount--;

        /* Function keys are being used for debug dumps. */
        if (func_key(scode)) continue;

        /* Perform make/break processing. */
        ch = make_break(scode);

        if (ch <= 0xFF) {
                /* A normal character. */
                buf[0] = ch;
                (void) in_process(tp, buf, 1);
        } else
        if (HOME <= ch && ch <= INSRT) {
                /* An ASCII escape sequence generated by the numeric pad. */
                buf[0] = ESC;
                buf[1] = '[';
                buf[2] = numpad_map[ch - HOME];
                (void) in_process(tp, buf, 3);
        } else
        if (ch == ALEFT) {
                /* Choose lower numbered console as current console. */
                select_console(ccurrent - 1);
                set_leds();
        } else
        if (ch == ARIGHT) {
                /* Choose higher numbered console as current console. */
                select_console(ccurrent + 1);
                set_leds();
        } else
        if (AF1 <= ch && ch <= AF12) {
                /* Alt-F1 is console, Alt-F2 is ttyc1, etc. */
                select_console(ch - AF1);
                set_leds();
        } else
        if (CF1 <= ch && ch <= CF12) {
            switch(ch) {
                case CF1: show_key_mappings(); break;
                case CF3: toggle_scroll(); break; /* hardware <-> software */
                case CF7: sigchar(&tty_table[CONSOLE], SIGQUIT); break;
                case CF8: sigchar(&tty_table[CONSOLE], SIGINT); break;
                case CF9: sigchar(&tty_table[CONSOLE], SIGKILL); break;
            }
        }
  }

  return 1;
}

/*===========================================================================*
 *                              make_break                                   *
 *===========================================================================*/
PRIVATE unsigned make_break(scode)
int scode;                      /* scan code of key just struck or released */
{
/* This routine can handle keyboards that interrupt only on key depression,
 * as well as keyboards that interrupt on key depression and key release.
 * For efficiency, the interrupt routine filters out most key releases.
 */
  int ch, make, escape;
  static int CAD_count = 0;

  /* Check for CTRL-ALT-DEL, and if found, halt the computer. This would
   * be better done in keyboard() in case TTY is hung, except control and
   * alt are set in the high level code.
   */
  if (ctrl && alt && (scode == DEL_SCAN || scode == INS_SCAN))
  {
        if (++CAD_count == 3) sys_abort(RBT_HALT);
        sys_kill(INIT_PROC_NR, SIGABRT);
        return -1;
  }

  /* High-order bit set on key release. */
  make = (scode & KEY_RELEASE) == 0;            /* true if pressed */

  ch = map_key(scode &= ASCII_MASK);            /* map to ASCII */

  escape = esc;         /* Key is escaped?  (true if added since the XT) */
  esc = 0;

  switch (ch) {
        case CTRL:              /* Left or right control key */
                *(escape ? &ctrl_r : &ctrl_l) = make;
                ctrl = ctrl_l | ctrl_r;
                break;
        case SHIFT:             /* Left or right shift key */
                *(scode == RSHIFT_SCAN ? &shift_r : &shift_l) = make;
                shift = shift_l | shift_r;
                break;
        case ALT:               /* Left or right alt key */
                *(escape ? &alt_r : &alt_l) = make;
                alt = alt_l | alt_r;
                break;
        case CALOCK:            /* Caps lock - toggle on 0 -> 1 transition */
                if (caps_down < make) {
                        locks[ccurrent] ^= CAPS_LOCK;
                        set_leds();
                }
                caps_down = make;
                break;
        case NLOCK:             /* Num lock */
                if (num_down < make) {
                        locks[ccurrent] ^= NUM_LOCK;
                        set_leds();
                }
                num_down = make;
                break;
        case SLOCK:             /* Scroll lock */
                if (scroll_down < make) {
                        locks[ccurrent] ^= SCROLL_LOCK;
                        set_leds();
                }
                scroll_down = make;
                break;
        case EXTKEY:            /* Escape keycode */
                esc = 1;                /* Next key is escaped */
                return(-1);
        default:                /* A normal key */
                if (make) return(ch);
  }

  /* Key release, or a shift type key. */
  return(-1);
}

/*===========================================================================*
 *                              set_leds                                     *
 *===========================================================================*/
PRIVATE void set_leds()
{
/* Set the LEDs on the caps, num, and scroll lock keys */
  int s;
  if (! machine.pc_at) return;  /* PC/XT doesn't have LEDs */

  kb_wait();                    /* wait for buffer empty  */
  if ((s=sys_outb(KEYBD, LED_CODE)) != OK)
      printf("Warning, sys_outb couldn't prepare for LED values: %d\n", s);
                                /* prepare keyboard to accept LED values */
  kb_ack();                     /* wait for ack response  */

  kb_wait();                    /* wait for buffer empty  */
  if ((s=sys_outb(KEYBD, locks[ccurrent])) != OK)
      printf("Warning, sys_outb couldn't give LED values: %d\n", s);
                                /* give keyboard LED values */
  kb_ack();                     /* wait for ack response  */
```

```
}

/*===========================================================================*
 *                              kb_wait                                      *
 *===========================================================================*/
PRIVATE int kb_wait()
{
/* Wait until the controller is ready; return zero if this times out. */

  int retries, status, temp;
  int s;

  retries = MAX_KB_BUSY_RETRIES + 1;    /* wait until not busy */
  do {
      s = sys_inb(KB_STATUS, &status);
      if (status & KB_OUT_FULL) {
          s = sys_inb(KEYBD, &temp);    /* discard value */
      }
      if (! (status & (KB_IN_FULL|KB_OUT_FULL)) )
          break;                        /* wait until ready */
  } while (--retries != 0);             /* continue unless timeout */
  return(retries);              /* zero on timeout, positive if ready */
}

/*===========================================================================*
 *                              kb_ack                                       *
 *===========================================================================*/
PRIVATE int kb_ack()
{
/* Wait until kbd acknowledges last command; return zero if this times out. */

  int retries, s;
  u8_t u8val;

  retries = MAX_KB_ACK_RETRIES + 1;
  do {
      s = sys_inb(KEYBD, &u8val);
      if (u8val == KB_ACK)
          break;                /* wait for ack */
  } while(--retries != 0);      /* continue unless timeout */

  return(retries);              /* nonzero if ack received */
}

/*===========================================================================*
 *                              kb_init                                      *
 *===========================================================================*/
PUBLIC void kb_init(tp)
tty_t *tp;
{
/* Initialize the keyboard driver. */

  tp->tty_devread = kb_read;    /* input function */
}

/*===========================================================================*
 *                              kb_init_once                                 *
 *===========================================================================*/
PUBLIC void kb_init_once(void)
{
  int i;

  set_leds();                   /* turn off numlock led */
  scan_keyboard();              /* discard leftover keystroke */

      /* Clear the function key observers array. Also see func_key(). */
      for (i=0; i<12; i++) {
          fkey_obs[i].proc_nr = NONE;   /* F1-F12 observers */
          fkey_obs[i].events = 0;       /* F1-F12 observers */
          sfkey_obs[i].proc_nr = NONE;  /* Shift F1-F12 observers */
          sfkey_obs[i].events = 0;      /* Shift F1-F12 observers */
      }

      /* Set interrupt handler and enable keyboard IRQ. */
      irq_hook_id = KEYBOARD_IRQ;       /* id to be returned on interrupt */
      if ((i=sys_irqsetpolicy(KEYBOARD_IRQ, IRQ_REENABLE, &irq_hook_id)) != OK)
          panic("TTY",  "Couldn't set keyboard IRQ policy", i);
      if ((i=sys_irqenable(&irq_hook_id)) != OK)
          panic("TTY", "Couldn't enable keyboard IRQs", i);
      kbd_irq_set |= (1 << KEYBOARD_IRQ);
}

/*===========================================================================*
 *                              kbd_loadmap                                  *
 *===========================================================================*/
PUBLIC int kbd_loadmap(m)
message *m;
{
/* Load a new keymap. */
  int result;
  result = sys_vircopy(m->PROC_NR, D, (vir_bytes) m->ADDRESS,
        SELF, D, (vir_bytes) keymap,
        (vir_bytes) sizeof(keymap));
  return(result);
}

/*===========================================================================*
 *                              do_fkey_ctl                                  *
 *===========================================================================*/
PUBLIC void do_fkey_ctl(m_ptr)
message *m_ptr;                 /* pointer to the request message */
{
/* This procedure allows processes to register a function key to receive
 * notifications if it is pressed. At most one binding per key can exist.
 */
  int i;
  int result;

  switch (m_ptr->FKEY_REQUEST) {        /* see what we must do */
  case FKEY_MAP:                        /* request for new mapping */
      result = OK;                      /* assume everything will be ok*/
      for (i=0; i < 12; i++) {          /* check F1-F12 keys */
          if (bit_isset(m_ptr->FKEY_FKEYS, i+1) ) {
              if (fkey_obs[i].proc_nr == NONE) {
                  fkey_obs[i].proc_nr = m_ptr->m_source;
                  fkey_obs[i].events = 0;
                  bit_unset(m_ptr->FKEY_FKEYS, i+1);
              } else {
                  printf("WARNING, fkey_map failed F%d\n", i+1);
                  result = EBUSY;       /* report failure, but try rest */
```

```
                }
            }
        }
        for (i=0; i < 12; i++) {          /* check Shift+F1-F12 keys */
            if (bit_isset(m_ptr->FKEY_SFKEYS, i+1) ) {
                if (sfkey_obs[i].proc_nr == NONE) {
                    sfkey_obs[i].proc_nr = m_ptr->m_source;
                    sfkey_obs[i].events = 0;
                    bit_unset(m_ptr->FKEY_SFKEYS, i+1);
                } else {
                    printf("WARNING, fkey_map failed Shift F%d\n", i+1);
                    result = EBUSY;       /* report failure but try rest */
                }
            }
        }
        break;
    case FKEY_UNMAP:
        result = OK;                      /* assume everything will be ok*/
        for (i=0; i < 12; i++) {          /* check F1-F12 keys */
            if (bit_isset(m_ptr->FKEY_FKEYS, i+1) ) {
                if (fkey_obs[i].proc_nr == m_ptr->m_source) {
                    fkey_obs[i].proc_nr = NONE;
                    fkey_obs[i].events = 0;
                    bit_unset(m_ptr->FKEY_FKEYS, i+1);
                } else {
                    result = EPERM;       /* report failure, but try rest */
                }
            }
        }
        for (i=0; i < 12; i++) {          /* check Shift+F1-F12 keys */
            if (bit_isset(m_ptr->FKEY_SFKEYS, i+1) ) {
                if (sfkey_obs[i].proc_nr == m_ptr->m_source) {
                    sfkey_obs[i].proc_nr = NONE;
                    sfkey_obs[i].events = 0;
                    bit_unset(m_ptr->FKEY_SFKEYS, i+1);
                } else {
                    result = EPERM;       /* report failure, but try rest */
                }
            }
        }
        break;
    case FKEY_EVENTS:
        m_ptr->FKEY_FKEYS = m_ptr->FKEY_SFKEYS = 0;
        for (i=0; i < 12; i++) {          /* check (Shift+) F1-F12 keys */
            if (fkey_obs[i].proc_nr == m_ptr->m_source) {
                if (fkey_obs[i].events) {
                    bit_set(m_ptr->FKEY_FKEYS, i+1);
                    fkey_obs[i].events = 0;
                }
            }
            if (sfkey_obs[i].proc_nr == m_ptr->m_source) {
                if (sfkey_obs[i].events) {
                    bit_set(m_ptr->FKEY_SFKEYS, i+1);
                    sfkey_obs[i].events = 0;
                }
            }
        }
        break;
    default:
            result =  EINVAL;             /* key cannot be observed */
```

```
  }

  /* Almost done, return result to caller. */
  m_ptr->m_type = result;
  send(m_ptr->m_source, m_ptr);
}

/*===========================================================================*
 *                              func_key                                     *
 *===========================================================================*/
PRIVATE int func_key(scode)
int scode;                      /* scan code for a function key */
{
/* This procedure traps function keys for debugging purposes. Observers of
 * function keys are kept in a global array. If a subject (a key) is pressed
 * the observer is notified of the event. Initialization of the arrays is done
 * in kb_init, where NONE is set to indicate there is no interest in the key.
 * Returns FALSE on a key release or if the key is not observable.
 */
  message m;
  int key;
  int proc_nr;
  int i,s;

  /* Ignore key releases. If this is a key press, get full key code. */
  if (scode & KEY_RELEASE) return(FALSE);       /* key release */
  key = map_key(scode);                         /* include modifiers */

  /* Key pressed, now see if there is an observer for the pressed key.
   *          F1-F12   observers are in fkey_obs array.
   *    SHIFT F1-F12   observers are in sfkey_req array.
   *    CTRL  F1-F12   reserved (see kb_read)
   *    ALT   F1-F12   reserved (see kb_read)
   * Other combinations are not in use. Note that Alt+Shift+F1-F12 is yet
   * defined in <minix/keymap.h>, and thus is easy for future extensions.
   */
  if (F1 <= key && key <= F12) {                /* F1-F12 */
      proc_nr = fkey_obs[key - F1].proc_nr;
      fkey_obs[key - F1].events ++ ;
  } else if (SF1 <= key && key <= SF12) {       /* Shift F2-F12 */
      proc_nr = sfkey_obs[key - SF1].proc_nr;
      sfkey_obs[key - SF1].events ++;
  }
  else {
      return(FALSE);                            /* not observable */
  }

  /* See if an observer is registered and send it a message. */
  if (proc_nr != NONE) {
      m.NOTIFY_TYPE = FKEY_PRESSED;
      notify(proc_nr);
  }
  return(TRUE);
}

/*===========================================================================*
 *                              show_key_mappings                            *
 *===========================================================================*/
PRIVATE void show_key_mappings()
{
    int i,s;
    struct proc proc;

    printf("\n");
    printf("System information.   Known function key mappings to request debug dumps:\n");
    printf("-------------------------------------------------------------------------\n");
    for (i=0; i<12; i++) {

      printf(" %sF%d: ", i+1<10? " ":"", i+1);
      if (fkey_obs[i].proc_nr != NONE) {
          if ((s=sys_getproc(&proc, fkey_obs[i].proc_nr))!=OK)
              printf("sys_getproc: %d\n", s);
          printf("%-14.14s", proc.p_name);
      } else {
          printf("%-14.14s", "<none>");
      }

      printf("    %sShift-F%d: ", i+1<10? " ":"", i+1);
      if (sfkey_obs[i].proc_nr != NONE) {
          if ((s=sys_getproc(&proc, sfkey_obs[i].proc_nr))!=OK)
              printf("sys_getproc: %d\n", s);
          printf("%-14.14s", proc.p_name);
      } else {
          printf("%-14.14s", "<none>");
      }
      printf("\n");
    }
    printf("\n");
    printf("Press one of the registered function keys to trigger a debug dump.\n");
    printf("\n");
}

/*===========================================================================*
 *                              scan_keyboard                                *
 *===========================================================================*/
PRIVATE int scan_keyboard()
{
/* Fetch the character from the keyboard hardware and acknowledge it. */
  pvb_pair_t byte_in[2], byte_out[2];

  byte_in[0].port = KEYBD;      /* get the scan code for the key struck */
  byte_in[1].port = PORT_B;     /* strobe the keyboard to ack the char */
  sys_vinb(byte_in, 2);         /* request actual input */

  pv_set(byte_out[0], PORT_B, byte_in[1].value | KBIT); /* strobe bit high */
  pv_set(byte_out[1], PORT_B, byte_in[1].value);        /* then strobe low */
  sys_voutb(byte_out, 2);       /* request actual output */

  return(byte_in[0].value);             /* return scan code */
}

/*===========================================================================*
 *                              do_panic_dumps                               *
 *===========================================================================*/
PUBLIC void do_panic_dumps(m)
message *m;                     /* request message to TTY */
{
/* Wait for keystrokes for printing debugging info and reboot. */
  int quiet, code;

  /* A panic! Allow debug dumps until user wants to shutdown. */
  printf("\nHit ESC to reboot, DEL to shutdown, F-keys for debug dumps\n");

  (void) scan_keyboard();       /* ack any old input */
  quiet = scan_keyboard();/* quiescent value (0 on PC, last code on AT)*/
  for (;;) {
        tickdelay(10);
        /* See if there are pending request for output, but don't block.
         * Diagnostics can span multiple printf()s, so do it in a loop.
         */
        while (nb_receive(ANY, m) == OK) {
                switch(m->m_type) {
                case FKEY_CONTROL: do_fkey_ctl(m);      break;
                case SYS_SIG:      do_new_kmess(m);     break;
                case DIAGNOSTICS:  do_diagnostics(m);   break;
                default:        ;       /* do nothing */
                }
                tickdelay(1);           /* allow more */
        }
        code = scan_keyboard();
        if (code != quiet) {
                /* A key has been pressed. */
                switch (code) {                 /* possibly abort MINIX */
                case ESC_SCAN:  sys_abort(RBT_REBOOT);  return;
                case DEL_SCAN:  sys_abort(RBT_HALT);    return;
                }
                (void) func_key(code);          /* check for function key */
                quiet = scan_keyboard();
        }
  }
}
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                drivers/tty/console.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/* Code and data for the IBM console driver.
 *
 * The 6845 video controller used by the IBM PC shares its video memory with
 * the CPU somewhere in the 0xB0000 memory bank.  To the 6845 this memory
 * consists of 16-bit words.  Each word has a character code in the low byte
 * and a so-called attribute byte in the high byte.  The CPU directly modifies
 * video memory to display characters, and sets two registers on the 6845 that
 * specify the video origin and the cursor position.  The video origin is the
 * place in video memory where the first character (upper left corner) can
 * be found.  Moving the origin is a fast way to scroll the screen.  Some
 * video adapters wrap around the top of video memory, so the origin can
 * move without bounds.  For other adapters screen memory must sometimes be
 * moved to reset the origin.  All computations on video memory use character
 * (word) addresses for simplicity and assume there is no wrapping.  The
 * assembly support functions translate the word addresses to byte addresses
 * and the scrolling function worries about wrapping.
 */

#include "../drivers.h"
#include <termios.h>
```

```
#include <minix/callnr.h>
#include <minix/com.h>
#include "tty.h"

#include "../../kernel/const.h"
#include "../../kernel/config.h"
#include "../../kernel/type.h"

/* Definitions used by the console driver. */
#define MONO_BASE    0xB0000L   /* base of mono video memory */
#define COLOR_BASE   0xB8000L   /* base of color video memory */
#define MONO_SIZE     0x1000    /* 4K mono video memory */
#define COLOR_SIZE    0x4000    /* 16K color video memory */
#define EGA_SIZE      0x8000    /* EGA & VGA have at least 32K */
#define BLANK_COLOR   0x0700    /* determines cursor color on blank screen */
#define SCROLL_UP          0    /* scroll forward */
#define SCROLL_DOWN        1    /* scroll backward */
#define BLANK_MEM ((u16_t *) 0) /* tells mem_vid_copy() to blank the screen */
#define CONS_RAM_WORDS    80    /* video ram buffer size */
#define MAX_ESC_PARMS      4    /* number of escape sequence params allowed */

/* Constants relating to the controller chips. */
#define M_6845         0x3B4    /* port for 6845 mono */
#define C_6845         0x3D4    /* port for 6845 color */
#define INDEX              0    /* 6845's index register */
#define DATA               1    /* 6845's data register */
#define STATUS             6    /* 6845's status register */
#define VID_ORG           12    /* 6845's origin register */
#define CURSOR            14    /* 6845's cursor register */

/* Beeper. */
#define BEEP_FREQ     0x0533    /* value to put into timer to set beep freq */
#define B_TIME             3    /* length of CTRL-G beep is ticks */

/* definitions used for font management */
#define GA_SEQUENCER_INDEX      0x3C4
#define GA_SEQUENCER_DATA       0x3C5
#define GA_GRAPHICS_INDEX       0x3CE
#define GA_GRAPHICS_DATA        0x3CF
#define GA_VIDEO_ADDRESS        0xA0000L
#define GA_FONT_SIZE            8192

/* Global variables used by the console driver and assembly support. */
PUBLIC int vid_index;           /* index of video segment in remote mem map */
PUBLIC u16_t vid_seg;
PUBLIC vir_bytes vid_off;       /* video ram is found at vid_seg:vid_off */
PUBLIC unsigned vid_size;       /* 0x2000 for color or 0x0800 for mono */
PUBLIC unsigned vid_mask;       /* 0x1FFF for color or 0x07FF for mono */
PUBLIC unsigned blank_color = BLANK_COLOR; /* display code for blank */

/* Private variables used by the console driver. */
PRIVATE int vid_port;           /* I/O port for accessing 6845 */
PRIVATE int wrap;               /* hardware can wrap? */
PRIVATE int softscroll;         /* 1 = software scrolling, 0 = hardware */
PRIVATE int beeping;            /* speaker is beeping? */
PRIVATE unsigned font_lines;    /* font lines per character */
PRIVATE unsigned scr_width;     /* # characters on a line */
PRIVATE unsigned scr_lines;     /* # lines on the screen */
PRIVATE unsigned scr_size;      /* # characters on the screen */

```

```
/* Per console data. */
typedef struct console {
  tty_t *c_tty;                 /* associated TTY struct */
  int c_column;                 /* current column number (0-origin) */
  int c_row;                    /* current row (0 at top of screen) */
  int c_rwords;                 /* number of WORDS (not bytes) in outqueue */
  unsigned c_start;             /* start of video memory of this console */
  unsigned c_limit;             /* limit of this console's video memory */
  unsigned c_org;               /* location in RAM where 6845 base points */
  unsigned c_cur;               /* current position of cursor in video RAM */
  unsigned c_attr;              /* character attribute */
  unsigned c_blank;             /* blank attribute */
  char c_reverse;               /* reverse video */
  char c_esc_state;             /* 0=normal, 1=ESC, 2=ESC[ */
  char c_esc_intro;             /* Distinguishing character following ESC */
  int *c_esc_parmp;             /* pointer to current escape parameter */
  int c_esc_parmv[MAX_ESC_PARMS];       /* list of escape parameters */
  u16_t c_ramqueue[CONS_RAM_WORDS];     /* buffer for video RAM */
} console_t;

PRIVATE int nr_cons= 1;         /* actual number of consoles */
PRIVATE console_t cons_table[NR_CONS];
PRIVATE console_t *curcons;     /* currently visible */

/* Color if using a color controller. */
#define color   (vid_port == C_6845)

/* Map from ANSI colors to the attributes used by the PC */
PRIVATE int ansi_colors[8] = {0, 4, 2, 6, 1, 5, 3, 7};

/* Structure used for font management */
struct sequence {
        unsigned short index;
        unsigned char port;
        unsigned char value;
};

FORWARD _PROTOTYPE( int cons_write, (struct tty *tp, int try)           );
FORWARD _PROTOTYPE( void cons_echo, (tty_t *tp, int c)                  );
FORWARD _PROTOTYPE( void out_char, (console_t *cons, int c)             );
FORWARD _PROTOTYPE( void putk, (int c)                                  );
FORWARD _PROTOTYPE( void beep, (void)                                   );
FORWARD _PROTOTYPE( void do_escape, (console_t *cons, int c)            );
FORWARD _PROTOTYPE( void flush, (console_t *cons)                       );
FORWARD _PROTOTYPE( void parse_escape, (console_t *cons, int c)         );
FORWARD _PROTOTYPE( void scroll_screen, (console_t *cons, int dir)      );
FORWARD _PROTOTYPE( void set_6845, (int reg, unsigned val)              );
FORWARD _PROTOTYPE( void get_6845, (int reg, unsigned *val)             );
FORWARD _PROTOTYPE( void stop_beep, (timer_t *tmrp)                     );
FORWARD _PROTOTYPE( void cons_org0, (void)                              );
FORWARD _PROTOTYPE( int ga_program, (struct sequence *seq)              );
FORWARD _PROTOTYPE( int cons_ioctl, (tty_t *tp, int)                    );

/*===========================================================================*
 *                              cons_write                                   *
 *===========================================================================*/
PRIVATE int cons_write(tp, try)
register struct tty *tp;        /* tells which terminal is to be used */
int try;
{
```

```
/* Copy as much data as possible to the output queue, then start I/O.  On
 * memory-mapped terminals, such as the IBM console, the I/O will also be
 * finished, and the counts updated.  Keep repeating until all I/O done.
 */

  int count;
  int result;
  register char *tbuf;
  char buf[64];
  console_t *cons = tp->tty_priv;

  if (try) return 1;    /* we can always write to console */

  /* Check quickly for nothing to do, so this can be called often without
   * unmodular tests elsewhere.
   */
  if ((count = tp->tty_outleft) == 0 || tp->tty_inhibited) return;

  /* Copy the user bytes to buf[] for decent addressing. Loop over the
   * copies, since the user buffer may be much larger than buf[].
   */
  do {
        if (count > sizeof(buf)) count = sizeof(buf);
        if ((result = sys_vircopy(tp->tty_outproc, D, tp->tty_out_vir,
                        SELF, D, (vir_bytes) buf, (vir_bytes) count)) != OK)
                break;
        tbuf = buf;

        /* Update terminal data structure. */
        tp->tty_out_vir += count;
        tp->tty_outcum += count;
        tp->tty_outleft -= count;

        /* Output each byte of the copy to the screen.  Avoid calling
         * out_char() for the "easy" characters, put them into the buffer
         * directly.
         */
        do {
                if ((unsigned) *tbuf < ' ' || cons->c_esc_state > 0
                        || cons->c_column >= scr_width
                        || cons->c_rwords >= buflen(cons->c_ramqueue))
                {
                        out_char(cons, *tbuf++);
                } else {
                        cons->c_ramqueue[cons->c_rwords++] =
                                        cons->c_attr | (*tbuf++ & BYTE);
                        cons->c_column++;
                }
        } while (--count != 0);
  } while ((count = tp->tty_outleft) != 0 && !tp->tty_inhibited);

  flush(cons);                  /* transfer anything buffered to the screen */

  /* Reply to the writer if all output is finished or if an error occured. */
  if (tp->tty_outleft == 0 || result != OK) {
        /* REVIVE is not possible. I/O on memory mapped consoles finishes. */
        tty_reply(tp->tty_outrepcode, tp->tty_outcaller, tp->tty_outproc,
                                                        tp->tty_outcum);
        tp->tty_outcum = 0;
  }
```

```
}

/*===========================================================================*
 *                              cons_echo                                    *
 *===========================================================================*/
PRIVATE void cons_echo(tp, c)
register tty_t *tp;             /* pointer to tty struct */
int c;                          /* character to be echoed */
{
/* Echo keyboard input (print & flush). */
  console_t *cons = tp->tty_priv;

  out_char(cons, c);
  flush(cons);
}

/*===========================================================================*
 *                              out_char                                     *
 *===========================================================================*/
PRIVATE void out_char(cons, c)
register console_t *cons;       /* pointer to console struct */
int c;                          /* character to be output */
{
/* Output a character on the console.  Check for escape sequences first. */
  if (cons->c_esc_state > 0) {
        parse_escape(cons, c);
        return;
  }

  switch(c) {
        case 000:               /* null is typically used for padding */
                return;         /* better not do anything */

        case 007:               /* ring the bell */
                flush(cons);    /* print any chars queued for output */
                beep();
                return;

        case '\b':              /* backspace */
                if (--cons->c_column < 0) {
                        if (--cons->c_row >= 0) cons->c_column += scr_width;
                }
                flush(cons);
                return;

        case '\n':              /* line feed */
                if ((cons->c_tty->tty_termios.c_oflag & (OPOST|ONLCR))
                                                == (OPOST|ONLCR)) {
                        cons->c_column = 0;
                }
                /*FALL THROUGH*/
        case 013:               /* CTRL-K */
        case 014:               /* CTRL-L */
                if (cons->c_row == scr_lines-1) {
                        scroll_screen(cons, SCROLL_UP);
                } else {
                        cons->c_row++;
                }
                flush(cons);
                return;

        case '\r':              /* carriage return */
                cons->c_column = 0;
                flush(cons);
                return;

        case '\t':              /* tab */
                cons->c_column = (cons->c_column + TAB_SIZE) & ~TAB_MASK;
                if (cons->c_column > scr_width) {
                        cons->c_column -= scr_width;
                        if (cons->c_row == scr_lines-1) {
                                scroll_screen(cons, SCROLL_UP);
                        } else {
                                cons->c_row++;
                        }
                }
                flush(cons);
                return;

        case 033:               /* ESC - start of an escape sequence */
                flush(cons);    /* print any chars queued for output */
                cons->c_esc_state = 1;  /* mark ESC as seen */
                return;

        default:                /* printable chars are stored in ramqueue */
                if (cons->c_column >= scr_width) {
                        if (!LINEWRAP) return;
                        if (cons->c_row == scr_lines-1) {
                                scroll_screen(cons, SCROLL_UP);
                        } else {
                                cons->c_row++;
                        }
                        cons->c_column = 0;
                        flush(cons);
                }
                if (cons->c_rwords == buflen(cons->c_ramqueue)) flush(cons);
                cons->c_ramqueue[cons->c_rwords++] = cons->c_attr | (c & BYTE);
                cons->c_column++;                       /* next column */
                return;
  }
}

/*===========================================================================*
 *                              scroll_screen                                *
 *===========================================================================*/
PRIVATE void scroll_screen(cons, dir)
register console_t *cons;       /* pointer to console struct */
int dir;                        /* SCROLL_UP or SCROLL_DOWN */
{
  unsigned new_line, new_org, chars;

  flush(cons);
  chars = scr_size - scr_width;         /* one screen minus one line */

  /* Scrolling the screen is a real nuisance due to the various incompatible
   * video cards.  This driver supports software scrolling (Hercules?),
   * hardware scrolling (mono and CGA cards) and hardware scrolling without
   * wrapping (EGA cards).  In the latter case we must make sure that
   *            c_start <= c_org && c_org + scr_size <= c_limit
   * holds, because EGA doesn't wrap around the end of video memory.
   */
  if (dir == SCROLL_UP) {
        /* Scroll one line up in 3 ways: soft, avoid wrap, use origin. */
        if (softscroll) {
                vid_vid_copy(cons->c_start + scr_width, cons->c_start, chars);
        } else
        if (!wrap && cons->c_org + scr_size + scr_width >= cons->c_limit) {
                vid_vid_copy(cons->c_org + scr_width, cons->c_start, chars);
                cons->c_org = cons->c_start;
        } else {
                cons->c_org = (cons->c_org + scr_width) & vid_mask;
        }
        new_line = (cons->c_org + chars) & vid_mask;
  } else {
        /* Scroll one line down in 3 ways: soft, avoid wrap, use origin. */
        if (softscroll) {
                vid_vid_copy(cons->c_start, cons->c_start + scr_width, chars);
        } else
        if (!wrap && cons->c_org < cons->c_start + scr_width) {
                new_org = cons->c_limit - scr_size;
                vid_vid_copy(cons->c_org, new_org + scr_width, chars);
                cons->c_org = new_org;
        } else {
                cons->c_org = (cons->c_org - scr_width) & vid_mask;
        }
        new_line = cons->c_org;
  }
  /* Blank the new line at top or bottom. */
  blank_color = cons->c_blank;
  mem_vid_copy(BLANK_MEM, new_line, scr_width);

  /* Set the new video origin. */
  if (cons == curcons) set_6845(VID_ORG, cons->c_org);
  flush(cons);
}

/*===========================================================================*
 *                              flush                                        *
 *===========================================================================*/
PRIVATE void flush(cons)
register console_t *cons;       /* pointer to console struct */
{
/* Send characters buffered in 'ramqueue' to screen memory, check the new
 * cursor position, compute the new hardware cursor position and set it.
 */
  unsigned cur;
  tty_t *tp = cons->c_tty;

  /* Have the characters in 'ramqueue' transferred to the screen. */
  if (cons->c_rwords > 0) {
        mem_vid_copy(cons->c_ramqueue, cons->c_cur, cons->c_rwords);
        cons->c_rwords = 0;

        /* TTY likes to know the current column and if echoing messed up. */
        tp->tty_position = cons->c_column;
        tp->tty_reprint = TRUE;
  }

  /* Check and update the cursor position. */
  if (cons->c_column < 0) cons->c_column = 0;
  if (cons->c_column > scr_width) cons->c_column = scr_width;
  if (cons->c_row < 0) cons->c_row = 0;
  if (cons->c_row >= scr_lines) cons->c_row = scr_lines - 1;
  cur = cons->c_org + cons->c_row * scr_width + cons->c_column;
  if (cur != cons->c_cur) {
        if (cons == curcons) set_6845(CURSOR, cur);
        cons->c_cur = cur;
  }
}

/*===========================================================================*
 *                              parse_escape                                 *
 *===========================================================================*/
PRIVATE void parse_escape(cons, c)
register console_t *cons;       /* pointer to console struct */
char c;                         /* next character in escape sequence */
{
/* The following ANSI escape sequences are currently supported.
 * If n and/or m are omitted, they default to 1.
 *   ESC [nA moves up n lines
 *   ESC [nB moves down n lines
 *   ESC [nC moves right n spaces
 *   ESC [nD moves left n spaces
 *   ESC [m;nH" moves cursor to (m,n)
 *   ESC [J clears screen from cursor
 *   ESC [K clears line from cursor
 *   ESC [nL inserts n lines ar cursor
 *   ESC [nM deletes n lines at cursor
 *   ESC [nP deletes n chars at cursor
 *   ESC [n@ inserts n chars at cursor
 *   ESC [nm enables rendition n (0=normal, 4=bold, 5=blinking, 7=reverse)
 *   ESC M scrolls the screen backwards if the cursor is on the top line
 */

  switch (cons->c_esc_state) {
    case 1:                     /* ESC seen */
        cons->c_esc_intro = '\0';
        cons->c_esc_parmp = bufend(cons->c_esc_parmv);
        do {
                *--cons->c_esc_parmp = 0;
        } while (cons->c_esc_parmp > cons->c_esc_parmv);
        switch (c) {
            case '[':   /* Control Sequence Introducer */
                cons->c_esc_intro = c;
                cons->c_esc_state = 2;
                break;
            case 'M':   /* Reverse Index */
                do_escape(cons, c);
                break;
            default:
                cons->c_esc_state = 0;
        }
        break;

    case 2:                     /* ESC [ seen */
        if (c >= '0' && c <= '9') {
                if (cons->c_esc_parmp < bufend(cons->c_esc_parmv))
                        *cons->c_esc_parmp = *cons->c_esc_parmp * 10 + (c-'0');
        } else
        if (c == ';') {
                if (cons->c_esc_parmp < bufend(cons->c_esc_parmv))
                        cons->c_esc_parmp++;
        } else {
                do_escape(cons, c);
        }
        break;
  }
}

/*===========================================================================*
 *                              do_escape                                    *
 *===========================================================================*/
PRIVATE void do_escape(cons, c)
register console_t *cons;       /* pointer to console struct */
char c;                         /* next character in escape sequence */
{
  int value, n;
  unsigned src, dst, count;
  int *parmp;

  /* Some of these things hack on screen RAM, so it had better be up to date */
  flush(cons);

  if (cons->c_esc_intro == '\0') {
        /* Handle a sequence beginning with just ESC */
        switch (c) {
            case 'M':           /* Reverse Index */
                if (cons->c_row == 0) {
                        scroll_screen(cons, SCROLL_DOWN);
                } else {
                        cons->c_row--;
                }
                flush(cons);
                break;

            default: break;
        }
  } else
  if (cons->c_esc_intro == '[') {
        /* Handle a sequence beginning with ESC [ and parameters */
        value = cons->c_esc_parmv[0];
        switch (c) {
            case 'A':           /* ESC [nA moves up n lines */
                n = (value == 0 ? 1 : value);
                cons->c_row -= n;
                flush(cons);
                break;

            case 'B':           /* ESC [nB moves down n lines */
                n = (value == 0 ? 1 : value);
                cons->c_row += n;
                flush(cons);
                break;

            case 'C':           /* ESC [nC moves right n spaces */
                n = (value == 0 ? 1 : value);
                cons->c_column += n;
                flush(cons);
                break;

            case 'D':           /* ESC [nD moves left n spaces */
                n = (value == 0 ? 1 : value);
                cons->c_column -= n;
                flush(cons);
                break;

            case 'H':           /* ESC [m;nH" moves cursor to (m,n) */
                cons->c_row = cons->c_esc_parmv[0] - 1;
                cons->c_column = cons->c_esc_parmv[1] - 1;
                flush(cons);
                break;

            case 'J':           /* ESC [sJ clears in display */
                switch (value) {
                    case 0:     /* Clear from cursor to end of screen */
                        count = scr_size - (cons->c_cur - cons->c_org);
                        dst = cons->c_cur;
                        break;
                    case 1:     /* Clear from start of screen to cursor */
                        count = cons->c_cur - cons->c_org;
                        dst = cons->c_org;
                        break;
                    case 2:     /* Clear entire screen */
                        count = scr_size;
                        dst = cons->c_org;
                        break;
                    default:    /* Do nothing */
                        count = 0;
                        dst = cons->c_org;
                }
                blank_color = cons->c_blank;
                mem_vid_copy(BLANK_MEM, dst, count);
                break;

            case 'K':           /* ESC [sK clears line from cursor */
                switch (value) {
                    case 0:     /* Clear from cursor to end of line */
                        count = scr_width - cons->c_column;
                        dst = cons->c_cur;
                        break;
                    case 1:     /* Clear from beginning of line to cursor */
                        count = cons->c_column;
                        dst = cons->c_cur - cons->c_column;
                        break;
                    case 2:     /* Clear entire line */
                        count = scr_width;
                        dst = cons->c_cur - cons->c_column;
                        break;
                    default:    /* Do nothing */
                        count = 0;
                        dst = cons->c_cur;
                }
                blank_color = cons->c_blank;
                mem_vid_copy(BLANK_MEM, dst, count);
                break;

            case 'L':           /* ESC [nL inserts n lines at cursor */
                n = value;
                if (n < 1) n = 1;
                if (n > (scr_lines - cons->c_row))
```

```
                                n = scr_lines - cons->c_row;

                        src = cons->c_org + cons->c_row * scr_width;
                        dst = src + n * scr_width;
                        count = (scr_lines - cons->c_row - n) * scr_width;
                        vid_vid_copy(src, dst, count);
                        blank_color = cons->c_blank;
                        mem_vid_copy(BLANK_MEM, src, n * scr_width);
                        break;

                    case 'M':           /* ESC [nM deletes n lines at cursor */
                        n = value;
                        if (n < 1) n = 1;
                        if (n > (scr_lines - cons->c_row))
                                n = scr_lines - cons->c_row;

                        dst = cons->c_org + cons->c_row * scr_width;
                        src = dst + n * scr_width;
                        count = (scr_lines - cons->c_row - n) * scr_width;
                        vid_vid_copy(src, dst, count);
                        blank_color = cons->c_blank;
                        mem_vid_copy(BLANK_MEM, dst + count, n * scr_width);
                        break;

                    case '@':           /* ESC [n@ inserts n chars at cursor */
                        n = value;
                        if (n < 1) n = 1;
                        if (n > (scr_width - cons->c_column))
                                n = scr_width - cons->c_column;

                        src = cons->c_cur;
                        dst = src + n;
                        count = scr_width - cons->c_column - n;
                        vid_vid_copy(src, dst, count);
                        blank_color = cons->c_blank;
                        mem_vid_copy(BLANK_MEM, src, n);
                        break;

                    case 'P':           /* ESC [nP deletes n chars at cursor */
                        n = value;
                        if (n < 1) n = 1;
                        if (n > (scr_width - cons->c_column))
                                n = scr_width - cons->c_column;

                        dst = cons->c_cur;
                        src = dst + n;
                        count = scr_width - cons->c_column - n;
                        vid_vid_copy(src, dst, count);
                        blank_color = cons->c_blank;
                        mem_vid_copy(BLANK_MEM, dst + count, n);
                        break;

                    case 'm':           /* ESC [nm enables rendition n */
                        for (parmp = cons->c_esc_parmv; parmp <= cons->c_esc_parmp
                                        && parmp < bufend(cons->c_esc_parmv); parmp++) {
                                if (cons->c_reverse) {
                                        /* Unswap fg and bg colors */
                                        cons->c_attr =  ((cons->c_attr & 0x7000) >> 4) |
                                                        ((cons->c_attr & 0x0700) << 4) |
                                                        ((cons->c_attr & 0x8800));
                                }
                                switch (n = *parmp) {
                                    case 0:     /* NORMAL */
                                        cons->c_attr = cons->c_blank = BLANK_COLOR;
                                        cons->c_reverse = FALSE;
                                        break;

                                    case 1:     /* BOLD  */
                                        /* Set intensity bit */
                                        cons->c_attr |= 0x0800;
                                        break;

                                    case 4:     /* UNDERLINE */
                                        if (color) {
                                                /* Change white to cyan, i.e. lose red
                                                 */
                                                cons->c_attr = (cons->c_attr & 0xBBFF);
                                        } else {
                                                /* Set underline attribute */
                                                cons->c_attr = (cons->c_attr & 0x99FF);
                                        }
                                        break;

                                    case 5:     /* BLINKING */
                                        /* Set the blink bit */
                                        cons->c_attr |= 0x8000;
                                        break;

                                    case 7:     /* REVERSE */
                                        cons->c_reverse = TRUE;
                                        break;

                                    default:    /* COLOR */
                                        if (n == 39) n = 37;    /* set default color */
                                        if (n == 49) n = 40;

                                        if (!color) {
                                                /* Don't mess up a monochrome screen */
                                        } else
                                        if (30 <= n && n <= 37) {
                                                /* Foreground color */
                                                cons->c_attr =
                                                        (cons->c_attr & 0xF8FF) |
                                                        (ansi_colors[(n - 30)] << 8);
                                                cons->c_blank =
                                                        (cons->c_blank & 0xF8FF) |
                                                        (ansi_colors[(n - 30)] << 8);
                                        } else
                                        if (40 <= n && n <= 47) {
                                                /* Background color */
                                                cons->c_attr =
                                                        (cons->c_attr & 0x8FFF) |
                                                        (ansi_colors[(n - 40)] << 12);
                                                cons->c_blank =
                                                        (cons->c_blank & 0x8FFF) |
                                                        (ansi_colors[(n - 40)] << 12);
                                        }
                                }
                                if (cons->c_reverse) {
                                        /* Swap fg and bg colors */
                                        cons->c_attr =  ((cons->c_attr & 0x7000) >> 4) |
                                                        ((cons->c_attr & 0x0700) << 4) |
                                                        ((cons->c_attr & 0x8800));
                                }
                        }
                        break;
                }
          }
          cons->c_esc_state = 0;
        }

        /*===========================================================================*
         *                              set_6845                                     *
         *===========================================================================*/
        PRIVATE void set_6845(reg, val)
        int reg;                        /* which register pair to set */
        unsigned val;                   /* 16-bit value to set it to */
        {
        /* Set a register pair inside the 6845.
         * Registers 12-13 tell the 6845 where in video ram to start
         * Registers 14-15 tell the 6845 where to put the cursor
         */
          pvb_pair_t char_out[4];
          pv_set(char_out[0], vid_port + INDEX, reg);   /* set index register */
          pv_set(char_out[1], vid_port + DATA, (val>>8) & BYTE);    /* high byte */
          pv_set(char_out[2], vid_port + INDEX, reg + 1);           /* again */
          pv_set(char_out[3], vid_port + DATA, val&BYTE);           /* low byte */
          sys_voutb(char_out, 4);                       /* do actual output */
        }

        /*===========================================================================*
         *                              get_6845                                     *
         *===========================================================================*/
        PRIVATE void get_6845(reg, val)
        int reg;                        /* which register pair to set */
        unsigned *val;                  /* 16-bit value to set it to */
        {
          char v1, v2;
        /* Get a register pair inside the 6845.  */
          sys_outb(vid_port + INDEX, reg);
          sys_inb(vid_port + DATA, &v1);
          sys_outb(vid_port + INDEX, reg+1);
          sys_inb(vid_port + DATA, &v2);
          *val = (v1 << 8) | v2;
        }

        /*===========================================================================*
         *                              beep                                         *
         *===========================================================================*/
        PRIVATE void beep()
        {
        /* Making a beeping sound on the speaker (output for CRTL-G).
         * This routine works by turning on the bits 0 and 1 in port B of the 8255
         * chip that drive the speaker.
         */
          static timer_t tmr_stop_beep;
          pvb_pair_t char_out[3];
          clock_t now;
          int port_b_val, s;

          /* Fetch current time in advance to prevent beeping delay. */
          if ((s=getuptime(&now)) != OK)
                panic("TTY","Console couldn’t get clock’s uptime.", s);
          if (!beeping) {
                /* Set timer channel 2, square wave, with given frequency. */
                pv_set(char_out[0], TIMER_MODE, 0xB6);
                pv_set(char_out[1], TIMER2, (BEEP_FREQ >> 0) & BYTE);
                pv_set(char_out[2], TIMER2, (BEEP_FREQ >> 8) & BYTE);
                if (sys_voutb(char_out, 3)==OK) {
                        if (sys_inb(PORT_B, &port_b_val)==OK &&
                            sys_outb(PORT_B, (port_b_val|3))==OK)
                                beeping = TRUE;
                }
          }
          /* Add a timer to the timers list. Possibly reschedule the alarm. */
          tmrs_settimer(&tty_timers, &tmr_stop_beep, now+B_TIME, stop_beep, NULL);
          if (tty_timers->tmr_exp_time != tty_next_timeout) {
                tty_next_timeout = tty_timers->tmr_exp_time;
                if ((s=sys_setalarm(tty_next_timeout, 1)) != OK)
                        panic("TTY","Console couldn’t set alarm.", s);
          }
        }

        /*===========================================================================*
         *                              stop_beep                                    *
         *===========================================================================*/
        PRIVATE void stop_beep(tmrp)
        timer_t *tmrp;
        {
        /* Turn off the beeper by turning off bits 0 and 1 in PORT_B. */
          int port_b_val;
          if (sys_inb(PORT_B, &port_b_val)==OK &&
                sys_outb(PORT_B, (port_b_val & ~3))==OK)
                        beeping = FALSE;
        }

        /*===========================================================================*
         *                              scr_init                                     *
         *===========================================================================*/
        PUBLIC void scr_init(tp)
        tty_t *tp;
        {
        /* Initialize the screen driver. */
          console_t *cons;
          phys_bytes vid_base;
          u16_t bios_columns, bios_crtbase, bios_fontlines;
          u8_t bios_rows;
          int line;
          int s;
          static int vdu_initialized = 0;
          unsigned page_size;

          /* Associate console and TTY. */
          line = tp - &tty_table[0];
          if (line >= nr_cons) return;
          cons = &cons_table[line];
          cons->c_tty = tp;
          tp->tty_priv = cons;

          /* Initialize the keyboard driver. */
```

```
    kb_init(tp);

    /* Fill in TTY function hooks. */
    tp->tty_devwrite = cons_write;
    tp->tty_echo = cons_echo;
    tp->tty_ioctl = cons_ioctl;

    /* Get the BIOS parameters that describe the VDU. */
    if (! vdu_initialized++) {

          /* How about error checking? What to do on failure??? */
          s=sys_vircopy(SELF, BIOS_SEG, (vir_bytes) VDU_SCREEN_COLS_ADDR,
                  SELF, D, (vir_bytes) &bios_columns, VDU_SCREEN_COLS_SIZE);
          s=sys_vircopy(SELF, BIOS_SEG, (vir_bytes) VDU_CRT_BASE_ADDR,
                  SELF, D, (vir_bytes) &bios_crtbase, VDU_CRT_BASE_SIZE);
          s=sys_vircopy(SELF, BIOS_SEG, (vir_bytes) VDU_SCREEN_ROWS_ADDR,
                  SELF, D, (vir_bytes) &bios_rows, VDU_SCREEN_ROWS_SIZE);
          s=sys_vircopy(SELF, BIOS_SEG, (vir_bytes) VDU_FONTLINES_ADDR,
                  SELF, D, (vir_bytes) &bios_fontlines, VDU_FONTLINES_SIZE);

          vid_port = bios_crtbase;
          scr_width = bios_columns;
          font_lines = bios_fontlines;
          scr_lines = machine.vdu_ega ? bios_rows+1 : 25;

          if (color) {
                  vid_base = COLOR_BASE;
                  vid_size = COLOR_SIZE;
          } else {
                  vid_base = MONO_BASE;
                  vid_size = MONO_SIZE;
          }
          if (machine.vdu_ega) vid_size = EGA_SIZE;
          wrap = ! machine.vdu_ega;

          s = sys_segctl(&vid_index, &vid_seg, &vid_off, vid_base, vid_size);

          vid_size >>= 1;         /* word count */
          vid_mask = vid_size - 1;

          /* Size of the screen (number of displayed characters.) */
          scr_size = scr_lines * scr_width;

          /* There can be as many consoles as video memory allows. */
          nr_cons = vid_size / scr_size;
          if (nr_cons > NR_CONS) nr_cons = NR_CONS;
          if (nr_cons > 1) wrap = 0;
          page_size = vid_size / nr_cons;
    }

    cons->c_start = line * page_size;
    cons->c_limit = cons->c_start + page_size;
    cons->c_cur = cons->c_org = cons->c_start;
    cons->c_attr = cons->c_blank = BLANK_COLOR;

    if (line != 0) {
          /* Clear the non-console vtys. */
          blank_color = BLANK_COLOR;
          mem_vid_copy(BLANK_MEM, cons->c_start, scr_size);
    } else {
          int i, n;
          /* Set the cursor of the console vty at the bottom. c_cur
           * is updated automatically later.
           */
          scroll_screen(cons, SCROLL_UP);
          cons->c_row = scr_lines - 1;
          cons->c_column = 0;
    }
    select_console(0);
    cons_ioctl(tp, 0);
  }

  /*===========================================================================*
   *                              kputc                                        *
   *===========================================================================*/
  PUBLIC void kputc(c)
  int c;
  {
          putk(c);
  }

  /*===========================================================================*
   *                              do_new_kmess                                 *
   *===========================================================================*/
  PUBLIC void do_new_kmess(m)
  message *m;
  {
  /* Notification for a new kernel message. */
    struct kmessages kmess;                       /* kmessages structure */
    static int prev_next = 0;                     /* previous next seen */
    int size, next;
    int bytes;
    int r;

    /* Try to get a fresh copy of the buffer with kernel messages. */
    sys_getkmessages(&kmess);

    /* Print only the new part. Determine how many new bytes there are with
     * help of the current and previous 'next' index. Note that the kernel
     * buffer is circular. This works fine if less then KMESS_BUF_SIZE bytes
     * is new data; else we miss % KMESS_BUF_SIZE here.
     * Check for size being positive, the buffer might as well be emptied!
     */
    if (kmess.km_size > 0) {
        bytes = ((kmess.km_next + KMESS_BUF_SIZE) - prev_next) % KMESS_BUF_SIZE;
        r=prev_next;                              /* start at previous old */
        while (bytes > 0) {
            putk( kmess.km_buf[(r%KMESS_BUF_SIZE)] );
            bytes --;
            r ++;
        }
        putk(0);                  /* terminate to flush output */
    }

    /* Almost done, store 'next' so that we can determine what part of the
     * kernel messages buffer to print next time a notification arrives.
     */
    prev_next = kmess.km_next;
  }
```

```
/*===========================================================================*
 *                              do_diagnostics                               *
 *===========================================================================*/
PUBLIC void do_diagnostics(m_ptr)
message *m_ptr;                 /* pointer to request message */
{
/* Print a string for a server. */
  char c;
  vir_bytes src;
  int count;
  int result = OK;
  int proc_nr = m_ptr->DIAG_PROC_NR;
  if (proc_nr == SELF) proc_nr = m_ptr->m_source;

  src = (vir_bytes) m_ptr->DIAG_PRINT_BUF;
  for (count = m_ptr->DIAG_BUF_COUNT; count > 0; count--) {
        if (sys_vircopy(proc_nr, D, src++, SELF, D, (vir_bytes) &c, 1) != OK) {
                result = EFAULT;
                break;
        }
        putk(c);
  }
  putk(0);                      /* always terminate, even with EFAULT */
  m_ptr->m_type = result;
  send(m_ptr->m_source, m_ptr);
}

/*===========================================================================*
 *                              putk                                         *
 *===========================================================================*/
PRIVATE void putk(c)
int c;                          /* character to print */
{
/* This procedure is used by the version of printf() that is linked with
 * the TTY driver.  The one in the library sends a message to FS, which is
 * not what is needed for printing within the TTY. This version just queues
 * the character and starts the output.
 */
  if (c != 0) {
        if (c == '\n') putk('\r');
        out_char(&cons_table[0], (int) c);
  } else {
        flush(&cons_table[0]);
  }
}

/*===========================================================================*
 *                              toggle_scroll                                *
 *===========================================================================*/
PUBLIC void toggle_scroll()
{
/* Toggle between hardware and software scroll. */

  cons_org0();
  softscroll = !softscroll;
  printf("%sware scrolling enabled.\n", softscroll ? "Soft" : "Hard");
}
```

```
/*===========================================================================*
 *                              cons_stop                                    *
 *===========================================================================*/
PUBLIC void cons_stop()
{
/* Prepare for halt or reboot. */
  cons_org0();
  softscroll = 1;
  select_console(0);
  cons_table[0].c_attr = cons_table[0].c_blank = BLANK_COLOR;
}

/*===========================================================================*
 *                              cons_org0                                    *
 *===========================================================================*/
PRIVATE void cons_org0()
{
/* Scroll video memory back to put the origin at 0. */
  int cons_line;
  console_t *cons;
  unsigned n;

  for (cons_line = 0; cons_line < nr_cons; cons_line++) {
        cons = &cons_table[cons_line];
        while (cons->c_org > cons->c_start) {
                n = vid_size - scr_size;        /* amount of unused memory */
                if (n > cons->c_org - cons->c_start)
                        n = cons->c_org - cons->c_start;
                vid_vid_copy(cons->c_org, cons->c_org - n, scr_size);
                cons->c_org -= n;
        }
        flush(cons);
  }
  select_console(ccurrent);
}

/*===========================================================================*
 *                              select_console                               *
 *===========================================================================*/
PUBLIC void select_console(int cons_line)
{
/* Set the current console to console number 'cons_line'. */

  if (cons_line < 0 || cons_line >= nr_cons) return;
  ccurrent = cons_line;
  curcons = &cons_table[cons_line];
  set_6845(VID_ORG, curcons->c_org);
  set_6845(CURSOR, curcons->c_cur);
}

/*===========================================================================*
 *                              con_loadfont                                 *
 *===========================================================================*/
PUBLIC int con_loadfont(m)
message *m;
{
/* Load a font into the EGA or VGA adapter. */
  int result;
  static struct sequence seq1[7] = {
        { GA_SEQUENCER_INDEX, 0x00, 0x01 },
        { GA_SEQUENCER_INDEX, 0x02, 0x04 },
        { GA_SEQUENCER_INDEX, 0x04, 0x07 },
        { GA_SEQUENCER_INDEX, 0x00, 0x03 },
        { GA_GRAPHICS_INDEX, 0x04, 0x02 },
        { GA_GRAPHICS_INDEX, 0x05, 0x00 },
        { GA_GRAPHICS_INDEX, 0x06, 0x00 },
  };
  static struct sequence seq2[7] = {
        { GA_SEQUENCER_INDEX, 0x00, 0x01 },
        { GA_SEQUENCER_INDEX, 0x02, 0x03 },
        { GA_SEQUENCER_INDEX, 0x04, 0x03 },
        { GA_SEQUENCER_INDEX, 0x00, 0x03 },
        { GA_GRAPHICS_INDEX, 0x04, 0x00 },
        { GA_GRAPHICS_INDEX, 0x05, 0x10 },
        { GA_GRAPHICS_INDEX, 0x06,    0 },
  };

  seq2[6].value= color ? 0x0E : 0x0A;

  if (!machine.vdu_ega) return(ENOTTY);
  result = ga_program(seq1);    /* bring font memory into view */

  result = sys_physcopy(m->PROC_NR, D, (vir_bytes) m->ADDRESS,
        NONE, PHYS_SEG, (phys_bytes) GA_VIDEO_ADDRESS, (phys_bytes)GA_FONT_SIZE);

  result = ga_program(seq2);    /* restore */

  return(result);
}

/*===========================================================================*
 *                              ga_program                                   *
 *===========================================================================*/
PRIVATE int ga_program(seq)
struct sequence *seq;
{
  pvb_pair_t char_out[14];
  int i;
  for (i=0; i<7; i++) {
      pv_set(char_out[2*i], seq->index, seq->port);
      pv_set(char_out[2*i+1], seq->index+1, seq->value);
      seq++;
  }
  return sys_voutb(char_out, 14);
}

/*===========================================================================*
 *                              cons_ioctl                                   *
 *===========================================================================*/
PRIVATE int cons_ioctl(tp, try)
tty_t *tp;
int try;
{
/* Set the screen dimensions. */

  tp->tty_winsize.ws_row= scr_lines;
  tp->tty_winsize.ws_col= scr_width;
  tp->tty_winsize.ws_xpixel= scr_width * 8;
  tp->tty_winsize.ws_ypixel= scr_lines * font_lines;
}
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                servers/pm/pm.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

/* This is the master header for PM.  It includes some other files
 * and defines the principal constants.
 */
#define _POSIX_SOURCE      1    /* tell headers to include POSIX stuff */
#define _MINIX             1    /* tell headers to include MINIX stuff */
#define _SYSTEM            1    /* tell headers that this is the kernel */

/* The following are so basic, all the *.c files get them automatically. */
#include <minix/config.h>       /* MUST be first */
#include <ansi.h>               /* MUST be second */
#include <sys/types.h>
#include <minix/const.h>
#include <minix/type.h>

#include <fcntl.h>
#include <unistd.h>
#include <minix/syslib.h>
#include <minix/sysutil.h>

#include <limits.h>
#include <errno.h>

#include "const.h"
#include "type.h"
#include "proto.h"
#include "glo.h"
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                servers/pm/const.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

/* Constants used by the Process Manager. */

#define NO_MEM ((phys_clicks) 0)  /* returned by alloc_mem() with mem is up */

#define NR_PIDS        30000    /* process ids range from 0 to NR_PIDS-1.
                                 * (magic constant: some old applications use
                                 * a 'short' instead of pid_t.)
                                 */

#define PM_PID             0    /* PM's process id number */
#define INIT_PID           1    /* INIT's process id number */

```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                              servers/pm/type.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

/* If there were any type definitions local to the Process Manager, they would
 * be here.  This file is included only for symmetry with the kernel and File
 * System, which do have some local type definitions.
 */

++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                              servers/pm/proto.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

/* Function prototypes. */

struct mproc;
struct stat;
struct mem_map;
struct memory;

#include <timers.h>

/* alloc.c */
_PROTOTYPE( phys_clicks alloc_mem, (phys_clicks clicks)                 );
_PROTOTYPE( void free_mem, (phys_clicks base, phys_clicks clicks)       );
_PROTOTYPE( void mem_init, (struct memory *chunks, phys_clicks *free)   );
#define swap_in()                       ((void)0)
#define swap_inqueue(rmp)               ((void)0)

/* break.c */
_PROTOTYPE( int adjust, (struct mproc *rmp,
                        vir_clicks data_clicks, vir_bytes sp)           );
_PROTOTYPE( int do_brk, (void)                                          );
_PROTOTYPE( int size_ok, (int file_type, vir_clicks tc, vir_clicks dc,
                        vir_clicks sc, vir_clicks dvir, vir_clicks s_vir) );

/* devio.c */
_PROTOTYPE( int do_dev_io, (void) );
_PROTOTYPE( int do_dev_io, (void) );

/* dmp.c */
_PROTOTYPE( int do_fkey_pressed, (void)                                         );

/* exec.c */
_PROTOTYPE( int do_exec, (void)                                         );
_PROTOTYPE( void rw_seg, (int rw, int fd, int proc, int seg,
                                                phys_bytes seg_bytes)   );
_PROTOTYPE( struct mproc *find_share, (struct mproc *mp_ign, Ino_t ino,
                        Dev_t dev, time_t ctime)                        );

/* forkexit.c */
_PROTOTYPE( int do_fork, (void)                                         );
_PROTOTYPE( int do_pm_exit, (void)                                      );
_PROTOTYPE( int do_waitpid, (void)                                      );
_PROTOTYPE( void pm_exit, (struct mproc *rmp, int exit_status)          );

/* getset.c */
_PROTOTYPE( int do_getset, (void)                                       );
```

```

/* main.c */
_PROTOTYPE( int main, (void)                                            );

/* misc.c */
_PROTOTYPE( int do_reboot, (void)                                       );
_PROTOTYPE( int do_getsysinfo, (void)                                   );
_PROTOTYPE( int do_getprocnr, (void)                                    );
_PROTOTYPE( int do_svrctl, (void)                                       );
_PROTOTYPE( int do_allocmem, (void)                                     );
_PROTOTYPE( int do_freemem, (void)                                      );
_PROTOTYPE( int do_getsetpriority, (void)                                               );

_PROTOTYPE( void setreply, (int proc_nr, int result)                    );

/* signal.c */
_PROTOTYPE( int do_alarm, (void)                                        );
_PROTOTYPE( int do_kill, (void)                                         );
_PROTOTYPE( int ksig_pending, (void)                                    );
_PROTOTYPE( int do_pause, (void)                                        );
_PROTOTYPE( int set_alarm, (int proc_nr, int sec)                       );
_PROTOTYPE( int check_sig, (pid_t proc_id, int signo)                   );
_PROTOTYPE( void sig_proc, (struct mproc *rmp, int sig_nr)              );
_PROTOTYPE( int do_sigaction, (void)                                    );
_PROTOTYPE( int do_sigpending, (void)                                   );
_PROTOTYPE( int do_sigprocmask, (void)                                  );
_PROTOTYPE( int do_sigreturn, (void)                                    );
_PROTOTYPE( int do_sigsuspend, (void)                                   );
_PROTOTYPE( void check_pending, (struct mproc *rmp)                     );

/* time.c */
_PROTOTYPE( int do_stime, (void)                                        );
_PROTOTYPE( int do_time, (void)                                         );
_PROTOTYPE( int do_times, (void)                                        );
_PROTOTYPE( int do_gettimeofday, (void)                                 );

/* timers.c */
_PROTOTYPE( void pm_set_timer, (timer_t *tp, int delta,
        tmr_func_t watchdog, int arg));
_PROTOTYPE( void pm_expire_timers, (clock_t now));
_PROTOTYPE( void pm_cancel_timer, (timer_t *tp));

/* trace.c */
_PROTOTYPE( int do_trace, (void)                                        );
_PROTOTYPE( void stop_proc, (struct mproc *rmp, int sig_nr)             );

/* utility.c */
_PROTOTYPE( pid_t get_free_pid, (void)                                  );
_PROTOTYPE( int allowed, (char *name_buf, struct stat *s_buf, int mask) );
_PROTOTYPE( int no_sys, (void)                                          );
_PROTOTYPE( void panic, (char *who, char *mess, int num)                );
_PROTOTYPE( void tell_fs, (int what, int p1, int p2, int p3)            );
_PROTOTYPE( int get_stack_ptr, (int proc_nr, vir_bytes *sp)             );
_PROTOTYPE( int get_mem_map, (int proc_nr, struct mem_map *mem_map)     );
_PROTOTYPE( char *find_param, (const char *key));
_PROTOTYPE( int proc_from_pid, (pid_t p));
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                servers/pm/glo.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

/* EXTERN should be extern except in table.c */
#ifdef _TABLE
#undef EXTERN
#define EXTERN
#endif

/* Global variables. */
EXTERN struct mproc *mp;        /* ptr to 'mproc' slot of current process */
EXTERN int procs_in_use;        /* how many processes are marked as IN_USE */
EXTERN char monitor_params[128*sizeof(char *)]; /* boot monitor parameters */
EXTERN struct kinfo kinfo;                      /* kernel information */

/* The parameters of the call are kept here. */
EXTERN message m_in;            /* the incoming message itself is kept here. */
EXTERN int who;                 /* caller's proc number */
EXTERN int call_nr;             /* system call number */

extern _PROTOTYPE (int (*call_vec[]), (void) ); /* system call handlers */
extern char core_name[];        /* file name where core images are produced */
EXTERN sigset_t core_sset;      /* which signals cause core images */
EXTERN sigset_t ign_sset;       /* which signals are by default ignored */

```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                servers/pm/mproc.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

/* This table has one slot per process.  It contains all the process management
 * information for each process.  Among other things, it defines the text, data
 * and stack segments, uids and gids, and various flags.  The kernel and file
 * systems have tables that are also indexed by process, with the contents
 * of corresponding slots referring to the same process in all three.
 */
#include <timers.h>

EXTERN struct mproc {
  struct mem_map mp_seg[NR_LOCAL_SEGS]; /* points to text, data, stack */
  char mp_exitstatus;           /* storage for status when process exits */
  char mp_sigstatus;            /* storage for signal # for killed procs */
  pid_t mp_pid;                 /* process id */
  pid_t mp_procgrp;             /* pid of process group (used for signals) */
  pid_t mp_wpid;                /* pid this process is waiting for */
  int mp_parent;                /* index of parent process */

  /* Child user and system times. Accounting done on child exit. */
  clock_t mp_child_utime;       /* cumulative user time of children */
  clock_t mp_child_stime;       /* cumulative sys time of children */

  /* Real and effective uids and gids. */
  uid_t mp_realuid;             /* process' real uid */
  uid_t mp_effuid;              /* process' effective uid */
  gid_t mp_realgid;             /* process' real gid */
```

```
   gid_t mp_effgid;              /* process' effective gid */

   /* File identification for sharing. */
   ino_t mp_ino;                 /* inode number of file */
   dev_t mp_dev;                 /* device number of file system */
   time_t mp_ctime;              /* inode changed time */

   /* Signal handling information. */
   sigset_t mp_ignore;           /* 1 means ignore the signal, 0 means don't */
   sigset_t mp_catch;            /* 1 means catch the signal, 0 means don't */
   sigset_t mp_sig2mess;         /* 1 means transform into notify message */
   sigset_t mp_sigmask;          /* signals to be blocked */
   sigset_t mp_sigmask2;         /* saved copy of mp_sigmask */
   sigset_t mp_sigpending;       /* pending signals to be handled */
   struct sigaction mp_sigact[_NSIG + 1]; /* as in sigaction(2) */
   vir_bytes mp_sigreturn;       /* address of C library __sigreturn function */
   struct timer mp_timer;        /* watchdog timer for alarm(2) */

   /* Backwards compatibility for signals. */
   sighandler_t mp_func;         /* all sigs vectored to a single user fcn */

   unsigned mp_flags;            /* flag bits */
   vir_bytes mp_procargs;        /* ptr to proc's initial stack arguments */
   struct mproc *mp_swapq;       /* queue of procs waiting to be swapped in */
   message mp_reply;             /* reply message to be sent to one */

   /* Scheduling priority. */
   signed int mp_nice;           /* nice is PRIO_MIN..PRIO_MAX, standard 0. */

   char mp_name[PROC_NAME_LEN];  /* process name */
 } mproc[NR_PROCS];

 /* Flag values */
 #define IN_USE          0x001   /* set when 'mproc' slot in use */
 #define WAITING         0x002   /* set by WAIT system call */
 #define ZOMBIE          0x004   /* set by EXIT, cleared by WAIT */
 #define PAUSED          0x008   /* set by PAUSE system call */
 #define ALARM_ON        0x010   /* set when SIGALRM timer started */
 #define SEPARATE        0x020   /* set if file is separate I & D space */
 #define TRACED          0x040   /* set if process is to be traced */
 #define STOPPED         0x080   /* set if process stopped for tracing */
 #define SIGSUSPENDED    0x100   /* set by SIGSUSPEND system call */
 #define REPLY           0x200   /* set if a reply message is pending */
 #define ONSWAP          0x400   /* set if data segment is swapped out */
 #define SWAPIN          0x800   /* set if on the "swap this in" queue */
 #define DONT_SWAP      0x1000   /* never swap out this process */
 #define PRIV_PROC      0x2000   /* system process, special privileges */

 #define NIL_MPROC ((struct mproc *) 0)

```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                servers/pm/param.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
 /* The following names are synonyms for the variables in the input message. */
 #define addr            m1_p1
 #define exec_name       m1_p1
 #define exec_len        m1_i1
 #define func            m6_f1
```

```
#define grp_id          m1_i1
#define namelen         m1_i2
#define pid             m1_i1
#define procnr          m1_i1
#define seconds         m1_i1
#define sig             m6_i1
#define stack_bytes     m1_i2
#define stack_ptr       m1_p2
#define status          m1_i1
#define usr_id          m1_i1
#define request         m2_i2
#define taddr           m2_l1
#define data            m2_l2
#define sig_nr          m1_i2
#define sig_nsa         m1_p1
#define sig_osa         m1_p2
#define sig_ret         m1_p3
#define sig_set         m2_l1
#define sig_how         m2_i1
#define sig_flags       m2_i2
#define sig_context     m2_p1
#define info_what       m1_i1
#define info_where      m1_p1
#define reboot_flag     m1_i1
#define reboot_code     m1_p1
#define reboot_strlen   m1_i2
#define svrctl_req      m2_i1
#define svrctl_argp     m2_p1
#define stime           m2_l1
#define memsize         m4_l1
#define membase         m4_l2

/* The following names are synonyms for the variables in a reply message. */
#define reply_res       m_type
#define reply_res2      m2_i1
#define reply_ptr       m2_p1
#define reply_mask      m2_l1
#define reply_trace     m2_l2
#define reply_time      m2_l1
#define reply_utime     m2_l2
#define reply_t1        m4_l1
#define reply_t2        m4_l2
#define reply_t3        m4_l3
#define reply_t4        m4_l4
#define reply_t5        m4_l5

/* The following names are used to inform the FS about certain events. */
#define tell_fs_arg1    m1_i1
#define tell_fs_arg2    m1_i2
#define tell_fs_arg3    m1_i3

```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                      servers/pm/table.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

/* This file contains the table used to map system call numbers onto the
 * routines that perform them.
 */

#define _TABLE

#include "pm.h"
#include <minix/callnr.h>
#include <signal.h>
#include "mproc.h"
#include "param.h"

/* Miscellaneous */
char core_name[] = "core";      /* file name where core images are produced */

_PROTOTYPE (int (*call_vec[NCALLS]), (void) ) = {
        no_sys,         /*  0 = unused  */
        do_pm_exit,     /*  1 = exit    */
        do_fork,        /*  2 = fork    */
        no_sys,         /*  3 = read    */
        no_sys,         /*  4 = write   */
        no_sys,         /*  5 = open    */
        no_sys,         /*  6 = close   */
        do_waitpid,     /*  7 = wait    */
        no_sys,         /*  8 = creat   */
        no_sys,         /*  9 = link    */
        no_sys,         /* 10 = unlink  */
        do_waitpid,     /* 11 = waitpid */
        no_sys,         /* 12 = chdir   */
        do_time,        /* 13 = time    */
        no_sys,         /* 14 = mknod   */
        no_sys,         /* 15 = chmod   */
        no_sys,         /* 16 = chown   */
        do_brk,         /* 17 = break   */
        no_sys,         /* 18 = stat    */
        no_sys,         /* 19 = lseek   */
        do_getset,      /* 20 = getpid  */
        no_sys,         /* 21 = mount   */
        no_sys,         /* 22 = umount  */
        do_getset,      /* 23 = setuid  */
        do_getset,      /* 24 = getuid  */
        do_stime,       /* 25 = stime   */
        do_trace,       /* 26 = ptrace  */
        do_alarm,       /* 27 = alarm   */
        no_sys,         /* 28 = fstat   */
        do_pause,       /* 29 = pause   */
        no_sys,         /* 30 = utime   */
        no_sys,         /* 31 = (stty)  */
        no_sys,         /* 32 = (gtty)  */
        no_sys,         /* 33 = access  */
        no_sys,         /* 34 = (nice)  */
        no_sys,         /* 35 = (ftime) */
        no_sys,         /* 36 = sync    */
        do_kill,        /* 37 = kill    */
        no_sys,         /* 38 = rename  */
        no_sys,         /* 39 = mkdir   */
        no_sys,         /* 40 = rmdir   */
        no_sys,         /* 41 = dup     */
        no_sys,         /* 42 = pipe    */
        do_times,       /* 43 = times   */
        no_sys,         /* 44 = (prof)  */
        no_sys,         /* 45 = unused  */
        do_getset,      /* 46 = setgid  */
        do_getset,      /* 47 = getgid  */
        no_sys,         /* 48 = (signal)*/
        no_sys,         /* 49 = unused  */
        no_sys,         /* 50 = unused  */
        no_sys,         /* 51 = (acct)  */
        no_sys,         /* 52 = (phys)  */
        no_sys,         /* 53 = (lock)  */
        no_sys,         /* 54 = ioctl   */
        no_sys,         /* 55 = fcntl   */
        no_sys,         /* 56 = (mpx)   */
        no_sys,         /* 57 = unused  */
        no_sys,         /* 58 = unused  */
        do_exec,        /* 59 = execve  */
        no_sys,         /* 60 = umask   */
        no_sys,         /* 61 = chroot  */
        do_getset,      /* 62 = setsid  */
        do_getset,      /* 63 = getpgrp */

        no_sys,         /* 64 = unused */
        no_sys,         /* 65 = UNPAUSE */
        no_sys,         /* 66 = unused  */
        no_sys,         /* 67 = REVIVE  */
        no_sys,         /* 68 = TASK_REPLY  */
        no_sys,         /* 69 = unused  */
        no_sys,         /* 70 = unused  */
        do_sigaction,   /* 71 = sigaction   */
        do_sigsuspend,  /* 72 = sigsuspend  */
        do_sigpending,  /* 73 = sigpending  */
        do_sigprocmask, /* 74 = sigprocmask */
        do_sigreturn,   /* 75 = sigreturn   */
        do_reboot,      /* 76 = reboot  */
        do_svrctl,      /* 77 = svrctl  */

        no_sys,         /* 78 = unused */
        do_getsysinfo,  /* 79 = getsysinfo */
        do_getprocnr,   /* 80 = getprocnr */
        no_sys,         /* 81 = unused */
        no_sys,         /* 82 = fstatfs */
        do_allocmem,    /* 83 = memalloc */
        do_freemem,     /* 84 = memfree */
        no_sys,         /* 85 = select */
        no_sys,         /* 86 = fchdir */
        no_sys,         /* 87 = fsync */
        do_getsetpriority,      /* 88 = getpriority */
        do_getsetpriority,      /* 89 = setpriority */
        do_time,        /* 90 = gettimeofday */
};
/* This should not fail with "array size is negative": */
extern int dummy[sizeof(call_vec) == NCALLS * sizeof(call_vec[0]) ? 1 : -1];
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                      servers/pm/main.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

/* This file contains the main program of the process manager and some related
 * procedures.  When MINIX starts up, the kernel runs for a little while,
 * initializing itself and its tasks, and then it runs PM and FS.  Both PM
 * and FS initialize themselves as far as they can. PM asks the kernel for
 * all free memory and starts serving requests.
 *
 * The entry points into this file are:
 *   main:      starts PM running
 *   setreply:  set the reply to be sent to process making an PM system call
 */

#include "pm.h"
#include <minix/keymap.h>
#include <minix/callnr.h>
#include <minix/com.h>
#include <signal.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/resource.h>
#include <string.h>
#include "mproc.h"
#include "param.h"

#include "../../kernel/const.h"
#include "../../kernel/config.h"
#include "../../kernel/type.h"
#include "../../kernel/proc.h"

FORWARD _PROTOTYPE( void get_work, (void)                               );
FORWARD _PROTOTYPE( void pm_init, (void)                                );
FORWARD _PROTOTYPE( int get_nice_value, (int queue)                     );
FORWARD _PROTOTYPE( void get_mem_chunks, (struct memory *mem_chunks)    );
FORWARD _PROTOTYPE( void patch_mem_chunks, (struct memory *mem_chunks,
        struct mem_map *map_ptr)        );

#define click_to_round_k(n) \
        ((unsigned) ((((unsigned long) (n) << CLICK_SHIFT) + 512) / 1024))

/*===========================================================================*
 *                              main                                         *
 *===========================================================================*/
PUBLIC int main()
{
/* Main routine of the process manager. */
  int result, s, proc_nr;
  struct mproc *rmp;
  sigset_t sigset;

  pm_init();                    /* initialize process manager tables */

  /* This is PM's main loop-  get work and do it, forever and forever. */
  while (TRUE) {
        get_work();             /* wait for an PM system call */

        /* Check for system notifications first. Special cases. */
        if (call_nr == SYN_ALARM) {
                pm_expire_timers(m_in.NOTIFY_TIMESTAMP);
                result = SUSPEND;               /* don't reply */
        } else if (call_nr == SYS_SIG) {        /* signals pending */
                sigset = m_in.NOTIFY_ARG;
                if (sigismember(&sigset, SIGKSIG))  (void) ksig_pending();
                result = SUSPEND;               /* don't reply */
        }
        /* Else, if the system call number is valid, perform the call. */
        else if ((unsigned) call_nr >= NCALLS) {
                result = ENOSYS;
        } else {
                result = (*call_vec[call_nr])();
        }

        /* Send the results back to the user to indicate completion. */
        if (result != SUSPEND) setreply(who, result);

        swap_in();              /* maybe a process can be swapped in? */

        /* Send out all pending reply messages, including the answer to
         * the call just made above.  The processes must not be swapped out.
         */
        for (proc_nr=0, rmp=mproc; proc_nr < NR_PROCS; proc_nr++, rmp++) {
                /* In the meantime, the process may have been killed by a
                 * signal (e.g. if a lethal pending signal was unblocked)
                 * without the PM realizing it. If the slot is no longer in
                 * use or just a zombie, don't try to reply.
                 */
                if ((rmp->mp_flags & (REPLY | ONSWAP | IN_USE | ZOMBIE)) ==
                   (REPLY | IN_USE)) {
                        if ((s=send(proc_nr, &rmp->mp_reply)) != OK) {
                                panic(__FILE__,"PM can't reply to", proc_nr);
                        }
                        rmp->mp_flags &= ~REPLY;
                }
        }
  }
  return(OK);
}

/*===========================================================================*
 *                              get_work                                     *
 *===========================================================================*/
PRIVATE void get_work()
{
/* Wait for the next message and extract useful information from it. */
  if (receive(ANY, &m_in) != OK) panic(__FILE__,"PM receive error", NO_NUM);
  who = m_in.m_source;          /* who sent the message */
  call_nr = m_in.m_type;        /* system call number */

  /* Process slot of caller. Misuse PM's own process slot if the kernel is
   * calling. This can happen in case of synchronous alarms (CLOCK) or or
   * event like pending kernel signals (SYSTEM).
   */
  mp = &mproc[who < 0 ? PM_PROC_NR : who];
}
```

```
/*===========================================================================*
 *                              setreply                                     *
 *===========================================================================*/
PUBLIC void setreply(proc_nr, result)
int proc_nr;                    /* process to reply to */
int result;                     /* result of call (usually OK or error #) */
{
/* Fill in a reply message to be sent later to a user process.  System calls
 * may occasionally fill in other fields, this is only for the main return
 * value, and for setting the "must send reply" flag.
 */
  register struct mproc *rmp = &mproc[proc_nr];

  rmp->mp_reply.reply_res = result;
  rmp->mp_flags |= REPLY;       /* reply pending */

  if (rmp->mp_flags & ONSWAP)
        swap_inqueue(rmp);      /* must swap this process back in */
}

/*===========================================================================*
 *                              pm_init                                      *
 *===========================================================================*/
PRIVATE void pm_init()
{
/* Initialize the process manager.
 * Memory use info is collected from the boot monitor, the kernel, and
 * all processes compiled into the system image. Initially this information
 * is put into an array mem_chunks. Elements of mem_chunks are struct memory,
 * and hold base, size pairs in units of clicks. This array is small, there
 * should be no more than 8 chunks. After the array of chunks has been built
 * the contents are used to initialize the hole list. Space for the hole list
 * is reserved as an array with twice as many elements as the maximum number
 * of processes allowed. It is managed as a linked list, and elements of the
 * array are struct hole, which, in addition to storage for a base and size in
 * click units also contain space for a link, a pointer to another element.
*/
  int s;
  static struct boot_image image[NR_BOOT_PROCS];
  register struct boot_image *ip;
  static char core_sigs[] = { SIGQUIT, SIGILL, SIGTRAP, SIGABRT,
                        SIGEMT, SIGFPE, SIGUSR1, SIGSEGV, SIGUSR2 };
  static char ign_sigs[] = { SIGCHLD };
  register struct mproc *rmp;
  register char *sig_ptr;
  phys_clicks total_clicks, minix_clicks, free_clicks;
  message mess;
  struct mem_map mem_map[NR_LOCAL_SEGS];
  struct memory mem_chunks[NR_MEMS];

  /* Initialize process table, including timers. */
  for (rmp=&mproc[0]; rmp<&mproc[NR_PROCS]; rmp++) {
        tmr_inittimer(&rmp->mp_timer);
  }

  /* Build the set of signals which cause core dumps, and the set of signals
   * that are by default ignored.
   */
  sigemptyset(&core_sset);
  for (sig_ptr = core_sigs; sig_ptr < core_sigs+sizeof(core_sigs); sig_ptr++)
```

```
                sigaddset(&core_sset, *sig_ptr);
        sigemptyset(&ign_sset);
        for (sig_ptr = ign_sigs; sig_ptr < ign_sigs+sizeof(ign_sigs); sig_ptr++)
                sigaddset(&ign_sset, *sig_ptr);

        /* Obtain a copy of the boot monitor parameters and the kernel info struct.
         * Parse the list of free memory chunks. This list is what the boot monitor
         * reported, but it must be corrected for the kernel and system processes.
         */
        if ((s=sys_getmonparams(monitor_params, sizeof(monitor_params))) != OK)
            panic(__FILE__,"get monitor params failed",s);
        get_mem_chunks(mem_chunks);
        if ((s=sys_getkinfo(&kinfo)) != OK)
            panic(__FILE__,"get kernel info failed",s);

        /* Get the memory map of the kernel to see how much memory it uses. */
        if ((s=get_mem_map(SYSTASK, mem_map)) != OK)
                panic(__FILE__,"couldn't get memory map of SYSTASK",s);
        minix_clicks = (mem_map[S].mem_phys+mem_map[S].mem_len)-mem_map[T].mem_phys;
        patch_mem_chunks(mem_chunks, mem_map);

        /* Initialize PM's process table. Request a copy of the system image table
         * that is defined at the kernel level to see which slots to fill in.
         */
        if (OK != (s=sys_getimage(image)))
                panic(__FILE__,"couldn't get image table: %d\n", s);
        procs_in_use = 0;                               /* start populating table */
        printf("Building process table:");              /* show what's happening */
        for (ip = &image[0]; ip < &image[NR_BOOT_PROCS]; ip++) {
                if (ip->proc_nr >= 0) {                 /* task have negative nrs */
                        procs_in_use += 1;              /* found user process */

                        /* Set process details found in the image table. */
                        rmp = &mproc[ip->proc_nr];
                        strncpy(rmp->mp_name, ip->proc_name, PROC_NAME_LEN);
                        rmp->mp_parent = RS_PROC_NR;
                        rmp->mp_nice = get_nice_value(ip->priority);
                        if (ip->proc_nr == INIT_PROC_NR) {      /* user process */
                                rmp->mp_pid = INIT_PID;
                                rmp->mp_flags |= IN_USE;
                                sigemptyset(&rmp->mp_ignore);
                        }
                        else {                                  /* system process */
                                rmp->mp_pid = get_free_pid();
                                rmp->mp_flags |= IN_USE | DONT_SWAP | PRIV_PROC;
                                sigfillset(&rmp->mp_ignore);
                        }
                        sigemptyset(&rmp->mp_sigmask);
                        sigemptyset(&rmp->mp_catch);
                        sigemptyset(&rmp->mp_sig2mess);

                        /* Get memory map for this process from the kernel. */
                        if ((s=get_mem_map(ip->proc_nr, rmp->mp_seg)) != OK)
                                panic(__FILE__,"couldn't get process entry",s);
                        if (rmp->mp_seg[T].mem_len != 0) rmp->mp_flags |= SEPARATE;
                        minix_clicks += rmp->mp_seg[S].mem_phys +
                                rmp->mp_seg[S].mem_len - rmp->mp_seg[T].mem_phys;
                        patch_mem_chunks(mem_chunks, rmp->mp_seg);

                        /* Tell FS about this system process. */
                        mess.PR_PROC_NR = ip->proc_nr;
                        mess.PR_PID = rmp->mp_pid;
                        if (OK != (s=send(FS_PROC_NR, &mess)))
                                panic(__FILE__,"can't sync up with FS", s);
                        printf(" %s", ip->proc_name);   /* display process name */
                }
          }
          printf(".\n");                                /* last process done */

          /* Override some details. PM is somewhat special. */
          mproc[PM_PROC_NR].mp_pid = PM_PID;            /* magically override pid */
          mproc[PM_PROC_NR].mp_parent = PM_PROC_NR;     /* PM doesn't have parent */

          /* Tell FS that no more system processes follow and synchronize. */
          mess.PR_PROC_NR = NONE;
          if (sendrec(FS_PROC_NR, &mess) != OK || mess.m_type != OK)
                panic(__FILE__,"can't sync up with FS", NO_NUM);

          /* Initialize tables to all physical memory and print memory information. */
          printf("Physical memory:");
          mem_init(mem_chunks, &free_clicks);
          total_clicks = minix_clicks + free_clicks;
          printf(" total %u KB,", click_to_round_k(total_clicks));
          printf(" system %u KB,", click_to_round_k(minix_clicks));
          printf(" free %u KB.\n", click_to_round_k(free_clicks));
        }

        /*===========================================================================*
         *                              get_nice_value                               *
         *===========================================================================*/
        PRIVATE int get_nice_value(queue)
        int queue;                              /* store mem chunks here */
        {
        /* Processes in the boot image have a priority assigned. The PM doesn't know
         * about priorities, but uses 'nice' values instead. The priority is between
         * MIN_USER_Q and MAX_USER_Q. We have to scale between PRIO_MIN and PRIO_MAX.
         */
          int nice_val = (queue - USER_Q) * (PRIO_MAX-PRIO_MIN+1) /
              (MIN_USER_Q-MAX_USER_Q+1);
          if (nice_val > PRIO_MAX) nice_val = PRIO_MAX; /* shouldn't happen */
          if (nice_val < PRIO_MIN) nice_val = PRIO_MIN; /* shouldn't happen */
          return nice_val;
        }

        /*===========================================================================*
         *                              get_mem_chunks                               *
         *===========================================================================*/
        PRIVATE void get_mem_chunks(mem_chunks)
        struct memory *mem_chunks;                      /* store mem chunks here */
        {
        /* Initialize the free memory list from the 'memory' boot variable.  Translate
         * the byte offsets and sizes in this list to clicks, properly truncated. Also
         * make sure that we don't exceed the maximum address space of the 286 or the
         * 8086, i.e. when running in 16-bit protected mode or real mode.
         */
          long base, size, limit;
          char *s, *end;                        /* use to parse boot variable */
          int i, done = 0;
          struct memory *memp;

          /* Initialize everything to zero. */
          for (i = 0; i < NR_MEMS; i++) {
                memp = &mem_chunks[i];          /* next mem chunk is stored here */
                memp->base = memp->size = 0;
          }

          /* The available memory is determined by MINIX' boot loader as a list of
           * (base:size)-pairs in boothead.s. The 'memory' boot variable is set in
           * in boot.s.  The format is "b0:s0,b1:s1,b2:s2", where b0:s0 is low mem,
           * b1:s1 is mem between 1M and 16M, b2:s2 is mem above 16M. Pairs b1:s1
           * and b2:s2 are combined if the memory is adjacent.
           */
          s = find_param("memory");             /* get memory boot variable */
          for (i = 0; i < NR_MEMS && !done; i++) {
                memp = &mem_chunks[i];          /* next mem chunk is stored here */
                base = size = 0;                /* initialize next base:size pair */
                if (*s != 0) {                  /* get fresh data, unless at end */

                    /* Read fresh base and expect colon as next char. */
                    base = strtoul(s, &end, 0x10);              /* get number */
                    if (end != s && *end == ':') s = ++end;     /* skip ':' */
                    else *s=0;                  /* terminate, should not happen */

                    /* Read fresh size and expect comma or assume end. */
                    size = strtoul(s, &end, 0x10);              /* get number */
                    if (end != s && *end == ',') s = ++end;     /* skip ',' */
                    else done = 1;
                }
                limit = base + size;
                base = (base + CLICK_SIZE-1) & ~(long)(CLICK_SIZE-1);
                limit &= ~(long)(CLICK_SIZE-1);
                if (limit <= base) continue;
                memp->base = base >> CLICK_SHIFT;
                memp->size = (limit - base) >> CLICK_SHIFT;
          }
        }

        /*===========================================================================*
         *                              patch_mem_chunks                             *
         *===========================================================================*/
        PRIVATE void patch_mem_chunks(mem_chunks, map_ptr)
        struct memory *mem_chunks;                      /* store mem chunks here */
        struct mem_map *map_ptr;                        /* memory to remove */
        {
        /* Remove server memory from the free memory list. The boot monitor
         * promises to put processes at the start of memory chunks. The
         * tasks all use same base address, so only the first task changes
         * the memory lists. The servers and init have their own memory
         * spaces and their memory will be removed from the list.
         */
          struct memory *memp;
          for (memp = mem_chunks; memp < &mem_chunks[NR_MEMS]; memp++) {
                if (memp->base == map_ptr[T].mem_phys) {
                        memp->base += map_ptr[T].mem_len + map_ptr[D].mem_len;
                        memp->size -= map_ptr[T].mem_len + map_ptr[D].mem_len;
                }
          }
        }
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                   servers/pm/forkexit.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

/* This file deals with creating processes (via FORK) and deleting them (via
 * EXIT/WAIT).  When a process forks, a new slot in the 'mproc' table is
 * allocated for it, and a copy of the parent's core image is made for the
 * child.  Then the kernel and file system are informed.  A process is removed
 * from the 'mproc' table when two events have occurred: (1) it has exited or
 * been killed by a signal, and (2) the parent has done a WAIT.  If the process
 * exits first, it continues to occupy a slot until the parent does a WAIT.
 *
 * The entry points into this file are:
 *   do_fork:    perform the FORK system call
 *   do_pm_exit: perform the EXIT system call (by calling pm_exit())
 *   pm_exit:    actually do the exiting
 *   do_wait:    perform the WAITPID or WAIT system call
 */

#include "pm.h"
#include <sys/wait.h>
#include <minix/callnr.h>
#include <minix/com.h>
#include <signal.h>
#include "mproc.h"
#include "param.h"

#define LAST_FEW            2   /* last few slots reserved for superuser */

FORWARD _PROTOTYPE (void cleanup, (register struct mproc *child) );

/*===========================================================================*
 *                              do_fork                                      *
 *===========================================================================*/
PUBLIC int do_fork()
{
/* The process pointed to by 'mp' has forked.  Create a child process. */
  register struct mproc *rmp;   /* pointer to parent */
  register struct mproc *rmc;   /* pointer to child */
  int child_nr, s;
  phys_clicks prog_clicks, child_base;
  phys_bytes prog_bytes, parent_abs, child_abs; /* Intel only */
  pid_t new_pid;

 /* If tables might fill up during FORK, don't even start since recovery half
  * way through is such a nuisance.
  */
  rmp = mp;
  if ((procs_in_use == NR_PROCS) ||
              (procs_in_use >= NR_PROCS-LAST_FEW && rmp->mp_effuid != 0))
  {
        printf("PM: warning, process table is full!\n");
        return(EAGAIN);
  }

  /* Determine how much memory to allocate.  Only the data and stack need to
   * be copied, because the text segment is either shared or of zero length.
   */
  prog_clicks = (phys_clicks) rmp->mp_seg[S].mem_len;
  prog_clicks += (rmp->mp_seg[S].mem_vir - rmp->mp_seg[D].mem_vir);
  prog_bytes = (phys_bytes) prog_clicks << CLICK_SHIFT;
  if ( (child_base = alloc_mem(prog_clicks)) == NO_MEM) return(ENOMEM);

  /* Create a copy of the parent's core image for the child. */
  child_abs = (phys_bytes) child_base << CLICK_SHIFT;
  parent_abs = (phys_bytes) rmp->mp_seg[D].mem_phys << CLICK_SHIFT;
  s = sys_abscopy(parent_abs, child_abs, prog_bytes);
  if (s < 0) panic(__FILE__,"do_fork can't copy", s);

  /* Find a slot in 'mproc' for the child process.  A slot must exist. */
  for (rmc = &mproc[0]; rmc < &mproc[NR_PROCS]; rmc++)
        if ( (rmc->mp_flags & IN_USE) == 0) break;

  /* Set up the child and its memory map; copy its 'mproc' slot from parent. */
  child_nr = (int)(rmc - mproc);        /* slot number of the child */
  procs_in_use++;
  *rmc = *rmp;                  /* copy parent's process slot to child's */
  rmc->mp_parent = who;                 /* record child's parent */
  /* inherit only these flags */
  rmc->mp_flags &= (IN_USE|SEPARATE|PRIV_PROC|DONT_SWAP);
  rmc->mp_child_utime = 0;              /* reset administration */
  rmc->mp_child_stime = 0;              /* reset administration */

  /* A separate I&D child keeps the parents text segment.  The data and stack
   * segments must refer to the new copy.
   */
  if (!(rmc->mp_flags & SEPARATE)) rmc->mp_seg[T].mem_phys = child_base;
  rmc->mp_seg[D].mem_phys = child_base;
  rmc->mp_seg[S].mem_phys = rmc->mp_seg[D].mem_phys +
                      (rmp->mp_seg[S].mem_vir - rmp->mp_seg[D].mem_vir);
  rmc->mp_exitstatus = 0;
  rmc->mp_sigstatus = 0;

  /* Find a free pid for the child and put it in the table. */
  new_pid = get_free_pid();
  rmc->mp_pid = new_pid;        /* assign pid to child */

  /* Tell kernel and file system about the (now successful) FORK. */
  sys_fork(who, child_nr);
  tell_fs(FORK, who, child_nr, rmc->mp_pid);

  /* Report child's memory map to kernel. */
  sys_newmap(child_nr, rmc->mp_seg);

  /* Reply to child to wake it up. */
  setreply(child_nr, 0);                /* only parent gets details */
  rmp->mp_reply.procnr = child_nr;      /* child's process number */
  return(new_pid);                      /* child's pid */
}

/*===========================================================================*
 *                              do_pm_exit                                   *
 *===========================================================================*/
PUBLIC int do_pm_exit()
{
/* Perform the exit(status) system call. The real work is done by pm_exit(),
 * which is also called when a process is killed by a signal.
 */
  pm_exit(mp, m_in.status);
  return(SUSPEND);              /* can't communicate from beyond the grave */
}

/*===========================================================================*
 *                              pm_exit                                      *
 *===========================================================================*/
PUBLIC void pm_exit(rmp, exit_status)
register struct mproc *rmp;     /* pointer to the process to be terminated */
int exit_status;                /* the process' exit status (for parent) */
{
/* A process is done.  Release most of the process' possessions.  If its
 * parent is waiting, release the rest, else keep the process slot and
 * become a zombie.
 */
  register int proc_nr;
  int parent_waiting, right_child;
  pid_t pidarg, procgrp;
  struct mproc *p_mp;
  clock_t t[5];

  proc_nr = (int) (rmp - mproc);        /* get process slot number */

  /* Remember a session leader's process group. */
  procgrp = (rmp->mp_pid == mp->mp_procgrp) ? mp->mp_procgrp : 0;

  /* If the exited process has a timer pending, kill it. */
  if (rmp->mp_flags & ALARM_ON) set_alarm(proc_nr, (unsigned) 0);

  /* Do accounting: fetch usage times and accumulate at parent. */
  sys_times(proc_nr, t);
  p_mp = &mproc[rmp->mp_parent];                        /* process' parent */
  p_mp->mp_child_utime += t[0] + rmp->mp_child_utime;   /* add user time */
  p_mp->mp_child_stime += t[1] + rmp->mp_child_stime;   /* add system time */

  /* Tell the kernel and FS that the process is no longer runnable. */
  tell_fs(EXIT, proc_nr, 0, 0);  /* file system can free the proc slot */
  sys_exit(proc_nr);

  /* Pending reply messages for the dead process cannot be delivered. */
  rmp->mp_flags &= ~REPLY;

  /* Release the memory occupied by the child. */
  if (find_share(rmp, rmp->mp_ino, rmp->mp_dev, rmp->mp_ctime) == NULL) {
        /* No other process shares the text segment, so free it. */
        free_mem(rmp->mp_seg[T].mem_phys, rmp->mp_seg[T].mem_len);
  }
  /* Free the data and stack segments. */
  free_mem(rmp->mp_seg[D].mem_phys,
      rmp->mp_seg[S].mem_vir
        + rmp->mp_seg[S].mem_len - rmp->mp_seg[D].mem_vir);

  /* The process slot can only be freed if the parent has done a WAIT. */
  rmp->mp_exitstatus = (char) exit_status;

  pidarg = p_mp->mp_wpid;               /* who's being waited for? */
  parent_waiting = p_mp->mp_flags & WAITING;
  right_child =                         /* child meets one of the 3 tests? */
        (pidarg == -1 || pidarg == rmp->mp_pid || -pidarg == rmp->mp_procgrp);

  if (parent_waiting && right_child) {
        cleanup(rmp);                   /* tell parent and release child slot */
  } else {
        rmp->mp_flags = IN_USE|ZOMBIE;  /* parent not waiting, zombify child */
        sig_proc(p_mp, SIGCHLD);        /* send parent a "child died" signal */
  }

  /* If the process has children, disinherit them.  INIT is the new parent. */
  for (rmp = &mproc[0]; rmp < &mproc[NR_PROCS]; rmp++) {
        if (rmp->mp_flags & IN_USE && rmp->mp_parent == proc_nr) {
                /* 'rmp' now points to a child to be disinherited. */
                rmp->mp_parent = INIT_PROC_NR;
                parent_waiting = mproc[INIT_PROC_NR].mp_flags & WAITING;
                if (parent_waiting && (rmp->mp_flags & ZOMBIE)) cleanup(rmp);
        }
  }

  /* Send a hangup to the process' process group if it was a session leader. */
  if (procgrp != 0) check_sig(-procgrp, SIGHUP);
}

/*===========================================================================*
 *                              do_waitpid                                   *
 *===========================================================================*/
PUBLIC int do_waitpid()
{
/* A process wants to wait for a child to terminate. If a child is already
 * waiting, go clean it up and let this WAIT call terminate.  Otherwise,
 * really wait.
 * A process calling WAIT never gets a reply in the usual way at the end
 * of the main loop (unless WNOHANG is set or no qualifying child exists).
 * If a child has already exited, the routine cleanup() sends the reply
 * to awaken the caller.
 * Both WAIT and WAITPID are handled by this code.
 */
  register struct mproc *rp;
  int pidarg, options, children;

  /* Set internal variables, depending on whether this is WAIT or WAITPID. */
  pidarg  = (call_nr == WAIT ? -1 : m_in.pid);     /* 1st param of waitpid */
  options = (call_nr == WAIT ?  0 : m_in.sig_nr);  /* 3rd param of waitpid */
  if (pidarg == 0) pidarg = -mp->mp_procgrp;    /* pidarg < 0 ==> proc grp */

  /* Is there a child waiting to be collected? At this point, pidarg != 0:
   *    pidarg  >  0 means pidarg is pid of a specific process to wait for
   *    pidarg == -1 means wait for any child
   *    pidarg  < -1 means wait for any child whose process group = -pidarg
   */
  children = 0;
  for (rp = &mproc[0]; rp < &mproc[NR_PROCS]; rp++) {
        if ( (rp->mp_flags & IN_USE) && rp->mp_parent == who) {
                /* The value of pidarg determines which children qualify. */
                if (pidarg  > 0 && pidarg != rp->mp_pid) continue;
                if (pidarg < -1 && -pidarg != rp->mp_procgrp) continue;

                children++;             /* this child is acceptable */
                if (rp->mp_flags & ZOMBIE) {
                        /* This child meets the pid test and has exited. */
                        cleanup(rp);    /* this child has already exited */
                        return(SUSPEND);
                }
                if ((rp->mp_flags & STOPPED) && rp->mp_sigstatus) {
                        /* This child meets the pid test and is being traced.*/
                        mp->mp_reply.reply_res2 = 0177|(rp->mp_sigstatus << 8);
                        rp->mp_sigstatus = 0;
                        return(rp->mp_pid);
                }
        }
  }

  /* No qualifying child has exited.  Wait for one, unless none exists. */
  if (children > 0) {
        /* At least 1 child meets the pid test exists, but has not exited. */
        if (options & WNOHANG) return(0);    /* parent does not want to wait */
        mp->mp_flags |= WAITING;             /* parent wants to wait */
        mp->mp_wpid = (pid_t) pidarg;        /* save pid for later */
        return(SUSPEND);                     /* do not reply, let it wait */
  } else {
        /* No child even meets the pid test.  Return error immediately. */
        return(ECHILD);                      /* no - parent has no children */
  }
}

/*===========================================================================*
 *                              cleanup                                      *
 *===========================================================================*/
PRIVATE void cleanup(child)
register struct mproc *child;   /* tells which process is exiting */
{
/* Finish off the exit of a process.  The process has exited or been killed
 * by a signal, and its parent is waiting.
 */
  struct mproc *parent = &mproc[child->mp_parent];
  int exitstatus;

  /* Wake up the parent by sending the reply message. */
  exitstatus = (child->mp_exitstatus << 8) | (child->mp_sigstatus & 0377);
  parent->mp_reply.reply_res2 = exitstatus;
  setreply(child->mp_parent, child->mp_pid);
  parent->mp_flags &= ~WAITING;         /* parent no longer waiting */

  /* Release the process table entry and reinitialize some field. */
  child->mp_pid = 0;
  child->mp_flags = 0;
  child->mp_child_utime = 0;
  child->mp_child_stime = 0;
  procs_in_use--;
}
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                servers/pm/exec.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/* This file handles the EXEC system call.  It performs the work as follows:
 *    - see if the permissions allow the file to be executed
 *    - read the header and extract the sizes
 *    - fetch the initial args and environment from the user space
 *    - allocate the memory for the new process
 *    - copy the initial stack from PM to the process
 *    - read in the text and data segments and copy to the process
 *    - take care of setuid and setgid bits
 *    - fix up 'mproc' table
 *    - tell kernel about EXEC
 *    - save offset to initial argc (for ps)
 *
 * The entry points into this file are:
 *   do_exec:    perform the EXEC system call
 *   rw_seg:     read or write a segment from or to a file
 *   find_share: find a process whose text segment can be shared
 */

#include "pm.h"
#include <sys/stat.h>
#include <minix/callnr.h>
#include <minix/com.h>
#include <a.out.h>
#include <signal.h>
#include <string.h>
#include "mproc.h"
#include "param.h"

FORWARD _PROTOTYPE( int new_mem, (struct mproc *sh_mp, vir_bytes text_bytes,
                vir_bytes data_bytes, vir_bytes bss_bytes,
                vir_bytes stk_bytes, phys_bytes tot_bytes)              );
FORWARD _PROTOTYPE( void patch_ptr, (char stack[ARG_MAX], vir_bytes base) );
FORWARD _PROTOTYPE( int insert_arg, (char stack[ARG_MAX],
                vir_bytes *stk_bytes, char *arg, int replace)           );
FORWARD _PROTOTYPE( char *patch_stack, (int fd, char stack[ARG_MAX],
                vir_bytes *stk_bytes, char *script)                     );
FORWARD _PROTOTYPE( int read_header, (int fd, int *ft, vir_bytes *text_bytes,
                vir_bytes *data_bytes, vir_bytes *bss_bytes,
                phys_bytes *tot_bytes, long *sym_bytes, vir_clicks sc,
                vir_bytes *pc)                                          );

#define ESCRIPT (-2000) /* Returned by read_header for a #! script. */
#define PTRSIZE sizeof(char *) /* Size of pointers in argv[] and envp[]. */

/*===========================================================================*
 *                              do_exec                                      *
 *===========================================================================*/
PUBLIC int do_exec()
{
/* Perform the execve(name, argv, envp) call.  The user library builds a
 * complete stack image, including pointers, args, environ, etc.  The stack
 * is copied to a buffer inside PM, and then to the new core image.
 */
  register struct mproc *rmp;
  struct mproc *sh_mp;
  int m, r, fd, ft, sn;
  static char mbuf[ARG_MAX];    /* buffer for stack and zeroes */
  static char name_buf[PATH_MAX]; /* the name of the file to exec */
  char *new_sp, *name, *basename;
  vir_bytes src, dst, text_bytes, data_bytes, bss_bytes, stk_bytes, vsp;
  phys_bytes tot_bytes;         /* total space for program, including gap */
  long sym_bytes;
  vir_clicks sc;
  struct stat s_buf[2], *s_p;
  vir_bytes pc;

  /* Do some validity checks. */
  rmp = mp;
  stk_bytes = (vir_bytes) m_in.stack_bytes;
  if (stk_bytes > ARG_MAX) return(ENOMEM);      /* stack too big */
  if (m_in.exec_len <= 0 || m_in.exec_len > PATH_MAX) return(EINVAL);

  /* Get the exec file name and see if the file is executable. */
  src = (vir_bytes) m_in.exec_name;
  dst = (vir_bytes) name_buf;
  r = sys_datacopy(who, (vir_bytes) src,
                PM_PROC_NR, (vir_bytes) dst, (phys_bytes) m_in.exec_len);
  if (r != OK) return(r);       /* file name not in user data segment */

  /* Fetch the stack from the user before destroying the old core image. */
  src = (vir_bytes) m_in.stack_ptr;
  dst = (vir_bytes) mbuf;
  r = sys_datacopy(who, (vir_bytes) src,
                        PM_PROC_NR, (vir_bytes) dst, (phys_bytes)stk_bytes);
  /* can't fetch stack (e.g. bad virtual addr) */
  if (r != OK) return(EACCES);

  r = 0;        /* r = 0 (first attempt), or 1 (interpreted script) */
  name = name_buf;      /* name of file to exec. */
  do {
        s_p = &s_buf[r];
        tell_fs(CHDIR, who, FALSE, 0);  /* switch to the user's FS environ */
        fd = allowed(name, s_p, X_BIT); /* is file executable? */
        if (fd < 0)  return(fd);                /* file was not executable */

        /* Read the file header and extract the segment sizes. */
        sc = (stk_bytes + CLICK_SIZE - 1) >> CLICK_SHIFT;

        m = read_header(fd, &ft, &text_bytes, &data_bytes, &bss_bytes,
                                        &tot_bytes, &sym_bytes, sc, &pc);
        if (m != ESCRIPT || ++r > 1) break;
  } while ((name = patch_stack(fd, mbuf, &stk_bytes, name_buf)) != NULL);

  if (m < 0) {
        close(fd);              /* something wrong with header */
        return(stk_bytes > ARG_MAX ? ENOMEM : ENOEXEC);
  }

  /* Can the process' text be shared with that of one already running? */
  sh_mp = find_share(rmp, s_p->st_ino, s_p->st_dev, s_p->st_ctime);

  /* Allocate new memory and release old memory.  Fix map and tell kernel. */
  r = new_mem(sh_mp, text_bytes, data_bytes, bss_bytes, stk_bytes, tot_bytes);
  if (r != OK) {
        close(fd);              /* insufficient core or program too big */
        return(r);
  }

  /* Save file identification to allow it to be shared. */
  rmp->mp_ino = s_p->st_ino;
  rmp->mp_dev = s_p->st_dev;
  rmp->mp_ctime = s_p->st_ctime;

  /* Patch up stack and copy it from PM to new core image. */
  vsp = (vir_bytes) rmp->mp_seg[S].mem_vir << CLICK_SHIFT;
  vsp += (vir_bytes) rmp->mp_seg[S].mem_len << CLICK_SHIFT;
  vsp -= stk_bytes;
  patch_ptr(mbuf, vsp);
  src = (vir_bytes) mbuf;
  r = sys_datacopy(PM_PROC_NR, (vir_bytes) src,
                      who, (vir_bytes) vsp, (phys_bytes)stk_bytes);
  if (r != OK) panic(__FILE__,"do_exec stack copy err on", who);

  /* Read in text and data segments. */
  if (sh_mp != NULL) {
        lseek(fd, (off_t) text_bytes, SEEK_CUR);  /* shared: skip text */
  } else {
        rw_seg(0, fd, who, T, text_bytes);
  }
  rw_seg(0, fd, who, D, data_bytes);

  close(fd);                    /* don't need exec file any more */

  /* Take care of setuid/setgid bits. */
  if ((rmp->mp_flags & TRACED) == 0) { /* suppress if tracing */
        if (s_buf[0].st_mode & I_SET_UID_BIT) {
                rmp->mp_effuid = s_buf[0].st_uid;
                tell_fs(SETUID,who, (int)rmp->mp_realuid, (int)rmp->mp_effuid);
        }
        if (s_buf[0].st_mode & I_SET_GID_BIT) {
                rmp->mp_effgid = s_buf[0].st_gid;
                tell_fs(SETGID,who, (int)rmp->mp_realgid, (int)rmp->mp_effgid);
        }
  }

  /* Save offset to initial argc (for ps) */
  rmp->mp_procargs = vsp;

  /* Fix 'mproc' fields, tell kernel that exec is done,  reset caught sigs. */
  for (sn = 1; sn <= _NSIG; sn++) {
        if (sigismember(&rmp->mp_catch, sn)) {
                sigdelset(&rmp->mp_catch, sn);
                rmp->mp_sigact[sn].sa_handler = SIG_DFL;
                sigemptyset(&rmp->mp_sigact[sn].sa_mask);
        }
  }

  rmp->mp_flags &= ~SEPARATE;   /* turn off SEPARATE bit */
  rmp->mp_flags |= ft;          /* turn it on for separate I & D files */
  new_sp = (char *) vsp;

  tell_fs(EXEC, who, 0, 0);     /* allow FS to handle FD_CLOEXEC files */

  /* System will save command line for debugging, ps(1) output, etc. */
  basename = strrchr(name, '/');
  if (basename == NULL) basename = name; else basename++;
  strncpy(rmp->mp_name, basename, PROC_NAME_LEN-1);
  rmp->mp_name[PROC_NAME_LEN] = '\0';
  sys_exec(who, new_sp, basename, pc);

  /* Cause a signal if this process is traced. */
  if (rmp->mp_flags & TRACED) check_sig(rmp->mp_pid, SIGTRAP);

  return(SUSPEND);              /* no reply, new program just runs */
}
```

```
/*===========================================================================*
 *                              read_header                                  *
 *===========================================================================*/
PRIVATE int read_header(fd, ft, text_bytes, data_bytes, bss_bytes,
                                                tot_bytes, sym_bytes, sc, pc)
int fd;                         /* file descriptor for reading exec file */
int *ft;                        /* place to return ft number */
vir_bytes *text_bytes;          /* place to return text size */
vir_bytes *data_bytes;          /* place to return initialized data size */
vir_bytes *bss_bytes;           /* place to return bss size */
phys_bytes *tot_bytes;          /* place to return total size */
long *sym_bytes;                /* place to return symbol table size */
vir_clicks sc;                  /* stack size in clicks */
vir_bytes *pc;                  /* program entry point (initial PC) */
{
/* Read the header and extract the text, data, bss and total sizes from it. */

  int m, ct;
  vir_clicks tc, dc, s_vir, dvir;
  phys_clicks totc;
  struct exec hdr;              /* a.out header is read in here */

  /* Read the header and check the magic number.  The standard MINIX header
   * is defined in <a.out.h>.  It consists of 8 chars followed by 6 longs.
   * Then come 4 more longs that are not used here.
   *    Byte 0: magic number 0x01
   *    Byte 1: magic number 0x03
   *    Byte 2: normal = 0x10 (not checked, 0 is OK), separate I/D = 0x20
   *    Byte 3: CPU type, Intel 16 bit = 0x04, Intel 32 bit = 0x10,
   *            Motorola = 0x0B, Sun SPARC = 0x17
   *    Byte 4: Header length = 0x20
   *    Bytes 5-7 are not used.
   *
   *    Now come the 6 longs
   *    Bytes  8-11: size of text segments in bytes
   *    Bytes 12-15: size of initialized data segment in bytes
   *    Bytes 16-19: size of bss in bytes
   *    Bytes 20-23: program entry point
   *    Bytes 24-27: total memory allocated to program (text, data + stack)
   *    Bytes 28-31: size of symbol table in bytes
   * The longs are represented in a machine dependent order,
   * little-endian on the 8088, big-endian on the 68000.
   * The header is followed directly by the text and data segments, and the
   * symbol table (if any). The sizes are given in the header. Only the
   * text and data segments are copied into memory by exec. The header is
   * used here only. The symbol table is for the benefit of a debugger and
   * is ignored here.
   */

  if ((m= read(fd, &hdr, A_MINHDR)) < 2) return(ENOEXEC);

  /* Interpreted script? */
  if (((char *) &hdr)[0] == '#' && ((char *) &hdr)[1] == '!') return(ESCRIPT);

  if (m != A_MINHDR) return(ENOEXEC);

  /* Check magic number, cpu type, and flags. */
  if (BADMAG(hdr)) return(ENOEXEC);
  if (hdr.a_cpu != A_I80386) return(ENOEXEC);
  if ((hdr.a_flags & ~(A_NSYM | A_EXEC | A_SEP)) != 0) return(ENOEXEC);

  *ft = ( (hdr.a_flags & A_SEP) ? SEPARATE : 0);    /* separate I & D or not */

  /* Get text and data sizes. */
  *text_bytes = (vir_bytes) hdr.a_text; /* text size in bytes */
  *data_bytes = (vir_bytes) hdr.a_data; /* data size in bytes */
  *bss_bytes  = (vir_bytes) hdr.a_bss;  /* bss size in bytes */
  *tot_bytes  = hdr.a_total;            /* total bytes to allocate for prog */
  *sym_bytes  = hdr.a_syms;             /* symbol table size in bytes */
  if (*tot_bytes == 0) return(ENOEXEC);

  if (*ft != SEPARATE) {
        /* If I & D space is not separated, it is all considered data. Text=0*/
        *data_bytes += *text_bytes;
        *text_bytes = 0;
  }
  *pc = hdr.a_entry;    /* initial address to start execution */

  /* Check to see if segment sizes are feasible. */
  tc = ((unsigned long) *text_bytes + CLICK_SIZE - 1) >> CLICK_SHIFT;
  dc = (*data_bytes + *bss_bytes + CLICK_SIZE - 1) >> CLICK_SHIFT;
  totc = (*tot_bytes + CLICK_SIZE - 1) >> CLICK_SHIFT;
  if (dc >= totc) return(ENOEXEC);      /* stack must be at least 1 click */
  dvir = (*ft == SEPARATE ? 0 : tc);
  s_vir = dvir + (totc - sc);
  m = (dvir + dc > s_vir) ? ENOMEM : OK;
  ct = hdr.a_hdrlen & BYTE;             /* header length */
  if (ct > A_MINHDR) lseek(fd, (off_t) ct, SEEK_SET); /* skip unused hdr */
  return(m);
}

/*===========================================================================*
 *                              new_mem                                      *
 *===========================================================================*/
PRIVATE int new_mem(sh_mp, text_bytes, data_bytes,
        bss_bytes,stk_bytes,tot_bytes)
struct mproc *sh_mp;            /* text can be shared with this process */
vir_bytes text_bytes;           /* text segment size in bytes */
vir_bytes data_bytes;           /* size of initialized data in bytes */
vir_bytes bss_bytes;            /* size of bss in bytes */
vir_bytes stk_bytes;            /* size of initial stack segment in bytes */
phys_bytes tot_bytes;           /* total memory to allocate, including gap */
{
/* Allocate new memory and release the old memory.  Change the map and report
 * the new map to the kernel.  Zero the new core image's bss, gap and stack.
 */

  register struct mproc *rmp = mp;
  vir_clicks text_clicks, data_clicks, gap_clicks, stack_clicks, tot_clicks;
  phys_clicks new_base;
  phys_bytes bytes, base, bss_offset;
  int s;

  /* No need to allocate text if it can be shared. */
  if (sh_mp != NULL) text_bytes = 0;

  /* Allow the old data to be swapped out to make room.  (Which is really a
   * waste of time, because we are going to throw it away anyway.)
   */
```

```
    rmp->mp_flags |= WAITING;

    /* Acquire the new memory.  Each of the 4 parts: text, (data+bss), gap,
     * and stack occupies an integral number of clicks, starting at click
     * boundary.  The data and bss parts are run together with no space.
     */
    text_clicks = ((unsigned long) text_bytes + CLICK_SIZE - 1) >> CLICK_SHIFT;
    data_clicks = (data_bytes + bss_bytes + CLICK_SIZE - 1) >> CLICK_SHIFT;
    stack_clicks = (stk_bytes + CLICK_SIZE - 1) >> CLICK_SHIFT;
    tot_clicks = (tot_bytes + CLICK_SIZE - 1) >> CLICK_SHIFT;
    gap_clicks = tot_clicks - data_clicks - stack_clicks;
    if ( (int) gap_clicks < 0) return(ENOMEM);

    /* Try to allocate memory for the new process. */
    new_base = alloc_mem(text_clicks + tot_clicks);
    if (new_base == NO_MEM) return(ENOMEM);

    /* We've got memory for the new core image.  Release the old one. */
    rmp = mp;

    if (find_share(rmp, rmp->mp_ino, rmp->mp_dev, rmp->mp_ctime) == NULL) {
          /* No other process shares the text segment, so free it. */
          free_mem(rmp->mp_seg[T].mem_phys, rmp->mp_seg[T].mem_len);
    }
    /* Free the data and stack segments. */
    free_mem(rmp->mp_seg[D].mem_phys,
     rmp->mp_seg[S].mem_vir + rmp->mp_seg[S].mem_len - rmp->mp_seg[D].mem_vir);

    /* We have now passed the point of no return.  The old core image has been
     * forever lost, memory for a new core image has been allocated.  Set up
     * and report new map.
     */
    if (sh_mp != NULL) {
          /* Share the text segment. */
          rmp->mp_seg[T] = sh_mp->mp_seg[T];
    } else {
          rmp->mp_seg[T].mem_phys = new_base;
          rmp->mp_seg[T].mem_vir = 0;
          rmp->mp_seg[T].mem_len = text_clicks;
    }
    rmp->mp_seg[D].mem_phys = new_base + text_clicks;
    rmp->mp_seg[D].mem_vir = 0;
    rmp->mp_seg[D].mem_len = data_clicks;
    rmp->mp_seg[S].mem_phys = rmp->mp_seg[D].mem_phys + data_clicks + gap_clicks;
    rmp->mp_seg[S].mem_vir = rmp->mp_seg[D].mem_vir + data_clicks + gap_clicks;
    rmp->mp_seg[S].mem_len = stack_clicks;

    sys_newmap(who, rmp->mp_seg);   /* report new map to the kernel */

    /* The old memory may have been swapped out, but the new memory is real. */
    rmp->mp_flags &= ~(WAITING|ONSWAP|SWAPIN);

    /* Zero the bss, gap, and stack segment. */
    bytes = (phys_bytes)(data_clicks + gap_clicks + stack_clicks) << CLICK_SHIFT;
    base = (phys_bytes) rmp->mp_seg[D].mem_phys << CLICK_SHIFT;
    bss_offset = (data_bytes >> CLICK_SHIFT) << CLICK_SHIFT;
    base += bss_offset;
    bytes -= bss_offset;

    if ((s=sys_memset(0, base, bytes)) != OK) {
          panic(__FILE__,"new_mem can't zero", s);
    }

    return(OK);
  }

  /*===========================================================================*
   *                              patch_ptr                                    *
   *===========================================================================*/
  PRIVATE void patch_ptr(stack, base)
  char stack[ARG_MAX];            /* pointer to stack image within PM */
  vir_bytes base;                 /* virtual address of stack base inside user */
  {
  /* When doing an exec(name, argv, envp) call, the user builds up a stack
   * image with arg and env pointers relative to the start of the stack.  Now
   * these pointers must be relocated, since the stack is not positioned at
   * address 0 in the user's address space.
   */

    char **ap, flag;
    vir_bytes v;

    flag = 0;                     /* counts number of 0-pointers seen */
    ap = (char **) stack;         /* points initially to 'nargs' */
    ap++;                         /* now points to argv[0] */
    while (flag < 2) {
          if (ap >= (char **) &stack[ARG_MAX]) return;    /* too bad */
          if (*ap != NULL) {
                  v = (vir_bytes) *ap;    /* v is relative pointer */
                  v += base;              /* relocate it */
                  *ap = (char *) v;       /* put it back */
          } else {
                  flag++;
          }
          ap++;
    }
  }

  /*===========================================================================*
   *                              insert_arg                                   *
   *===========================================================================*/
  PRIVATE int insert_arg(stack, stk_bytes, arg, replace)
  char stack[ARG_MAX];            /* pointer to stack image within PM */
  vir_bytes *stk_bytes;           /* size of initial stack */
  char *arg;                      /* argument to prepend/replace as new argv[0] */
  int replace;
  {
  /* Patch the stack so that arg will become argv[0].  Be careful, the stack may
   * be filled with garbage, although it normally looks like this:
   *      nargs argv[0] ... argv[nargs-1] NULL envp[0] ... NULL
   * followed by the strings "pointed" to by the argv[i] and the envp[i].  The
   * pointers are really offsets from the start of stack.
   * Return true iff the operation succeeded.
   */
    int offset, a0, a1, old_bytes = *stk_bytes;

    /* Prepending arg adds at least one string and a zero byte. */
    offset = strlen(arg) + 1;

    a0 = (int) ((char **) stack)[1];      /* argv[0] */
    if (a0 < 4 * PTRSIZE || a0 >= old_bytes) return(FALSE);

    a1 = a0;                      /* a1 will point to the strings to be moved */
    if (replace) {
          /* Move a1 to the end of argv[0][] (argv[1] if nargs > 1). */
          do {
                  if (a1 == old_bytes) return(FALSE);
                  --offset;
          } while (stack[a1++] != 0);
    } else {
          offset += PTRSIZE;      /* new argv[0] needs new pointer in argv[] */
          a0 += PTRSIZE;          /* location of new argv[0][]. */
    }

    /* stack will grow by offset bytes (or shrink by -offset bytes) */
    if ((*stk_bytes += offset) > ARG_MAX) return(FALSE);

    /* Reposition the strings by offset bytes */
    memmove(stack + a1 + offset, stack + a1, old_bytes - a1);

    strcpy(stack + a0, arg);      /* Put arg in the new space. */

    if (!replace) {
          /* Make space for a new argv[0]. */
          memmove(stack + 2 * PTRSIZE, stack + 1 * PTRSIZE, a0 - 2 * PTRSIZE);

          ((char **) stack)[0]++; /* nargs++; */
    }
    /* Now patch up argv[] and envp[] by offset. */
    patch_ptr(stack, (vir_bytes) offset);
    ((char **) stack)[1] = (char *) a0;   /* set argv[0] correctly */
    return(TRUE);
  }

  /*===========================================================================*
   *                              patch_stack                                  *
   *===========================================================================*/
  PRIVATE char *patch_stack(fd, stack, stk_bytes, script)
  int fd;                         /* file descriptor to open script file */
  char stack[ARG_MAX];            /* pointer to stack image within PM */
  vir_bytes *stk_bytes;           /* size of initial stack */
  char *script;                   /* name of script to interpret */
  {
  /* Patch the argument vector to include the path name of the script to be
   * interpreted, and all strings on the #! line.  Returns the path name of
   * the interpreter.
   */
    char *sp, *interp = NULL;
    int n;
    enum { INSERT=FALSE, REPLACE=TRUE };

    /* Make script[] the new argv[0]. */
    if (!insert_arg(stack, stk_bytes, script, REPLACE)) return(NULL);

    if (lseek(fd, 2L, 0) == -1                    /* just behind the #! */
      || (n= read(fd, script, PATH_MAX)) < 0      /* read line one */
      || (sp= memchr(script, '\n', n)) == NULL)   /* must be a proper line */
          return(NULL);

    /* Move sp backwards through script[], prepending each string to stack. */
    for (;;) {
          /* skip spaces behind argument. */
          while (sp > script && (*--sp == ' ' || *sp == '\t')) {}
          if (sp == script) break;

          sp[1] = 0;
          /* Move to the start of the argument. */
          while (sp > script && sp[-1] != ' ' && sp[-1] != '\t') --sp;

          interp = sp;
          if (!insert_arg(stack, stk_bytes, sp, INSERT)) return(NULL);
    }

    /* Round *stk_bytes up to the size of a pointer for alignment contraints. */
    *stk_bytes= ((*stk_bytes + PTRSIZE - 1) / PTRSIZE) * PTRSIZE;

    close(fd);
    return(interp);
  }

  /*===========================================================================*
   *                              rw_seg                                       *
   *===========================================================================*/
  PUBLIC void rw_seg(rw, fd, proc, seg, seg_bytes0)
  int rw;                         /* 0 = read, 1 = write */
  int fd;                         /* file descriptor to read from / write to */
  int proc;                       /* process number */
  int seg;                        /* T, D, or S */
  phys_bytes seg_bytes0;          /* how much is to be transferred? */
  {
  /* Transfer text or data from/to a file and copy to/from a process segment.
   * This procedure is a little bit tricky.  The logical way to transfer a
   * segment would be block by block and copying each block to/from the user
   * space one at a time.  This is too slow, so we do something dirty here,
   * namely send the user space and virtual address to the file system in the
   * upper 10 bits of the file descriptor, and pass it the user virtual address
   * instead of a PM address.  The file system extracts these parameters when
   * gets a read or write call from the process manager, which is the only
   * process that is permitted to use this trick.  The file system then copies
   * the whole segment directly to/from user space, bypassing PM completely.
   *
   * The byte count on read is usually smaller than the segment count, because
   * a segment is padded out to a click multiple, and the data segment is only
   * partially initialized.
   */

    int new_fd, bytes, r;
    char *ubuf_ptr;
    struct mem_map *sp = &mproc[proc].mp_seg[seg];
    phys_bytes seg_bytes = seg_bytes0;

    new_fd = (proc << 7) | (seg << 5) | fd;
    ubuf_ptr = (char *) ((vir_bytes) sp->mem_vir << CLICK_SHIFT);

    while (seg_bytes != 0) {
  #define PM_CHUNK_SIZE 8192
          bytes = MIN((INT_MAX / PM_CHUNK_SIZE) * PM_CHUNK_SIZE, seg_bytes);
          if (rw == 0) {
                  r = read(new_fd, ubuf_ptr, bytes);
          } else {
                  r = write(new_fd, ubuf_ptr, bytes);
          }
          if (r != bytes) break;
          ubuf_ptr += bytes;
          seg_bytes -= bytes;
    }
  }

  /*===========================================================================*
   *                              find_share                                   *
   *===========================================================================*/
  PUBLIC struct mproc *find_share(mp_ign, ino, dev, ctime)
  struct mproc *mp_ign;           /* process that should not be looked at */
  ino_t ino;                      /* parameters that uniquely identify a file */
  dev_t dev;
  time_t ctime;
  {
  /* Look for a process that is the file <ino, dev, ctime> in execution.  Don't
   * accidentally "find" mp_ign, because it is the process on whose behalf this
   * call is made.
   */
    struct mproc *sh_mp;
    for (sh_mp = &mproc[0]; sh_mp < &mproc[NR_PROCS]; sh_mp++) {

          if (!(sh_mp->mp_flags & SEPARATE)) continue;
          if (sh_mp == mp_ign) continue;
          if (sh_mp->mp_ino != ino) continue;
          if (sh_mp->mp_dev != dev) continue;
          if (sh_mp->mp_ctime != ctime) continue;
          return sh_mp;
    }
    return(NULL);
  }
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                servers/pm/break.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/* The MINIX model of memory allocation reserves a fixed amount of memory for
 * the combined text, data, and stack segments.  The amount used for a child
 * process created by FORK is the same as the parent had.  If the child does
 * an EXEC later, the new size is taken from the header of the file EXEC'ed.
 *
 * The layout in memory consists of the text segment, followed by the data
 * segment, followed by a gap (unused memory), followed by the stack segment.
 * The data segment grows upward and the stack grows downward, so each can
 * take memory from the gap.  If they meet, the process must be killed.  The
 * procedures in this file deal with the growth of the data and stack segments.
 *
 * The entry points into this file are:
 *   do_brk:      BRK/SBRK system calls to grow or shrink the data segment
 *   adjust:      see if a proposed segment adjustment is allowed
 *   size_ok:     see if the segment sizes are feasible
 */

#include "pm.h"
#include <signal.h>
#include "mproc.h"
```

```
#include "param.h"

#define DATA_CHANGED       1    /* flag value when data segment size changed */
#define STACK_CHANGED      2    /* flag value when stack size changed */

/*===========================================================================*
 *                              do_brk                                       *
 *===========================================================================*/
PUBLIC int do_brk()
{
/* Perform the brk(addr) system call.
 *
 * The call is complicated by the fact that on some machines (e.g., 8088),
 * the stack pointer can grow beyond the base of the stack segment without
 * anybody noticing it.
 * The parameter, 'addr' is the new virtual address in D space.
 */

  register struct mproc *rmp;
  int r;
  vir_bytes v, new_sp;
  vir_clicks new_clicks;

  rmp = mp;
  v = (vir_bytes) m_in.addr;
  new_clicks = (vir_clicks) ( ((long) v + CLICK_SIZE - 1) >> CLICK_SHIFT);
  if (new_clicks < rmp->mp_seg[D].mem_vir) {
        rmp->mp_reply.reply_ptr = (char *) -1;
        return(ENOMEM);
  }
  new_clicks -= rmp->mp_seg[D].mem_vir;
  if ((r=get_stack_ptr(who, &new_sp)) != OK)    /* ask kernel for sp value */
        panic(__FILE__,"couldn't get stack pointer", r);
  r = adjust(rmp, new_clicks, new_sp);
  rmp->mp_reply.reply_ptr = (r == OK ? m_in.addr : (char *) -1);
  return(r);                    /* return new address or -1 */
}

/*===========================================================================*
 *                              adjust                                       *
 *===========================================================================*/
PUBLIC int adjust(rmp, data_clicks, sp)
register struct mproc *rmp;     /* whose memory is being adjusted? */
vir_clicks data_clicks;         /* how big is data segment to become? */
vir_bytes sp;                   /* new value of sp */
{
/* See if data and stack segments can coexist, adjusting them if need be.
 * Memory is never allocated or freed.  Instead it is added or removed from the
 * gap between data segment and stack segment.  If the gap size becomes
 * negative, the adjustment of data or stack fails and ENOMEM is returned.
 */

  register struct mem_map *mem_sp, *mem_dp;
  vir_clicks sp_click, gap_base, lower, old_clicks;
  int changed, r, ft;
  long base_of_stack, delta;    /* longs avoid certain problems */

  mem_dp = &rmp->mp_seg[D];     /* pointer to data segment map */
  mem_sp = &rmp->mp_seg[S];     /* pointer to stack segment map */
  changed = 0;                  /* set when either segment changed */

  if (mem_sp->mem_len == 0) return(OK); /* don't bother init */

  /* See if stack size has gone negative (i.e., sp too close to 0xFFFF...) */
  base_of_stack = (long) mem_sp->mem_vir + (long) mem_sp->mem_len;
  sp_click = sp >> CLICK_SHIFT; /* click containing sp */
  if (sp_click >= base_of_stack) return(ENOMEM);        /* sp too high */

  /* Compute size of gap between stack and data segments. */
  delta = (long) mem_sp->mem_vir - (long) sp_click;
  lower = (delta > 0 ? sp_click : mem_sp->mem_vir);

  /* Add a safety margin for future stack growth. Impossible to do right. */
#define SAFETY_BYTES  (384 * sizeof(char *))
#define SAFETY_CLICKS ((SAFETY_BYTES + CLICK_SIZE - 1) / CLICK_SIZE)
  gap_base = mem_dp->mem_vir + data_clicks + SAFETY_CLICKS;
  if (lower < gap_base) return(ENOMEM); /* data and stack collided */

  /* Update data length (but not data orgin) on behalf of brk() system call. */
  old_clicks = mem_dp->mem_len;
  if (data_clicks != mem_dp->mem_len) {
        mem_dp->mem_len = data_clicks;
        changed |= DATA_CHANGED;
  }

  /* Update stack length and origin due to change in stack pointer. */
  if (delta > 0) {
        mem_sp->mem_vir -= delta;
        mem_sp->mem_phys -= delta;
        mem_sp->mem_len += delta;
        changed |= STACK_CHANGED;
  }

  /* Do the new data and stack segment sizes fit in the address space? */
  ft = (rmp->mp_flags & SEPARATE);
  r = (rmp->mp_seg[D].mem_vir + rmp->mp_seg[D].mem_len >
          rmp->mp_seg[S].mem_vir) ? ENOMEM : OK;
  if (r == OK) {
        if (changed) sys_newmap((int)(rmp - mproc), rmp->mp_seg);
        return(OK);
  }

  /* New sizes don't fit or require too many page/segment registers. Restore.*/
  if (changed & DATA_CHANGED) mem_dp->mem_len = old_clicks;
  if (changed & STACK_CHANGED) {
        mem_sp->mem_vir += delta;
        mem_sp->mem_phys += delta;
        mem_sp->mem_len -= delta;
  }
  return(ENOMEM);
}
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                servers/pm/signal.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

/* This file handles signals, which are asynchronous events and are generally
 * a messy and unpleasant business.  Signals can be generated by the KILL
 * system call, or from the keyboard (SIGINT) or from the clock (SIGALRM).
 * In all cases control eventually passes to check_sig() to see which processes
 * can be signaled.  The actual signaling is done by sig_proc().
 *
 * The entry points into this file are:
 *   do_sigaction:   perform the SIGACTION system call
 *   do_sigpending:  perform the SIGPENDING system call
 *   do_sigprocmask: perform the SIGPROCMASK system call
 *   do_sigreturn:   perform the SIGRETURN system call
 *   do_sigsuspend:  perform the SIGSUSPEND system call
 *   do_kill:   perform the KILL system call
 *   do_alarm:  perform the ALARM system call by calling set_alarm()
 *   set_alarm: tell the clock task to start or stop a timer
 *   do_pause:  perform the PAUSE system call
 *   ksig_pending: the kernel notified about pending signals
 *   sig_proc:  interrupt or terminate a signaled process
 *   check_sig: check which processes to signal with sig_proc()
 *   check_pending:  check if a pending signal can now be delivered
 */

#include "pm.h"
#include <sys/stat.h>
#include <sys/ptrace.h>
#include <minix/callnr.h>
#include <minix/com.h>
#include <signal.h>
#include <sys/sigcontext.h>
#include <string.h>
#include "mproc.h"
#include "param.h"

#define CORE_MODE       0777    /* mode to use on core image files */
#define DUMPED          0200    /* bit set in status when core dumped */

FORWARD _PROTOTYPE( void dump_core, (struct mproc *rmp)                 );
FORWARD _PROTOTYPE( void unpause, (int pro)                             );
FORWARD _PROTOTYPE( void handle_sig, (int proc_nr, sigset_t sig_map)    );
FORWARD _PROTOTYPE( void cause_sigalrm, (struct timer *tp)              );

/*===========================================================================*
 *                              do_sigaction                                 *
 *===========================================================================*/
PUBLIC int do_sigaction()
{
  int r;
  struct sigaction svec;
  struct sigaction *svp;

  if (m_in.sig_nr == SIGKILL) return(OK);
  if (m_in.sig_nr < 1 || m_in.sig_nr > _NSIG) return (EINVAL);
  svp = &mp->mp_sigact[m_in.sig_nr];
  if ((struct sigaction *) m_in.sig_osa != (struct sigaction *) NULL) {
        r = sys_datacopy(PM_PROC_NR,(vir_bytes) svp,
                who, (vir_bytes) m_in.sig_osa, (phys_bytes) sizeof(svec));
        if (r != OK) return(r);
  }

  if ((struct sigaction *) m_in.sig_nsa == (struct sigaction *) NULL)
        return(OK);

  /* Read in the sigaction structure. */
  r = sys_datacopy(who, (vir_bytes) m_in.sig_nsa,
                PM_PROC_NR, (vir_bytes) &svec, (phys_bytes) sizeof(svec));
  if (r != OK) return(r);

  if (svec.sa_handler == SIG_IGN) {
        sigaddset(&mp->mp_ignore, m_in.sig_nr);
        sigdelset(&mp->mp_sigpending, m_in.sig_nr);
        sigdelset(&mp->mp_catch, m_in.sig_nr);
        sigdelset(&mp->mp_sig2mess, m_in.sig_nr);
  } else if (svec.sa_handler == SIG_DFL) {
        sigdelset(&mp->mp_ignore, m_in.sig_nr);
        sigdelset(&mp->mp_catch, m_in.sig_nr);
        sigdelset(&mp->mp_sig2mess, m_in.sig_nr);
  } else if (svec.sa_handler == SIG_MESS) {
        if (! (mp->mp_flags & PRIV_PROC)) return(EPERM);
        sigdelset(&mp->mp_ignore, m_in.sig_nr);
        sigaddset(&mp->mp_sig2mess, m_in.sig_nr);
        sigdelset(&mp->mp_catch, m_in.sig_nr);
  } else {
        sigdelset(&mp->mp_ignore, m_in.sig_nr);
        sigaddset(&mp->mp_catch, m_in.sig_nr);
        sigdelset(&mp->mp_sig2mess, m_in.sig_nr);
  }
  mp->mp_sigact[m_in.sig_nr].sa_handler = svec.sa_handler;
  sigdelset(&svec.sa_mask, SIGKILL);
  mp->mp_sigact[m_in.sig_nr].sa_mask = svec.sa_mask;
  mp->mp_sigact[m_in.sig_nr].sa_flags = svec.sa_flags;
  mp->mp_sigreturn = (vir_bytes) m_in.sig_ret;
  return(OK);
}

/*===========================================================================*
 *                              do_sigpending                                *
 *===========================================================================*/
PUBLIC int do_sigpending()
{
  mp->mp_reply.reply_mask = (long) mp->mp_sigpending;
  return OK;
}

/*===========================================================================*
 *                              do_sigprocmask                               *
 *===========================================================================*/
PUBLIC int do_sigprocmask()
{
/* Note that the library interface passes the actual mask in sigmask_set,
 * not a pointer to the mask, in order to save a copy.  Similarly,
 * the old mask is placed in the return message which the library
 * interface copies (if requested) to the user specified address.
 *
 * The library interface must set SIG_INQUIRE if the 'act' argument
 * is NULL.
 */

  int i;

  mp->mp_reply.reply_mask = (long) mp->mp_sigmask;

  switch (m_in.sig_how) {
      case SIG_BLOCK:
        sigdelset((sigset_t *)&m_in.sig_set, SIGKILL);
        for (i = 1; i <= _NSIG; i++) {
                if (sigismember((sigset_t *)&m_in.sig_set, i))
                        sigaddset(&mp->mp_sigmask, i);
        }
        break;

      case SIG_UNBLOCK:
        for (i = 1; i <= _NSIG; i++) {
                if (sigismember((sigset_t *)&m_in.sig_set, i))
                        sigdelset(&mp->mp_sigmask, i);
        }
        check_pending(mp);
        break;

      case SIG_SETMASK:
        sigdelset((sigset_t *) &m_in.sig_set, SIGKILL);
        mp->mp_sigmask = (sigset_t) m_in.sig_set;
        check_pending(mp);
        break;

      case SIG_INQUIRE:
        break;

      default:
        return(EINVAL);
        break;
  }
  return OK;
}

/*===========================================================================*
 *                              do_sigsuspend                                *
 *===========================================================================*/
PUBLIC int do_sigsuspend()
{
  mp->mp_sigmask2 = mp->mp_sigmask;     /* save the old mask */
  mp->mp_sigmask = (sigset_t) m_in.sig_set;
  sigdelset(&mp->mp_sigmask, SIGKILL);
  mp->mp_flags |= SIGSUSPENDED;
  check_pending(mp);
  return(SUSPEND);
}

/*===========================================================================*
 *                              do_sigreturn                                 *
 *===========================================================================*/
PUBLIC int do_sigreturn()
{
/* A user signal handler is done.  Restore context and check for
 * pending unblocked signals.
 */
```

```

  int r;

  mp->mp_sigmask = (sigset_t) m_in.sig_set;
  sigdelset(&mp->mp_sigmask, SIGKILL);

  r = sys_sigreturn(who, (struct sigmsg *) m_in.sig_context);
  check_pending(mp);
  return(r);
}

/*===========================================================================*
 *                              do_kill                                      *
 *===========================================================================*/
PUBLIC int do_kill()
{
/* Perform the kill(pid, signo) system call. */

  return check_sig(m_in.pid, m_in.sig_nr);
}

/*===========================================================================*
 *                              ksig_pending                                 *
 *===========================================================================*/
PUBLIC int ksig_pending()
{
/* Certain signals, such as segmentation violations originate in the kernel.
 * When the kernel detects such signals, it notifies the PM to take further
 * action. The PM requests the kernel to send messages with the process
 * slot and bit map for all signaled processes. The File System, for example,
 * uses this mechanism to signal writing on broken pipes (SIGPIPE).
 *
 * The kernel has notified the PM about pending signals. Request pending
 * signals until all signals are handled. If there are no more signals,
 * NONE is returned in the process number field.
 */
 int proc_nr;
 sigset_t sig_map;

 while (TRUE) {
   sys_getksig(&proc_nr, &sig_map);     /* get an arbitrary pending signal */
   if (NONE == proc_nr) {               /* stop if no more pending signals */
        break;
   } else {
        handle_sig(proc_nr, sig_map);   /* handle the received signal */
        sys_endksig(proc_nr);           /* tell kernel it's done */
   }
 }
 return(SUSPEND);                       /* prevents sending reply */
}

/*===========================================================================*
 *                              handle_sig                                   *
 *===========================================================================*/
PRIVATE void handle_sig(proc_nr, sig_map)
int proc_nr;
sigset_t sig_map;
{
  register struct mproc *rmp;
  int i;
  pid_t proc_id, id;

  rmp = &mproc[proc_nr];
  if ((rmp->mp_flags & (IN_USE | ZOMBIE)) != IN_USE) return;
  proc_id = rmp->mp_pid;
  mp = &mproc[0];                       /* pretend signals are from PM */
  mp->mp_procgrp = rmp->mp_procgrp;     /* get process group right */

  /* Check each bit in turn to see if a signal is to be sent.  Unlike
   * kill(), the kernel may collect several unrelated signals for a
   * process and pass them to PM in one blow.  Thus loop on the bit
   * map. For SIGINT and SIGQUIT, use proc_id 0 to indicate a broadcast
   * to the recipient's process group.  For SIGKILL, use proc_id -1 to
   * indicate a systemwide broadcast.
   */
  for (i = 1; i <= _NSIG; i++) {
        if (!sigismember(&sig_map, i)) continue;
        switch (i) {
            case SIGINT:
            case SIGQUIT:
                id = 0; break;  /* broadcast to process group */
            case SIGKILL:
                id = -1; break; /* broadcast to all except INIT */
            default:
                id = proc_id;
                break;
        }
        check_sig(id, i);
  }
}

/*===========================================================================*
 *                              do_alarm                                     *
 *===========================================================================*/
PUBLIC int do_alarm()
{
/* Perform the alarm(seconds) system call. */
  return(set_alarm(who, m_in.seconds));
}

/*===========================================================================*
 *                              set_alarm                                    *
 *===========================================================================*/
PUBLIC int set_alarm(proc_nr, sec)
int proc_nr;                    /* process that wants the alarm */
int sec;                        /* how many seconds delay before the signal */
{
/* This routine is used by do_alarm() to set the alarm timer.  It is also used
 * to turn the timer off when a process exits with the timer still on.
 */
  clock_t ticks;        /* number of ticks for alarm */
  clock_t exptime;      /* needed for remaining time on previous alarm */
  clock_t uptime;       /* current system time */
  int remaining;        /* previous time left in seconds */
  int s;

  /* First determine remaining time of previous alarm, if set. */
  if (mproc[proc_nr].mp_flags & ALARM_ON) {
        if ( (s=getuptime(&uptime)) != OK)
                panic(__FILE__,"set_alarm couldn't get uptime", s);
        exptime = *tmr_exp_time(&mproc[proc_nr].mp_timer);
        remaining = (int) ((exptime - uptime + (HZ-1))/HZ);
        if (remaining < 0) remaining = 0;
  } else {
        remaining = 0;
  }

  /* Tell the clock task to provide a signal message when the time comes.
   *
   * Large delays cause a lot of problems.  First, the alarm system call
   * takes an unsigned seconds count and the library has cast it to an int.
   * That probably works, but on return the library will convert "negative"
   * unsigneds to errors.  Presumably no one checks for these errors, so
   * force this call through.  Second, If unsigned and long have the same
   * size, converting from seconds to ticks can easily overflow.  Finally,
   * the kernel has similar overflow bugs adding ticks.
   *
   * Fixing this requires a lot of ugly casts to fit the wrong interface
   * types and to avoid overflow traps.  ALRM_EXP_TIME has the right type
   * (clock_t) although it is declared as long.  How can variables like
   * this be declared properly without combinatorial explosion of message
   * types?
   */
  ticks = (clock_t) (HZ * (unsigned long) (unsigned) sec);
  if ( (unsigned long) ticks / HZ != (unsigned) sec)
        ticks = LONG_MAX;       /* eternity (really TMR_NEVER) */

  if (ticks != 0) {
        pm_set_timer(&mproc[proc_nr].mp_timer, ticks, cause_sigalrm, proc_nr);
        mproc[proc_nr].mp_flags |=  ALARM_ON;
  } else if (mproc[proc_nr].mp_flags & ALARM_ON) {
        pm_cancel_timer(&mproc[proc_nr].mp_timer);
        mproc[proc_nr].mp_flags &= ~ALARM_ON;
  }
  return(remaining);
}

/*===========================================================================*
 *                              cause_sigalrm                                *
 *===========================================================================*/
PRIVATE void cause_sigalrm(tp)
struct timer *tp;
{
  int proc_nr;
  register struct mproc *rmp;

  proc_nr = tmr_arg(tp)->ta_int;        /* get process from timer */
  rmp = &mproc[proc_nr];

  if ((rmp->mp_flags & (IN_USE | ZOMBIE)) != IN_USE) return;
  if ((rmp->mp_flags & ALARM_ON) == 0) return;
  rmp->mp_flags &= ~ALARM_ON;
  check_sig(rmp->mp_pid, SIGALRM);
}

/*===========================================================================*
 *                              do_pause                                     *
 *===========================================================================*/
PUBLIC int do_pause()
{
```

```
/* Perform the pause() system call. */

  mp->mp_flags |= PAUSED;
  return(SUSPEND);
}

/*===========================================================================*
 *                              sig_proc                                     *
 *===========================================================================*/
PUBLIC void sig_proc(rmp, signo)
register struct mproc *rmp;     /* pointer to the process to be signaled */
int signo;                      /* signal to send to process (1 to _NSIG) */
{
/* Send a signal to a process.  Check to see if the signal is to be caught,
 * ignored, tranformed into a message (for system processes) or blocked.
 *  - If the signal is to be transformed into a message, request the KERNEL to
 * send the target process a system notification with the pending signal as an
 * argument.
 *  - If the signal is to be caught, request the KERNEL to push a sigcontext
 * structure and a sigframe structure onto the catcher's stack.  Also, KERNEL
 * will reset the program counter and stack pointer, so that when the process
 * next runs, it will be executing the signal handler. When the signal handler
 * returns,  sigreturn(2) will be called.  Then KERNEL will restore the signal
 * context from the sigcontext structure.
 * If there is insufficient stack space, kill the process.
 */

  vir_bytes new_sp;
  int s;
  int slot;
  int sigflags;
  struct sigmsg sm;

  slot = (int) (rmp - mproc);
  if ((rmp->mp_flags & (IN_USE | ZOMBIE)) != IN_USE) {
        printf("PM: signal %d sent to %s process %d\n",
                signo, (rmp->mp_flags & ZOMBIE) ? "zombie" : "dead", slot);
        panic(__FILE__,"", NO_NUM);
  }
  if ((rmp->mp_flags & TRACED) && signo != SIGKILL) {
        /* A traced process has special handling. */
        unpause(slot);
        stop_proc(rmp, signo);  /* a signal causes it to stop */
        return;
  }
  /* Some signals are ignored by default. */
  if (sigismember(&rmp->mp_ignore, signo)) {
        return;
  }
  if (sigismember(&rmp->mp_sigmask, signo)) {
        /* Signal should be blocked. */
        sigaddset(&rmp->mp_sigpending, signo);
        return;
  }
  sigflags = rmp->mp_sigact[signo].sa_flags;
  if (sigismember(&rmp->mp_catch, signo)) {
        if (rmp->mp_flags & SIGSUSPENDED)
                sm.sm_mask = rmp->mp_sigmask2;
        else
                sm.sm_mask = rmp->mp_sigmask;
        sm.sm_signo = signo;
        sm.sm_sighandler = (vir_bytes) rmp->mp_sigact[signo].sa_handler;
        sm.sm_sigreturn = rmp->mp_sigreturn;
        if ((s=get_stack_ptr(slot, &new_sp)) != OK)
                panic(__FILE__,"couldn't get new stack pointer",s);
        sm.sm_stkptr = new_sp;

        /* Make room for the sigcontext and sigframe struct. */
        new_sp -= sizeof(struct sigcontext)
                                 + 3 * sizeof(char *) + 2 * sizeof(int);

        if (adjust(rmp, rmp->mp_seg[D].mem_len, new_sp) != OK)
                goto doterminate;

        rmp->mp_sigmask |= rmp->mp_sigact[signo].sa_mask;
        if (sigflags & SA_NODEFER)
                sigdelset(&rmp->mp_sigmask, signo);
        else
                sigaddset(&rmp->mp_sigmask, signo);

        if (sigflags & SA_RESETHAND) {
                sigdelset(&rmp->mp_catch, signo);
                rmp->mp_sigact[signo].sa_handler = SIG_DFL;
        }

        if (OK == (s=sys_sigsend(slot, &sm))) {

                sigdelset(&rmp->mp_sigpending, signo);
                /* If process is hanging on PAUSE, WAIT, SIGSUSPEND, tty,
                 * pipe, etc., release it.
                 */
                unpause(slot);
                return;
        }
        panic(__FILE__, "warning, sys_sigsend failed", s);
  }
  else if (sigismember(&rmp->mp_sig2mess, signo)) {
        if (OK != (s=sys_kill(slot,signo)))
                panic(__FILE__, "warning, sys_kill failed", s);
        return;
  }

doterminate:
  /* Signal should not or cannot be caught.  Take default action. */
  if (sigismember(&ign_sset, signo)) return;

  rmp->mp_sigstatus = (char) signo;
  if (sigismember(&core_sset, signo)) {
        /* Switch to the user's FS environment and dump core. */
        tell_fs(CHDIR, slot, FALSE, 0);
        dump_core(rmp);
  }
  pm_exit(rmp, 0);              /* terminate process */
}

/*===========================================================================*
 *                              check_sig                                    *
 *===========================================================================*/
PUBLIC int check_sig(proc_id, signo)
pid_t proc_id;                  /* pid of proc to sig, or 0 or -1, or -pgrp */
```

```
int signo;                      /* signal to send to process (0 to _NSIG) */
{
/* Check to see if it is possible to send a signal.  The signal may have to be
 * sent to a group of processes.  This routine is invoked by the KILL system
 * call, and also when the kernel catches a DEL or other signal.
 */

  register struct mproc *rmp;
  int count;                    /* count # of signals sent */
  int error_code;

  if (signo < 0 || signo > _NSIG) return(EINVAL);

  /* Return EINVAL for attempts to send SIGKILL to INIT alone. */
  if (proc_id == INIT_PID && signo == SIGKILL) return(EINVAL);

  /* Search the proc table for processes to signal.  (See forkexit.c about
   * pid magic.)
   */
  count = 0;
  error_code = ESRCH;
  for (rmp = &mproc[0]; rmp < &mproc[NR_PROCS]; rmp++) {
        if (!(rmp->mp_flags & IN_USE)) continue;
        if ((rmp->mp_flags & ZOMBIE) && signo != 0) continue;

        /* Check for selection. */
        if (proc_id > 0 && proc_id != rmp->mp_pid) continue;
        if (proc_id == 0 && mp->mp_procgrp != rmp->mp_procgrp) continue;
        if (proc_id == -1 && rmp->mp_pid <= INIT_PID) continue;
        if (proc_id < -1 && rmp->mp_procgrp != -proc_id) continue;

        /* Check for permission. */
        if (mp->mp_effuid != SUPER_USER
            && mp->mp_realuid != rmp->mp_realuid
            && mp->mp_effuid != rmp->mp_realuid
            && mp->mp_realuid != rmp->mp_effuid
            && mp->mp_effuid != rmp->mp_effuid) {
                error_code = EPERM;
                continue;
        }

        count++;
        if (signo == 0) continue;

        /* 'sig_proc' will handle the disposition of the signal.  The
         * signal may be caught, blocked, ignored, or cause process
         * termination, possibly with core dump.
         */
        sig_proc(rmp, signo);

        if (proc_id > 0) break; /* only one process being signaled */
  }

  /* If the calling process has killed itself, don't reply. */
  if ((mp->mp_flags & (IN_USE | ZOMBIE)) != IN_USE) return(SUSPEND);
  return(count > 0 ? OK : error_code);
}
```

```
/*===========================================================================*
 *                              check_pending                                *
 *===========================================================================*/
PUBLIC void check_pending(rmp)
register struct mproc *rmp;
{
  /* Check to see if any pending signals have been unblocked.  The
   * first such signal found is delivered.
   *
   * If multiple pending unmasked signals are found, they will be
   * delivered sequentially.
   *
   * There are several places in this file where the signal mask is
   * changed.  At each such place, check_pending() should be called to
   * check for newly unblocked signals.
   */

  int i;

  for (i = 1; i <= _NSIG; i++) {
        if (sigismember(&rmp->mp_sigpending, i) &&
                !sigismember(&rmp->mp_sigmask, i)) {
                sigdelset(&rmp->mp_sigpending, i);
                sig_proc(rmp, i);
                break;
        }
  }
}

/*===========================================================================*
 *                              unpause                                      *
 *===========================================================================*/
PRIVATE void unpause(pro)
int pro;                        /* which process number */
{
/* A signal is to be sent to a process.  If that process is hanging on a
 * system call, the system call must be terminated with EINTR.  Possible
 * calls are PAUSE, WAIT, READ and WRITE, the latter two for pipes and ttys.
 * First check if the process is hanging on an PM call.  If not, tell FS,
 * so it can check for READs and WRITEs from pipes, ttys and the like.
 */

  register struct mproc *rmp;

  rmp = &mproc[pro];

  /* Check to see if process is hanging on a PAUSE, WAIT or SIGSUSPEND call. */
  if (rmp->mp_flags & (PAUSED | WAITING | SIGSUSPENDED)) {
        rmp->mp_flags &= ~(PAUSED | WAITING | SIGSUSPENDED);
        setreply(pro, EINTR);
        return;
  }

  /* Process is not hanging on an PM call.  Ask FS to take a look. */
  tell_fs(UNPAUSE, pro, 0, 0);
}
```

```
/*===========================================================================*
 *                              dump_core                                    *
 *===========================================================================*/
PRIVATE void dump_core(rmp)
register struct mproc *rmp;     /* whose core is to be dumped */
{
/* Make a core dump on the file "core", if possible. */

  int s, fd, seg, slot;
  vir_bytes current_sp;
  long trace_data, trace_off;

  slot = (int) (rmp - mproc);

  /* Can core file be written?  We are operating in the user's FS environment,
   * so no special permission checks are needed.
   */
  if (rmp->mp_realuid != rmp->mp_effuid) return;
  if ( (fd = open(core_name, O_WRONLY | O_CREAT | O_TRUNC | O_NONBLOCK,
                                                CORE_MODE)) < 0) return;
  rmp->mp_sigstatus |= DUMPED;

  /* Make sure the stack segment is up to date.
   * We don't want adjust() to fail unless current_sp is preposterous,
   * but it might fail due to safety checking.  Also, we don't really want
   * the adjust() for sending a signal to fail due to safety checking.
   * Maybe make SAFETY_BYTES a parameter.
   */
  if ((s=get_stack_ptr(slot, &current_sp)) != OK)
        panic(__FILE__,"couldn't get new stack pointer",s);
  adjust(rmp, rmp->mp_seg[D].mem_len, current_sp);

  /* Write the memory map of all segments to begin the core file. */
  if (write(fd, (char *) rmp->mp_seg, (unsigned) sizeof rmp->mp_seg)
      != (unsigned) sizeof rmp->mp_seg) {
        close(fd);
        return;
  }

  /* Write out the whole kernel process table entry to get the regs. */
  trace_off = 0;
  while (sys_trace(T_GETUSER, slot, trace_off, &trace_data) == OK) {
        if (write(fd, (char *) &trace_data, (unsigned) sizeof (long))
            != (unsigned) sizeof (long)) {
                close(fd);
                return;
        }
        trace_off += sizeof (long);
  }

  /* Loop through segments and write the segments themselves out. */
  for (seg = 0; seg < NR_LOCAL_SEGS; seg++) {
        rw_seg(1, fd, slot, seg,
                (phys_bytes) rmp->mp_seg[seg].mem_len << CLICK_SHIFT);
  }
  close(fd);
}
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                servers/pm/timers.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

/* PM watchdog timer management. These functions in this file provide
 * a convenient interface to the timers library that manages a list of
 * watchdog timers. All details of scheduling an alarm at the CLOCK task
 * are hidden behind this interface.
 * Only system processes are allowed to set an alarm timer at the kernel.
 * Therefore, the PM maintains a local list of timers for user processes
 * that requested an alarm signal.
 *
 * The entry points into this file are:
 *   pm_set_timer:      reset and existing or set a new watchdog timer
 *   pm_expire_timers:  check for expired timers and run watchdog functions
 *   pm_cancel_timer:   remove a time from the list of timers
 *
 */

#include "pm.h"

#include <timers.h>
#include <minix/syslib.h>
#include <minix/com.h>

PRIVATE timer_t *pm_timers = NULL;

/*===========================================================================*
 *                              pm_set_timer                                 *
 *===========================================================================*/
PUBLIC void pm_set_timer(timer_t *tp, int ticks, tmr_func_t watchdog, int arg)
{
        int r;
        clock_t now, prev_time = 0, next_time;

        if ((r = getuptime(&now)) != OK)
                panic(__FILE__, "PM couldn't get uptime", NO_NUM);

        /* Set timer argument and add timer to the list. */
        tmr_arg(tp)->ta_int = arg;
        prev_time = tmrs_settimer(&pm_timers,tp,now+ticks,watchdog,&next_time);

        /* Reschedule our synchronous alarm if necessary. */
        if (! prev_time || prev_time > next_time) {
                if (sys_setalarm(next_time, 1) != OK)
                        panic(__FILE__, "PM set timer couldn't set alarm.", NO_NUM);
        }

        return;
}

/*===========================================================================*
 *                              pm_expire_timers                             *
 *===========================================================================*/
PUBLIC void pm_expire_timers(clock_t now)
{
        clock_t next_time;

        /* Check for expired timers and possibly reschedule an alarm. */
        tmrs_exptimers(&pm_timers, now, &next_time);
        if (next_time > 0) {
                if (sys_setalarm(next_time, 1) != OK)
                        panic(__FILE__, "PM expire timer couldn't set alarm.", NO_NUM);
        }
}

/*===========================================================================*
 *                              pm_cancel_timer                              *
 *===========================================================================*/
PUBLIC void pm_cancel_timer(timer_t *tp)
{
        clock_t next_time, prev_time;
        prev_time = tmrs_clrtimer(&pm_timers, tp, &next_time);

        /* If the earliest timer has been removed, we have to set the alarm to
     * the next timer, or cancel the alarm altogether if the last timer has
     * been cancelled (next_time will be 0 then).
         */
        if (prev_time < next_time || ! next_time) {
                if (sys_setalarm(next_time, 1) != OK)
                        panic(__FILE__, "PM expire timer couldn't set alarm.", NO_NUM);
        }
}
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                servers/pm/time.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/* This file takes care of those system calls that deal with time.
 *
 * The entry points into this file are
 *   do_time:           perform the TIME system call
 *   do_stime:          perform the STIME system call
 *   do_times:          perform the TIMES system call
 */

#include "pm.h"
#include <minix/callnr.h>
#include <minix/com.h>
#include <signal.h>
#include "mproc.h"
#include "param.h"

PRIVATE time_t boottime;

/*===========================================================================*
 *                              do_time                                      *
 *===========================================================================*/
PUBLIC int do_time()
{
/* Perform the time(tp) system call. This returns the time in seconds since
 * 1.1.1970.  MINIX is an astrophysically naive system that assumes the earth
 * rotates at a constant rate and that such things as leap seconds do not
 * exist.
 */
  clock_t uptime;
  int s;

  if ( (s=getuptime(&uptime)) != OK)
        panic(__FILE__,"do_time couldn't get uptime", s);

  mp->mp_reply.reply_time = (time_t) (boottime + (uptime/HZ));
  mp->mp_reply.reply_utime = (uptime%HZ)*1000000/HZ;
  return(OK);
}

/*===========================================================================*
 *                              do_stime                                     *
 *===========================================================================*/
PUBLIC int do_stime()
{
/* Perform the stime(tp) system call. Retrieve the system's uptime (ticks
 * since boot) and store the time in seconds at system boot in the global
 * variable 'boottime'.
 */
  clock_t uptime;
  int s;

  if (mp->mp_effuid != SUPER_USER) {
      return(EPERM);
  }
  if ( (s=getuptime(&uptime)) != OK)
      panic(__FILE__,"do_stime couldn't get uptime", s);
  boottime = (long) m_in.stime - (uptime/HZ);

  /* Also inform FS about the new system time. */
  tell_fs(STIME, boottime, 0, 0);

  return(OK);
}

/*===========================================================================*
 *                              do_times                                     *
 *===========================================================================*/
PUBLIC int do_times()
{
/* Perform the times(buffer) system call. */
  register struct mproc *rmp = mp;
  clock_t t[5];
  int s;

  if (OK != (s=sys_times(who, t)))
      panic(__FILE__,"do_times couldn't get times", s);
  rmp->mp_reply.reply_t1 = t[0];                /* user time */
  rmp->mp_reply.reply_t2 = t[1];                /* system time */
  rmp->mp_reply.reply_t3 = rmp->mp_child_utime; /* child user time */
  rmp->mp_reply.reply_t4 = rmp->mp_child_stime; /* child system time */
  rmp->mp_reply.reply_t5 = t[4];                /* uptime since boot */

  return(OK);
}
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                servers/pm/getset.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

/* This file handles the 4 system calls that get and set uids and gids.
 * It also handles getpid(), setsid(), and getpgrp().  The code for each
 * one is so tiny that it hardly seemed worthwhile to make each a separate
 * function.
 */

#include "pm.h"
#include <minix/callnr.h>
#include <signal.h>
#include "mproc.h"
#include "param.h"

/*===========================================================================*
 *                              do_getset                                    *
 *===========================================================================*/
PUBLIC int do_getset()
{
/* Handle GETUID, GETGID, GETPID, GETPGRP, SETUID, SETGID, SETSID.  The four
 * GETs and SETSID return their primary results in 'r'.  GETUID, GETGID, and
 * GETPID also return secondary results (the effective IDs, or the parent
 * process ID) in 'reply_res2', which is returned to the user.
 */

  register struct mproc *rmp = mp;
  register int r;

  switch(call_nr) {
        case GETUID:
                r = rmp->mp_realuid;
                rmp->mp_reply.reply_res2 = rmp->mp_effuid;
                break;

        case GETGID:
                r = rmp->mp_realgid;
                rmp->mp_reply.reply_res2 = rmp->mp_effgid;
                break;

        case GETPID:
                r = mproc[who].mp_pid;
                rmp->mp_reply.reply_res2 = mproc[rmp->mp_parent].mp_pid;
                break;

        case SETUID:
                if (rmp->mp_realuid != (uid_t) m_in.usr_id &&
                                rmp->mp_effuid != SUPER_USER)
                        return(EPERM);
                rmp->mp_realuid = (uid_t) m_in.usr_id;
                rmp->mp_effuid = (uid_t) m_in.usr_id;
                tell_fs(SETUID, who, rmp->mp_realuid, rmp->mp_effuid);
                r = OK;
                break;

        case SETGID:
                if (rmp->mp_realgid != (gid_t) m_in.grp_id &&
                                rmp->mp_effuid != SUPER_USER)
                        return(EPERM);
                rmp->mp_realgid = (gid_t) m_in.grp_id;
                rmp->mp_effgid = (gid_t) m_in.grp_id;
                tell_fs(SETGID, who, rmp->mp_realgid, rmp->mp_effgid);
                r = OK;
                break;

        case SETSID:
                if (rmp->mp_procgrp == rmp->mp_pid) return(EPERM);
                rmp->mp_procgrp = rmp->mp_pid;
                tell_fs(SETSID, who, 0, 0);
                /* fall through */

        case GETPGRP:
                r = rmp->mp_procgrp;
                break;

        default:
                r = EINVAL;
                break;
  }
  return(r);
}
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                    servers/pm/misc.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/* Miscellaneous system calls.                          Author: Kees J. Bot
 *                                                              31 Mar 2000
 * The entry points into this file are:
 *   do_reboot: kill all processes, then reboot system
 *   do_svrctl: process manager control
 *   do_getsysinfo: request copy of PM data structure  (Jorrit N. Herder)
 *   do_getprocnr: lookup process slot number  (Jorrit N. Herder)
 *   do_memalloc: allocate a chunk of memory  (Jorrit N. Herder)
 *   do_memfree: deallocate a chunk of memory  (Jorrit N. Herder)
 *   do_getsetpriority: get/set process priority
 */

#include "pm.h"
#include <minix/callnr.h>
#include <signal.h>
#include <sys/svrctl.h>
#include <sys/resource.h>
#include <minix/com.h>
#include <string.h>
#include "mproc.h"
#include "param.h"

/*===========================================================================*
 *                              do_allocmem                                  *
 *===========================================================================*/
PUBLIC int do_allocmem()
{
  vir_clicks mem_clicks;
  phys_clicks mem_base;

  mem_clicks = (m_in.memsize + CLICK_SIZE -1 ) >> CLICK_SHIFT;
  mem_base = alloc_mem(mem_clicks);
  if (mem_base == NO_MEM) return(ENOMEM);
  mp->mp_reply.membase =  (phys_bytes) (mem_base << CLICK_SHIFT);
  return(OK);
}

/*===========================================================================*
 *                              do_freemem                                   *
 *===========================================================================*/
PUBLIC int do_freemem()
{
  vir_clicks mem_clicks;
  phys_clicks mem_base;

  mem_clicks = (m_in.memsize + CLICK_SIZE -1 ) >> CLICK_SHIFT;
  mem_base = (m_in.membase + CLICK_SIZE -1 ) >> CLICK_SHIFT;
  free_mem(mem_base, mem_clicks);
  return(OK);
}

/*===========================================================================*
 *                              do_getsysinfo                                *
 *===========================================================================*/
PUBLIC int do_getsysinfo()
{
  struct mproc *proc_addr;
  vir_bytes src_addr, dst_addr;
  struct kinfo kinfo;
  size_t len;
  int s;

  switch(m_in.info_what) {
  case SI_KINFO:                        /* kernel info is obtained via PM */
        sys_getkinfo(&kinfo);
        src_addr = (vir_bytes) &kinfo;
        len = sizeof(struct kinfo);
        break;
  case SI_PROC_ADDR:                    /* get address of PM process table */
        proc_addr = &mproc[0];
        src_addr = (vir_bytes) &proc_addr;
        len = sizeof(struct mproc *);
        break;
  case SI_PROC_TAB:                     /* copy entire process table */
        src_addr = (vir_bytes) mproc;
        len = sizeof(struct mproc) * NR_PROCS;
        break;
  default:
        return(EINVAL);
  }

  dst_addr = (vir_bytes) m_in.info_where;
  if (OK != (s=sys_datacopy(SELF, src_addr, who, dst_addr, len)))
        return(s);
  return(OK);
}
```

```
/*===========================================================================*
 *                              do_getprocnr                                 *
 *===========================================================================*/
PUBLIC int do_getprocnr()
{
  register struct mproc *rmp;
  static char search_key[PROC_NAME_LEN+1];
  int key_len;
  int s;

  if (m_in.pid >= 0) {                          /* lookup process by pid */
        for (rmp = &mproc[0]; rmp < &mproc[NR_PROCS]; rmp++) {
                if ((rmp->mp_flags & IN_USE) && (rmp->mp_pid==m_in.pid)) {
                        mp->mp_reply.procnr = (int) (rmp - mproc);
                        return(OK);
                }
        }
        return(ESRCH);
  } else if (m_in.namelen > 0) {                /* lookup process by name */
        key_len = MIN(m_in.namelen, PROC_NAME_LEN);
        if (OK != (s=sys_datacopy(who, (vir_bytes) m_in.addr,
                        SELF, (vir_bytes) search_key, key_len)))
                return(s);
        search_key[key_len] = '\0';     /* terminate for safety */
        for (rmp = &mproc[0]; rmp < &mproc[NR_PROCS]; rmp++) {
                if ((rmp->mp_flags & IN_USE) &&
                        strncmp(rmp->mp_name, search_key, key_len)==0) {
                        mp->mp_reply.procnr = (int) (rmp - mproc);
                        return(OK);
                }
        }
        return(ESRCH);
  } else {                              /* return own process number */
        mp->mp_reply.procnr = who;
  }
  return(OK);
}

/*===========================================================================*
 *                              do_reboot                                    *
 *===========================================================================*/
#define REBOOT_CODE     "delay; boot"
PUBLIC int do_reboot()
{
  char monitor_code[32*sizeof(char *)];
  int code_len;
  int abort_flag;

  if (mp->mp_effuid != SUPER_USER) return(EPERM);

  switch (m_in.reboot_flag) {
  case RBT_HALT:
  case RBT_PANIC:
  case RBT_RESET:
        abort_flag = m_in.reboot_flag;
        break;
  case RBT_REBOOT:
        code_len = strlen(REBOOT_CODE) + 1;
        strncpy(monitor_code, REBOOT_CODE, code_len);
        abort_flag = RBT_MONITOR;
```

```
                break;
          case RBT_MONITOR:
                code_len = m_in.reboot_strlen + 1;
                if (code_len > sizeof(monitor_code)) return(EINVAL);
                if (sys_datacopy(who, (vir_bytes) m_in.reboot_code,
                        PM_PROC_NR, (vir_bytes) monitor_code,
                        (phys_bytes) (code_len)) != OK) return(EFAULT);
                if (monitor_code[code_len-1] != 0) return(EINVAL);
                abort_flag = RBT_MONITOR;
                break;
          default:
                return(EINVAL);
          }

          check_sig(-1, SIGKILL);               /* kill all processes except init */
          tell_fs(REBOOT,0,0,0);                /* tell FS to prepare for shutdown */

          /* Ask the kernel to abort. All system services, including the PM, will
           * get a HARD_STOP notification. Await the notification in the main loop.
           */
          sys_abort(abort_flag, PM_PROC_NR, monitor_code, code_len);
          return(SUSPEND);                      /* don't reply to killed process */
        }

        /*===========================================================================*
         *                              do_getsetpriority                            *
         *===========================================================================*/
        PUBLIC int do_getsetpriority()
        {
                int arg_which, arg_who, arg_pri;
                int rmp_nr;
                struct mproc *rmp;

                arg_which = m_in.m1_i1;
                arg_who = m_in.m1_i2;
                arg_pri = m_in.m1_i3;   /* for SETPRIORITY */

                /* Code common to GETPRIORITY and SETPRIORITY. */

                /* Only support PRIO_PROCESS for now. */
                if (arg_which != PRIO_PROCESS)
                        return(EINVAL);

                if (arg_who == 0)
                        rmp_nr = who;
                else
                        if ((rmp_nr = proc_from_pid(arg_who)) < 0)
                                return(ESRCH);

                rmp = &mproc[rmp_nr];

                if (mp->mp_effuid != SUPER_USER &&
                   mp->mp_effuid != rmp->mp_effuid && mp->mp_effuid != rmp->mp_realuid)
                        return EPERM;

                /* If GET, that's it. */
                if (call_nr == GETPRIORITY) {
                        return(rmp->mp_nice - PRIO_MIN);
                }

                /* Only root is allowed to reduce the nice level. */
                if (rmp->mp_nice > arg_pri && mp->mp_effuid != SUPER_USER)
                        return(EACCES);

                /* We're SET, and it's allowed. Do it and tell kernel. */
                rmp->mp_nice = arg_pri;
                return sys_nice(rmp_nr, arg_pri);
        }

        /*===========================================================================*
         *                              do_svrctl                                    *
         *===========================================================================*/
        PUBLIC int do_svrctl()
        {
          int s, req;
          vir_bytes ptr;
        #define MAX_LOCAL_PARAMS 2
          static struct {
                char name[30];
                char value[30];
          } local_param_overrides[MAX_LOCAL_PARAMS];
          static int local_params = 0;

          req = m_in.svrctl_req;
          ptr = (vir_bytes) m_in.svrctl_argp;

          /* Is the request indeed for the MM? */
          if (((req >> 8) & 0xFF) != 'M') return(EINVAL);

          /* Control operations local to the PM. */
          switch(req) {
          case MMSETPARAM:
          case MMGETPARAM: {
              struct sysgetenv sysgetenv;
              char search_key[64];
              char *val_start;
              size_t val_len;
              size_t copy_len;

              /* Copy sysgetenv structure to PM. */
              if (sys_datacopy(who, ptr, SELF, (vir_bytes) &sysgetenv,
                      sizeof(sysgetenv)) != OK) return(EFAULT);

              /* Set a param override? */
              if (req == MMSETPARAM) {
                if (local_params >= MAX_LOCAL_PARAMS) return ENOSPC;
                if (sysgetenv.keylen <= 0
                 || sysgetenv.keylen >=
                         sizeof(local_param_overrides[local_params].name)
                 || sysgetenv.vallen <= 0
                 || sysgetenv.vallen >=
                         sizeof(local_param_overrides[local_params].value))
                        return EINVAL;

                  if ((s = sys_datacopy(who, (vir_bytes) sysgetenv.key,
                    SELF, (vir_bytes) local_param_overrides[local_params].name,
                       sysgetenv.keylen)) != OK)
                        return s;
                  if ((s = sys_datacopy(who, (vir_bytes) sysgetenv.val,
                    SELF, (vir_bytes) local_param_overrides[local_params].value,
                      sysgetenv.keylen)) != OK)
                        return s;
                    local_param_overrides[local_params].name[sysgetenv.keylen] = '\0';
                    local_param_overrides[local_params].value[sysgetenv.vallen] = '\0';

                local_params++;

                return OK;
              }

              if (sysgetenv.keylen == 0) {      /* copy all parameters */
                  val_start = monitor_params;
                  val_len = sizeof(monitor_params);
              }
              else {                            /* lookup value for key */
                  int p;
                  /* Try to get a copy of the requested key. */
                  if (sysgetenv.keylen > sizeof(search_key)) return(EINVAL);
                  if ((s = sys_datacopy(who, (vir_bytes) sysgetenv.key,
                          SELF, (vir_bytes) search_key, sysgetenv.keylen)) != OK)
                      return(s);

                  /* Make sure key is null-terminated and lookup value.
                   * First check local overrides.
                   */
                  search_key[sysgetenv.keylen-1]= '\0';
                  for(p = 0; p < local_params; p++) {
                        if (!strcmp(search_key, local_param_overrides[p].name)) {
                                val_start = local_param_overrides[p].value;
                                break;
                        }
                  }
                  if (p >= local_params && (val_start = find_param(search_key)) == NULL)
                       return(ESRCH);
                  val_len = strlen(val_start) + 1;
              }

              /* See if it fits in the client's buffer. */
              if (val_len > sysgetenv.vallen)
                return E2BIG;

              /* Value found, make the actual copy (as far as possible). */
              copy_len = MIN(val_len, sysgetenv.vallen);
              if ((s=sys_datacopy(SELF, (vir_bytes) val_start,
                      who, (vir_bytes) sysgetenv.val, copy_len)) != OK)
                  return(s);

              return OK;
          }
          default:
                return(EINVAL);
          }
        }
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                      servers/fs/fs.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

/* This is the master header for fs.  It includes some other files
 * and defines the principal constants.
 */
#define _POSIX_SOURCE      1    /* tell headers to include POSIX stuff */
#define _MINIX             1    /* tell headers to include MINIX stuff */
#define _SYSTEM            1    /* tell headers that this is the kernel */

#define VERBOSE            0    /* show messages during initialization? */

/* The following are so basic, all the *.c files get them automatically. */
#include <minix/config.h>       /* MUST be first */
#include <ansi.h>               /* MUST be second */
#include <sys/types.h>
#include <minix/const.h>
#include <minix/type.h>
#include <minix/dmap.h>

#include <limits.h>
#include <errno.h>

#include <minix/syslib.h>
#include <minix/sysutil.h>

#include "const.h"
#include "type.h"
#include "proto.h"
#include "glo.h"
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                      servers/fs/const.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

/* Tables sizes */
#define V1_NR_DZONES       7    /* # direct zone numbers in a V1 inode */
#define V1_NR_TZONES       9    /* total # zone numbers in a V1 inode */
#define V2_NR_DZONES       7    /* # direct zone numbers in a V2 inode */
#define V2_NR_TZONES      10    /* total # zone numbers in a V2 inode */

#define NR_FILPS         128    /* # slots in filp table */
#define NR_INODES         64    /* # slots in "in core" inode table */
#define NR_SUPERS          8    /* # slots in super block table */
#define NR_LOCKS           8    /* # slots in the file locking table */

/* The type of sizeof may be (unsigned) long.  Use the following macro for
 * taking the sizes of small objects so that there are no surprises like
 * (small) long constants being passed to routines expecting an int.
 */
#define usizeof(t) ((unsigned) sizeof(t))

/* File system types. */
#define SUPER_MAGIC   0x137F    /* magic number contained in super-block */
#define SUPER_REV     0x7F13    /* magic # when 68000 disk read on PC or vv */
```

```
#define SUPER_V2      0x2468    /* magic # for V2 file systems */
#define SUPER_V2_REV  0x6824    /* V2 magic written on PC, read on 68K or vv */
#define SUPER_V3      0x4d5a    /* magic # for V3 file systems */

#define V1                 1    /* version number of V1 file systems */
#define V2                 2    /* version number of V2 file systems */
#define V3                 3    /* version number of V3 file systems */

/* Miscellaneous constants */
#define SU_UID   ((uid_t) 0)    /* super_user's uid_t */
#define SYS_UID  ((uid_t) 0)    /* uid_t for processes MM and INIT */
#define SYS_GID  ((gid_t) 0)    /* gid_t for processes MM and INIT */
#define NORMAL             0    /* forces get_block to do disk read */
#define NO_READ            1    /* prevents get_block from doing disk read */
#define PREFETCH           2    /* tells get_block not to read or mark dev */

#define XPIPE   (-NR_TASKS-1)   /* used in fp_task when susp'd on pipe */
#define XLOCK   (-NR_TASKS-2)   /* used in fp_task when susp'd on lock */
#define XPOPEN  (-NR_TASKS-3)   /* used in fp_task when susp'd on pipe open */
#define XSELECT (-NR_TASKS-4)   /* used in fp_task when susp'd on select */

#define NO_BIT   ((bit_t) 0)    /* returned by alloc_bit() to signal failure */

#define DUP_MASK        0100    /* mask to distinguish dup2 from dup */

#define LOOK_UP            0 /* tells search_dir to lookup string */
#define ENTER              1 /* tells search_dir to make dir entry */
#define DELETE             2 /* tells search_dir to delete entry */
#define IS_EMPTY           3 /* tells search_dir to ret. OK or ENOTEMPTY */

#define CLEAN              0    /* disk and memory copies identical */
#define DIRTY              1    /* disk and memory copies differ */
#define ATIME            002    /* set if atime field needs updating */
#define CTIME            004    /* set if ctime field needs updating */
#define MTIME            010    /* set if mtime field needs updating */

#define BYTE_SWAP          0    /* tells conv2/conv4 to swap bytes */

#define END_OF_FILE   (-104)    /* eof detected */

#define ROOT_INODE         1            /* inode number for root directory */
#define BOOT_BLOCK  ((block_t) 0)       /* block number of boot block */
#define SUPER_BLOCK_BYTES (1024)        /* bytes offset */
#define START_BLOCK     2               /* first block of FS (not counting SB) */

#define DIR_ENTRY_SIZE       usizeof (struct direct)  /* # bytes/dir entry   */
#define NR_DIR_ENTRIES(b)   ((b)/DIR_ENTRY_SIZE)  /* # dir entries/blk   */
#define SUPER_SIZE      usizeof (struct super_block)  /* super_block size    */
#define PIPE_SIZE(b)          (V1_NR_DZONES*(b))  /* pipe size in bytes  */

#define FS_BITMAP_CHUNKS(b) ((b)/usizeof (bitchunk_t))/* # map chunks/blk   */
#define FS_BITCHUNK_BITS                (usizeof(bitchunk_t) * CHAR_BIT)
#define FS_BITS_PER_BLOCK(b)    (FS_BITMAP_CHUNKS(b) * FS_BITCHUNK_BITS)

/* Derived sizes pertaining to the V1 file system. */
#define V1_ZONE_NUM_SIZE           usizeof (zone1_t)  /* # bytes in V1 zone  */
#define V1_INODE_SIZE             usizeof (d1_inode)  /* bytes in V1 dsk ino */

/* # zones/indir block */
#define V1_INDIRECTS (STATIC_BLOCK_SIZE/V1_ZONE_NUM_SIZE)
```

```

/* # V1 dsk inodes/blk */
#define V1_INODES_PER_BLOCK (STATIC_BLOCK_SIZE/V1_INODE_SIZE)

/* Derived sizes pertaining to the V2 file system. */
#define V2_ZONE_NUM_SIZE            usizeof (zone_t)  /* # bytes in V2 zone  */
#define V2_INODE_SIZE             usizeof (d2_inode)  /* bytes in V2 dsk ino */
#define V2_INDIRECTS(b)   ((b)/V2_ZONE_NUM_SIZE)  /* # zones/indir block */
#define V2_INODES_PER_BLOCK(b) ((b)/V2_INODE_SIZE)/* # V2 dsk inodes/blk */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                servers/fs/type.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/* Declaration of the V1 inode as it is on the disk (not in core). */
typedef struct {                /* V1.x disk inode */
  mode_t d1_mode;               /* file type, protection, etc. */
  uid_t d1_uid;                 /* user id of the file's owner */
  off_t d1_size;                /* current file size in bytes */
  time_t d1_mtime;              /* when was file data last changed */
  u8_t d1_gid;                  /* group number */
  u8_t d1_nlinks;               /* how many links to this file */
  u16_t d1_zone[V1_NR_TZONES];  /* block nums for direct, ind, and dbl ind */
} d1_inode;

/* Declaration of the V2 inode as it is on the disk (not in core). */
typedef struct {                /* V2.x disk inode */
  mode_t d2_mode;               /* file type, protection, etc. */
  u16_t d2_nlinks;              /* how many links to this file. HACK! */
  uid_t d2_uid;                 /* user id of the file's owner. */
  u16_t d2_gid;                 /* group number HACK! */
  off_t d2_size;                /* current file size in bytes */
  time_t d2_atime;              /* when was file data last accessed */
  time_t d2_mtime;              /* when was file data last changed */
  time_t d2_ctime;              /* when was inode data last changed */
  zone_t d2_zone[V2_NR_TZONES]; /* block nums for direct, ind, and dbl ind */
} d2_inode;
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                servers/fs/proto.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/* Function prototypes. */

#include "timers.h"

/* Structs used in prototypes must be declared as such first. */
struct buf;
struct filp;
struct inode;
struct super_block;

/* cache.c */
_PROTOTYPE( zone_t alloc_zone, (Dev_t dev, zone_t z)                   );
_PROTOTYPE( void flushall, (Dev_t dev)                                 );
_PROTOTYPE( void free_zone, (Dev_t dev, zone_t numb)                   );
_PROTOTYPE( struct buf *get_block, (Dev_t dev, block_t block,int only_search));
_PROTOTYPE( void invalidate, (Dev_t device)                          );
_PROTOTYPE( void put_block, (struct buf *bp, int block_type)         );
_PROTOTYPE( void rw_block, (struct buf *bp, int rw_flag)             );
_PROTOTYPE( void rw_scattered, (Dev_t dev,
                        struct buf **bufq, int bufqsize, int rw_flag)  );

/* device.c */
_PROTOTYPE( int dev_open, (Dev_t dev, int proc, int flags)           );
_PROTOTYPE( void dev_close, (Dev_t dev)                              );
_PROTOTYPE( int dev_io, (int op, Dev_t dev, int proc, void *buf,
                        off_t pos, int bytes, int flags)             );
_PROTOTYPE( int gen_opcl, (int op, Dev_t dev, int proc, int flags)   );
_PROTOTYPE( void gen_io, (int task_nr, message *mess_ptr)            );
_PROTOTYPE( int no_dev, (int op, Dev_t dev, int proc, int flags)     );
_PROTOTYPE( int tty_opcl, (int op, Dev_t dev, int proc, int flags)   );
_PROTOTYPE( int ctty_opcl, (int op, Dev_t dev, int proc, int flags)  );
_PROTOTYPE( int clone_opcl, (int op, Dev_t dev, int proc, int flags) );
_PROTOTYPE( void ctty_io, (int task_nr, message *mess_ptr)           );
_PROTOTYPE( int do_ioctl, (void)                                     );
_PROTOTYPE( int do_setsid, (void)                                    );
_PROTOTYPE( void dev_status, (message *)                             );

/* dmp.c */
_PROTOTYPE( int do_fkey_pressed, (void)                              );

/* dmap.c */
_PROTOTYPE( int do_devctl, (void)                                    );
_PROTOTYPE( void build_dmap, (void)                                  );
_PROTOTYPE( int map_driver, (int major, int proc_nr, int dev_style)  );

/* filedes.c */
_PROTOTYPE( struct filp *find_filp, (struct inode *rip, mode_t bits) );
_PROTOTYPE( int get_fd, (int start, mode_t bits, int *k, struct filp **fpt) );
_PROTOTYPE( struct filp *get_filp, (int fild)                        );

/* inode.c */
_PROTOTYPE( struct inode *alloc_inode, (dev_t dev, mode_t bits)      );
_PROTOTYPE( void dup_inode, (struct inode *ip)                       );
_PROTOTYPE( void free_inode, (Dev_t dev, Ino_t numb)                 );
_PROTOTYPE( struct inode *get_inode, (Dev_t dev, int numb)           );
_PROTOTYPE( void put_inode, (struct inode *rip)                      );
_PROTOTYPE( void update_times, (struct inode *rip)                   );
_PROTOTYPE( void rw_inode, (struct inode *rip, int rw_flag)          );
_PROTOTYPE( void wipe_inode, (struct inode *rip)                     );

/* link.c */
_PROTOTYPE( int do_link, (void)                                      );
_PROTOTYPE( int do_unlink, (void)                                    );
_PROTOTYPE( int do_rename, (void)                                    );
_PROTOTYPE( void truncate, (struct inode *rip)                       );

/* lock.c */
_PROTOTYPE( int lock_op, (struct filp *f, int req)                   );
_PROTOTYPE( void lock_revive, (void)                                 );

/* main.c */
_PROTOTYPE( int main, (void)                                         );
_PROTOTYPE( void reply, (int whom, int result)                       );

/* misc.c */
```

```
_PROTOTYPE( int do_dup, (void)                                          );
_PROTOTYPE( int do_exit, (void)                                         );
_PROTOTYPE( int do_fcntl, (void)                                        );
_PROTOTYPE( int do_fork, (void)                                         );
_PROTOTYPE( int do_exec, (void)                                         );
_PROTOTYPE( int do_revive, (void)                                       );
_PROTOTYPE( int do_set, (void)                                          );
_PROTOTYPE( int do_sync, (void)                                         );
_PROTOTYPE( int do_fsync, (void)                                        );
_PROTOTYPE( int do_reboot, (void)                                       );
_PROTOTYPE( int do_svrctl, (void)                                       );
_PROTOTYPE( int do_getsysinfo, (void)                                   );

/* mount.c */
_PROTOTYPE( int do_mount, (void)                                        );
_PROTOTYPE( int do_umount, (void)                                       );
_PROTOTYPE( int unmount, (Dev_t dev)                                    );

/* open.c */
_PROTOTYPE( int do_close, (void)                                        );
_PROTOTYPE( int do_creat, (void)                                        );
_PROTOTYPE( int do_lseek, (void)                                        );
_PROTOTYPE( int do_mknod, (void)                                        );
_PROTOTYPE( int do_mkdir, (void)                                        );
_PROTOTYPE( int do_open, (void)                                         );

/* path.c */
_PROTOTYPE( struct inode *advance,(struct inode *dirp, char string[NAME_MAX]));
_PROTOTYPE( int search_dir, (struct inode *ldir_ptr,
                        char string [NAME_MAX], ino_t *numb, int flag)  );
_PROTOTYPE( struct inode *eat_path, (char *path)                        );
_PROTOTYPE( struct inode *last_dir, (char *path, char string [NAME_MAX]));

/* pipe.c */
_PROTOTYPE( int do_pipe, (void)                                         );
_PROTOTYPE( int do_unpause, (void)                                      );
_PROTOTYPE( int pipe_check, (struct inode *rip, int rw_flag,
                         int oflags, int bytes, off_t position, int *canwrite, int notouch));
_PROTOTYPE( void release, (struct inode *ip, int call_nr, int count)    );
_PROTOTYPE( void revive, (int proc_nr, int bytes)                       );
_PROTOTYPE( void suspend, (int task)                                    );
_PROTOTYPE( int select_request_pipe, (struct filp *f, int *ops, int bl) );
_PROTOTYPE( int select_cancel_pipe, (struct filp *f)                    );
_PROTOTYPE( int select_match_pipe, (struct filp *f)                     );

/* protect.c */
_PROTOTYPE( int do_access, (void)                                       );
_PROTOTYPE( int do_chmod, (void)                                        );
_PROTOTYPE( int do_chown, (void)                                        );
_PROTOTYPE( int do_umask, (void)                                        );
_PROTOTYPE( int forbidden, (struct inode *rip, mode_t access_desired)   );
_PROTOTYPE( int read_only, (struct inode *ip)                           );

/* read.c */
_PROTOTYPE( int do_read, (void)                                         );
_PROTOTYPE( struct buf *rahead, (struct inode *rip, block_t baseblock,
                        off_t position, unsigned bytes_ahead)           );
_PROTOTYPE( void read_ahead, (void)                                     );
_PROTOTYPE( block_t read_map, (struct inode *rip, off_t position)       );
_PROTOTYPE( int read_write, (int rw_flag)                               );
_PROTOTYPE( zone_t rd_indir, (struct buf *bp, int index)               );

/* stadir.c */
_PROTOTYPE( int do_chdir, (void)                                        );
_PROTOTYPE( int do_fchdir, (void)                                       );
_PROTOTYPE( int do_chroot, (void)                                       );
_PROTOTYPE( int do_fstat, (void)                                        );
_PROTOTYPE( int do_stat, (void)                                         );
_PROTOTYPE( int do_fstatfs, (void)                                      );

/* super.c */
_PROTOTYPE( bit_t alloc_bit, (struct super_block *sp, int map, bit_t origin));
_PROTOTYPE( void free_bit, (struct super_block *sp, int map,
                                                bit_t bit_returned)     );
_PROTOTYPE( struct super_block *get_super, (Dev_t dev)                  );
_PROTOTYPE( int mounted, (struct inode *rip)                            );
_PROTOTYPE( int read_super, (struct super_block *sp)                    );
_PROTOTYPE( int get_block_size, (dev_t dev)                             );

/* time.c */
_PROTOTYPE( int do_stime, (void)                                        );
_PROTOTYPE( int do_utime, (void)                                        );

/* utility.c */
_PROTOTYPE( time_t clock_time, (void)                                   );
_PROTOTYPE( unsigned conv2, (int norm, int w)                           );
_PROTOTYPE( long conv4, (int norm, long x)                              );
_PROTOTYPE( int fetch_name, (char *path, int len, int flag)             );
_PROTOTYPE( int no_sys, (void)                                          );
_PROTOTYPE( void panic, (char *who, char *mess, int num)                );

/* write.c */
_PROTOTYPE( void clear_zone, (struct inode *rip, off_t pos, int flag)   );
_PROTOTYPE( int do_write, (void)                                        );
_PROTOTYPE( struct buf *new_block, (struct inode *rip, off_t position)  );
_PROTOTYPE( void zero_block, (struct buf *bp)                           );

/* select.c */
_PROTOTYPE( int do_select, (void)                                       );
_PROTOTYPE( int select_callback, (struct filp *, int ops)               );
_PROTOTYPE( void select_forget, (int fproc)                             );
_PROTOTYPE( void select_timeout_check, (timer_t *)                      );
_PROTOTYPE( void init_select, (void)                                    );
_PROTOTYPE( int select_notified, (int major, int minor, int ops)        );

/* timers.c */
_PROTOTYPE( void fs_set_timer, (timer_t *tp, int delta, tmr_func_t watchdog, int arg));
_PROTOTYPE( void fs_expire_timers, (clock_t now)                        );
_PROTOTYPE( void fs_cancel_timer, (timer_t *tp)                         );
_PROTOTYPE( void fs_init_timer, (timer_t *tp)                           );

/* cdprobe.c */
_PROTOTYPE( int cdprobe, (void)                                         );
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                servers/fs/glo.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

/* EXTERN should be extern except for the table file */
#ifdef _TABLE
#undef EXTERN
#define EXTERN
#endif

/* File System global variables */
EXTERN struct fproc *fp;        /* pointer to caller's fproc struct */
EXTERN int super_user;          /* 1 if caller is super_user, else 0 */
EXTERN int susp_count;          /* number of procs suspended on pipe */
EXTERN int nr_locks;            /* number of locks currently in place */
EXTERN int reviving;            /* number of pipe processes to be revived */
EXTERN off_t rdahedpos;         /* position to read ahead */
EXTERN struct inode *rdahed_inode;      /* pointer to inode to read ahead */
EXTERN Dev_t root_dev;          /* device number of the root device */
EXTERN time_t boottime;         /* time in seconds at system boot */

/* The parameters of the call are kept here. */
EXTERN message m_in;            /* the input message itself */
EXTERN message m_out;           /* the output message used for reply */
EXTERN int who;                 /* caller's proc number */
EXTERN int call_nr;             /* system call number */
EXTERN char user_path[PATH_MAX];/* storage for user path name */

/* The following variables are used for returning results to the caller. */
EXTERN int err_code;            /* temporary storage for error number */
EXTERN int rdwt_err;            /* status of last disk i/o request */

/* Data initialized elsewhere. */
extern _PROTOTYPE (int (*call_vec[]), (void) ); /* sys call table */
extern char dot1[2];   /* dot1 (&dot1[0]) and dot2 (&dot2[0]) have a special */
extern char dot2[3];   /* meaning to search_dir: no access permission check. */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                servers/fs/fproc.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

/* This is the per-process information.  A slot is reserved for each potential
 * process. Thus NR_PROCS must be the same as in the kernel. It is not
 * possible or even necessary to tell when a slot is free here.
 */
EXTERN struct fproc {
  mode_t fp_umask;              /* mask set by umask system call */
  struct inode *fp_workdir;     /* pointer to working directory's inode */
  struct inode *fp_rootdir;     /* pointer to current root dir (see chroot) */
  struct filp *fp_filp[OPEN_MAX];/* the file descriptor table */
  uid_t fp_realuid;             /* real user id */
  uid_t fp_effuid;              /* effective user id */
  gid_t fp_realgid;             /* real group id */
  gid_t fp_effgid;              /* effective group id */
  dev_t fp_tty;                 /* major/minor of controlling tty */
  int fp_fd;                    /* place to save fd if rd/wr can't finish */
  char *fp_buffer;              /* place to save buffer if rd/wr can't finish*/
  int  fp_nbytes;               /* place to save bytes if rd/wr can't finish */
  int  fp_cum_io_partial;       /* partial byte count if rd/wr can't finish */
  char fp_suspended;            /* set to indicate process hanging */
  char fp_revived;              /* set to indicate process being revived */
  char fp_task;                 /* which task is proc suspended on */
  char fp_sesldr;               /* true if proc is a session leader */
  pid_t fp_pid;                 /* process id */
  long fp_cloexec;              /* bit map for POSIX Table 6-2 FD_CLOEXEC */
} fproc[NR_PROCS];

/* Field values. */
#define NOT_SUSPENDED      0    /* process is not suspended on pipe or task */
#define SUSPENDED          1    /* process is suspended on pipe or task */
#define NOT_REVIVING       0    /* process is not being revived */
#define REVIVING           1    /* process is being revived from suspension */
#define PID_FREE           0    /* process slot free */

/* Check is process number is acceptable - includes system processes. */
#define isokprocnr(n)   ((unsigned)((n)+NR_TASKS) < NR_PROCS + NR_TASKS)

```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                    servers/fs/buf.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/* Buffer (block) cache.  To acquire a block, a routine calls get_block(),
 * telling which block it wants.  The block is then regarded as "in use"
 * and has its 'b_count' field incremented.  All the blocks that are not
 * in use are chained together in an LRU list, with 'front' pointing
 * to the least recently used block, and 'rear' to the most recently used
 * block.  A reverse chain, using the field b_prev is also maintained.
 * Usage for LRU is measured by the time the put_block() is done.  The second
 * parameter to put_block() can violate the LRU order and put a block on the
 * front of the list, if it will probably not be needed soon.  If a block
 * is modified, the modifying routine must set b_dirt to DIRTY, so the block
 * will eventually be rewritten to the disk.
 */

#include <sys/dir.h>                    /* need struct direct */
#include <dirent.h>

EXTERN struct buf {
  /* Data portion of the buffer. */
  union {
    char b__data[MAX_BLOCK_SIZE];                     /* ordinary user data */
/* directory block */
    struct direct b__dir[NR_DIR_ENTRIES(MAX_BLOCK_SIZE)];
/* V1 indirect block */
    zone1_t b__v1_ind[V1_INDIRECTS];
/* V2 indirect block */
    zone_t  b__v2_ind[V2_INDIRECTS(MAX_BLOCK_SIZE)];
/* V1 inode block */
    d1_inode b__v1_ino[V1_INODES_PER_BLOCK];
/* V2 inode block */
    d2_inode b__v2_ino[V2_INODES_PER_BLOCK(MAX_BLOCK_SIZE)];
/* bit map block */
    bitchunk_t b__bitmap[FS_BITMAP_CHUNKS(MAX_BLOCK_SIZE)];
  } b;

  /* Header portion of the buffer. */
  struct buf *b_next;           /* used to link all free bufs in a chain */
  struct buf *b_prev;           /* used to link all free bufs the other way */
  struct buf *b_hash;           /* used to link bufs on hash chains */
  block_t b_blocknr;            /* block number of its (minor) device */
  dev_t b_dev;                  /* major | minor device where block resides */
  char b_dirt;                  /* CLEAN or DIRTY */
  char b_count;                 /* number of users of this buffer */
} buf[NR_BUFS];

/* A block is free if b_dev == NO_DEV. */

#define NIL_BUF ((struct buf *) 0)      /* indicates absence of a buffer */

/* These defs make it possible to use to bp->b_data instead of bp->b.b__data */
#define b_data   b.b__data
#define b_dir    b.b__dir
#define b_v1_ind b.b__v1_ind
#define b_v2_ind b.b__v2_ind
#define b_v1_ino b.b__v1_ino
#define b_v2_ino b.b__v2_ino
#define b_bitmap b.b__bitmap

EXTERN struct buf *buf_hash[NR_BUF_HASH];       /* the buffer hash table */

EXTERN struct buf *front;       /* points to least recently used free block */
EXTERN struct buf *rear;        /* points to most recently used free block */
EXTERN int bufs_in_use;         /* # bufs currently in use (not on free list)*/

/* When a block is released, the type of usage is passed to put_block(). */
#define WRITE_IMMED   0100 /* block should be written to disk now */
#define ONE_SHOT      0200 /* set if block not likely to be needed soon */

#define INODE_BLOCK        0                             /* inode block */
#define DIRECTORY_BLOCK    1                             /* directory block */
#define INDIRECT_BLOCK     2                             /* pointer block */
#define MAP_BLOCK          3                             /* bit map */
#define FULL_DATA_BLOCK    5                             /* data, fully used */
#define PARTIAL_DATA_BLOCK 6                             /* data, partly used*/

#define HASH_MASK (NR_BUF_HASH - 1)     /* mask for hashing block numbers */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                    servers/fs/file.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/* This is the filp table.  It is an intermediary between file descriptors and
 * inodes.  A slot is free if filp_count == 0.
 */

EXTERN struct filp {
  mode_t filp_mode;             /* RW bits, telling how file is opened */
  int filp_flags;               /* flags from open and fcntl */
  int filp_count;               /* how many file descriptors share this slot?*/
  struct inode *filp_ino;       /* pointer to the inode */
  off_t filp_pos;               /* file position */

  /* the following fields are for select() and are owned by the generic
   * select() code (i.e., fd-type-specific select() code can't touch these).
   */
  int filp_selectors;           /* select()ing processes blocking on this fd */
  int filp_select_ops;          /* interested in these SEL_* operations */

  /* following are for fd-type-specific select() */
  int filp_pipe_select_ops;
} filp[NR_FILPS];

#define FILP_CLOSED     0       /* filp_mode: associated device closed */

#define NIL_FILP (struct filp *) 0      /* indicates absence of a filp slot */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                servers/fs/lock.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/* This is the file locking table.  Like the filp table, it points to the
 * inode table, however, in this case to achieve advisory locking.
 */
EXTERN struct file_lock {
  short lock_type;              /* F_RDLOCK or F_WRLOCK; 0 means unused slot */
  pid_t lock_pid;               /* pid of the process holding the lock */
  struct inode *lock_inode;     /* pointer to the inode locked */
  off_t lock_first;             /* offset of first byte locked */
  off_t lock_last;              /* offset of last byte locked */
} file_lock[NR_LOCKS];
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                servers/fs/inode.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/* Inode table.  This table holds inodes that are currently in use.  In some
 * cases they have been opened by an open() or creat() system call, in other
 * cases the file system itself needs the inode for one reason or another,
 * such as to search a directory for a path name.
 * The first part of the struct holds fields that are present on the
 * disk; the second part holds fields not present on the disk.
 * The disk inode part is also declared in "type.h" as 'd1_inode' for V1
 * file systems and 'd2_inode' for V2 file systems.
 */

EXTERN struct inode {
  mode_t i_mode;                /* file type, protection, etc. */
  nlink_t i_nlinks;             /* how many links to this file */
  uid_t i_uid;                  /* user id of the file's owner */
  gid_t i_gid;                  /* group number */
  off_t i_size;                 /* current file size in bytes */
  time_t i_atime;               /* time of last access (V2 only) */
  time_t i_mtime;               /* when was file data last changed */
  time_t i_ctime;               /* when was inode itself changed (V2 only)*/
  zone_t i_zone[V2_NR_TZONES]; /* zone numbers for direct, ind, and dbl ind */

  /* The following items are not present on the disk. */
  dev_t i_dev;                  /* which device is the inode on */
  ino_t i_num;                  /* inode number on its (minor) device */
  int i_count;                  /* # times inode used; 0 means slot is free */
  int i_ndzones;                /* # direct zones (Vx_NR_DZONES) */
  int i_nindirs;                /* # indirect zones per indirect block */
  struct super_block *i_sp;     /* pointer to super block for inode's device */
  char i_dirt;                  /* CLEAN or DIRTY */
  char i_pipe;                  /* set to I_PIPE if pipe */
  char i_mount;                 /* this bit is set if file mounted on */
  char i_seek;                  /* set on LSEEK, cleared on READ/WRITE */
  char i_update;                /* the ATIME, CTIME, and MTIME bits are here */
} inode[NR_INODES];

#define NIL_INODE (struct inode *) 0    /* indicates absence of inode slot */

/* Field values.  Note that CLEAN and DIRTY are defined in "const.h" */
#define NO_PIPE           0     /* i_pipe is NO_PIPE if inode is not a pipe */
#define I_PIPE            1     /* i_pipe is I_PIPE if inode is a pipe */
#define NO_MOUNT          0     /* i_mount is NO_MOUNT if file not mounted on*/
#define I_MOUNT           1     /* i_mount is I_MOUNT if file mounted on */
#define NO_SEEK           0     /* i_seek = NO_SEEK if last op was not SEEK */
#define ISEEK             1     /* i_seek = ISEEK if last op was SEEK */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                servers/fs/param.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/* The following names are synonyms for the variables in the input message. */
#define acc_time      m2_l1
#define addr          m1_i3
#define buffer        m1_p1
#define child         m1_i2
#define co_mode       m1_i1
#define eff_grp_id    m1_i3
#define eff_user_id   m1_i3
#define erki          m1_p1
#define fd            m1_i1
#define fd2           m1_i2
#define ioflags       m1_i3
#define group         m1_i3
#define real_grp_id   m1_i2
#define ls_fd         m2_i1
#define mk_mode       m1_i2
#define mk_z0         m1_i3
#define mode          m3_i2
#define c_mode        m1_i3
#define c_name        m1_p1
#define name          m3_p1
#define name1         m1_p1
#define name2         m1_p2
#define name_length   m3_i1
#define name1_length  m1_i1
#define name2_length  m1_i2
#define nbytes        m1_i2
#define owner         m1_i2
#define parent        m1_i1
#define pathname      m3_ca1
#define pid           m1_i3
#define pro           m1_i1
#define ctl_req       m4_l1
#define driver_nr     m4_l2
#define dev_nr        m4_l3
```

```
#define dev_style     m4_l4
#define rd_only       m1_i3
#define real_user_id  m1_i2
#define request       m1_i2
#define sig           m1_i2
#define slot1         m1_i1
#define tp            m2_l1
#define utime_actime  m2_l1
#define utime_modtime m2_l2
#define utime_file    m2_p1
#define utime_length  m2_i1
#define utime_strlen  m2_i2
#define whence        m2_i2
#define svrctl_req    m2_i1
#define svrctl_argp   m2_p1
#define pm_stime      m1_i1
#define info_what     m1_i1
#define info_where    m1_p1

/* The following names are synonyms for the variables in the output message. */
#define reply_type    m_type
#define reply_l1      m2_l1
#define reply_i1      m1_i1
#define reply_i2      m1_i2
#define reply_t1      m4_l1
#define reply_t2      m4_l2
#define reply_t3      m4_l3
#define reply_t4      m4_l4
#define reply_t5      m4_l5
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                     servers/fs/super.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/* Super block table.  The root file system and every mounted file system
 * has an entry here.  The entry holds information about the sizes of the bit
 * maps and inodes.  The s_ninodes field gives the number of inodes available
 * for files and directories, including the root directory.  Inode 0 is
 * on the disk, but not used.  Thus s_ninodes = 4 means that 5 bits will be
 * used in the bit map, bit 0, which is always 1 and not used, and bits 1-4
 * for files and directories.  The disk layout is:
 *
 *    Item        # blocks
 *    boot block      1
 *    super block     1    (offset 1kB)
 *    inode map     s_imap_blocks
 *    zone map      s_zmap_blocks
 *    inodes        (s_ninodes + 'inodes per block' - 1)/'inodes per block'
 *    unused        whatever is needed to fill out the current zone
 *    data zones    (s_zones - s_firstdatazone) << s_log_zone_size
 *
 * A super_block slot is free if s_dev == NO_DEV.
 */

EXTERN struct super_block {
  ino_t s_ninodes;              /* # usable inodes on the minor device */
  zone1_t  s_nzones;            /* total device size, including bit maps etc */
  short s_imap_blocks;          /* # of blocks used by inode bit map */
  short s_zmap_blocks;          /* # of blocks used by zone bit map */
```

```
   zone1_t s_firstdatazone;      /* number of first data zone */
   short s_log_zone_size;        /* log2 of blocks/zone */
   short s_pad;                  /* try to avoid compiler-dependent padding */
   off_t s_max_size;             /* maximum file size on this device */
   zone_t s_zones;               /* number of zones (replaces s_nzones in V2) */
   short s_magic;                /* magic number to recognize super-blocks */

   /* The following items are valid on disk only for V3 and above */

   /* The block size in bytes. Minimum MIN_BLOCK SIZE. SECTOR_SIZE
    * multiple. If V1 or V2 filesystem, this should be
    * initialised to STATIC_BLOCK_SIZE. Maximum MAX_BLOCK_SIZE.
    */
   short s_pad2;                 /* try to avoid compiler-dependent padding */
   unsigned short s_block_size;  /* block size in bytes. */
   char s_disk_version;          /* filesystem format sub-version */

   /* The following items are only used when the super_block is in memory. */
   struct inode *s_isup;         /* inode for root dir of mounted file sys */
   struct inode *s_imount;       /* inode mounted on */
   unsigned s_inodes_per_block;  /* precalculated from magic number */
   dev_t s_dev;                  /* whose super block is this? */
   int s_rd_only;                /* set to 1 iff file sys mounted read only */
   int s_native;                 /* set to 1 iff not byte swapped file system */
   int s_version;                /* file system version, zero means bad magic */
   int s_ndzones;                /* # direct zones in an inode */
   int s_nindirs;                /* # indirect zones per indirect block */
   bit_t s_isearch;              /* inodes below this bit number are in use */
   bit_t s_zsearch;              /* all zones below this bit number are in use*/
 } super_block[NR_SUPERS];

 #define NIL_SUPER (struct super_block *) 0
 #define IMAP           0        /* operating on the inode bit map */
 #define ZMAP           1        /* operating on the zone bit map */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                servers/fs/table.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/* This file contains the table used to map system call numbers onto the
 * routines that perform them.
 */

#define _TABLE

#include "fs.h"
#include <minix/callnr.h>
#include <minix/com.h>
#include "buf.h"
#include "file.h"
#include "fproc.h"
#include "inode.h"
#include "lock.h"
#include "super.h"

PUBLIC _PROTOTYPE (int (*call_vec[]), (void) ) = {
        no_sys,         /*  0 = unused  */
        do_exit,        /*  1 = exit    */
        do_fork,        /*  2 = fork    */
        do_read,        /*  3 = read    */
        do_write,       /*  4 = write   */
        do_open,        /*  5 = open    */
        do_close,       /*  6 = close   */
        no_sys,         /*  7 = wait    */
        do_creat,       /*  8 = creat   */
        do_link,        /*  9 = link    */
        do_unlink,      /* 10 = unlink  */
        no_sys,         /* 11 = waitpid */
        do_chdir,       /* 12 = chdir   */
        no_sys,         /* 13 = time    */
        do_mknod,       /* 14 = mknod   */
        do_chmod,       /* 15 = chmod   */
        do_chown,       /* 16 = chown   */
        no_sys,         /* 17 = break   */
        do_stat,        /* 18 = stat    */
        do_lseek,       /* 19 = lseek   */
        no_sys,         /* 20 = getpid  */
        do_mount,       /* 21 = mount   */
        do_umount,      /* 22 = umount  */
        do_set,         /* 23 = setuid  */
        no_sys,         /* 24 = getuid  */
        do_stime,       /* 25 = stime   */
        no_sys,         /* 26 = ptrace  */
        no_sys,         /* 27 = alarm   */
        do_fstat,       /* 28 = fstat   */
        no_sys,         /* 29 = pause   */
        do_utime,       /* 30 = utime   */
        no_sys,         /* 31 = (stty)  */
        no_sys,         /* 32 = (gtty)  */
        do_access,      /* 33 = access  */
        no_sys,         /* 34 = (nice)  */
        no_sys,         /* 35 = (ftime) */
        do_sync,        /* 36 = sync    */
        no_sys,         /* 37 = kill    */
        do_rename,      /* 38 = rename  */
        do_mkdir,       /* 39 = mkdir   */
        do_unlink,      /* 40 = rmdir   */
        do_dup,         /* 41 = dup     */
        do_pipe,        /* 42 = pipe    */
        no_sys,         /* 43 = times   */
        no_sys,         /* 44 = (prof)  */
        no_sys,         /* 45 = unused  */
        do_set,         /* 46 = setgid  */
        no_sys,         /* 47 = getgid  */
        no_sys,         /* 48 = (signal)*/
        no_sys,         /* 49 = unused  */
        no_sys,         /* 50 = unused  */
        no_sys,         /* 51 = (acct)  */
        no_sys,         /* 52 = (phys)  */
        no_sys,         /* 53 = (lock)  */
        do_ioctl,       /* 54 = ioctl   */
        do_fcntl,       /* 55 = fcntl   */
        no_sys,         /* 56 = (mpx)   */
        no_sys,         /* 57 = unused  */
        no_sys,         /* 58 = unused  */
        do_exec,        /* 59 = execve  */
        do_umask,       /* 60 = umask   */
        do_chroot,      /* 61 = chroot  */
        do_setsid,      /* 62 = setsid  */
        no_sys,         /* 63 = getpgrp */

        no_sys,         /* 64 = KSIG: signals originating in the kernel */
        do_unpause,     /* 65 = UNPAUSE */
        no_sys,         /* 66 = unused  */
        do_revive,      /* 67 = REVIVE  */
        no_sys,         /* 68 = TASK_REPLY      */
        no_sys,         /* 69 = unused */
        no_sys,         /* 70 = unused */
        no_sys,         /* 71 = si */
        no_sys,         /* 72 = sigsuspend */
        no_sys,         /* 73 = sigpending */
        no_sys,         /* 74 = sigprocmask */
        no_sys,         /* 75 = sigreturn */
        do_reboot,      /* 76 = reboot */
        do_svrctl,      /* 77 = svrctl */

        no_sys,         /* 78 = unused */
        do_getsysinfo,  /* 79 = getsysinfo */
        no_sys,         /* 80 = unused */
        do_devctl,      /* 81 = devctl */
        do_fstatfs,     /* 82 = fstatfs */
        no_sys,         /* 83 = memalloc */
        no_sys,         /* 84 = memfree */
        do_select,      /* 85 = select */
        do_fchdir,      /* 86 = fchdir */
        do_fsync,       /* 87 = fsync */
        no_sys,         /* 88 = getpriority */
        no_sys,         /* 89 = setpriority */
        no_sys,         /* 90 = gettimeofday */
};
/* This should not fail with "array size is negative": */
extern int dummy[sizeof(call_vec) == NCALLS * sizeof(call_vec[0]) ? 1 : -1];

```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                 servers/fs/cache.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/* The file system maintains a buffer cache to reduce the number of disk
 * accesses needed.  Whenever a read or write to the disk is done, a check is
 * first made to see if the block is in the cache.  This file manages the
 * cache.
 *
 * The entry points into this file are:
 *   get_block:   request to fetch a block for reading or writing from cache
 *   put_block:   return a block previously requested with get_block
 *   alloc_zone:  allocate a new zone (to increase the length of a file)
 *   free_zone:   release a zone (when a file is removed)
 *   rw_block:    read or write a block from the disk itself
 *   invalidate:  remove all the cache blocks on some device
 */

#include "fs.h"
#include <minix/com.h>
#include "buf.h"
#include "file.h"
#include "fproc.h"
#include "super.h"
```

```

FORWARD _PROTOTYPE( void rm_lru, (struct buf *bp) );

/*===========================================================================*
 *                              get_block                                    *
 *===========================================================================*/
PUBLIC struct buf *get_block(dev, block, only_search)
register dev_t dev;             /* on which device is the block? */
register block_t block;         /* which block is wanted? */
int only_search;                /* if NO_READ, don't read, else act normal */
{
/* Check to see if the requested block is in the block cache.  If so, return
 * a pointer to it.  If not, evict some other block and fetch it (unless
 * 'only_search' is 1).  All the blocks in the cache that are not in use
 * are linked together in a chain, with 'front' pointing to the least recently
 * used block and 'rear' to the most recently used block.  If 'only_search' is
 * 1, the block being requested will be overwritten in its entirety, so it is
 * only necessary to see if it is in the cache; if it is not, any free buffer
 * will do.  It is not necessary to actually read the block in from disk.
 * If 'only_search' is PREFETCH, the block need not be read from the disk,
 * and the device is not to be marked on the block, so callers can tell if
 * the block returned is valid.
 * In addition to the LRU chain, there is also a hash chain to link together
 * blocks whose block numbers end with the same bit strings, for fast lookup.
 */

  int b;
  register struct buf *bp, *prev_ptr;

  /* Search the hash chain for (dev, block). Do_read() can use
   * get_block(NO_DEV ...) to get an unnamed block to fill with zeros when
   * someone wants to read from a hole in a file, in which case this search
   * is skipped
   */
  if (dev != NO_DEV) {
        b = (int) block & HASH_MASK;
        bp = buf_hash[b];
        while (bp != NIL_BUF) {
                if (bp->b_blocknr == block && bp->b_dev == dev) {
                        /* Block needed has been found. */
                        if (bp->b_count == 0) rm_lru(bp);
                        bp->b_count++;  /* record that block is in use */

                        return(bp);
                } else {
                        /* This block is not the one sought. */
                        bp = bp->b_hash; /* move to next block on hash chain */
                }
        }
  }

  /* Desired block is not on available chain.  Take oldest block ('front'). */
  if ((bp = front) == NIL_BUF) panic(__FILE__,"all buffers in use", NR_BUFS);
  rm_lru(bp);

  /* Remove the block that was just taken from its hash chain. */
  b = (int) bp->b_blocknr & HASH_MASK;
  prev_ptr = buf_hash[b];
  if (prev_ptr == bp) {
        buf_hash[b] = bp->b_hash;
  } else {
        /* The block just taken is not on the front of its hash chain. */
        while (prev_ptr->b_hash != NIL_BUF)
                if (prev_ptr->b_hash == bp) {
                        prev_ptr->b_hash = bp->b_hash;  /* found it */
                        break;
                } else {
                        prev_ptr = prev_ptr->b_hash;    /* keep looking */
                }
  }

  /* If the block taken is dirty, make it clean by writing it to the disk.
   * Avoid hysteresis by flushing all other dirty blocks for the same device.
   */
  if (bp->b_dev != NO_DEV) {
        if (bp->b_dirt == DIRTY) flushall(bp->b_dev);
  }

  /* Fill in block's parameters and add it to the hash chain where it goes. */
  bp->b_dev = dev;              /* fill in device number */
  bp->b_blocknr = block;        /* fill in block number */
  bp->b_count++;                /* record that block is being used */
  b = (int) bp->b_blocknr & HASH_MASK;
  bp->b_hash = buf_hash[b];
  buf_hash[b] = bp;             /* add to hash list */

  /* Go get the requested block unless searching or prefetching. */
  if (dev != NO_DEV) {
        if (only_search == PREFETCH) bp->b_dev = NO_DEV;
        else
        if (only_search == NORMAL) {
                rw_block(bp, READING);
        }
  }
  return(bp);                   /* return the newly acquired block */
}

/*===========================================================================*
 *                              put_block                                    *
 *===========================================================================*/
PUBLIC void put_block(bp, block_type)
register struct buf *bp;        /* pointer to the buffer to be released */
int block_type;                 /* INODE_BLOCK, DIRECTORY_BLOCK, or whatever */
{
/* Return a block to the list of available blocks.   Depending on 'block_type'
 * it may be put on the front or rear of the LRU chain.  Blocks that are
 * expected to be needed again shortly (e.g., partially full data blocks)
 * go on the rear; blocks that are unlikely to be needed again shortly
 * (e.g., full data blocks) go on the front.  Blocks whose loss can hurt
 * the integrity of the file system (e.g., inode blocks) are written to
 * disk immediately if they are dirty.
 */
  if (bp == NIL_BUF) return;    /* it is easier to check here than in caller */

  bp->b_count--;                /* there is one use fewer now */
  if (bp->b_count != 0) return; /* block is still in use */

  bufs_in_use--;                /* one fewer block buffers in use */

  /* Put this block back on the LRU chain.  If the ONE_SHOT bit is set in
```

```
   * 'block_type', the block is not likely to be needed again shortly, so put
   * it on the front of the LRU chain where it will be the first one to be
   * taken when a free buffer is needed later.
   */
  if (bp->b_dev == DEV_RAM || block_type & ONE_SHOT) {
        /* Block probably won't be needed quickly. Put it on front of chain.
         * It will be the next block to be evicted from the cache.
         */
        bp->b_prev = NIL_BUF;
        bp->b_next = front;
        if (front == NIL_BUF)
                rear = bp;      /* LRU chain was empty */
        else
                front->b_prev = bp;
        front = bp;
  } else {
        /* Block probably will be needed quickly.  Put it on rear of chain.
         * It will not be evicted from the cache for a long time.
         */
        bp->b_prev = rear;
        bp->b_next = NIL_BUF;
        if (rear == NIL_BUF)
                front = bp;
        else
                rear->b_next = bp;
        rear = bp;
  }

  /* Some blocks are so important (e.g., inodes, indirect blocks) that they
   * should be written to the disk immediately to avoid messing up the file
   * system in the event of a crash.
   */
  if ((block_type & WRITE_IMMED) && bp->b_dirt==DIRTY && bp->b_dev != NO_DEV) {
                rw_block(bp, WRITING);
  }
}

/*===========================================================================*
 *                              alloc_zone                                   *
 *===========================================================================*/
PUBLIC zone_t alloc_zone(dev, z)
dev_t dev;                      /* device where zone wanted */
zone_t z;                       /* try to allocate new zone near this one */
{
/* Allocate a new zone on the indicated device and return its number. */

  int major, minor;
  bit_t b, bit;
  struct super_block *sp;

  /* Note that the routine alloc_bit() returns 1 for the lowest possible
   * zone, which corresponds to sp->s_firstdatazone.  To convert a value
   * between the bit number, 'b', used by alloc_bit() and the zone number, 'z',
   * stored in the inode, use the formula:
   *     z = b + sp->s_firstdatazone - 1
   * Alloc_bit() never returns 0, since this is used for NO_BIT (failure).
   */
  sp = get_super(dev);

  /* If z is 0, skip initial part of the map known to be fully in use. */
  if (z == sp->s_firstdatazone) {
        bit = sp->s_zsearch;
  } else {
        bit = (bit_t) z - (sp->s_firstdatazone - 1);
  }
  b = alloc_bit(sp, ZMAP, bit);
  if (b == NO_BIT) {
        err_code = ENOSPC;
        major = (int) (sp->s_dev >> MAJOR) & BYTE;
        minor = (int) (sp->s_dev >> MINOR) & BYTE;
        printf("No space on %sdevice %d/%d\n",
                sp->s_dev == root_dev ? "root " : "", major, minor);
        return(NO_ZONE);
  }
  if (z == sp->s_firstdatazone) sp->s_zsearch = b;      /* for next time */
  return(sp->s_firstdatazone - 1 + (zone_t) b);
}

/*===========================================================================*
 *                              free_zone                                    *
 *===========================================================================*/
PUBLIC void free_zone(dev, numb)
dev_t dev;                              /* device where zone located */
zone_t numb;                            /* zone to be returned */
{
/* Return a zone. */

  register struct super_block *sp;
  bit_t bit;

  /* Locate the appropriate super_block and return bit. */
  sp = get_super(dev);
  if (numb < sp->s_firstdatazone || numb >= sp->s_zones) return;
  bit = (bit_t) (numb - (sp->s_firstdatazone - 1));
  free_bit(sp, ZMAP, bit);
  if (bit < sp->s_zsearch) sp->s_zsearch = bit;
}

/*===========================================================================*
 *                              rw_block                                     *
 *===========================================================================*/
PUBLIC void rw_block(bp, rw_flag)
register struct buf *bp;        /* buffer pointer */
int rw_flag;                    /* READING or WRITING */
{
/* Read or write a disk block. This is the only routine in which actual disk
 * I/O is invoked. If an error occurs, a message is printed here, but the error
 * is not reported to the caller.  If the error occurred while purging a block
 * from the cache, it is not clear what the caller could do about it anyway.
 */

  int r, op;
  off_t pos;
  dev_t dev;
  int block_size;

  block_size = get_block_size(bp->b_dev);

  if ( (dev = bp->b_dev) != NO_DEV) {
        pos = (off_t) bp->b_blocknr * block_size;
        op = (rw_flag == READING ? DEV_READ : DEV_WRITE);
        r = dev_io(op, dev, FS_PROC_NR, bp->b_data, pos, block_size, 0);
        if (r != block_size) {
            if (r >= 0) r = END_OF_FILE;
            if (r != END_OF_FILE)
              printf("Unrecoverable disk error on device %d/%d, block %ld\n",
                        (dev>>MAJOR)&BYTE, (dev>>MINOR)&BYTE, bp->b_blocknr);
                bp->b_dev = NO_DEV;     /* invalidate block */

                /* Report read errors to interested parties. */
                if (rw_flag == READING) rdwt_err = r;
        }
  }

  bp->b_dirt = CLEAN;
}

/*===========================================================================*
 *                              invalidate                                   *
 *===========================================================================*/
PUBLIC void invalidate(device)
dev_t device;                   /* device whose blocks are to be purged */
{
/* Remove all the blocks belonging to some device from the cache. */

  register struct buf *bp;

  for (bp = &buf[0]; bp < &buf[NR_BUFS]; bp++)
        if (bp->b_dev == device) bp->b_dev = NO_DEV;
}

/*===========================================================================*
 *                              flushall                                     *
 *===========================================================================*/
PUBLIC void flushall(dev)
dev_t dev;                      /* device to flush */
{
/* Flush all dirty blocks for one device. */

  register struct buf *bp;
  static struct buf *dirty[NR_BUFS];    /* static so it isn't on stack */
  int ndirty;

  for (bp = &buf[0], ndirty = 0; bp < &buf[NR_BUFS]; bp++)
        if (bp->b_dirt == DIRTY && bp->b_dev == dev) dirty[ndirty++] = bp;
  rw_scattered(dev, dirty, ndirty, WRITING);
}

/*===========================================================================*
 *                              rw_scattered                                 *
 *===========================================================================*/
PUBLIC void rw_scattered(dev, bufq, bufqsize, rw_flag)
dev_t dev;                      /* major-minor device number */
struct buf **bufq;              /* pointer to array of buffers */
int bufqsize;                   /* number of buffers */
int rw_flag;                    /* READING or WRITING */
{
/* Read or write scattered data from a device. */

  register struct buf *bp;
  int gap;
  register int i;
  register iovec_t *iop;
  static iovec_t iovec[NR_IOREQS];  /* static so it isn't on stack */
  int j, r;
  int block_size;

  block_size = get_block_size(dev);

  /* (Shell) sort buffers on b_blocknr. */
  gap = 1;
  do
        gap = 3 * gap + 1;
  while (gap <= bufqsize);
  while (gap != 1) {
        gap /= 3;
        for (j = gap; j < bufqsize; j++) {
                for (i = j - gap;
                     i >= 0 && bufq[i]->b_blocknr > bufq[i + gap]->b_blocknr;
                     i -= gap) {
                        bp = bufq[i];
                        bufq[i] = bufq[i + gap];
                        bufq[i + gap] = bp;
                }
        }
  }

  /* Set up I/O vector and do I/O.  The result of dev_io is OK if everything
   * went fine, otherwise the error code for the first failed transfer.
   */
  while (bufqsize > 0) {
        for (j = 0, iop = iovec; j < NR_IOREQS && j < bufqsize; j++, iop++) {
                bp = bufq[j];
                if (bp->b_blocknr != bufq[0]->b_blocknr + j) break;
                iop->iov_addr = (vir_bytes) bp->b_data;
                iop->iov_size = block_size;
        }
        r = dev_io(rw_flag == WRITING ? DEV_SCATTER : DEV_GATHER,
                dev, FS_PROC_NR, iovec,
                (off_t) bufq[0]->b_blocknr * block_size, j, 0);

        /* Harvest the results.  Dev_io reports the first error it may have
         * encountered, but we only care if it's the first block that failed.
         */
        for (i = 0, iop = iovec; i < j; i++, iop++) {
                bp = bufq[i];
                if (iop->iov_size != 0) {
                        /* Transfer failed. An error? Do we care? */
                        if (r != OK && i == 0) {
                                printf(
                                "fs: I/O error on device %d/%d, block %lu\n",
                                        (dev>>MAJOR)&BYTE, (dev>>MINOR)&BYTE,
                                        bp->b_blocknr);
                                bp->b_dev = NO_DEV;     /* invalidate block */
                        }
                        break;
                }
                if (rw_flag == READING) {
                        bp->b_dev = dev;        /* validate block */
                        put_block(bp, PARTIAL_DATA_BLOCK);
                } else {
                        bp->b_dirt = CLEAN;
                }
        }
        bufq += i;
        bufqsize -= i;
        if (rw_flag == READING) {
                /* Don't bother reading more than the device is willing to
                 * give at this time.  Don't forget to release those extras.
                 */
                while (bufqsize > 0) {
                        put_block(*bufq++, PARTIAL_DATA_BLOCK);
                        bufqsize--;
                }
        }
        if (rw_flag == WRITING && i == 0) {
                /* We're not making progress, this means we might keep
                 * looping. Buffers remain dirty if un-written. Buffers are
                 * lost if invalidate()d or LRU-removed while dirty. This
                 * is better than keeping unwritable blocks around forever..
                 */
                break;
        }
  }
}

/*===========================================================================*
 *                              rm_lru                                       *
 *===========================================================================*/
PRIVATE void rm_lru(bp)
struct buf *bp;
{
/* Remove a block from its LRU chain. */
  struct buf *next_ptr, *prev_ptr;

  bufs_in_use++;
  next_ptr = bp->b_next;        /* successor on LRU chain */
  prev_ptr = bp->b_prev;        /* predecessor on LRU chain */
  if (prev_ptr != NIL_BUF)
        prev_ptr->b_next = next_ptr;
  else
        front = next_ptr;       /* this block was at front of chain */

  if (next_ptr != NIL_BUF)
        next_ptr->b_prev = prev_ptr;
  else
        rear = prev_ptr;        /* this block was at rear of chain */
}
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                    servers/fs/inode.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/* This file manages the inode table.  There are procedures to allocate and
 * deallocate inodes, acquire, erase, and release them, and read and write
 * them from the disk.
 *
 * The entry points into this file are
 *   get_inode:    search inode table for a given inode; if not there,
 *                 read it
 *   put_inode:    indicate that an inode is no longer needed in memory
 *   alloc_inode:  allocate a new, unused inode
 *   wipe_inode:   erase some fields of a newly allocated inode
 *   free_inode:   mark an inode as available for a new file
 *   update_times: update atime, ctime, and mtime
 *   rw_inode:     read a disk block and extract an inode, or corresp. write
 *   old_icopy:    copy to/from in-core inode struct and disk inode (V1.x)
 *   new_icopy:    copy to/from in-core inode struct and disk inode (V2.x)
 *   dup_inode:    indicate that someone else is using an inode table entry
 */

#include "fs.h"
#include "buf.h"
#include "file.h"
#include "fproc.h"
#include "inode.h"
#include "super.h"

FORWARD _PROTOTYPE( void old_icopy, (struct inode *rip, d1_inode *dip,
                                                int direction, int norm));
FORWARD _PROTOTYPE( void new_icopy, (struct inode *rip, d2_inode *dip,
                                                int direction, int norm));

/*===========================================================================*
 *                              get_inode                                    *
 *===========================================================================*/
PUBLIC struct inode *get_inode(dev, numb)
dev_t dev;                      /* device on which inode resides */
int numb;                       /* inode number (ANSI: may not be unshort) */
{
/* Find a slot in the inode table, load the specified inode into it, and
 * return a pointer to the slot.  If 'dev' == NO_DEV, just return a free slot.
 */

  register struct inode *rip, *xp;

  /* Search the inode table both for (dev, numb) and a free slot. */
  xp = NIL_INODE;
  for (rip = &inode[0]; rip < &inode[NR_INODES]; rip++) {
        if (rip->i_count > 0) { /* only check used slots for (dev, numb) */
                if (rip->i_dev == dev && rip->i_num == numb) {
                        /* This is the inode that we are looking for. */
                        rip->i_count++;
                        return(rip);    /* (dev, numb) found */
                }
        } else {
                xp = rip;       /* remember this free slot for later */
        }
  }

  /* Inode we want is not currently in use.  Did we find a free slot? */
  if (xp == NIL_INODE) {        /* inode table completely full */
        err_code = ENFILE;
        return(NIL_INODE);
  }

  /* A free inode slot has been located.  Load the inode into it. */
  xp->i_dev = dev;
  xp->i_num = numb;
  xp->i_count = 1;
  if (dev != NO_DEV) rw_inode(xp, READING);     /* get inode from disk */
  xp->i_update = 0;             /* all the times are initially up-to-date */

  return(xp);
}

/*===========================================================================*
 *                              put_inode                                    *
 *===========================================================================*/
PUBLIC void put_inode(rip)
register struct inode *rip;     /* pointer to inode to be released */
{
/* The caller is no longer using this inode.  If no one else is using it either
 * write it back to the disk immediately.  If it has no links, truncate it and
 * return it to the pool of available inodes.
 */

  if (rip == NIL_INODE) return; /* checking here is easier than in caller */
  if (--rip->i_count == 0) {    /* i_count == 0 means no one is using it now */
        if (rip->i_nlinks == 0) {
                /* i_nlinks == 0 means free the inode. */
                truncate(rip);  /* return all the disk blocks */
                rip->i_mode = I_NOT_ALLOC;      /* clear I_TYPE field */
                rip->i_dirt = DIRTY;
                free_inode(rip->i_dev, rip->i_num);
        } else {
                if (rip->i_pipe == I_PIPE) truncate(rip);
        }
        rip->i_pipe = NO_PIPE;  /* should always be cleared */
        if (rip->i_dirt == DIRTY) rw_inode(rip, WRITING);
  }
}

/*===========================================================================*
 *                              alloc_inode                                  *
 *===========================================================================*/
PUBLIC struct inode *alloc_inode(dev_t dev, mode_t bits)
{
/* Allocate a free inode on 'dev', and return a pointer to it. */

  register struct inode *rip;
  register struct super_block *sp;
  int major, minor, inumb;
  bit_t b;

  sp = get_super(dev);  /* get pointer to super_block */
  if (sp->s_rd_only) {  /* can't allocate an inode on a read only device. */
        err_code = EROFS;
        return(NIL_INODE);
  }

  /* Acquire an inode from the bit map. */
  b = alloc_bit(sp, IMAP, sp->s_isearch);
  if (b == NO_BIT) {
        err_code = ENFILE;
        major = (int) (sp->s_dev >> MAJOR) & BYTE;
        minor = (int) (sp->s_dev >> MINOR) & BYTE;
        printf("Out of i-nodes on %sdevice %d/%d\n",
                sp->s_dev == root_dev ? "root " : "", major, minor);
        return(NIL_INODE);
  }
  sp->s_isearch = b;            /* next time start here */
  inumb = (int) b;              /* be careful not to pass unshort as param */

  /* Try to acquire a slot in the inode table. */
  if ((rip = get_inode(NO_DEV, inumb)) == NIL_INODE) {
        /* No inode table slots available.  Free the inode just allocated. */
        free_bit(sp, IMAP, b);
  } else {
        /* An inode slot is available. Put the inode just allocated into it. */
        rip->i_mode = bits;             /* set up RWX bits */
        rip->i_nlinks = 0;              /* initial no links */
        rip->i_uid = fp->fp_effuid;     /* file's uid is owner's */
        rip->i_gid = fp->fp_effgid;     /* ditto group id */
        rip->i_dev = dev;               /* mark which device it is on */
        rip->i_ndzones = sp->s_ndzones; /* number of direct zones */
        rip->i_nindirs = sp->s_nindirs; /* number of indirect zones per blk*/
        rip->i_sp = sp;                 /* pointer to super block */

        /* Fields not cleared already are cleared in wipe_inode().  They have
         * been put there because truncate() needs to clear the same fields if
         * the file happens to be open while being truncated.  It saves space
         * not to repeat the code twice.
         */
        wipe_inode(rip);
  }

  return(rip);
}

/*===========================================================================*
 *                              wipe_inode                                   *
 *===========================================================================*/
PUBLIC void wipe_inode(rip)
register struct inode *rip;     /* the inode to be erased */
{
/* Erase some fields in the inode.  This function is called from alloc_inode()
 * when a new inode is to be allocated, and from truncate(), when an existing
 * inode is to be truncated.
 */

  register int i;

  rip->i_size = 0;
  rip->i_update = ATIME | CTIME | MTIME;        /* update all times later */
  rip->i_dirt = DIRTY;
  for (i = 0; i < V2_NR_TZONES; i++) rip->i_zone[i] = NO_ZONE;
}

/*===========================================================================*
 *                              free_inode                                   *
 *===========================================================================*/
PUBLIC void free_inode(dev, inumb)
dev_t dev;                      /* on which device is the inode */
ino_t inumb;                    /* number of inode to be freed */
{
/* Return an inode to the pool of unallocated inodes. */

```

```
  register struct super_block *sp;
  bit_t b;

  /* Locate the appropriate super_block. */
  sp = get_super(dev);
  if (inumb <= 0 || inumb > sp->s_ninodes) return;
  b = inumb;
  free_bit(sp, IMAP, b);
  if (b < sp->s_isearch) sp->s_isearch = b;
}

/*===========================================================================*
 *                              update_times                                 *
 *===========================================================================*/
PUBLIC void update_times(rip)
register struct inode *rip;     /* pointer to inode to be read/written */
{
/* Various system calls are required by the standard to update atime, ctime,
 * or mtime.  Since updating a time requires sending a message to the clock
 * task--an expensive business--the times are marked for update by setting
 * bits in i_update.  When a stat, fstat, or sync is done, or an inode is
 * released, update_times() may be called to actually fill in the times.
 */

  time_t cur_time;
  struct super_block *sp;

  sp = rip->i_sp;               /* get pointer to super block. */
  if (sp->s_rd_only) return;    /* no updates for read-only file systems */

  cur_time = clock_time();
  if (rip->i_update & ATIME) rip->i_atime = cur_time;
  if (rip->i_update & CTIME) rip->i_ctime = cur_time;
  if (rip->i_update & MTIME) rip->i_mtime = cur_time;
  rip->i_update = 0;            /* they are all up-to-date now */
}

/*===========================================================================*
 *                              rw_inode                                     *
 *===========================================================================*/
PUBLIC void rw_inode(rip, rw_flag)
register struct inode *rip;     /* pointer to inode to be read/written */
int rw_flag;                    /* READING or WRITING */
{
/* An entry in the inode table is to be copied to or from the disk. */

  register struct buf *bp;
  register struct super_block *sp;
  d1_inode *dip;
  d2_inode *dip2;
  block_t b, offset;

  /* Get the block where the inode resides. */
  sp = get_super(rip->i_dev);   /* get pointer to super block */
  rip->i_sp = sp;               /* inode must contain super block pointer */
  offset = sp->s_imap_blocks + sp->s_zmap_blocks + 2;
  b = (block_t) (rip->i_num - 1)/sp->s_inodes_per_block + offset;
  bp = get_block(rip->i_dev, b, NORMAL);
  dip  = bp->b_v1_ino + (rip->i_num - 1) % V1_INODES_PER_BLOCK;
  dip2 = bp->b_v2_ino + (rip->i_num - 1) %
```

```
          V2_INODES_PER_BLOCK(sp->s_block_size);

  /* Do the read or write. */
  if (rw_flag == WRITING) {
        if (rip->i_update) update_times(rip);   /* times need updating */
        if (sp->s_rd_only == FALSE) bp->b_dirt = DIRTY;
  }

  /* Copy the inode from the disk block to the in-core table or vice versa.
   * If the fourth parameter below is FALSE, the bytes are swapped.
   */
  if (sp->s_version == V1)
        old_icopy(rip, dip,  rw_flag, sp->s_native);
  else
        new_icopy(rip, dip2, rw_flag, sp->s_native);

  put_block(bp, INODE_BLOCK);
  rip->i_dirt = CLEAN;
}

/*===========================================================================*
 *                              old_icopy                                    *
 *===========================================================================*/
PRIVATE void old_icopy(rip, dip, direction, norm)
register struct inode *rip;     /* pointer to the in-core inode struct */
register d1_inode *dip;         /* pointer to the d1_inode inode struct */
int direction;                  /* READING (from disk) or WRITING (to disk) */
int norm;                       /* TRUE = do not swap bytes; FALSE = swap */

{
/* The V1.x IBM disk, the V1.x 68000 disk, and the V2 disk (same for IBM and
 * 68000) all have different inode layouts.  When an inode is read or written
 * this routine handles the conversions so that the information in the inode
 * table is independent of the disk structure from which the inode came.
 * The old_icopy routine copies to and from V1 disks.
 */

  int i;

  if (direction == READING) {
        /* Copy V1.x inode to the in-core table, swapping bytes if need be. */
        rip->i_mode    = conv2(norm, (int) dip->d1_mode);
        rip->i_uid     = conv2(norm, (int) dip->d1_uid );
        rip->i_size    = conv4(norm,       dip->d1_size);
        rip->i_mtime   = conv4(norm,       dip->d1_mtime);
        rip->i_atime   = rip->i_mtime;
        rip->i_ctime   = rip->i_mtime;
        rip->i_nlinks  = dip->d1_nlinks;                /* 1 char */
        rip->i_gid     = dip->d1_gid;                   /* 1 char */
        rip->i_ndzones = V1_NR_DZONES;
        rip->i_nindirs = V1_INDIRECTS;
        for (i = 0; i < V1_NR_TZONES; i++)
                rip->i_zone[i] = conv2(norm, (int) dip->d1_zone[i]);
  } else {
        /* Copying V1.x inode to disk from the in-core table. */
        dip->d1_mode   = conv2(norm, (int) rip->i_mode);
        dip->d1_uid    = conv2(norm, (int) rip->i_uid );
        dip->d1_size   = conv4(norm,       rip->i_size);
        dip->d1_mtime  = conv4(norm,       rip->i_mtime);
        dip->d1_nlinks = rip->i_nlinks;                 /* 1 char */
        dip->d1_gid    = rip->i_gid;                    /* 1 char */
        for (i = 0; i < V1_NR_TZONES; i++)
                dip->d1_zone[i] = conv2(norm, (int) rip->i_zone[i]);
  }
}

/*===========================================================================*
 *                              new_icopy                                    *
 *===========================================================================*/
PRIVATE void new_icopy(rip, dip, direction, norm)
register struct inode *rip;     /* pointer to the in-core inode struct */
register d2_inode *dip; /* pointer to the d2_inode struct */
int direction;                  /* READING (from disk) or WRITING (to disk) */
int norm;                       /* TRUE = do not swap bytes; FALSE = swap */

{
/* Same as old_icopy, but to/from V2 disk layout. */

  int i;

  if (direction == READING) {
        /* Copy V2.x inode to the in-core table, swapping bytes if need be. */
        rip->i_mode    = conv2(norm,dip->d2_mode);
        rip->i_uid     = conv2(norm,dip->d2_uid);
        rip->i_nlinks  = conv2(norm,dip->d2_nlinks);
        rip->i_gid     = conv2(norm,dip->d2_gid);
        rip->i_size    = conv4(norm,dip->d2_size);
        rip->i_atime   = conv4(norm,dip->d2_atime);
        rip->i_ctime   = conv4(norm,dip->d2_ctime);
        rip->i_mtime   = conv4(norm,dip->d2_mtime);
        rip->i_ndzones = V2_NR_DZONES;
        rip->i_nindirs = V2_INDIRECTS(rip->i_sp->s_block_size);
        for (i = 0; i < V2_NR_TZONES; i++)
                rip->i_zone[i] = conv4(norm, (long) dip->d2_zone[i]);
  } else {
        /* Copying V2.x inode to disk from the in-core table. */
        dip->d2_mode   = conv2(norm,rip->i_mode);
        dip->d2_uid    = conv2(norm,rip->i_uid);
        dip->d2_nlinks = conv2(norm,rip->i_nlinks);
        dip->d2_gid    = conv2(norm,rip->i_gid);
        dip->d2_size   = conv4(norm,rip->i_size);
        dip->d2_atime  = conv4(norm,rip->i_atime);
        dip->d2_ctime  = conv4(norm,rip->i_ctime);
        dip->d2_mtime  = conv4(norm,rip->i_mtime);
        for (i = 0; i < V2_NR_TZONES; i++)
                dip->d2_zone[i] = conv4(norm, (long) rip->i_zone[i]);
  }
}

/*===========================================================================*
 *                              dup_inode                                    *
 *===========================================================================*/
PUBLIC void dup_inode(ip)
struct inode *ip;               /* The inode to be duplicated. */
{
/* This routine is a simplified form of get_inode() for the case where
 * the inode pointer is already known.
 */

  ip->i_count++;
```

```
}
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                    servers/fs/super.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/* This file manages the super block table and the related data structures,
 * namely, the bit maps that keep track of which zones and which inodes are
 * allocated and which are free.  When a new inode or zone is needed, the
 * appropriate bit map is searched for a free entry.
 *
 * The entry points into this file are
 *   alloc_bit:       somebody wants to allocate a zone or inode; find one
 *   free_bit:        indicate that a zone or inode is available for allocation
 *   get_super:       search the 'superblock' table for a device
 *   mounted:         tells if file inode is on mounted (or ROOT) file system
 *   read_super:      read a superblock
 */

#include "fs.h"
#include <string.h>
#include <minix/com.h>
#include "buf.h"
#include "inode.h"
#include "super.h"
#include "const.h"

/*===========================================================================*
 *                              alloc_bit                                    *
 *===========================================================================*/
PUBLIC bit_t alloc_bit(sp, map, origin)
struct super_block *sp;         /* the filesystem to allocate from */
int map;                        /* IMAP (inode map) or ZMAP (zone map) */
bit_t origin;                   /* number of bit to start searching at */
{
/* Allocate a bit from a bit map and return its bit number. */

  block_t start_block;          /* first bit block */
  bit_t map_bits;               /* how many bits are there in the bit map? */
  unsigned bit_blocks;          /* how many blocks are there in the bit map? */
  unsigned block, word, bcount;
  struct buf *bp;
  bitchunk_t *wptr, *wlim, k;
  bit_t i, b;

  if (sp->s_rd_only)
        panic(__FILE__,"can't allocate bit on read-only filesys.", NO_NUM);

  if (map == IMAP) {
        start_block = START_BLOCK;
        map_bits = sp->s_ninodes + 1;
        bit_blocks = sp->s_imap_blocks;
  } else {
        start_block = START_BLOCK + sp->s_imap_blocks;
        map_bits = sp->s_zones - (sp->s_firstdatazone - 1);
        bit_blocks = sp->s_zmap_blocks;
  }

  /* Figure out where to start the bit search (depends on 'origin'). */
  if (origin >= map_bits) origin = 0;   /* for robustness */

  /* Locate the starting place. */
  block = origin / FS_BITS_PER_BLOCK(sp->s_block_size);
  word = (origin % FS_BITS_PER_BLOCK(sp->s_block_size)) / FS_BITCHUNK_BITS;

  /* Iterate over all blocks plus one, because we start in the middle. */
  bcount = bit_blocks + 1;
  do {
        bp = get_block(sp->s_dev, start_block + block, NORMAL);
        wlim = &bp->b_bitmap[FS_BITMAP_CHUNKS(sp->s_block_size)];

        /* Iterate over the words in block. */
        for (wptr = &bp->b_bitmap[word]; wptr < wlim; wptr++) {

                /* Does this word contain a free bit? */
                if (*wptr == (bitchunk_t) ~0) continue;

                /* Find and allocate the free bit. */
                k = conv2(sp->s_native, (int) *wptr);
                for (i = 0; (k & (1 << i)) != 0; ++i) {}

                /* Bit number from the start of the bit map. */
                b = ((bit_t) block * FS_BITS_PER_BLOCK(sp->s_block_size))
                    + (wptr - &bp->b_bitmap[0]) * FS_BITCHUNK_BITS
                    + i;

                /* Don't allocate bits beyond the end of the map. */
                if (b >= map_bits) break;

                /* Allocate and return bit number. */
                k |= 1 << i;
                *wptr = conv2(sp->s_native, (int) k);
                bp->b_dirt = DIRTY;
                put_block(bp, MAP_BLOCK);
                return(b);
        }
        put_block(bp, MAP_BLOCK);
        if (++block >= bit_blocks) block = 0;   /* last block, wrap around */
        word = 0;
  } while (--bcount > 0);
  return(NO_BIT);               /* no bit could be allocated */
}

/*===========================================================================*
 *                              free_bit                                     *
 *===========================================================================*/
PUBLIC void free_bit(sp, map, bit_returned)
struct super_block *sp;         /* the filesystem to operate on */
int map;                        /* IMAP (inode map) or ZMAP (zone map) */
bit_t bit_returned;             /* number of bit to insert into the map */
{
/* Return a zone or inode by turning off its bitmap bit. */

  unsigned block, word, bit;
  struct buf *bp;
  bitchunk_t k, mask;
  block_t start_block;

  if (sp->s_rd_only)
        panic(__FILE__,"can't free bit on read-only filesys.", NO_NUM);

  if (map == IMAP) {
        start_block = START_BLOCK;
  } else {
        start_block = START_BLOCK + sp->s_imap_blocks;
  }
  block = bit_returned / FS_BITS_PER_BLOCK(sp->s_block_size);
  word = (bit_returned % FS_BITS_PER_BLOCK(sp->s_block_size))
         / FS_BITCHUNK_BITS;

  bit = bit_returned % FS_BITCHUNK_BITS;
  mask = 1 << bit;

  bp = get_block(sp->s_dev, start_block + block, NORMAL);

  k = conv2(sp->s_native, (int) bp->b_bitmap[word]);
  if (!(k & mask)) {
        panic(__FILE__,map == IMAP ? "tried to free unused inode" :
              "tried to free unused block", NO_NUM);
  }

  k &= ~mask;
  bp->b_bitmap[word] = conv2(sp->s_native, (int) k);
  bp->b_dirt = DIRTY;

  put_block(bp, MAP_BLOCK);
}

/*===========================================================================*
 *                              get_super                                    *
 *===========================================================================*/
PUBLIC struct super_block *get_super(dev)
dev_t dev;                      /* device number whose super_block is sought */
{
/* Search the superblock table for this device.  It is supposed to be there. */

  register struct super_block *sp;

  if (dev == NO_DEV)
        panic(__FILE__,"request for super_block of NO_DEV", NO_NUM);

  for (sp = &super_block[0]; sp < &super_block[NR_SUPERS]; sp++)
        if (sp->s_dev == dev) return(sp);

  /* Search failed.  Something wrong. */
  panic(__FILE__,"can't find superblock for device (in decimal)", (int) dev);

  return(NIL_SUPER);            /* to keep the compiler and lint quiet */
}

/*===========================================================================*
 *                              get_block_size                               *
 *===========================================================================*/
PUBLIC int get_block_size(dev_t dev)
{
/* Search the superblock table for this device. */
```

```

  register struct super_block *sp;

  if (dev == NO_DEV)
        panic(__FILE__,"request for block size of NO_DEV", NO_NUM);

  for (sp = &super_block[0]; sp < &super_block[NR_SUPERS]; sp++) {
        if (sp->s_dev == dev) {
                return(sp->s_block_size);
        }
  }

  /* no mounted filesystem? use this block size then. */
  return MIN_BLOCK_SIZE;
}

/*===========================================================================*
 *                              mounted                                      *
 *===========================================================================*/
PUBLIC int mounted(rip)
register struct inode *rip;     /* pointer to inode */
{
/* Report on whether the given inode is on a mounted (or ROOT) file system. */

  register struct super_block *sp;
  register dev_t dev;

  dev = (dev_t) rip->i_zone[0];
  if (dev == root_dev) return(TRUE);    /* inode is on root file system */

  for (sp = &super_block[0]; sp < &super_block[NR_SUPERS]; sp++)
        if (sp->s_dev == dev) return(TRUE);

  return(FALSE);
}

/*===========================================================================*
 *                              read_super                                   *
 *===========================================================================*/
PUBLIC int read_super(sp)
register struct super_block *sp; /* pointer to a superblock */
{
/* Read a superblock. */
  dev_t dev;
  int magic;
  int version, native, r;
  static char sbbuf[MIN_BLOCK_SIZE];

  dev = sp->s_dev;              /* save device (will be overwritten by copy) */
  if (dev == NO_DEV)
        panic(__FILE__,"request for super_block of NO_DEV", NO_NUM);
  r = dev_io(DEV_READ, dev, FS_PROC_NR,
        sbbuf, SUPER_BLOCK_BYTES, MIN_BLOCK_SIZE, 0);
  if (r != MIN_BLOCK_SIZE) {
        return EINVAL;
  }
  memcpy(sp, sbbuf, sizeof(*sp));
  sp->s_dev = NO_DEV;           /* restore later */
  magic = sp->s_magic;          /* determines file system type */

  /* Get file system version and type. */
  if (magic == SUPER_MAGIC || magic == conv2(BYTE_SWAP, SUPER_MAGIC)) {
        version = V1;
        native  = (magic == SUPER_MAGIC);
  } else if (magic == SUPER_V2 || magic == conv2(BYTE_SWAP, SUPER_V2)) {
        version = V2;
        native  = (magic == SUPER_V2);
  } else if (magic == SUPER_V3) {
        version = V3;
        native = 1;
  } else {
        return(EINVAL);
  }

  /* If the super block has the wrong byte order, swap the fields; the magic
   * number doesn't need conversion. */
  sp->s_ninodes =       conv4(native, sp->s_ninodes);
  sp->s_nzones =        conv2(native, (int) sp->s_nzones);
  sp->s_imap_blocks =   conv2(native, (int) sp->s_imap_blocks);
  sp->s_zmap_blocks =   conv2(native, (int) sp->s_zmap_blocks);
  sp->s_firstdatazone = conv2(native, (int) sp->s_firstdatazone);
  sp->s_log_zone_size = conv2(native, (int) sp->s_log_zone_size);
  sp->s_max_size =      conv4(native, sp->s_max_size);
  sp->s_zones =         conv4(native, sp->s_zones);

  /* In V1, the device size was kept in a short, s_nzones, which limited
   * devices to 32K zones.  For V2, it was decided to keep the size as a
   * long.  However, just changing s_nzones to a long would not work, since
   * then the position of s_magic in the super block would not be the same
   * in V1 and V2 file systems, and there would be no way to tell whether
   * a newly mounted file system was V1 or V2.  The solution was to introduce
   * a new variable, s_zones, and copy the size there.
   *
   * Calculate some other numbers that depend on the version here too, to
   * hide some of the differences.
   */
  if (version == V1) {
        sp->s_block_size = STATIC_BLOCK_SIZE;
        sp->s_zones = sp->s_nzones;     /* only V1 needs this copy */
        sp->s_inodes_per_block = V1_INODES_PER_BLOCK;
        sp->s_ndzones = V1_NR_DZONES;
        sp->s_nindirs = V1_INDIRECTS;
  } else {
        if (version == V2)
                sp->s_block_size = STATIC_BLOCK_SIZE;
        if (sp->s_block_size < MIN_BLOCK_SIZE)
                return EINVAL;
        sp->s_inodes_per_block = V2_INODES_PER_BLOCK(sp->s_block_size);
        sp->s_ndzones = V2_NR_DZONES;
        sp->s_nindirs = V2_INDIRECTS(sp->s_block_size);
  }

  if (sp->s_block_size < MIN_BLOCK_SIZE) {
        return EINVAL;
  }
  if (sp->s_block_size > MAX_BLOCK_SIZE) {
        printf("Filesystem block size is %d kB; maximum filesystem\n"
        "block size is %d kB. This limit can be increased by recompiling.\n",
        sp->s_block_size/1024, MAX_BLOCK_SIZE/1024);
        return EINVAL;
  }
  if ((sp->s_block_size % 512) != 0) {
        return EINVAL;
  }
  if (SUPER_SIZE > sp->s_block_size) {
        return EINVAL;
  }
  if ((sp->s_block_size % V2_INODE_SIZE) != 0 ||
     (sp->s_block_size % V1_INODE_SIZE) != 0) {
        return EINVAL;
  }

  sp->s_isearch = 0;            /* inode searches initially start at 0 */
  sp->s_zsearch = 0;            /* zone searches initially start at 0 */
  sp->s_version = version;
  sp->s_native  = native;

  /* Make a few basic checks to see if super block looks reasonable. */
  if (sp->s_imap_blocks < 1 || sp->s_zmap_blocks < 1
                                || sp->s_ninodes < 1 || sp->s_zones < 1
                                || (unsigned) sp->s_log_zone_size > 4) {
        printf("not enough imap or zone map blocks, \n");
        printf("or not enough inodes, or not enough zones, "
                "or zone size too large\n");
        return(EINVAL);
  }
  sp->s_dev = dev;              /* restore device number */
  return(OK);
}
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                     servers/fs/filedes.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/* This file contains the procedures that manipulate file descriptors.
 *
 * The entry points into this file are
 *   get_fd:    look for free file descriptor and free filp slots
 *   get_filp:  look up the filp entry for a given file descriptor
 *   find_filp: find a filp slot that points to a given inode
 */

#include "fs.h"
#include "file.h"
#include "fproc.h"
#include "inode.h"

/*===========================================================================*
 *                              get_fd                                       *
 *===========================================================================*/
PUBLIC int get_fd(int start, mode_t bits, int *k, struct filp **fpt)
{
/* Look for a free file descriptor and a free filp slot.  Fill in the mode word
 * in the latter, but don't claim either one yet, since the open() or creat()
 * may yet fail.
 */

  register struct filp *f;
  register int i;

  *k = -1;                      /* we need a way to tell if file desc found */

  /* Search the fproc fp_filp table for a free file descriptor. */
  for (i = start; i < OPEN_MAX; i++) {
        if (fp->fp_filp[i] == NIL_FILP) {
                /* A file descriptor has been located. */
                *k = i;
                break;
        }
  }

  /* Check to see if a file descriptor has been found. */
  if (*k < 0) return(EMFILE);   /* this is why we initialized k to -1 */

  /* Now that a file descriptor has been found, look for a free filp slot. */
  for (f = &filp[0]; f < &filp[NR_FILPS]; f++) {
        if (f->filp_count == 0) {
                f->filp_mode = bits;
                f->filp_pos = 0L;
                f->filp_selectors = 0;
                f->filp_select_ops = 0;
                f->filp_pipe_select_ops = 0;
                f->filp_flags = 0;
                *fpt = f;
                return(OK);
        }
  }

  /* If control passes here, the filp table must be full.  Report that back. */
  return(ENFILE);
}

/*===========================================================================*
 *                              get_filp                                     *
 *===========================================================================*/
PUBLIC struct filp *get_filp(fild)
int fild;                       /* file descriptor */
{
/* See if 'fild' refers to a valid file descr.  If so, return its filp ptr. */

  err_code = EBADF;
  if (fild < 0 || fild >= OPEN_MAX ) return(NIL_FILP);
  return(fp->fp_filp[fild]);    /* may also be NIL_FILP */
}

/*===========================================================================*
 *                              find_filp                                    *
 *===========================================================================*/
PUBLIC struct filp *find_filp(register struct inode *rip, mode_t bits)
{
/* Find a filp slot that refers to the inode 'rip' in a way as described
 * by the mode bit 'bits'. Used for determining whether somebody is still
 * interested in either end of a pipe.  Also used when opening a FIFO to
 * find partners to share a filp field with (to shared the file position).
 * Like 'get_fd' it performs its job by linear search through the filp table.
 */

  register struct filp *f;

  for (f = &filp[0]; f < &filp[NR_FILPS]; f++) {
        if (f->filp_count != 0 && f->filp_ino == rip && (f->filp_mode & bits)){
                return(f);
        }
  }

  /* If control passes here, the filp wasn't there.  Report that back. */
  return(NIL_FILP);
}
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                              servers/fs/lock.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/* This file handles advisory file locking as required by POSIX.
 *
 * The entry points into this file are
 *   lock_op:   perform locking operations for FCNTL system call
 *   lock_revive: revive processes when a lock is released
 */

#include "fs.h"
#include <minix/com.h>
#include <fcntl.h>
#include <unistd.h>
#include "file.h"
#include "fproc.h"
#include "inode.h"
#include "lock.h"
#include "param.h"

/*===========================================================================*
 *                              lock_op                                      *
 *===========================================================================*/
PUBLIC int lock_op(f, req)
struct filp *f;
int req;                        /* either F_SETLK or F_SETLKW */
{
/* Perform the advisory locking required by POSIX. */

  int r, ltype, i, conflict = 0, unlocking = 0;
  mode_t mo;
  off_t first, last;
  struct flock flock;
  vir_bytes user_flock;
  struct file_lock *flp, *flp2, *empty;

  /* Fetch the flock structure from user space. */
  user_flock = (vir_bytes) m_in.name1;
  r = sys_datacopy(who, (vir_bytes) user_flock,
        FS_PROC_NR, (vir_bytes) &flock, (phys_bytes) sizeof(flock));
  if (r != OK) return(EINVAL);

  /* Make some error checks. */
  ltype = flock.l_type;
  mo = f->filp_mode;
  if (ltype != F_UNLCK && ltype != F_RDLCK && ltype != F_WRLCK) return(EINVAL);
  if (req == F_GETLK && ltype == F_UNLCK) return(EINVAL);
  if ( (f->filp_ino->i_mode & I_TYPE) != I_REGULAR) return(EINVAL);
```

```
        if (req != F_GETLK && ltype == F_RDLCK && (mo & R_BIT) == 0) return(EBADF);
        if (req != F_GETLK && ltype == F_WRLCK && (mo & W_BIT) == 0) return(EBADF);

        /* Compute the first and last bytes in the lock region. */
        switch (flock.l_whence) {
              case SEEK_SET:  first = 0; break;
              case SEEK_CUR:  first = f->filp_pos; break;
              case SEEK_END:  first = f->filp_ino->i_size; break;
              default:        return(EINVAL);
        }
        /* Check for overflow. */
        if (((long)flock.l_start > 0) && ((first + flock.l_start) < first))
              return(EINVAL);
        if (((long)flock.l_start < 0) && ((first + flock.l_start) > first))
              return(EINVAL);
        first = first + flock.l_start;
        last = first + flock.l_len - 1;
        if (flock.l_len == 0) last = MAX_FILE_POS;
        if (last < first) return(EINVAL);

        /* Check if this region conflicts with any existing lock. */
        empty = (struct file_lock *) 0;
        for (flp = &file_lock[0]; flp < & file_lock[NR_LOCKS]; flp++) {
              if (flp->lock_type == 0) {
                      if (empty == (struct file_lock *) 0) empty = flp;
                      continue;       /* 0 means unused slot */
              }
              if (flp->lock_inode != f->filp_ino) continue;   /* different file */
              if (last < flp->lock_first) continue;   /* new one is in front */
              if (first > flp->lock_last) continue;   /* new one is afterwards */
              if (ltype == F_RDLCK && flp->lock_type == F_RDLCK) continue;
              if (ltype != F_UNLCK && flp->lock_pid == fp->fp_pid) continue;

              /* There might be a conflict.  Process it. */
              conflict = 1;
              if (req == F_GETLK) break;

              /* If we are trying to set a lock, it just failed. */
              if (ltype == F_RDLCK || ltype == F_WRLCK) {
                      if (req == F_SETLK) {
                              /* For F_SETLK, just report back failure. */
                              return(EAGAIN);
                      } else {
                              /* For F_SETLKW, suspend the process. */
                              suspend(XLOCK);
                              return(SUSPEND);
                      }
              }

              /* We are clearing a lock and we found something that overlaps. */
              unlocking = 1;
              if (first <= flp->lock_first && last >= flp->lock_last) {
                      flp->lock_type = 0;     /* mark slot as unused */
                      nr_locks--;             /* number of locks is now 1 less */
                      continue;
              }

              /* Part of a locked region has been unlocked. */
              if (first <= flp->lock_first) {
                      flp->lock_first = last + 1;
```

```
                        continue;
                }

                if (last >= flp->lock_last) {
                        flp->lock_last = first - 1;
                        continue;
                }

                /* Bad luck. A lock has been split in two by unlocking the middle. */
                if (nr_locks == NR_LOCKS) return(ENOLCK);
                for (i = 0; i < NR_LOCKS; i++)
                        if (file_lock[i].lock_type == 0) break;
                flp2 = &file_lock[i];
                flp2->lock_type = flp->lock_type;
                flp2->lock_pid = flp->lock_pid;
                flp2->lock_inode = flp->lock_inode;
                flp2->lock_first = last + 1;
                flp2->lock_last = flp->lock_last;
                flp->lock_last = first - 1;
                nr_locks++;
        }
        if (unlocking) lock_revive();

        if (req == F_GETLK) {
                if (conflict) {
                        /* GETLK and conflict. Report on the conflicting lock. */
                        flock.l_type = flp->lock_type;
                        flock.l_whence = SEEK_SET;
                        flock.l_start = flp->lock_first;
                        flock.l_len = flp->lock_last - flp->lock_first + 1;
                        flock.l_pid = flp->lock_pid;

                } else {
                        /* It is GETLK and there is no conflict. */
                        flock.l_type = F_UNLCK;
                }

                /* Copy the flock structure back to the caller. */
                r = sys_datacopy(FS_PROC_NR, (vir_bytes) &flock,
                        who, (vir_bytes) user_flock, (phys_bytes) sizeof(flock));
                return(r);
        }

        if (ltype == F_UNLCK) return(OK);       /* unlocked a region with no locks */

        /* There is no conflict.  If space exists, store new lock in the table. */
        if (empty == (struct file_lock *) 0) return(ENOLCK);  /* table full */
        empty->lock_type = ltype;
        empty->lock_pid = fp->fp_pid;
        empty->lock_inode = f->filp_ino;
        empty->lock_first = first;
        empty->lock_last = last;
        nr_locks++;
        return(OK);
}
```

```
/*===========================================================================*
 *                              lock_revive                                  *
 *===========================================================================*/
PUBLIC void lock_revive()
{
/* Go find all the processes that are waiting for any kind of lock and
 * revive them all.  The ones that are still blocked will block again when
 * they run.  The others will complete.  This strategy is a space-time
 * tradeoff.  Figuring out exactly which ones to unblock now would take
 * extra code, and the only thing it would win would be some performance in
 * extremely rare circumstances (namely, that somebody actually used
 * locking).
 */

  int task;
  struct fproc *fptr;

  for (fptr = &fproc[INIT_PROC_NR + 1]; fptr < &fproc[NR_PROCS]; fptr++){
        task = -fptr->fp_task;
        if (fptr->fp_suspended == SUSPENDED && task == XLOCK) {
                revive( (int) (fptr - fproc), 0);
        }
  }
}
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                      servers/fs/main.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/* This file contains the main program of the File System.  It consists of
 * a loop that gets messages requesting work, carries out the work, and sends
 * replies.
 *
 * The entry points into this file are:
 *   main:      main program of the File System
 *   reply:     send a reply to a process after the requested work is done
 *
 */

struct super_block;             /* proto.h needs to know this */

#include "fs.h"
#include <fcntl.h>
#include <string.h>
#include <stdio.h>
#include <signal.h>
#include <stdlib.h>
#include <sys/ioc_memory.h>
#include <sys/svrctl.h>
#include <minix/callnr.h>
#include <minix/com.h>
#include <minix/keymap.h>
#include <minix/const.h>
#include "buf.h"
#include "file.h"
#include "fproc.h"
#include "inode.h"
#include "param.h"
#include "super.h"
```

```

FORWARD _PROTOTYPE( void fs_init, (void)                                );
FORWARD _PROTOTYPE( int igetenv, (char *var, int optional)              );
FORWARD _PROTOTYPE( void get_work, (void)                               );
FORWARD _PROTOTYPE( void load_ram, (void)                               );
FORWARD _PROTOTYPE( void load_super, (Dev_t super_dev)                  );

/*===========================================================================*
 *                              main                                         *
 *===========================================================================*/
PUBLIC int main()
{
/* This is the main program of the file system.  The main loop consists of
 * three major activities: getting new work, processing the work, and sending
 * the reply.  This loop never terminates as long as the file system runs.
 */
  sigset_t sigset;
  int error;

  fs_init();

  /* This is the main loop that gets work, processes it, and sends replies. */
  while (TRUE) {
        get_work();             /* sets who and call_nr */

        fp = &fproc[who];       /* pointer to proc table struct */
        super_user = (fp->fp_effuid == SU_UID ? TRUE : FALSE);   /* su? */

        /* Check for special control messages first. */
        if (call_nr == SYS_SIG) {
                sigset = m_in.NOTIFY_ARG;
                if (sigismember(&sigset, SIGKSTOP)) {
                        do_sync();
                        sys_exit(0);            /* never returns */
                }
        } else if (call_nr == SYN_ALARM) {
                /* Not a user request; system has expired one of our timers,
                 * currently only in use for select(). Check it.
                 */
                fs_expire_timers(m_in.NOTIFY_TIMESTAMP);
        } else if ((call_nr & NOTIFY_MESSAGE)) {
                /* Device notifies us of an event. */
                dev_status(&m_in);
        } else {
                /* Call the internal function that does the work. */
                if (call_nr < 0 || call_nr >= NCALLS) {
                        error = ENOSYS;
                        printf("FS, warning illegal %d system call by %d\n", call_nr, who);
                } else if (fp->fp_pid == PID_FREE) {
                        error = ENOSYS;
                        printf("FS, bad process, who = %d, call_nr = %d, slot1 = %d\n",
                                 who, call_nr, m_in.slot1);
                } else {
                        error = (*call_vec[call_nr])();
                }

                /* Copy the results back to the user and send reply. */
                if (error != SUSPEND) { reply(who, error); }
                if (rdahed_inode != NIL_INODE) {
                        read_ahead(); /* do block read ahead */
                }
        }
  }
  return(OK);                           /* shouldn't come here */
}

/*===========================================================================*
 *                              get_work                                     *
 *===========================================================================*/
PRIVATE void get_work()
{
  /* Normally wait for new input.  However, if 'reviving' is
   * nonzero, a suspended process must be awakened.
   */
  register struct fproc *rp;

  if (reviving != 0) {
        /* Revive a suspended process. */
        for (rp = &fproc[0]; rp < &fproc[NR_PROCS]; rp++)
                if (rp->fp_revived == REVIVING) {
                        who = (int)(rp - fproc);
                        call_nr = rp->fp_fd & BYTE;
                        m_in.fd = (rp->fp_fd >>8) & BYTE;
                        m_in.buffer = rp->fp_buffer;
                        m_in.nbytes = rp->fp_nbytes;
                        rp->fp_suspended = NOT_SUSPENDED; /*no longer hanging*/
                        rp->fp_revived = NOT_REVIVING;
                        reviving--;
                        return;
                }
        panic(__FILE__,"get_work couldn't revive anyone", NO_NUM);
  }

  /* Normal case.  No one to revive. */
  if (receive(ANY, &m_in) != OK) panic(__FILE__,"fs receive error", NO_NUM);
  who = m_in.m_source;
  call_nr = m_in.m_type;
}

/*===========================================================================*
 *                              buf_pool                                     *
 *===========================================================================*/
PRIVATE void buf_pool(void)
{
/* Initialize the buffer pool. */

  register struct buf *bp;

  bufs_in_use = 0;
  front = &buf[0];
  rear = &buf[NR_BUFS - 1];

  for (bp = &buf[0]; bp < &buf[NR_BUFS]; bp++) {
        bp->b_blocknr = NO_BLOCK;
        bp->b_dev = NO_DEV;
        bp->b_next = bp + 1;
        bp->b_prev = bp - 1;
  }
  buf[0].b_prev = NIL_BUF;
  buf[NR_BUFS - 1].b_next = NIL_BUF;

  for (bp = &buf[0]; bp < &buf[NR_BUFS]; bp++) bp->b_hash = bp->b_next;
  buf_hash[0] = front;

}

/*===========================================================================*
 *                              reply                                        *
 *===========================================================================*/
PUBLIC void reply(whom, result)
int whom;                       /* process to reply to */
int result;                     /* result of the call (usually OK or error #) */
{
/* Send a reply to a user process. It may fail (if the process has just
 * been killed by a signal), so don't check the return code.  If the send
 * fails, just ignore it.
 */
  int s;
  m_out.reply_type = result;
  s = send(whom, &m_out);
  if (s != OK) printf("FS: couldn't send reply %d: %d\n", result, s);
}

/*===========================================================================*
 *                              fs_init                                      *
 *===========================================================================*/
PRIVATE void fs_init()
{
/* Initialize global variables, tables, etc. */
  register struct inode *rip;
  register struct fproc *rfp;
  message mess;
  int s;

  /* Initialize the process table with help of the process manager messages.
   * Expect one message for each system process with its slot number and pid.
   * When no more processes follow, the magic process number NONE is sent.
   * Then, stop and synchronize with the PM.
   */
  do {
        if (OK != (s=receive(PM_PROC_NR, &mess)))
                panic(__FILE__,"FS couldn't receive from PM", s);
        if (NONE == mess.PR_PROC_NR) break;

        rfp = &fproc[mess.PR_PROC_NR];
        rfp->fp_pid = mess.PR_PID;
        rfp->fp_realuid = (uid_t) SYS_UID;
        rfp->fp_effuid = (uid_t) SYS_UID;
        rfp->fp_realgid = (gid_t) SYS_GID;
        rfp->fp_effgid = (gid_t) SYS_GID;
        rfp->fp_umask = ~0;

  } while (TRUE);                       /* continue until process NONE */
  mess.m_type = OK;                     /* tell PM that we succeeded */
  s=send(PM_PROC_NR, &mess);            /* send synchronization message */

  /* All process table entries have been set. Continue with FS initialization.
   * Certain relations must hold for the file system to work at all. Some
   * extra block_size requirements are checked at super-block-read-in time.
   */
  if (OPEN_MAX > 127) panic(__FILE__,"OPEN_MAX > 127", NO_NUM);
  if (NR_BUFS < 6) panic(__FILE__,"NR_BUFS < 6", NO_NUM);
  if (V1_INODE_SIZE != 32) panic(__FILE__,"V1 inode size != 32", NO_NUM);
  if (V2_INODE_SIZE != 64) panic(__FILE__,"V2 inode size != 64", NO_NUM);
  if (OPEN_MAX > 8 * sizeof(long))
        panic(__FILE__,"Too few bits in fp_cloexec", NO_NUM);

  /* The following initializations are needed to let dev_opcl succeed .*/
  fp = (struct fproc *) NULL;
  who = FS_PROC_NR;

  buf_pool();                   /* initialize buffer pool */
  build_dmap();                 /* build device table and map boot driver */
  load_ram();                   /* init RAM disk, load if it is root */
  load_super(root_dev);         /* load super block for root device */
  init_select();                /* init select() structures */

  /* The root device can now be accessed; set process directories. */
  for (rfp=&fproc[0]; rfp < &fproc[NR_PROCS]; rfp++) {
        if (rfp->fp_pid != PID_FREE) {
                rip = get_inode(root_dev, ROOT_INODE);
                dup_inode(rip);
                rfp->fp_rootdir = rip;
                rfp->fp_workdir = rip;
        }
  }
}

/*===========================================================================*
 *                              igetenv                                      *
 *===========================================================================*/
PRIVATE int igetenv(key, optional)
char *key;
int optional;
{
/* Ask kernel for an integer valued boot environment variable. */
  char value[64];
  int i;

  if ((i = env_get_param(key, value, sizeof(value))) != OK) {
      if (!optional)
        printf("FS: Warning, couldn't get monitor param: %d\n", i);
      return 0;
  }
  return(atoi(value));
}

/*===========================================================================*
 *                              load_ram                                     *
 *===========================================================================*/
PRIVATE void load_ram(void)
{
/* Allocate a RAM disk with size given in the boot parameters. If a RAM disk
 * image is given, the copy the entire image device block-by-block to a RAM
 * disk with the same size as the image.
 * If the root device is not set, the RAM disk will be used as root instead.
 */
  register struct buf *bp, *bp1;
  u32_t lcount, ram_size_kb;
  zone_t zones;
  struct super_block *sp, *dsp;
  block_t b;
  Dev_t image_dev;
  static char sbbuf[MIN_BLOCK_SIZE];
  int block_size_image, block_size_ram, ramfs_block_size;
  int s;

  /* Get some boot environment variables. */
  root_dev = igetenv("rootdev", 0);
  image_dev = igetenv("ramimagedev", 0);
  ram_size_kb = igetenv("ramsize", 0);

  /* Open the root device. */
  if (dev_open(root_dev, FS_PROC_NR, R_BIT|W_BIT) != OK)
        panic(__FILE__,"Cannot open root device",NO_NUM);

  /* If we must initialize a ram disk, get details from the image device. */
  if (root_dev == DEV_RAM) {
        u32_t fsmax, probedev;

        /* If we are running from CD, see if we can find it. */
        if (igetenv("cdproberoot", 1) && (probedev=cdprobe()) != NO_DEV) {
                char devnum[10];
                struct sysgetenv env;

                /* If so, this is our new RAM image device. */
                image_dev = probedev;

                /* Tell PM about it, so userland can find out about it
                 * with sysenv interface.
                 */
                env.key = "cdproberoot";
                env.keylen = strlen(env.key);
                sprintf(devnum, "%d", (int) probedev);
                env.val = devnum;
                env.vallen = strlen(devnum);
                svrctl(MMSETPARAM, &env);
        }

        /* Open image device for RAM root. */
        if (dev_open(image_dev, FS_PROC_NR, R_BIT) != OK)
                panic(__FILE__,"Cannot open RAM image device", NO_NUM);

        /* Get size of RAM disk image from the super block. */
        sp = &super_block[0];
        sp->s_dev = image_dev;
        if (read_super(sp) != OK)
                panic(__FILE__,"Bad RAM disk image FS", NO_NUM);

        lcount = sp->s_zones << sp->s_log_zone_size;    /* # blks on root dev*/

        /* Stretch the RAM disk file system to the boot parameters size, but
         * no further than the last zone bit map block allows.
         */
        if (ram_size_kb*1024 < lcount*sp->s_block_size)
                ram_size_kb = lcount*sp->s_block_size/1024;
        fsmax = (u32_t) sp->s_zmap_blocks * CHAR_BIT * sp->s_block_size;
        fsmax = (fsmax + (sp->s_firstdatazone-1)) << sp->s_log_zone_size;
        if (ram_size_kb*1024 > fsmax*sp->s_block_size)
                ram_size_kb = fsmax*sp->s_block_size/1024;
  }

  /* Tell RAM driver how big the RAM disk must be. */
  m_out.m_type = DEV_IOCTL;
  m_out.PROC_NR = FS_PROC_NR;
  m_out.DEVICE = RAM_DEV;
  m_out.REQUEST = MIOCRAMSIZE;                    /* I/O control to use */
  m_out.POSITION = (ram_size_kb * 1024);          /* request in bytes */
  if ((s=sendrec(MEM_PROC_NR, &m_out)) != OK)
        panic("FS","sendrec from MEM failed", s);
  else if (m_out.REP_STATUS != OK) {
        /* Report and continue, unless RAM disk is required as root FS. */
        if (root_dev != DEV_RAM) {
                report("FS","can't set RAM disk size", m_out.REP_STATUS);
                return;
        } else {
                panic(__FILE__,"can't set RAM disk size", m_out.REP_STATUS);
        }
  }

  /* See if we must load the RAM disk image, otherwise return. */
  if (root_dev != DEV_RAM)
        return;

  /* Copy the blocks one at a time from the image to the RAM disk. */
  printf("Loading RAM disk onto /dev/ram:\33[23CLoaded:    0 KB");

  inode[0].i_mode = I_BLOCK_SPECIAL;    /* temp inode for rahead() */
  inode[0].i_size = LONG_MAX;
  inode[0].i_dev = image_dev;
  inode[0].i_zone[0] = image_dev;

  block_size_ram = get_block_size(DEV_RAM);
  block_size_image = get_block_size(image_dev);

  /* RAM block size has to be a multiple of the root image block
   * size to make copying easier.
   */
  if (block_size_image % block_size_ram) {
        printf("\nram block size: %d image block size: %d\n",
                block_size_ram, block_size_image);
        panic(__FILE__, "ram disk block size must be a multiple of "
                "the image disk block size", NO_NUM);
  }

  /* Loading blocks from image device. */
  for (b = 0; b < (block_t) lcount; b++) {
        int rb, factor;
        bp = rahead(&inode[0], b, (off_t)block_size_image * b, block_size_image);
        factor = block_size_image/block_size_ram;
        for(rb = 0; rb < factor; rb++) {
                bp1 = get_block(root_dev, b * factor + rb, NO_READ);
                memcpy(bp1->b_data, bp->b_data + rb * block_size_ram,
                        (size_t) block_size_ram);
                bp1->b_dirt = DIRTY;
                put_block(bp1, FULL_DATA_BLOCK);
        }
        put_block(bp, FULL_DATA_BLOCK);
        if (b % 11 == 0)
        printf("\b\b\b\b\b\b\b\b\b%6ld KB", ((long) b * block_size_image)/1024L);
  }

  /* Commit changes to RAM so dev_io will see it. */
  do_sync();

  printf("\rRAM disk of %u KB loaded onto /dev/ram.", (unsigned) ram_size_kb);
  if (root_dev == DEV_RAM) printf(" Using RAM disk as root FS.");
  printf("  \n");

  /* Invalidate and close the image device. */
  invalidate(image_dev);
  dev_close(image_dev);

  /* Resize the RAM disk root file system. */
  if (dev_io(DEV_READ, root_dev, FS_PROC_NR,
        sbbuf, SUPER_BLOCK_BYTES, MIN_BLOCK_SIZE, 0) != MIN_BLOCK_SIZE) {
        printf("WARNING: ramdisk read for resizing failed\n");
  }
  dsp = (struct super_block *) sbbuf;
  if (dsp->s_magic == SUPER_V3)
        ramfs_block_size = dsp->s_block_size;
  else
        ramfs_block_size = STATIC_BLOCK_SIZE;
  zones = (ram_size_kb * 1024 / ramfs_block_size) >> sp->s_log_zone_size;

  dsp->s_nzones = conv2(sp->s_native, (u16_t) zones);
  dsp->s_zones = conv4(sp->s_native, zones);
  if (dev_io(DEV_WRITE, root_dev, FS_PROC_NR,
        sbbuf, SUPER_BLOCK_BYTES, MIN_BLOCK_SIZE, 0) != MIN_BLOCK_SIZE) {
        printf("WARNING: ramdisk write for resizing failed\n");
  }
}

/*===========================================================================*
 *                              load_super                                   *
 *===========================================================================*/
PRIVATE void load_super(super_dev)
dev_t super_dev;                        /* place to get superblock from */
{
  int bad;
  register struct super_block *sp;
  register struct inode *rip;

  /* Initialize the super_block table. */
  for (sp = &super_block[0]; sp < &super_block[NR_SUPERS]; sp++)
        sp->s_dev = NO_DEV;

  /* Read in super_block for the root file system. */
  sp = &super_block[0];
  sp->s_dev = super_dev;

  /* Check super_block for consistency. */
  bad = (read_super(sp) != OK);
  if (!bad) {
        rip = get_inode(super_dev, ROOT_INODE); /* inode for root dir */
        if ( (rip->i_mode & I_TYPE) != I_DIRECTORY || rip->i_nlinks < 3) bad++;
  }
  if (bad) panic(__FILE__,"Invalid root file system", NO_NUM);

  sp->s_imount = rip;
  dup_inode(rip);
  sp->s_isup = rip;
  sp->s_rd_only = 0;
  return;
}
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                servers/fs/open.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/* This file contains the procedures for creating, opening, closing, and
 * seeking on files.
 *
 * The entry points into this file are
 *   do_creat:  perform the CREAT system call
 *   do_open:   perform the OPEN system call
 *   do_mknod:  perform the MKNOD system call
 *   do_mkdir:  perform the MKDIR system call
 *   do_close:  perform the CLOSE system call
 *   do_lseek:  perform the LSEEK system call
 */

#include "fs.h"
#include <sys/stat.h>
#include <fcntl.h>
#include <minix/callnr.h>
#include <minix/com.h>
#include "buf.h"
#include "file.h"
#include "fproc.h"
#include "inode.h"
#include "lock.h"
#include "param.h"
#include "super.h"

#define offset m2_l1

PRIVATE char mode_map[] = {R_BIT, W_BIT, R_BIT|W_BIT, 0};

FORWARD _PROTOTYPE( int common_open, (int oflags, mode_t omode)         );
FORWARD _PROTOTYPE( int pipe_open, (struct inode *rip,mode_t bits,int oflags));
FORWARD _PROTOTYPE( struct inode *new_node, (char *path, mode_t bits,
                                                        zone_t z0)      );

/*===========================================================================*
 *                              do_creat                                     *
 *===========================================================================*/
PUBLIC int do_creat()
{
/* Perform the creat(name, mode) system call. */
  int r;

  if (fetch_name(m_in.name, m_in.name_length, M3) != OK) return(err_code);
  r = common_open(O_WRONLY | O_CREAT | O_TRUNC, (mode_t) m_in.mode);
  return(r);
}
```

```
/*===========================================================================*
 *                               do_open                                     *
 *===========================================================================*/
PUBLIC int do_open()
{
/* Perform the open(name, flags,...) system call. */

  int create_mode = 0;          /* is really mode_t but this gives problems */
  int r;

  /* If O_CREAT is set, open has three parameters, otherwise two. */
  if (m_in.mode & O_CREAT) {
        create_mode = m_in.c_mode;
        r = fetch_name(m_in.c_name, m_in.name1_length, M1);
  } else {
        r = fetch_name(m_in.name, m_in.name_length, M3);
  }

  if (r != OK) return(err_code); /* name was bad */
  r = common_open(m_in.mode, create_mode);
  return(r);
}

/*===========================================================================*
 *                               common_open                                 *
 *===========================================================================*/
PRIVATE int common_open(register int oflags, mode_t omode)
{
/* Common code from do_creat and do_open. */

  register struct inode *rip;
  int r, b, exist = TRUE;
  dev_t dev;
  mode_t bits;
  off_t pos;
  struct filp *fil_ptr, *filp2;

  /* Remap the bottom two bits of oflags. */
  bits = (mode_t) mode_map[oflags & O_ACCMODE];

  /* See if file descriptor and filp slots are available. */
  if ( (r = get_fd(0, bits, &m_in.fd, &fil_ptr)) != OK) return(r);

  /* If O_CREATE is set, try to make the file. */
  if (oflags & O_CREAT) {
        /* Create a new inode by calling new_node(). */
        omode = I_REGULAR | (omode & ALL_MODES & fp->fp_umask);
        rip = new_node(user_path, omode, NO_ZONE);
        r = err_code;
        if (r == OK) exist = FALSE;      /* we just created the file */
        else if (r != EEXIST) return(r); /* other error */
        else exist = !(oflags & O_EXCL); /* file exists, if the O_EXCL
                                            flag is set this is an error */
  } else {
         /* Scan path name. */
        if ( (rip = eat_path(user_path)) == NIL_INODE) return(err_code);
  }

  /* Claim the file descriptor and filp slot and fill them in. */
  fp->fp_filp[m_in.fd] = fil_ptr;
  fil_ptr->filp_count = 1;
  fil_ptr->filp_ino = rip;
  fil_ptr->filp_flags = oflags;

  /* Only do the normal open code if we didn't just create the file. */
  if (exist) {
        /* Check protections. */
        if ((r = forbidden(rip, bits)) == OK) {
                /* Opening reg. files directories and special files differ. */
                switch (rip->i_mode & I_TYPE) {
                   case I_REGULAR:
                        /* Truncate regular file if O_TRUNC. */
                        if (oflags & O_TRUNC) {
                                if ((r = forbidden(rip, W_BIT)) !=OK) break;
                                truncate(rip);
                                wipe_inode(rip);
                                /* Send the inode from the inode cache to the
                                 * block cache, so it gets written on the next
                                 * cache flush.
                                 */
                                rw_inode(rip, WRITING);
                        }
                        break;

                   case I_DIRECTORY:
                        /* Directories may be read but not written. */
                        r = (bits & W_BIT ? EISDIR : OK);
                        break;

                   case I_CHAR_SPECIAL:
                   case I_BLOCK_SPECIAL:
                        /* Invoke the driver for special processing. */
                        dev = (dev_t) rip->i_zone[0];
                        r = dev_open(dev, who, bits | (oflags & ~O_ACCMODE));
                        break;

                   case I_NAMED_PIPE:
                        oflags |= O_APPEND;     /* force append mode */
                        fil_ptr->filp_flags = oflags;
                        r = pipe_open(rip, bits, oflags);
                        if (r != ENXIO) {
                                /* See if someone else is doing a rd or wt on
                                 * the FIFO.  If so, use its filp entry so the
                                 * file position will be automatically shared.
                                 */
                                b = (bits & R_BIT ? R_BIT : W_BIT);
                                fil_ptr->filp_count = 0; /* don't find self */
                                if ((filp2 = find_filp(rip, b)) != NIL_FILP) {
                                        /* Co-reader or writer found. Use it.*/
                                        fp->fp_filp[m_in.fd] = filp2;
                                        filp2->filp_count++;
                                        filp2->filp_ino = rip;
                                        filp2->filp_flags = oflags;

                                        /* i_count was incremented incorrectly
                                         * by eatpath above, not knowing that
                                         * we were going to use an existing
                                         * filp entry.  Correct this error.
                                         */
                                        rip->i_count--;
                                } else {
                                        /* Nobody else found.  Restore filp. */
                                        fil_ptr->filp_count = 1;
                                        if (b == R_BIT)
                                             pos = rip->i_zone[V2_NR_DZONES+0];
                                        else
                                             pos = rip->i_zone[V2_NR_DZONES+1];
                                        fil_ptr->filp_pos = pos;
                                }
                        }
                        break;
                }
        }
  }

  /* If error, release inode. */
  if (r != OK) {
        if (r == SUSPEND) return(r);            /* Oops, just suspended */
        fp->fp_filp[m_in.fd] = NIL_FILP;
        fil_ptr->filp_count= 0;
        put_inode(rip);
        return(r);
  }

  return(m_in.fd);
}

/*===========================================================================*
 *                              new_node                                     *
 *===========================================================================*/
PRIVATE struct inode *new_node(char *path, mode_t bits, zone_t z0)
{
/* New_node() is called by common_open(), do_mknod(), and do_mkdir().
 * In all cases it allocates a new inode, makes a directory entry for it on
 * the path 'path', and initializes it.  It returns a pointer to the inode if
 * it can do this; otherwise it returns NIL_INODE.  It always sets 'err_code'
 * to an appropriate value (OK or an error code).
 */

  register struct inode *rlast_dir_ptr, *rip;
  register int r;
  char string[NAME_MAX];

  /* See if the path can be opened down to the last directory. */
  if ((rlast_dir_ptr = last_dir(path, string)) == NIL_INODE) return(NIL_INODE);

  /* The final directory is accessible. Get final component of the path. */
  rip = advance(rlast_dir_ptr, string);
  if ( rip == NIL_INODE && err_code == ENOENT) {
        /* Last path component does not exist.  Make new directory entry. */
        if ( (rip = alloc_inode(rlast_dir_ptr->i_dev, bits)) == NIL_INODE) {
                /* Can't creat new inode: out of inodes. */
                put_inode(rlast_dir_ptr);
                return(NIL_INODE);
        }

        /* Force inode to the disk before making directory entry to make
         * the system more robust in the face of a crash: an inode with
         * no directory entry is much better than the opposite.
         */
        rip->i_nlinks++;
        rip->i_zone[0] = z0;            /* major/minor device numbers */
        rw_inode(rip, WRITING);         /* force inode to disk now */

        /* New inode acquired.  Try to make directory entry. */
        if ((r = search_dir(rlast_dir_ptr, string, &rip->i_num,ENTER)) != OK) {
                put_inode(rlast_dir_ptr);
                rip->i_nlinks--;        /* pity, have to free disk inode */
                rip->i_dirt = DIRTY;    /* dirty inodes are written out */
                put_inode(rip); /* this call frees the inode */
                err_code = r;
                return(NIL_INODE);
        }

  } else {
        /* Either last component exists, or there is some problem. */
        if (rip != NIL_INODE)
                r = EEXIST;
        else
                r = err_code;
  }

  /* Return the directory inode and exit. */
  put_inode(rlast_dir_ptr);
  err_code = r;
  return(rip);
}

/*===========================================================================*
 *                              pipe_open                                    *
 *===========================================================================*/
PRIVATE int pipe_open(register struct inode *rip, register mode_t bits,
        register int oflags)
{
/*  This function is called from common_open. It checks if
 *  there is at least one reader/writer pair for the pipe, if not
 *  it suspends the caller, otherwise it revives all other blocked
 *  processes hanging on the pipe.
 */

  rip->i_pipe = I_PIPE;
  if (find_filp(rip, bits & W_BIT ? R_BIT : W_BIT) == NIL_FILP) {
        if (oflags & O_NONBLOCK) {
                if (bits & W_BIT) return(ENXIO);
        } else {
                suspend(XPOPEN);        /* suspend caller */
                return(SUSPEND);
        }
  } else if (susp_count > 0) {/* revive blocked processes */
        release(rip, OPEN, susp_count);
        release(rip, CREAT, susp_count);
  }
  return(OK);
}

/*===========================================================================*
 *                              do_mknod                                     *
 *===========================================================================*/
PUBLIC int do_mknod()
{
```

```
/* Perform the mknod(name, mode, addr) system call. */

  register mode_t bits, mode_bits;
  struct inode *ip;

  /* Only the super_user may make nodes other than fifos. */
  mode_bits = (mode_t) m_in.mk_mode;            /* mode of the inode */
  if (!super_user && ((mode_bits & I_TYPE) != I_NAMED_PIPE)) return(EPERM);
  if (fetch_name(m_in.name1, m_in.name1_length, M1) != OK) return(err_code);
  bits = (mode_bits & I_TYPE) | (mode_bits & ALL_MODES & fp->fp_umask);
  ip = new_node(user_path, bits, (zone_t) m_in.mk_z0);
  put_inode(ip);
  return(err_code);
}

/*===========================================================================*
 *                              do_mkdir                                     *
 *===========================================================================*/
PUBLIC int do_mkdir()
{
/* Perform the mkdir(name, mode) system call. */

  int r1, r2;                   /* status codes */
  ino_t dot, dotdot;            /* inode numbers for . and .. */
  mode_t bits;                  /* mode bits for the new inode */
  char string[NAME_MAX];        /* last component of the new dir's path name */
  register struct inode *rip, *ldirp;

  /* Check to see if it is possible to make another link in the parent dir. */
  if (fetch_name(m_in.name1, m_in.name1_length, M1) != OK) return(err_code);
  ldirp = last_dir(user_path, string);  /* pointer to new dir's parent */
  if (ldirp == NIL_INODE) return(err_code);
  if (ldirp->i_nlinks >= (ldirp->i_sp->s_version == V1 ?
         CHAR_MAX : SHRT_MAX)) {
        put_inode(ldirp);       /* return parent */
        return(EMLINK);
  }

  /* Next make the inode. If that fails, return error code. */
  bits = I_DIRECTORY | (m_in.mode & RWX_MODES & fp->fp_umask);
  rip = new_node(user_path, bits, (zone_t) 0);
  if (rip == NIL_INODE || err_code == EEXIST) {
        put_inode(rip);         /* can't make dir: it already exists */
        put_inode(ldirp);       /* return parent too */
        return(err_code);
  }

  /* Get the inode numbers for . and .. to enter in the directory. */
  dotdot = ldirp->i_num;        /* parent's inode number */
  dot = rip->i_num;             /* inode number of the new dir itself */

  /* Now make dir entries for . and .. unless the disk is completely full. */
  /* Use dot1 and dot2, so the mode of the directory isn't important. */
  rip->i_mode = bits;   /* set mode */
  r1 = search_dir(rip, dot1, &dot, ENTER);      /* enter . in the new dir */
  r2 = search_dir(rip, dot2, &dotdot, ENTER);   /* enter .. in the new dir */

  /* If both . and .. were successfully entered, increment the link counts. */
  if (r1 == OK && r2 == OK) {
        /* Normal case.  It was possible to enter . and .. in the new dir. */
        rip->i_nlinks++;        /* this accounts for . */
        ldirp->i_nlinks++;      /* this accounts for .. */
        ldirp->i_dirt = DIRTY;  /* mark parent's inode as dirty */
  } else {
        /* It was not possible to enter . or .. probably disk was full. */
        (void) search_dir(ldirp, string, (ino_t *) 0, DELETE);
        rip->i_nlinks--;        /* undo the increment done in new_node() */
  }
  rip->i_dirt = DIRTY;          /* either way, i_nlinks has changed */

  put_inode(ldirp);             /* return the inode of the parent dir */
  put_inode(rip);               /* return the inode of the newly made dir */
  return(err_code);             /* new_node() always sets 'err_code' */
}

/*===========================================================================*
 *                              do_close                                     *
 *===========================================================================*/
PUBLIC int do_close()
{
/* Perform the close(fd) system call. */

  register struct filp *rfilp;
  register struct inode *rip;
  struct file_lock *flp;
  int rw, mode_word, lock_count;
  dev_t dev;

  /* First locate the inode that belongs to the file descriptor. */
  if ( (rfilp = get_filp(m_in.fd)) == NIL_FILP) return(err_code);
  rip = rfilp->filp_ino;        /* 'rip' points to the inode */

  if (rfilp->filp_count - 1 == 0 && rfilp->filp_mode != FILP_CLOSED) {
        /* Check to see if the file is special. */
        mode_word = rip->i_mode & I_TYPE;
        if (mode_word == I_CHAR_SPECIAL || mode_word == I_BLOCK_SPECIAL) {
                dev = (dev_t) rip->i_zone[0];
                if (mode_word == I_BLOCK_SPECIAL)  {
                        /* Invalidate cache entries unless special is mounted
                         * or ROOT
                         */
                        if (!mounted(rip)) {
                                (void) do_sync();       /* purge cache */
                                invalidate(dev);
                        }
                }
                /* Do any special processing on device close. */
                dev_close(dev);
        }
  }

  /* If the inode being closed is a pipe, release everyone hanging on it. */
  if (rip->i_pipe == I_PIPE) {
        rw = (rfilp->filp_mode & R_BIT ? WRITE : READ);
        release(rip, rw, NR_PROCS);
  }

  /* If a write has been done, the inode is already marked as DIRTY. */
  if (--rfilp->filp_count == 0) {
        if (rip->i_pipe == I_PIPE && rip->i_count > 1) {
                /* Save the file position in the i-node in case needed later.
                 * The read and write positions are saved separately.  The
                 * last 3 zones in the i-node are not used for (named) pipes.
                 */
                if (rfilp->filp_mode == R_BIT)
                        rip->i_zone[V2_NR_DZONES+0] = (zone_t) rfilp->filp_pos;
                else
                        rip->i_zone[V2_NR_DZONES+1] = (zone_t) rfilp->filp_pos;
        }
        put_inode(rip);
  }

  fp->fp_cloexec &= ~(1L << m_in.fd);   /* turn off close-on-exec bit */
  fp->fp_filp[m_in.fd] = NIL_FILP;

  /* Check to see if the file is locked.  If so, release all locks. */
  if (nr_locks == 0) return(OK);
  lock_count = nr_locks;        /* save count of locks */
  for (flp = &file_lock[0]; flp < &file_lock[NR_LOCKS]; flp++) {
        if (flp->lock_type == 0) continue;      /* slot not in use */
        if (flp->lock_inode == rip && flp->lock_pid == fp->fp_pid) {
                flp->lock_type = 0;
                nr_locks--;
        }
  }
  if (nr_locks < lock_count) lock_revive();     /* lock released */
  return(OK);
}

/*===========================================================================*
 *                              do_lseek                                     *
 *===========================================================================*/
PUBLIC int do_lseek()
{
/* Perform the lseek(ls_fd, offset, whence) system call. */

  register struct filp *rfilp;
  register off_t pos;

  /* Check to see if the file descriptor is valid. */
  if ( (rfilp = get_filp(m_in.ls_fd)) == NIL_FILP) return(err_code);

  /* No lseek on pipes. */
  if (rfilp->filp_ino->i_pipe == I_PIPE) return(ESPIPE);

  /* The value of 'whence' determines the start position to use. */
  switch(m_in.whence) {
        case 0: pos = 0;        break;
        case 1: pos = rfilp->filp_pos;  break;
        case 2: pos = rfilp->filp_ino->i_size;  break;
        default: return(EINVAL);
  }

  /* Check for overflow. */
  if (((long)m_in.offset > 0) && ((long)(pos + m_in.offset) < (long)pos))
        return(EINVAL);
  if (((long)m_in.offset < 0) && ((long)(pos + m_in.offset) > (long)pos))
        return(EINVAL);
  pos = pos + m_in.offset;

  if (pos != rfilp->filp_pos)
        rfilp->filp_ino->i_seek = ISEEK;        /* inhibit read ahead */
  rfilp->filp_pos = pos;
  m_out.reply_l1 = pos;         /* insert the long into the output message */
  return(OK);
}
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                      servers/fs/read.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/* This file contains the heart of the mechanism used to read (and write)
 * files.  Read and write requests are split up into chunks that do not cross
 * block boundaries.  Each chunk is then processed in turn.  Reads on special
 * files are also detected and handled.
 *
 * The entry points into this file are
 *   do_read:    perform the READ system call by calling read_write
 *   read_write: actually do the work of READ and WRITE
 *   read_map:   given an inode and file position, look up its zone number
 *   rd_indir:   read an entry in an indirect block
 *   read_ahead: manage the block read ahead business
 */

#include "fs.h"
#include <fcntl.h>
#include <minix/com.h>
#include "buf.h"
#include "file.h"
#include "fproc.h"
#include "inode.h"
#include "param.h"
#include "super.h"

FORWARD _PROTOTYPE( int rw_chunk, (struct inode *rip, off_t position,
        unsigned off, int chunk, unsigned left, int rw_flag,
        char *buff, int seg, int usr, int block_size, int *completed));

/*===========================================================================*
 *                              do_read                                      *
 *===========================================================================*/
PUBLIC int do_read()
{
  return(read_write(READING));
}

/*===========================================================================*
 *                              read_write                                   *
 *===========================================================================*/
PUBLIC int read_write(rw_flag)
int rw_flag;                    /* READING or WRITING */
{
/* Perform read(fd, buffer, nbytes) or write(fd, buffer, nbytes) call. */

  register struct inode *rip;
  register struct filp *f;
  off_t bytes_left, f_size, position;
  unsigned int off, cum_io;
  int op, oflags, r, chunk, usr, seg, block_spec, char_spec;
  int regular, partial_pipe = 0, partial_cnt = 0;
  mode_t mode_word;
  struct filp *wf;
  int block_size;
  int completed, r2 = OK;
  phys_bytes p;

  /* left unfinished rw_chunk()s from previous call! this can't happen.
   * it means something has gone wrong we can't repair now.
   */
  if (bufs_in_use < 0) {
        panic(__FILE__,"start - bufs_in_use negative", bufs_in_use);
  }

  /* MM loads segments by putting funny things in upper 10 bits of 'fd'. */
  if (who == PM_PROC_NR && (m_in.fd & (~BYTE)) ) {
        usr = m_in.fd >> 7;
        seg = (m_in.fd >> 5) & 03;
        m_in.fd &= 037;         /* get rid of user and segment bits */
  } else {
        usr = who;              /* normal case */
        seg = D;
  }

  /* If the file descriptor is valid, get the inode, size and mode. */
  if (m_in.nbytes < 0) return(EINVAL);
  if ((f = get_filp(m_in.fd)) == NIL_FILP) return(err_code);
  if (((f->filp_mode) & (rw_flag == READING ? R_BIT : W_BIT)) == 0) {
        return(f->filp_mode == FILP_CLOSED ? EIO : EBADF);
  }
  if (m_in.nbytes == 0)
         return(0);     /* so char special files need not check for 0*/

  /* check if user process has the memory it needs.
   * if not, copying will fail later.
   * do this after 0-check above because umap doesn't want to map 0 bytes.
   */
  if ((r = sys_umap(usr, seg, (vir_bytes) m_in.buffer, m_in.nbytes, &p)) != OK)
        return r;
  position = f->filp_pos;
  oflags = f->filp_flags;
  rip = f->filp_ino;
  f_size = rip->i_size;
  r = OK;
  if (rip->i_pipe == I_PIPE) {
        /* fp->fp_cum_io_partial is only nonzero when doing partial writes */
        cum_io = fp->fp_cum_io_partial;
  } else {
        cum_io = 0;
  }
  op = (rw_flag == READING ? DEV_READ : DEV_WRITE);
  mode_word = rip->i_mode & I_TYPE;
  regular = mode_word == I_REGULAR || mode_word == I_NAMED_PIPE;

  if ((char_spec = (mode_word == I_CHAR_SPECIAL ? 1 : 0))) {
        if (rip->i_zone[0] == NO_DEV)
                panic(__FILE__,"read_write tries to read from "
                        "character device NO_DEV", NO_NUM);
        block_size = get_block_size(rip->i_zone[0]);
  }
  if ((block_spec = (mode_word == I_BLOCK_SPECIAL ? 1 : 0))) {
        f_size = ULONG_MAX;
        if (rip->i_zone[0] == NO_DEV)
                panic(__FILE__,"read_write tries to read from "
                " block device NO_DEV", NO_NUM);
        block_size = get_block_size(rip->i_zone[0]);
  }

  if (!char_spec && !block_spec)
        block_size = rip->i_sp->s_block_size;

  rdwt_err = OK;                /* set to EIO if disk error occurs */

  /* Check for character special files. */
  if (char_spec) {
        dev_t dev;
        dev = (dev_t) rip->i_zone[0];
        r = dev_io(op, dev, usr, m_in.buffer, position, m_in.nbytes, oflags);
        if (r >= 0) {
                cum_io = r;
                position += r;
                r = OK;
        }
  } else {
        if (rw_flag == WRITING && block_spec == 0) {
                /* Check in advance to see if file will grow too big. */
                if (position > rip->i_sp->s_max_size - m_in.nbytes)
                        return(EFBIG);

                /* Check for O_APPEND flag. */
                if (oflags & O_APPEND) position = f_size;

                /* Clear the zone containing present EOF if hole about
                 * to be created.  This is necessary because all unwritten
                 * blocks prior to the EOF must read as zeros.
                 */
                if (position > f_size) clear_zone(rip, f_size, 0);
        }

        /* Pipes are a little different.  Check. */
        if (rip->i_pipe == I_PIPE) {
               r = pipe_check(rip, rw_flag, oflags,
                        m_in.nbytes, position, &partial_cnt, 0);
               if (r <= 0) return(r);
        }

        if (partial_cnt > 0) partial_pipe = 1;

        /* Split the transfer into chunks that don’t span two blocks. */
        while (m_in.nbytes != 0) {

                off = (unsigned int) (position % block_size);/* offset in blk*/
                if (partial_pipe) {  /* pipes only */
                        chunk = MIN(partial_cnt, block_size - off);
                } else
                        chunk = MIN(m_in.nbytes, block_size - off);
                if (chunk < 0) chunk = block_size - off;

                if (rw_flag == READING) {
                        bytes_left = f_size - position;
                        if (position >= f_size) break;  /* we are beyond EOF */
                        if (chunk > bytes_left) chunk = (int) bytes_left;
                }

                /* Read or write 'chunk' bytes. */
                r = rw_chunk(rip, position, off, chunk, (unsigned) m_in.nbytes,
                             rw_flag, m_in.buffer, seg, usr, block_size, &completed);

                if (r != OK) break;     /* EOF reached */
                if (rdwt_err < 0) break;

                /* Update counters and pointers. */
                m_in.buffer += chunk;   /* user buffer address */
                m_in.nbytes -= chunk;   /* bytes yet to be read */
                cum_io += chunk;        /* bytes read so far */
                position += chunk;      /* position within the file */

                if (partial_pipe) {
                        partial_cnt -= chunk;
                        if (partial_cnt <= 0)  break;
                }
        }
  }

  /* On write, update file size and access time. */
  if (rw_flag == WRITING) {
        if (regular || mode_word == I_DIRECTORY) {
                if (position > f_size) rip->i_size = position;
        }
  } else {
        if (rip->i_pipe == I_PIPE) {
                if ( position >= rip->i_size) {
                        /* Reset pipe pointers. */
                        rip->i_size = 0;        /* no data left */
                        position = 0;           /* reset reader(s) */
                        wf = find_filp(rip, W_BIT);
                        if (wf != NIL_FILP) wf->filp_pos = 0;
                }
        }
  }
  f->filp_pos = position;

  /* Check to see if read-ahead is called for, and if so, set it up. */
  if (rw_flag == READING && rip->i_seek == NO_SEEK && position % block_size== 0
                && (regular || mode_word == I_DIRECTORY)) {
        rdahed_inode = rip;
        rdahedpos = position;
  }
  rip->i_seek = NO_SEEK;

  if (rdwt_err != OK) r = rdwt_err;     /* check for disk error */
  if (rdwt_err == END_OF_FILE) r = OK;

  /* if user-space copying failed, read/write failed. */
  if (r == OK && r2 != OK) {
        r = r2;
  }
  if (r == OK) {
        if (rw_flag == READING) rip->i_update |= ATIME;
        if (rw_flag == WRITING) rip->i_update |= CTIME | MTIME;
        rip->i_dirt = DIRTY;            /* inode is thus now dirty */
        if (partial_pipe) {
                partial_pipe = 0;
                        /* partial write on pipe with */
                /* O_NONBLOCK, return write count */
                if (!(oflags & O_NONBLOCK)) {
                        fp->fp_cum_io_partial = cum_io;
                        suspend(XPIPE);   /* partial write on pipe with */
                        return(SUSPEND);  /* nbyte > PIPE_SIZE - non-atomic */
                }
        }
        fp->fp_cum_io_partial = 0;
        return(cum_io);
  }
  if (bufs_in_use < 0) {
        panic(__FILE__,"end - bufs_in_use negative", bufs_in_use);
  }
  return(r);
}

/*===========================================================================*
 *                              rw_chunk                                     *
 *===========================================================================*/
PRIVATE int rw_chunk(rip, position, off, chunk, left, rw_flag, buff,
 seg, usr, block_size, completed)
register struct inode *rip;     /* pointer to inode for file to be rd/wr */
off_t position;                 /* position within file to read or write */
unsigned off;                   /* off within the current block */
int chunk;                      /* number of bytes to read or write */
unsigned left;                  /* max number of bytes wanted after position */
int rw_flag;                    /* READING or WRITING */
char *buff;                     /* virtual address of the user buffer */
int seg;                        /* T or D segment in user space */
int usr;                        /* which user process */
int block_size;                 /* block size of FS operating on */
int *completed;                 /* number of bytes copied */
{
/* Read or write (part of) a block. */

  register struct buf *bp;
  register int r = OK;
  int n, block_spec;
  block_t b;
  dev_t dev;

  *completed = 0;

  block_spec = (rip->i_mode & I_TYPE) == I_BLOCK_SPECIAL;
  if (block_spec) {
        b = position/block_size;
        dev = (dev_t) rip->i_zone[0];
  } else {
        b = read_map(rip, position);
        dev = rip->i_dev;
  }

  if (!block_spec && b == NO_BLOCK) {
        if (rw_flag == READING) {
                /* Reading from a nonexistent block.  Must read as all zeros.*/
                bp = get_block(NO_DEV, NO_BLOCK, NORMAL);    /* get a buffer */
                zero_block(bp);
        } else {
                /* Writing to a nonexistent block. Create and enter in inode.*/
                if ((bp= new_block(rip, position)) == NIL_BUF)return(err_code);
        }
  } else if (rw_flag == READING) {
        /* Read and read ahead if convenient. */
        bp = rahead(rip, b, position, left);
  } else {
        /* Normally an existing block to be partially overwritten is first read
         * in.  However, a full block need not be read in.  If it is already in
         * the cache, acquire it, otherwise just acquire a free buffer.
         */
        n = (chunk == block_size ? NO_READ : NORMAL);
        if (!block_spec && off == 0 && position >= rip->i_size) n = NO_READ;
        bp = get_block(dev, b, n);
  }

  /* In all cases, bp now points to a valid buffer. */
  if (bp == NIL_BUF) {
        panic(__FILE__,"bp not valid in rw_chunk, this can't happen", NO_NUM);
  }
  if (rw_flag == WRITING && chunk != block_size && !block_spec &&
                                        position >= rip->i_size && off == 0) {
        zero_block(bp);
  }

  if (rw_flag == READING) {
        /* Copy a chunk from the block buffer to user space. */
        r = sys_vircopy(FS_PROC_NR, D, (phys_bytes) (bp->b_data+off),
                        usr, seg, (phys_bytes) buff,
                        (phys_bytes) chunk);
  } else {
        /* Copy a chunk from user space to the block buffer. */
        r = sys_vircopy(usr, seg, (phys_bytes) buff,
                        FS_PROC_NR, D, (phys_bytes) (bp->b_data+off),
                        (phys_bytes) chunk);
        bp->b_dirt = DIRTY;
  }
  n = (off + chunk == block_size ? FULL_DATA_BLOCK : PARTIAL_DATA_BLOCK);
  put_block(bp, n);

  return(r);
}


/*===========================================================================*
 *                              read_map                                     *
 *===========================================================================*/
PUBLIC block_t read_map(rip, position)
register struct inode *rip;     /* ptr to inode to map from */
off_t position;                 /* position in file whose blk wanted */
{
/* Given an inode and a position within the corresponding file, locate the
 * block (not zone) number in which that position is to be found and return it.
 */

```

```
  register struct buf *bp;
  register zone_t z;
  int scale, boff, dzones, nr_indirects, index, zind, ex;
  block_t b;
  long excess, zone, block_pos;

  scale = rip->i_sp->s_log_zone_size;   /* for block-zone conversion */
  block_pos = position/rip->i_sp->s_block_size; /* relative blk # in file */
  zone = block_pos >> scale;    /* position's zone */
  boff = (int) (block_pos - (zone << scale) ); /* relative blk # within zone */
  dzones = rip->i_ndzones;
  nr_indirects = rip->i_nindirs;

  /* Is 'position' to be found in the inode itself? */
  if (zone < dzones) {
        zind = (int) zone;      /* index should be an int */
        z = rip->i_zone[zind];
        if (z == NO_ZONE) return(NO_BLOCK);
        b = ((block_t) z << scale) + boff;
        return(b);
  }

  /* It is not in the inode, so it must be single or double indirect. */
  excess = zone - dzones;       /* first Vx_NR_DZONES don't count */

  if (excess < nr_indirects) {
        /* 'position' can be located via the single indirect block. */
        z = rip->i_zone[dzones];
  } else {
        /* 'position' can be located via the double indirect block. */
        if ( (z = rip->i_zone[dzones+1]) == NO_ZONE) return(NO_BLOCK);
        excess -= nr_indirects;                 /* single indir doesn't count*/
        b = (block_t) z << scale;
        bp = get_block(rip->i_dev, b, NORMAL);  /* get double indirect block */
        index = (int) (excess/nr_indirects);
        z = rd_indir(bp, index);                /* z= zone for single*/
        put_block(bp, INDIRECT_BLOCK);          /* release double ind block */
        excess = excess % nr_indirects;         /* index into single ind blk */
  }

  /* 'z' is zone num for single indirect block; 'excess' is index into it. */
  if (z == NO_ZONE) return(NO_BLOCK);
  b = (block_t) z << scale;                     /* b is blk # for single ind */
  bp = get_block(rip->i_dev, b, NORMAL);        /* get single indirect block */
  ex = (int) excess;                            /* need an integer */
  z = rd_indir(bp, ex);                         /* get block pointed to */
  put_block(bp, INDIRECT_BLOCK);                /* release single indir blk */
  if (z == NO_ZONE) return(NO_BLOCK);
  b = ((block_t) z << scale) + boff;
  return(b);
}

/*===========================================================================*
 *                              rd_indir                                     *
 *===========================================================================*/
PUBLIC zone_t rd_indir(bp, index)
struct buf *bp;                 /* pointer to indirect block */
int index;                      /* index into *bp */
{
/* Given a pointer to an indirect block, read one entry.  The reason for
```

```
 * making a separate routine out of this is that there are four cases:
 * V1 (IBM and 68000), and V2 (IBM and 68000).
 */

  struct super_block *sp;
  zone_t zone;                  /* V2 zones are longs (shorts in V1) */

  sp = get_super(bp->b_dev);    /* need super block to find file sys type */

  /* read a zone from an indirect block */
  if (sp->s_version == V1)
        zone = (zone_t) conv2(sp->s_native, (int)  bp->b_v1_ind[index]);
  else
        zone = (zone_t) conv4(sp->s_native, (long) bp->b_v2_ind[index]);

  if (zone != NO_ZONE &&
                (zone < (zone_t) sp->s_firstdatazone || zone >= sp->s_zones)) {
        printf("Illegal zone number %ld in indirect block, index %d\n",
               (long) zone, index);
        panic(__FILE__,"check file system", NO_NUM);
  }
  return(zone);
}

/*===========================================================================*
 *                              read_ahead                                   *
 *===========================================================================*/
PUBLIC void read_ahead()
{
/* Read a block into the cache before it is needed. */
  int block_size;
  register struct inode *rip;
  struct buf *bp;
  block_t b;

  rip = rdahed_inode;           /* pointer to inode to read ahead from */
  block_size = get_block_size(rip->i_dev);
  rdahed_inode = NIL_INODE;     /* turn off read ahead */
  if ( (b = read_map(rip, rdahedpos)) == NO_BLOCK) return;      /* at EOF */
  bp = rahead(rip, b, rdahedpos, block_size);
  put_block(bp, PARTIAL_DATA_BLOCK);
}

/*===========================================================================*
 *                              rahead                                       *
 *===========================================================================*/
PUBLIC struct buf *rahead(rip, baseblock, position, bytes_ahead)
register struct inode *rip;     /* pointer to inode for file to be read */
block_t baseblock;              /* block at current position */
off_t position;                 /* position within file */
unsigned bytes_ahead;           /* bytes beyond position for immediate use */
{
/* Fetch a block from the cache or the device.  If a physical read is
 * required, prefetch as many more blocks as convenient into the cache.
 * This usually covers bytes_ahead and is at least BLOCKS_MINIMUM.
 * The device driver may decide it knows better and stop reading at a
 * cylinder boundary (or after an error).  Rw_scattered() puts an optional
 * flag on all reads to allow this.
 */
  int block_size;
```

```
/* Minimum number of blocks to prefetch. */
# define BLOCKS_MINIMUM         (NR_BUFS < 50 ? 18 : 32)
  int block_spec, scale, read_q_size;
  unsigned int blocks_ahead, fragment;
  block_t block, blocks_left;
  off_t ind1_pos;
  dev_t dev;
  struct buf *bp;
  static struct buf *read_q[NR_BUFS];

  block_spec = (rip->i_mode & I_TYPE) == I_BLOCK_SPECIAL;
  if (block_spec) {
        dev = (dev_t) rip->i_zone[0];
  } else {
        dev = rip->i_dev;
  }
  block_size = get_block_size(dev);

  block = baseblock;
  bp = get_block(dev, block, PREFETCH);
  if (bp->b_dev != NO_DEV) return(bp);

  /* The best guess for the number of blocks to prefetch:  A lot.
   * It is impossible to tell what the device looks like, so we don't even
   * try to guess the geometry, but leave it to the driver.
   *
   * The floppy driver can read a full track with no rotational delay, and it
   * avoids reading partial tracks if it can, so handing it enough buffers to
   * read two tracks is perfect.  (Two, because some diskette types have
   * an odd number of sectors per track, so a block may span tracks.)
   *
   * The disk drivers don't try to be smart.  With todays disks it is
   * impossible to tell what the real geometry looks like, so it is best to
   * read as much as you can.  With luck the caching on the drive allows
   * for a little time to start the next read.
   *
   * The current solution below is a bit of a hack, it just reads blocks from
   * the current file position hoping that more of the file can be found.  A
   * better solution must look at the already available zone pointers and
   * indirect blocks (but don't call read_map!).
   */

  fragment = position % block_size;
  position -= fragment;
  bytes_ahead += fragment;

  blocks_ahead = (bytes_ahead + block_size - 1) / block_size;

  if (block_spec && rip->i_size == 0) {
        blocks_left = NR_IOREQS;
  } else {
        blocks_left = (rip->i_size - position + block_size - 1) / block_size;

        /* Go for the first indirect block if we are in its neighborhood. */
        if (!block_spec) {
                scale = rip->i_sp->s_log_zone_size;
                ind1_pos = (off_t) rip->i_ndzones * (block_size << scale);
                if (position <= ind1_pos && rip->i_size > ind1_pos) {
                        blocks_ahead++;
                        blocks_left++;
                }
        }
  }

  /* No more than the maximum request. */
  if (blocks_ahead > NR_IOREQS) blocks_ahead = NR_IOREQS;

  /* Read at least the minimum number of blocks, but not after a seek. */
  if (blocks_ahead < BLOCKS_MINIMUM && rip->i_seek == NO_SEEK)
        blocks_ahead = BLOCKS_MINIMUM;

  /* Can't go past end of file. */
  if (blocks_ahead > blocks_left) blocks_ahead = blocks_left;

  read_q_size = 0;

  /* Acquire block buffers. */
  for (;;) {
        read_q[read_q_size++] = bp;

        if (--blocks_ahead == 0) break;

        /* Don't trash the cache, leave 4 free. */
        if (bufs_in_use >= NR_BUFS - 4) break;

        block++;

        bp = get_block(dev, block, PREFETCH);
        if (bp->b_dev != NO_DEV) {
                /* Oops, block already in the cache, get out. */
                put_block(bp, FULL_DATA_BLOCK);
                break;
        }
  }
  rw_scattered(dev, read_q, read_q_size, READING);
  return(get_block(dev, baseblock, NORMAL));
}
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                  servers/fs/write.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/* This file is the counterpart of "read.c".  It contains the code for writing
 * insofar as this is not contained in read_write().
 *
 * The entry points into this file are
 *   do_write:     call read_write to perform the WRITE system call
 *   clear_zone:   erase a zone in the middle of a file
 *   new_block:    acquire a new block
 */

#include "fs.h"
#include <string.h>
#include "buf.h"
#include "file.h"
#include "fproc.h"
#include "inode.h"
```

```
#include "super.h"

FORWARD _PROTOTYPE( int write_map, (struct inode *rip, off_t position,
                        zone_t new_zone)                              );

FORWARD _PROTOTYPE( void wr_indir, (struct buf *bp, int index, zone_t zone) );

/*===========================================================================*
 *                              do_write                                     *
 *===========================================================================*/
PUBLIC int do_write()
{
/* Perform the write(fd, buffer, nbytes) system call. */

  return(read_write(WRITING));
}

/*===========================================================================*
 *                              write_map                                    *
 *===========================================================================*/
PRIVATE int write_map(rip, position, new_zone)
register struct inode *rip;     /* pointer to inode to be changed */
off_t position;                 /* file address to be mapped */
zone_t new_zone;                /* zone # to be inserted */
{
/* Write a new zone into an inode. */
  int scale, ind_ex, new_ind, new_dbl, zones, nr_indirects, single, zindex, ex;
  zone_t z, z1;
  register block_t b;
  long excess, zone;
  struct buf *bp;

  rip->i_dirt = DIRTY;          /* inode will be changed */
  bp = NIL_BUF;
  scale = rip->i_sp->s_log_zone_size;           /* for zone-block conversion */
        /* relative zone # to insert */
  zone = (position/rip->i_sp->s_block_size) >> scale;
  zones = rip->i_ndzones;       /* # direct zones in the inode */
  nr_indirects = rip->i_nindirs;/* # indirect zones per indirect block */

  /* Is 'position' to be found in the inode itself? */
  if (zone < zones) {
        zindex = (int) zone;    /* we need an integer here */
        rip->i_zone[zindex] = new_zone;
        return(OK);
  }

  /* It is not in the inode, so it must be single or double indirect. */
  excess = zone - zones;        /* first Vx_NR_DZONES don't count */
  new_ind = FALSE;
  new_dbl = FALSE;

  if (excess < nr_indirects) {
        /* 'position' can be located via the single indirect block. */
        z1 = rip->i_zone[zones];        /* single indirect zone */
        single = TRUE;
  } else {
        /* 'position' can be located via the double indirect block. */
        if ( (z = rip->i_zone[zones+1]) == NO_ZONE) {
                /* Create the double indirect block. */
                if ( (z = alloc_zone(rip->i_dev, rip->i_zone[0])) == NO_ZONE)
                        return(err_code);
                rip->i_zone[zones+1] = z;
                new_dbl = TRUE; /* set flag for later */
        }

        /* Either way, 'z' is zone number for double indirect block. */
        excess -= nr_indirects; /* single indirect doesn't count */
        ind_ex = (int) (excess / nr_indirects);
        excess = excess % nr_indirects;
        if (ind_ex >= nr_indirects) return(EFBIG);
        b = (block_t) z << scale;
        bp = get_block(rip->i_dev, b, (new_dbl ? NO_READ : NORMAL));
        if (new_dbl) zero_block(bp);
        z1 = rd_indir(bp, ind_ex);
        single = FALSE;
  }

  /* z1 is now single indirect zone; 'excess' is index. */
  if (z1 == NO_ZONE) {
        /* Create indirect block and store zone # in inode or dbl indir blk. */
        z1 = alloc_zone(rip->i_dev, rip->i_zone[0]);
        if (single)
                rip->i_zone[zones] = z1;        /* update inode */
        else
                wr_indir(bp, ind_ex, z1);       /* update dbl indir */

        new_ind = TRUE;
        if (bp != NIL_BUF) bp->b_dirt = DIRTY;  /* if double ind, it is dirty*/
        if (z1 == NO_ZONE) {
                put_block(bp, INDIRECT_BLOCK);  /* release dbl indirect blk */
                return(err_code);       /* couldn't create single ind */
        }
  }
  put_block(bp, INDIRECT_BLOCK);        /* release double indirect blk */

  /* z1 is indirect block's zone number. */
  b = (block_t) z1 << scale;
  bp = get_block(rip->i_dev, b, (new_ind ? NO_READ : NORMAL) );
  if (new_ind) zero_block(bp);
  ex = (int) excess;                    /* we need an int here */
  wr_indir(bp, ex, new_zone);
  bp->b_dirt = DIRTY;
  put_block(bp, INDIRECT_BLOCK);

  return(OK);
}

/*===========================================================================*
 *                              wr_indir                                     *
 *===========================================================================*/
PRIVATE void wr_indir(bp, index, zone)
struct buf *bp;                 /* pointer to indirect block */
int index;                      /* index into *bp */
zone_t zone;                    /* zone to write */
{
/* Given a pointer to an indirect block, write one entry. */

  struct super_block *sp;

  sp = get_super(bp->b_dev);    /* need super block to find file sys type */

  /* write a zone into an indirect block */
  if (sp->s_version == V1)
        bp->b_v1_ind[index] = (zone1_t) conv2(sp->s_native, (int)  zone);
  else
        bp->b_v2_ind[index] = (zone_t)  conv4(sp->s_native, (long) zone);
}

/*===========================================================================*
 *                              clear_zone                                   *
 *===========================================================================*/
PUBLIC void clear_zone(rip, pos, flag)
register struct inode *rip;     /* inode to clear */
off_t pos;                      /* points to block to clear */
int flag;                       /* 0 if called by read_write, 1 by new_block */
{
/* Zero a zone, possibly starting in the middle.  The parameter 'pos' gives
 * a byte in the first block to be zeroed.  Clearzone() is called from
 * read_write and new_block().
 */

  register struct buf *bp;
  register block_t b, blo, bhi;
  register off_t next;
  register int scale;
  register zone_t zone_size;

  /* If the block size and zone size are the same, clear_zone() not needed. */
  scale = rip->i_sp->s_log_zone_size;
  if (scale == 0) return;

  zone_size = (zone_t) rip->i_sp->s_block_size << scale;
  if (flag == 1) pos = (pos/zone_size) * zone_size;
  next = pos + rip->i_sp->s_block_size - 1;

  /* If 'pos' is in the last block of a zone, do not clear the zone. */
  if (next/zone_size != pos/zone_size) return;
  if ( (blo = read_map(rip, next)) == NO_BLOCK) return;
  bhi = (  ((blo>>scale)+1) << scale)   - 1;

  /* Clear all the blocks between 'blo' and 'bhi'. */
  for (b = blo; b <= bhi; b++) {
        bp = get_block(rip->i_dev, b, NO_READ);
        zero_block(bp);
        put_block(bp, FULL_DATA_BLOCK);
  }
}

/*===========================================================================*
 *                              new_block                                    *
 *===========================================================================*/
PUBLIC struct buf *new_block(rip, position)
register struct inode *rip;     /* pointer to inode */
off_t position;                 /* file pointer */
{
/* Acquire a new block and return a pointer to it.  Doing so may require
 * allocating a complete zone, and then returning the initial block.
 * On the other hand, the current zone may still have some unused blocks.
 */
```

```

  register struct buf *bp;
  block_t b, base_block;
  zone_t z;
  zone_t zone_size;
  int scale, r;
  struct super_block *sp;

  /* Is another block available in the current zone? */
  if ( (b = read_map(rip, position)) == NO_BLOCK) {
        /* Choose first zone if possible. */
        /* Lose if the file is nonempty but the first zone number is NO_ZONE
         * corresponding to a zone full of zeros.  It would be better to
         * search near the last real zone.
         */
        if (rip->i_zone[0] == NO_ZONE) {
                sp = rip->i_sp;
                z = sp->s_firstdatazone;
        } else {
                z = rip->i_zone[0];     /* hunt near first zone */
        }
        if ( (z = alloc_zone(rip->i_dev, z)) == NO_ZONE) return(NIL_BUF);
        if ( (r = write_map(rip, position, z)) != OK) {
                free_zone(rip->i_dev, z);
                err_code = r;
                return(NIL_BUF);
        }

        /* If we are not writing at EOF, clear the zone, just to be safe. */
        if ( position != rip->i_size) clear_zone(rip, position, 1);
        scale = rip->i_sp->s_log_zone_size;
        base_block = (block_t) z << scale;
        zone_size = (zone_t) rip->i_sp->s_block_size << scale;
        b = base_block + (block_t)((position % zone_size)/rip->i_sp->s_block_size);
  }

  bp = get_block(rip->i_dev, b, NO_READ);
  zero_block(bp);
  return(bp);
}

/*===========================================================================*
 *                              zero_block                                   *
 *===========================================================================*/
PUBLIC void zero_block(bp)
register struct buf *bp;        /* pointer to buffer to zero */
{
/* Zero a block. */
  memset(bp->b_data, 0, MAX_BLOCK_SIZE);
  bp->b_dirt = DIRTY;
}
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                    servers/fs/pipe.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

/* This file deals with the suspension and revival of processes.  A process can
 * be suspended because it wants to read or write from a pipe and can't, or
 * because it wants to read or write from a special file and can't.  When a
 * process can't continue it is suspended, and revived later when it is able
 * to continue.
 *
 * The entry points into this file are
 *   do_pipe:     perform the PIPE system call
 *   pipe_check:  check to see that a read or write on a pipe is feasible now
 *   suspend:     suspend a process that cannot do a requested read or write
 *   release:     check to see if a suspended process can be released and do
 *                it
 *   revive:      mark a suspended process as able to run again
 *   do_unpause:  a signal has been sent to a process; see if it suspended
 */

#include "fs.h"
#include <fcntl.h>
#include <signal.h>
#include <minix/callnr.h>
#include <minix/com.h>
#include <sys/select.h>
#include <sys/time.h>
#include "file.h"
#include "fproc.h"
#include "inode.h"
#include "param.h"
#include "super.h"
#include "select.h"

/*===========================================================================*
 *                              do_pipe                                      *
 *===========================================================================*/
PUBLIC int do_pipe()
{
/* Perform the pipe(fil_des) system call. */

  register struct fproc *rfp;
  register struct inode *rip;
  int r;
  struct filp *fil_ptr0, *fil_ptr1;
  int fil_des[2];               /* reply goes here */

  /* Acquire two file descriptors. */
  rfp = fp;
  if ( (r = get_fd(0, R_BIT, &fil_des[0], &fil_ptr0)) != OK) return(r);
  rfp->fp_filp[fil_des[0]] = fil_ptr0;
  fil_ptr0->filp_count = 1;
  if ( (r = get_fd(0, W_BIT, &fil_des[1], &fil_ptr1)) != OK) {
        rfp->fp_filp[fil_des[0]] = NIL_FILP;
        fil_ptr0->filp_count = 0;
        return(r);
  }
  rfp->fp_filp[fil_des[1]] = fil_ptr1;
  fil_ptr1->filp_count = 1;

  /* Make the inode on the pipe device. */
  if ( (rip = alloc_inode(root_dev, I_REGULAR) ) == NIL_INODE) {
        rfp->fp_filp[fil_des[0]] = NIL_FILP;
        fil_ptr0->filp_count = 0;
        rfp->fp_filp[fil_des[1]] = NIL_FILP;
        fil_ptr1->filp_count = 0;
        return(err_code);
  }

  if (read_only(rip) != OK)
        panic(__FILE__,"pipe device is read only", NO_NUM);

  rip->i_pipe = I_PIPE;
  rip->i_mode &= ~I_REGULAR;
  rip->i_mode |= I_NAMED_PIPE;  /* pipes and FIFOs have this bit set */
  fil_ptr0->filp_ino = rip;
  fil_ptr0->filp_flags = O_RDONLY;
  dup_inode(rip);               /* for double usage */
  fil_ptr1->filp_ino = rip;
  fil_ptr1->filp_flags = O_WRONLY;
  rw_inode(rip, WRITING);       /* mark inode as allocated */
  m_out.reply_i1 = fil_des[0];
  m_out.reply_i2 = fil_des[1];
  rip->i_update = ATIME | CTIME | MTIME;
  return(OK);
}

/*===========================================================================*
 *                              pipe_check                                   *
 *===========================================================================*/
PUBLIC int pipe_check(rip, rw_flag, oflags, bytes, position, canwrite, notouch)
register struct inode *rip;     /* the inode of the pipe */
int rw_flag;                    /* READING or WRITING */
int oflags;                     /* flags set by open or fcntl */
register int bytes;             /* bytes to be read or written (all chunks) */
register off_t position;        /* current file position */
int *canwrite;                  /* return: number of bytes we can write */
int notouch;                    /* check only */
{
/* Pipes are a little different.  If a process reads from an empty pipe for
 * which a writer still exists, suspend the reader.  If the pipe is empty
 * and there is no writer, return 0 bytes.  If a process is writing to a
 * pipe and no one is reading from it, give a broken pipe error.
 */

  /* If reading, check for empty pipe. */
  if (rw_flag == READING) {
        if (position >= rip->i_size) {
                /* Process is reading from an empty pipe. */
                int r = 0;
                if (find_filp(rip, W_BIT) != NIL_FILP) {
                        /* Writer exists */
                        if (oflags & O_NONBLOCK) {
                                r = EAGAIN;
                        } else {
                                if (!notouch)
                                        suspend(XPIPE); /* block reader */
                                r = SUSPEND;
                        }
```

```
                                /* If need be, activate sleeping writers. */
                                if (susp_count > 0 && !notouch)
                                        release(rip, WRITE, susp_count);
                        }
                        return(r);
                }
  } else {
                /* Process is writing to a pipe. */
                if (find_filp(rip, R_BIT) == NIL_FILP) {
                        /* Tell kernel to generate a SIGPIPE signal. */
                        if (!notouch)
                                sys_kill((int)(fp - fproc), SIGPIPE);
                        return(EPIPE);
                }

                if (position + bytes > PIPE_SIZE(rip->i_sp->s_block_size)) {
                        if ((oflags & O_NONBLOCK)
                         && bytes < PIPE_SIZE(rip->i_sp->s_block_size))
                                return(EAGAIN);
                        else if ((oflags & O_NONBLOCK)
                        && bytes > PIPE_SIZE(rip->i_sp->s_block_size)) {
                        if ( (*canwrite = (PIPE_SIZE(rip->i_sp->s_block_size)
                                - position)) > 0)  {
                                        /* Do a partial write. Need to wakeup reader */
                                        if (!notouch)
                                                release(rip, READ, susp_count);
                                        return(1);
                                } else {
                                        return(EAGAIN);
                                }
                             }
                        if (bytes > PIPE_SIZE(rip->i_sp->s_block_size)) {
                                if ((*canwrite = PIPE_SIZE(rip->i_sp->s_block_size)
                                        - position) > 0) {
                                        /* Do a partial write. Need to wakeup reader
                                         * since we'll suspend ourself in read_write()
                                         */
                                        release(rip, READ, susp_count);
                                        return(1);
                                }
                        }
                        if (!notouch)
                                suspend(XPIPE); /* stop writer -- pipe full */
                        return(SUSPEND);
                }

                /* Writing to an empty pipe.  Search for suspended reader. */
                if (position == 0 && !notouch)
                        release(rip, READ, susp_count);
  }

  *canwrite = 0;
  return(1);
}

/*===========================================================================*
 *                              suspend                                      *
 *===========================================================================*/
PUBLIC void suspend(task)
int task;                       /* who is proc waiting for? (PIPE = pipe) */
```

```
{
/* Take measures to suspend the processing of the present system call.
 * Store the parameters to be used upon resuming in the process table.
 * (Actually they are not used when a process is waiting for an I/O device,
 * but they are needed for pipes, and it is not worth making the distinction.)
 * The SUSPEND pseudo error should be returned after calling suspend().
 */

  if (task == XPIPE || task == XPOPEN) susp_count++;/* #procs susp'ed on pipe*/
  fp->fp_suspended = SUSPENDED;
  fp->fp_fd = m_in.fd << 8 | call_nr;
  fp->fp_task = -task;
  if (task == XLOCK) {
        fp->fp_buffer = (char *) m_in.name1;    /* third arg to fcntl() */
        fp->fp_nbytes = m_in.request;           /* second arg to fcntl() */
  } else {
        fp->fp_buffer = m_in.buffer;            /* for reads and writes */
        fp->fp_nbytes = m_in.nbytes;
  }
}

/*===========================================================================*
 *                              release                                      *
 *===========================================================================*/
PUBLIC void release(ip, call_nr, count)
register struct inode *ip;      /* inode of pipe */
int call_nr;                    /* READ, WRITE, OPEN or CREAT */
int count;                      /* max number of processes to release */
{
/* Check to see if any process is hanging on the pipe whose inode is in 'ip'.
 * If one is, and it was trying to perform the call indicated by 'call_nr',
 * release it.
 */

  register struct fproc *rp;
  struct filp *f;

  /* Trying to perform the call also includes SELECTing on it with that
   * operation.
   */
  if (call_nr == READ || call_nr == WRITE) {
          int op;
          if (call_nr == READ)
                op = SEL_RD;
          else
                op = SEL_WR;
          for(f = &filp[0]; f < &filp[NR_FILPS]; f++) {
                if (f->filp_count < 1 || !(f->filp_pipe_select_ops & op) ||
                   f->filp_ino != ip)
                        continue;
                 select_callback(f, op);
                f->filp_pipe_select_ops &= ~op;
        }
  }

  /* Search the proc table. */
  for (rp = &fproc[0]; rp < &fproc[NR_PROCS]; rp++) {
        if (rp->fp_suspended == SUSPENDED &&
                        rp->fp_revived == NOT_REVIVING &&
                        (rp->fp_fd & BYTE) == call_nr &&
                        rp->fp_filp[rp->fp_fd>>8]->filp_ino == ip) {
                revive((int)(rp - fproc), 0);
                susp_count--;   /* keep track of who is suspended */
                if (--count == 0) return;
        }
  }
}

/*===========================================================================*
 *                              revive                                       *
 *===========================================================================*/
PUBLIC void revive(proc_nr, returned)
int proc_nr;                    /* process to revive */
int returned;                   /* if hanging on task, how many bytes read */
{
/* Revive a previously blocked process. When a process hangs on tty, this
 * is the way it is eventually released.
 */

  register struct fproc *rfp;
  register int task;

  if (proc_nr < 0 || proc_nr >= NR_PROCS)
        panic(__FILE__,"revive err", proc_nr);
  rfp = &fproc[proc_nr];
  if (rfp->fp_suspended == NOT_SUSPENDED || rfp->fp_revived == REVIVING)return;

  /* The 'reviving' flag only applies to pipes.  Processes waiting for TTY get
   * a message right away.  The revival process is different for TTY and pipes.
   * For select and TTY revival, the work is already done, for pipes it is not:
   *  the proc must be restarted so it can try again.
   */
  task = -rfp->fp_task;
  if (task == XPIPE || task == XLOCK) {
        /* Revive a process suspended on a pipe or lock. */
        rfp->fp_revived = REVIVING;
        reviving++;             /* process was waiting on pipe or lock */
  } else {
        rfp->fp_suspended = NOT_SUSPENDED;
        if (task == XPOPEN) /* process blocked in open or create */
                reply(proc_nr, rfp->fp_fd>>8);
        else if (task == XSELECT) {
                reply(proc_nr, returned);
        } else {
                /* Revive a process suspended on TTY or other device. */
                rfp->fp_nbytes = returned;      /*pretend it wants only what there is*/
                reply(proc_nr, returned);       /* unblock the process */
        }
  }
}

/*===========================================================================*
 *                              do_unpause                                   *
 *===========================================================================*/
PUBLIC int do_unpause()
{
/* A signal has been sent to a user who is paused on the file system.
 * Abort the system call with the EINTR error message.
 */

```

```
  register struct fproc *rfp;
  int proc_nr, task, fild;
  struct filp *f;
  dev_t dev;
  message mess;

  if (who > PM_PROC_NR) return(EPERM);
  proc_nr = m_in.pro;
  if (proc_nr < 0 || proc_nr >= NR_PROCS)
        panic(__FILE__,"unpause err 1", proc_nr);
  rfp = &fproc[proc_nr];
  if (rfp->fp_suspended == NOT_SUSPENDED) return(OK);
  task = -rfp->fp_task;

  switch (task) {
        case XPIPE:             /* process trying to read or write a pipe */
                break;

        case XLOCK:             /* process trying to set a lock with FCNTL */
                break;

        case XSELECT:           /* process blocking on select() */
                select_forget(proc_nr);
                break;

        case XPOPEN:            /* process trying to open a fifo */
                break;

        default:                /* process trying to do device I/O (e.g. tty)*/
                fild = (rfp->fp_fd >> 8) & BYTE;/* extract file descriptor */
                if (fild < 0 || fild >= OPEN_MAX)
                        panic(__FILE__,"unpause err 2",NO_NUM);
                f = rfp->fp_filp[fild];
                dev = (dev_t) f->filp_ino->i_zone[0];   /* device hung on */
                mess.TTY_LINE = (dev >> MINOR) & BYTE;
                mess.PROC_NR = proc_nr;

                /* Tell kernel R or W. Mode is from current call, not open. */
                mess.COUNT = (rfp->fp_fd & BYTE) == READ ? R_BIT : W_BIT;
                mess.m_type = CANCEL;
                fp = rfp;       /* hack - ctty_io uses fp */
                (*dmap[(dev >> MAJOR) & BYTE].dmap_io)(task, &mess);
  }

  rfp->fp_suspended = NOT_SUSPENDED;
  reply(proc_nr, EINTR);        /* signal interrupted call */
  return(OK);
}

/*===========================================================================*
 *                              select_request_pipe                          *
 *===========================================================================*/
PUBLIC int select_request_pipe(struct filp *f, int *ops, int block)
{
        int orig_ops, r = 0, err, canwrite;
        orig_ops = *ops;
        if ((*ops & SEL_RD)) {
                if ((err = pipe_check(f->filp_ino, READING, 0,
                        1, f->filp_pos, &canwrite, 1)) != SUSPEND)
                        r |= SEL_RD;
                if (err < 0 && err != SUSPEND && (*ops & SEL_ERR))
                        r |= SEL_ERR;
        }
        if ((*ops & SEL_WR)) {
                if ((err = pipe_check(f->filp_ino, WRITING, 0,
                        1, f->filp_pos, &canwrite, 1)) != SUSPEND)
                        r |= SEL_WR;
                if (err < 0 && err != SUSPEND && (*ops & SEL_ERR))
                        r |= SEL_ERR;
        }

        *ops = r;

        if (!r && block) {
                f->filp_pipe_select_ops |= orig_ops;
        }

        return SEL_OK;
}

/*===========================================================================*
 *                              select_match_pipe                            *
 *===========================================================================*/
PUBLIC int select_match_pipe(struct filp *f)
{
        /* recognize either pipe or named pipe (FIFO) */
        if (f && f->filp_ino && (f->filp_ino->i_mode & I_NAMED_PIPE))
                return 1;
        return 0;
}
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                servers/fs/path.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/* This file contains the procedures that look up path names in the directory
 * system and determine the inode number that goes with a given path name.
 *
 *  The entry points into this file are
 *   eat_path:   the 'main' routine of the path-to-inode conversion mechanism
 *   last_dir:   find the final directory on a given path
 *   advance:    parse one component of a path name
 *   search_dir: search a directory for a string and return its inode number
 */

#include "fs.h"
#include <string.h>
#include <minix/callnr.h>
#include "buf.h"
#include "file.h"
#include "fproc.h"
#include "inode.h"
#include "super.h"

PUBLIC char dot1[2] = ".";      /* used for search_dir to bypass the access */
PUBLIC char dot2[3] = "..";     /* permissions for . and ..                 */

FORWARD _PROTOTYPE( char *get_name, (char *old_name, char string [NAME_MAX]) );

/*===========================================================================*
 *                              eat_path                                     *
 *===========================================================================*/
PUBLIC struct inode *eat_path(path)
char *path;                     /* the path name to be parsed */
{
/* Parse the path 'path' and put its inode in the inode table. If not possible,
 * return NIL_INODE as function value and an error code in 'err_code'.
 */

  register struct inode *ldip, *rip;
  char string[NAME_MAX];        /* hold 1 path component name here */

  /* First open the path down to the final directory. */
  if ( (ldip = last_dir(path, string)) == NIL_INODE) {
        return(NIL_INODE);      /* we couldn't open final directory */
        }

  /* The path consisting only of "/" is a special case, check for it. */
  if (string[0] == '\0') return(ldip);

  /* Get final component of the path. */
  rip = advance(ldip, string);
  put_inode(ldip);
  return(rip);
}

/*===========================================================================*
 *                              last_dir                                     *
 *===========================================================================*/
PUBLIC struct inode *last_dir(path, string)
char *path;                     /* the path name to be parsed */
char string[NAME_MAX];          /* the final component is returned here */
{
/* Given a path, 'path', located in the fs address space, parse it as
 * far as the last directory, fetch the inode for the last directory into
 * the inode table, and return a pointer to the inode.  In
 * addition, return the final component of the path in 'string'.
 * If the last directory can't be opened, return NIL_INODE and
 * the reason for failure in 'err_code'.
 */

  register struct inode *rip;
  register char *new_name;
  register struct inode *new_ip;

  /* Is the path absolute or relative?  Initialize 'rip' accordingly. */
  rip = (*path == '/' ? fp->fp_rootdir : fp->fp_workdir);

  /* If dir has been removed or path is empty, return ENOENT. */
  if (rip->i_nlinks == 0 || *path == '\0') {
        err_code = ENOENT;
        return(NIL_INODE);
  }

  dup_inode(rip);               /* inode will be returned with put_inode */

  /* Scan the path component by component. */
  while (TRUE) {
        /* Extract one component. */
        if ( (new_name = get_name(path, string)) == (char*) 0) {
                put_inode(rip); /* bad path in user space */
                return(NIL_INODE);
        }
        if (*new_name == '\0') {
                if ( (rip->i_mode & I_TYPE) == I_DIRECTORY) {
                        return(rip);    /* normal exit */
                } else {
                        /* last file of path prefix is not a directory */
                        put_inode(rip);
                        err_code = ENOTDIR;
                        return(NIL_INODE);
                }
        }

        /* There is more path.  Keep parsing. */
        new_ip = advance(rip, string);
        put_inode(rip);         /* rip either obsolete or irrelevant */
        if (new_ip == NIL_INODE) return(NIL_INODE);

        /* The call to advance() succeeded.  Fetch next component. */
        path = new_name;
        rip = new_ip;
  }
}

/*===========================================================================*
 *                              get_name                                     *
 *===========================================================================*/
PRIVATE char *get_name(old_name, string)
char *old_name;                 /* path name to parse */
char string[NAME_MAX];          /* component extracted from 'old_name' */
{
/* Given a pointer to a path name in fs space, 'old_name', copy the next
 * component to 'string' and pad with zeros.  A pointer to that part of
 * the name as yet unparsed is returned.  Roughly speaking,
 * 'get_name' = 'old_name' - 'string'.
 *
 * This routine follows the standard convention that /usr/ast, /usr//ast,
 * //usr///ast and /usr/ast/ are all equivalent.
 */

  register int c;
  register char *np, *rnp;

  np = string;                  /* 'np' points to current position */
  rnp = old_name;               /* 'rnp' points to unparsed string */
  while ( (c = *rnp) == '/') rnp++;     /* skip leading slashes */

  /* Copy the unparsed path, 'old_name', to the array, 'string'. */
  while ( rnp < &old_name[PATH_MAX]  &&  c != '/'   &&  c != '\0') {
        if (np < &string[NAME_MAX]) *np++ = c;
        c = *++rnp;             /* advance to next character */
  }

  /* To make /usr/ast/ equivalent to /usr/ast, skip trailing slashes. */
  while (c == '/' && rnp < &old_name[PATH_MAX]) c = *++rnp;

  if (np < &string[NAME_MAX]) *np = '\0';       /* Terminate string */

  if (rnp >= &old_name[PATH_MAX]) {
        err_code = ENAMETOOLONG;
        return((char *) 0);
  }
  return(rnp);
}

/*===========================================================================*
 *                              advance                                      *
 *===========================================================================*/
PUBLIC struct inode *advance(dirp, string)
struct inode *dirp;             /* inode for directory to be searched */
char string[NAME_MAX];          /* component name to look for */
{
/* Given a directory and a component of a path, look up the component in
 * the directory, find the inode, open it, and return a pointer to its inode
 * slot.  If it can't be done, return NIL_INODE.
 */

  register struct inode *rip;
  struct inode *rip2;
  register struct super_block *sp;
  int r, inumb;
  dev_t mnt_dev;
  ino_t numb;

  /* If 'string' is empty, yield same inode straight away. */
  if (string[0] == '\0') { return(get_inode(dirp->i_dev, (int) dirp->i_num)); }

  /* Check for NIL_INODE. */
  if (dirp == NIL_INODE) { return(NIL_INODE); }

  /* If 'string' is not present in the directory, signal error. */
  if ( (r = search_dir(dirp, string, &numb, LOOK_UP)) != OK) {
        err_code = r;
        return(NIL_INODE);
  }

  /* Don't go beyond the current root directory, unless the string is dot2. */
  if (dirp == fp->fp_rootdir && strcmp(string, "..") == 0 && string != dot2)
                return(get_inode(dirp->i_dev, (int) dirp->i_num));

  /* The component has been found in the directory.  Get inode. */
  if ( (rip = get_inode(dirp->i_dev, (int) numb)) == NIL_INODE)  {
        return(NIL_INODE);
        }

  if (rip->i_num == ROOT_INODE)
        if (dirp->i_num == ROOT_INODE) {
            if (string[1] == '.') {
                for (sp = &super_block[1]; sp < &super_block[NR_SUPERS]; sp++){
                        if (sp->s_dev == rip->i_dev) {
                                /* Release the root inode.  Replace by the
                                 * inode mounted on.
                                 */
                                put_inode(rip);
                                mnt_dev = sp->s_imount->i_dev;
                                inumb = (int) sp->s_imount->i_num;
                                rip2 = get_inode(mnt_dev, inumb);
                                rip = advance(rip2, string);
                                put_inode(rip2);
                                break;
                        }
                }
            }
        }
  if (rip == NIL_INODE) return(NIL_INODE);

  /* See if the inode is mounted on.  If so, switch to root directory of the
   * mounted file system.  The super_block provides the linkage between the
   * inode mounted on and the root directory of the mounted file system.
   */
  while (rip != NIL_INODE && rip->i_mount == I_MOUNT) {
        /* The inode is indeed mounted on. */
        for (sp = &super_block[0]; sp < &super_block[NR_SUPERS]; sp++) {
                if (sp->s_imount == rip) {
                        /* Release the inode mounted on.  Replace by the
                         * inode of the root inode of the mounted device.
                         */
                        put_inode(rip);
                        rip = get_inode(sp->s_dev, ROOT_INODE);
                        break;
                }
        }
  }
  return(rip);          /* return pointer to inode's component */
}

/*===========================================================================*
 *                              search_dir                                   *
 *===========================================================================*/
PUBLIC int search_dir(ldir_ptr, string, numb, flag)
register struct inode *ldir_ptr; /* ptr to inode for dir to search */
char string[NAME_MAX];           /* component to search for */
ino_t *numb;                     /* pointer to inode number */
int flag;                        /* LOOK_UP, ENTER, DELETE or IS_EMPTY */
{
/* This function searches the directory whose inode is pointed to by 'ldip':
 * if (flag == ENTER)  enter 'string' in the directory with inode # '*numb';
 * if (flag == DELETE) delete 'string' from the directory;
 * if (flag == LOOK_UP) search for 'string' and return inode # in 'numb';
 * if (flag == IS_EMPTY) return OK if only . and .. in dir else ENOTEMPTY;
 *
 *    if 'string' is dot1 or dot2, no access permissions are checked.
 */

  register struct direct *dp = NULL;
  register struct buf *bp = NULL;
  int i, r, e_hit, t, match;
  mode_t bits;
  off_t pos;
  unsigned new_slots, old_slots;
  block_t b;
  struct super_block *sp;
  int extended = 0;

  /* If 'ldir_ptr' is not a pointer to a dir inode, error. */
  if ( (ldir_ptr->i_mode & I_TYPE) != I_DIRECTORY) return(ENOTDIR);

  r = OK;

  if (flag != IS_EMPTY) {
        bits = (flag == LOOK_UP ? X_BIT : W_BIT | X_BIT);

        if (string == dot1 || string == dot2) {
                if (flag != LOOK_UP) r = read_only(ldir_ptr);
                                     /* only a writable device is required. */
        }
        else r = forbidden(ldir_ptr, bits); /* check access permissions */
  }
  if (r != OK) return(r);

  /* Step through the directory one block at a time. */
  old_slots = (unsigned) (ldir_ptr->i_size/DIR_ENTRY_SIZE);
  new_slots = 0;
  e_hit = FALSE;
  match = 0;                    /* set when a string match occurs */

  for (pos = 0; pos < ldir_ptr->i_size; pos += ldir_ptr->i_sp->s_block_size) {
        b = read_map(ldir_ptr, pos);    /* get block number */

        /* Since directories don't have holes, 'b' cannot be NO_BLOCK. */
        bp = get_block(ldir_ptr->i_dev, b, NORMAL);     /* get a dir block */

        if (bp == NO_BLOCK)
                panic(__FILE__,"get_block returned NO_BLOCK", NO_NUM);

        /* Search a directory block. */
        for (dp = &bp->b_dir[0];
                dp < &bp->b_dir[NR_DIR_ENTRIES(ldir_ptr->i_sp->s_block_size)];
                dp++) {
                if (++new_slots > old_slots) { /* not found, but room left */
                        if (flag == ENTER) e_hit = TRUE;
                        break;
                }

                /* Match occurs if string found. */
                if (flag != ENTER && dp->d_ino != 0) {
                        if (flag == IS_EMPTY) {
                                /* If this test succeeds, dir is not empty. */
                                if (strcmp(dp->d_name, "." ) != 0 &&
                                    strcmp(dp->d_name, "..") != 0) match = 1;
                        } else {
                                if (strncmp(dp->d_name, string, NAME_MAX) == 0) {
                                        match = 1;
                                }
                        }
                }

                if (match) {
                        /* LOOK_UP or DELETE found what it wanted. */
                        r = OK;
                        if (flag == IS_EMPTY) r = ENOTEMPTY;
                        else if (flag == DELETE) {
                                /* Save d_ino for recovery. */
                                t = NAME_MAX - sizeof(ino_t);
                                *((ino_t *) &dp->d_name[t]) = dp->d_ino;
                                dp->d_ino = 0;  /* erase entry */
                                bp->b_dirt = DIRTY;
                                ldir_ptr->i_update |= CTIME | MTIME;
                                ldir_ptr->i_dirt = DIRTY;
                        } else {
                                sp = ldir_ptr->i_sp;    /* 'flag' is LOOK_UP */
                                *numb = conv4(sp->s_native, (int) dp->d_ino);
                        }
                        put_block(bp, DIRECTORY_BLOCK);
                        return(r);
                }

                /* Check for free slot for the benefit of ENTER. */
                if (flag == ENTER && dp->d_ino == 0) {
                        e_hit = TRUE;   /* we found a free slot */
                        break;
                }
        }

        /* The whole block has been searched or ENTER has a free slot. */
        if (e_hit) break;       /* e_hit set if ENTER can be performed now */
        put_block(bp, DIRECTORY_BLOCK); /* otherwise, continue searching dir */
  }

  /* The whole directory has now been searched. */
  if (flag != ENTER) {
        return(flag == IS_EMPTY ? OK : ENOENT);
  }

  /* This call is for ENTER.  If no free slot has been found so far, try to
   * extend directory.
   */
  if (e_hit == FALSE) { /* directory is full and no room left in last block */
        new_slots++;            /* increase directory size by 1 entry */
        if (new_slots == 0) return(EFBIG); /* dir size limited by slot count */
        if ( (bp = new_block(ldir_ptr, ldir_ptr->i_size)) == NIL_BUF)
                return(err_code);
        dp = &bp->b_dir[0];
        extended = 1;
  }

  /* 'bp' now points to a directory block with space. 'dp' points to slot. */
  (void) memset(dp->d_name, 0, (size_t) NAME_MAX); /* clear entry */
  for (i = 0; string[i] && i < NAME_MAX; i++) dp->d_name[i] = string[i];
  sp = ldir_ptr->i_sp;
  dp->d_ino = conv4(sp->s_native, (int) *numb);
  bp->b_dirt = DIRTY;
  put_block(bp, DIRECTORY_BLOCK);
  ldir_ptr->i_update |= CTIME | MTIME;  /* mark mtime for update later */
  ldir_ptr->i_dirt = DIRTY;
  if (new_slots > old_slots) {
        ldir_ptr->i_size = (off_t) new_slots * DIR_ENTRY_SIZE;
        /* Send the change to disk if the directory is extended. */
        if (extended) rw_inode(ldir_ptr, WRITING);
  }
  return(OK);
}
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                      servers/fs/mount.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/* This file performs the MOUNT and UMOUNT system calls.
 *
 * The entry points into this file are
 *   do_mount:  perform the MOUNT system call
 *   do_umount: perform the UMOUNT system call
 */

#include "fs.h"
#include <fcntl.h>
#include <minix/com.h>
#include <sys/stat.h>
#include "buf.h"
#include "file.h"
#include "fproc.h"
#include "inode.h"
#include "param.h"
#include "super.h"

FORWARD _PROTOTYPE( dev_t name_to_dev, (char *path)                     );

/*===========================================================================*
 *                              do_mount                                     *
 *===========================================================================*/
PUBLIC int do_mount()
{
/* Perform the mount(name, mfile, rd_only) system call. */

  register struct inode *rip, *root_ip;
  struct super_block *xp, *sp;
  dev_t dev;
  mode_t bits;
  int rdir, mdir;               /* TRUE iff {root|mount} file is dir */
  int r, found;

  /* Only the super-user may do MOUNT. */
  if (!super_user) return(EPERM);

  /* If 'name' is not for a block special file, return error. */
  if (fetch_name(m_in.name1, m_in.name1_length, M1) != OK) return(err_code);
  if ( (dev = name_to_dev(user_path)) == NO_DEV) return(err_code);

  /* Scan super block table to see if dev already mounted & find a free slot.*/
  sp = NIL_SUPER;
  found = FALSE;
  for (xp = &super_block[0]; xp < &super_block[NR_SUPERS]; xp++) {
        if (xp->s_dev == dev) found = TRUE;     /* is it mounted already? */
        if (xp->s_dev == NO_DEV) sp = xp;       /* record free slot */
  }
  if (found) return(EBUSY);     /* already mounted */
  if (sp == NIL_SUPER) return(ENFILE);  /* no super block available */

  /* Open the device the file system lives on. */
  if (dev_open(dev, who, m_in.rd_only ? R_BIT : (R_BIT|W_BIT)) != OK)
        return(EINVAL);

  /* Make the cache forget about blocks it has open on the filesystem */
  (void) do_sync();
  invalidate(dev);

  /* Fill in the super block. */
  sp->s_dev = dev;              /* read_super() needs to know which dev */
  r = read_super(sp);

  /* Is it recognized as a Minix filesystem? */
  if (r != OK) {
        dev_close(dev);
        sp->s_dev = NO_DEV;
        return(r);
  }

  /* Now get the inode of the file to be mounted on. */
  if (fetch_name(m_in.name2, m_in.name2_length, M1) != OK) {
        dev_close(dev);
        sp->s_dev = NO_DEV;
        return(err_code);
  }
  if ( (rip = eat_path(user_path)) == NIL_INODE) {
        dev_close(dev);
        sp->s_dev = NO_DEV;
        return(err_code);
  }

  /* It may not be busy. */
  r = OK;
  if (rip->i_count > 1) r = EBUSY;

  /* It may not be special. */
  bits = rip->i_mode & I_TYPE;
  if (bits == I_BLOCK_SPECIAL || bits == I_CHAR_SPECIAL) r = ENOTDIR;

  /* Get the root inode of the mounted file system. */
  root_ip = NIL_INODE;          /* if 'r' not OK, make sure this is defined */
  if (r == OK) {
        if ( (root_ip = get_inode(dev, ROOT_INODE)) == NIL_INODE) r = err_code;
  }
  if (root_ip != NIL_INODE && root_ip->i_mode == 0) {
        r = EINVAL;
  }

  /* File types of 'rip' and 'root_ip' may not conflict. */
  if (r == OK) {
        mdir = ((rip->i_mode & I_TYPE) == I_DIRECTORY);  /* TRUE iff dir */
        rdir = ((root_ip->i_mode & I_TYPE) == I_DIRECTORY);
        if (!mdir && rdir) r = EISDIR;
  }

  /* If error, return the super block and both inodes; release the maps. */
  if (r != OK) {
        put_inode(rip);
        put_inode(root_ip);
        (void) do_sync();
        invalidate(dev);
        dev_close(dev);
        sp->s_dev = NO_DEV;
        return(r);
  }

  /* Nothing else can go wrong.  Perform the mount. */
  rip->i_mount = I_MOUNT;       /* this bit says the inode is mounted on */
  sp->s_imount = rip;
  sp->s_isup = root_ip;
  sp->s_rd_only = m_in.rd_only;
  return(OK);
}

/*===========================================================================*
 *                              do_umount                                    *
 *===========================================================================*/
PUBLIC int do_umount()
{
/* Perform the umount(name) system call. */
  dev_t dev;

  /* Only the super-user may do UMOUNT. */
  if (!super_user) return(EPERM);

  /* If 'name' is not for a block special file, return error. */
  if (fetch_name(m_in.name, m_in.name_length, M3) != OK) return(err_code);
  if ( (dev = name_to_dev(user_path)) == NO_DEV) return(err_code);

  return(unmount(dev));
}

/*===========================================================================*
 *                              unmount                                      *
 *===========================================================================*/
PUBLIC int unmount(dev)
Dev_t dev;
{
/* Unmount a file system by device number. */
  register struct inode *rip;
  struct super_block *sp, *sp1;
  int count;

  /* See if the mounted device is busy.  Only 1 inode using it should be
   * open -- the root inode -- and that inode only 1 time.
   */
  count = 0;
  for (rip = &inode[0]; rip< &inode[NR_INODES]; rip++)
        if (rip->i_count > 0 && rip->i_dev == dev) count += rip->i_count;
  if (count > 1) return(EBUSY); /* can't umount a busy file system */

  /* Find the super block. */
  sp = NIL_SUPER;
  for (sp1 = &super_block[0]; sp1 < &super_block[NR_SUPERS]; sp1++) {
        if (sp1->s_dev == dev) {
                sp = sp1;
                break;
        }
  }

  /* Sync the disk, and invalidate cache. */
  (void) do_sync();             /* force any cached blocks out of memory */
  invalidate(dev);              /* invalidate cache entries for this dev */
  if (sp == NIL_SUPER) {
        return(EINVAL);
  }

  /* Close the device the file system lives on. */
  dev_close(dev);

  /* Finish off the unmount. */
  sp->s_imount->i_mount = NO_MOUNT;     /* inode returns to normal */
  put_inode(sp->s_imount);      /* release the inode mounted on */
  put_inode(sp->s_isup);        /* release the root inode of the mounted fs */
  sp->s_imount = NIL_INODE;
  sp->s_dev = NO_DEV;
  return(OK);
}

/*===========================================================================*
 *                              name_to_dev                                  *
 *===========================================================================*/
PRIVATE dev_t name_to_dev(path)
char *path;                     /* pointer to path name */
{
/* Convert the block special file 'path' to a device number.  If 'path'
 * is not a block special file, return error code in 'err_code'.
 */

  register struct inode *rip;
  register dev_t dev;

  /* If 'path' can't be opened, give up immediately. */
  if ( (rip = eat_path(path)) == NIL_INODE) return(NO_DEV);

  /* If 'path' is not a block special file, return error. */
  if ( (rip->i_mode & I_TYPE) != I_BLOCK_SPECIAL) {
        err_code = ENOTBLK;
        put_inode(rip);
        return(NO_DEV);
  }

  /* Extract the device number. */
  dev = (dev_t) rip->i_zone[0];
  put_inode(rip);
  return(dev);
}
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                servers/fs/link.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/* This file handles the LINK and UNLINK system calls.  It also deals with
 * deallocating the storage used by a file when the last UNLINK is done to a
 * file and the blocks must be returned to the free block pool.
 *
 * The entry points into this file are
 *   do_link:   perform the LINK system call
 *   do_unlink: perform the UNLINK and RMDIR system calls
 *   do_rename: perform the RENAME system call
 *   truncate:  release all the blocks associated with an inode
 */
```

```

#include "fs.h"
#include <sys/stat.h>
#include <string.h>
#include <minix/com.h>
#include <minix/callnr.h>
#include "buf.h"
#include "file.h"
#include "fproc.h"
#include "inode.h"
#include "param.h"
#include "super.h"

#define SAME 1000

FORWARD _PROTOTYPE( int remove_dir, (struct inode *rldirp, struct inode *rip,
                        char dir_name[NAME_MAX])                        );

FORWARD _PROTOTYPE( int unlink_file, (struct inode *dirp, struct inode *rip,
                        char file_name[NAME_MAX])                       );

/*===========================================================================*
 *                              do_link                                      *
 *===========================================================================*/
PUBLIC int do_link()
{
/* Perform the link(name1, name2) system call. */

  register struct inode *ip, *rip;
  register int r;
  char string[NAME_MAX];
  struct inode *new_ip;

  /* See if 'name' (file to be linked) exists. */
  if (fetch_name(m_in.name1, m_in.name1_length, M1) != OK) return(err_code);
  if ( (rip = eat_path(user_path)) == NIL_INODE) return(err_code);

  /* Check to see if the file has maximum number of links already. */
  r = OK;
  if (rip->i_nlinks >= (rip->i_sp->s_version == V1 ? CHAR_MAX : SHRT_MAX))
        r = EMLINK;

  /* Only super_user may link to directories. */
  if (r == OK)
        if ( (rip->i_mode & I_TYPE) == I_DIRECTORY && !super_user) r = EPERM;

  /* If error with 'name', return the inode. */
  if (r != OK) {
        put_inode(rip);
        return(r);
  }

  /* Does the final directory of 'name2' exist? */
  if (fetch_name(m_in.name2, m_in.name2_length, M1) != OK) {
        put_inode(rip);
        return(err_code);
  }
  if ( (ip = last_dir(user_path, string)) == NIL_INODE) r = err_code;

  /* If 'name2' exists in full (even if no space) set 'r' to error. */
  if (r == OK) {
        if ( (new_ip = advance(ip, string)) == NIL_INODE) {
                r = err_code;
                if (r == ENOENT) r = OK;
        } else {
                put_inode(new_ip);
                r = EEXIST;
        }
  }

  /* Check for links across devices. */
  if (r == OK)
        if (rip->i_dev != ip->i_dev) r = EXDEV;

  /* Try to link. */
  if (r == OK)
        r = search_dir(ip, string, &rip->i_num, ENTER);

  /* If success, register the linking. */
  if (r == OK) {
        rip->i_nlinks++;
        rip->i_update |= CTIME;
        rip->i_dirt = DIRTY;
  }

  /* Done.  Release both inodes. */
  put_inode(rip);
  put_inode(ip);
  return(r);
}

/*===========================================================================*
 *                              do_unlink                                    *
 *===========================================================================*/
PUBLIC int do_unlink()
{
/* Perform the unlink(name) or rmdir(name) system call. The code for these two
 * is almost the same.  They differ only in some condition testing.  Unlink()
 * may be used by the superuser to do dangerous things; rmdir() may not.
 */

  register struct inode *rip;
  struct inode *rldirp;
  int r;
  char string[NAME_MAX];

  /* Get the last directory in the path. */
  if (fetch_name(m_in.name, m_in.name_length, M3) != OK) return(err_code);
  if ( (rldirp = last_dir(user_path, string)) == NIL_INODE)
        return(err_code);

  /* The last directory exists.  Does the file also exist? */
  r = OK;
  if ( (rip = advance(rldirp, string)) == NIL_INODE) r = err_code;

  /* If error, return inode. */
  if (r != OK) {
        put_inode(rldirp);
        return(r);
  }

  /* Do not remove a mount point. */
  if (rip->i_num == ROOT_INODE) {
        put_inode(rldirp);
        put_inode(rip);
        return(EBUSY);
  }

  /* Now test if the call is allowed, separately for unlink() and rmdir(). */
  if (call_nr == UNLINK) {
        /* Only the su may unlink directories, but the su can unlink any dir.*/
        if ( (rip->i_mode & I_TYPE) == I_DIRECTORY && !super_user) r = EPERM;

        /* Don't unlink a file if it is the root of a mounted file system. */
        if (rip->i_num == ROOT_INODE) r = EBUSY;

        /* Actually try to unlink the file; fails if parent is mode 0 etc. */
        if (r == OK) r = unlink_file(rldirp, rip, string);

  } else {
        r = remove_dir(rldirp, rip, string); /* call is RMDIR */
  }

  /* If unlink was possible, it has been done, otherwise it has not. */
  put_inode(rip);
  put_inode(rldirp);
  return(r);
}

/*===========================================================================*
 *                              do_rename                                    *
 *===========================================================================*/
PUBLIC int do_rename()
{
/* Perform the rename(name1, name2) system call. */

  struct inode *old_dirp, *old_ip;      /* ptrs to old dir, file inodes */
  struct inode *new_dirp, *new_ip;      /* ptrs to new dir, file inodes */
  struct inode *new_superdirp, *next_new_superdirp;
  int r = OK;                           /* error flag; initially no error */
  int odir, ndir;                       /* TRUE iff {old|new} file is dir */
  int same_pdir;                        /* TRUE iff parent dirs are the same */
  char old_name[NAME_MAX], new_name[NAME_MAX];
  ino_t numb;
  int r1;

  /* See if 'name1' (existing file) exists.  Get dir and file inodes. */
  if (fetch_name(m_in.name1, m_in.name1_length, M1) != OK) return(err_code);
  if ( (old_dirp = last_dir(user_path, old_name))==NIL_INODE) return(err_code);

  if ( (old_ip = advance(old_dirp, old_name)) == NIL_INODE) r = err_code;

  /* See if 'name2' (new name) exists.  Get dir and file inodes. */
  if (fetch_name(m_in.name2, m_in.name2_length, M1) != OK) r = err_code;
  if ( (new_dirp = last_dir(user_path, new_name)) == NIL_INODE) r = err_code;
  new_ip = advance(new_dirp, new_name); /* not required to exist */

  if (old_ip != NIL_INODE)
        odir = ((old_ip->i_mode & I_TYPE) == I_DIRECTORY);  /* TRUE iff dir */

  /* If it is ok, check for a variety of possible errors. */
  if (r == OK) {
        same_pdir = (old_dirp == new_dirp);

        /* The old inode must not be a superdirectory of the new last dir. */
        if (odir && !same_pdir) {
                dup_inode(new_superdirp = new_dirp);
                while (TRUE) {          /* may hang in a file system loop */
                        if (new_superdirp == old_ip) {
                                r = EINVAL;
                                break;
                        }
                        next_new_superdirp = advance(new_superdirp, dot2);
                        put_inode(new_superdirp);
                        if (next_new_superdirp == new_superdirp)
                                break;  /* back at system root directory */
                        new_superdirp = next_new_superdirp;
                        if (new_superdirp == NIL_INODE) {
                                /* Missing ".." entry.  Assume the worst. */
                                r = EINVAL;
                                break;
                        }
                }
                put_inode(new_superdirp);
        }

        /* The old or new name must not be . or .. */
        if (strcmp(old_name, ".")==0 || strcmp(old_name, "..")==0 ||
            strcmp(new_name, ".")==0 || strcmp(new_name, "..")==0) r = EINVAL;

        /* Both parent directories must be on the same device. */
        if (old_dirp->i_dev != new_dirp->i_dev) r = EXDEV;

        /* Parent dirs must be writable, searchable and on a writable device */
        if ((r1 = forbidden(old_dirp, W_BIT | X_BIT)) != OK ||
            (r1 = forbidden(new_dirp, W_BIT | X_BIT)) != OK) r = r1;

        /* Some tests apply only if the new path exists. */
        if (new_ip == NIL_INODE) {
                /* don't rename a file with a file system mounted on it. */
                if (old_ip->i_dev != old_dirp->i_dev) r = EXDEV;
                if (odir && new_dirp->i_nlinks >=
                    (new_dirp->i_sp->s_version == V1 ? CHAR_MAX : SHRT_MAX) &&
                    !same_pdir && r == OK) r = EMLINK;
        } else {
                if (old_ip == new_ip) r = SAME; /* old=new */

                /* has the old file or new file a file system mounted on it? */
                if (old_ip->i_dev != new_ip->i_dev) r = EXDEV;

                ndir = ((new_ip->i_mode & I_TYPE) == I_DIRECTORY); /* dir ? */
                if (odir == TRUE && ndir == FALSE) r = ENOTDIR;
                if (odir == FALSE && ndir == TRUE) r = EISDIR;
        }
  }

  /* If a process has another root directory than the system root, we might
   * "accidently" be moving it's working directory to a place where it's
   * root directory isn't a super directory of it anymore. This can make
   * the function chroot useless. If chroot will be used often we should
   * probably check for it here.
   */

  /* The rename will probably work. Only two things can go wrong now:
   * 1. being unable to remove the new file. (when new file already exists)
   * 2. being unable to make the new directory entry. (new file doesn't exists)
   *     [directory has to grow by one block and cannot because the disk
   *      is completely full].
   */
  if (r == OK) {
        if (new_ip != NIL_INODE) {
                  /* There is already an entry for 'new'. Try to remove it. */
                if (odir)
                        r = remove_dir(new_dirp, new_ip, new_name);
                else
                        r = unlink_file(new_dirp, new_ip, new_name);
        }
        /* if r is OK, the rename will succeed, while there is now an
         * unused entry in the new parent directory.
         */
  }

  if (r == OK) {
        /* If the new name will be in the same parent directory as the old one,
         * first remove the old name to free an entry for the new name,
         * otherwise first try to create the new name entry to make sure
         * the rename will succeed.
         */
        numb = old_ip->i_num;           /* inode number of old file */

        if (same_pdir) {
                r = search_dir(old_dirp, old_name, (ino_t *) 0, DELETE);
                                                /* shouldn't go wrong. */
                if (r==OK) (void) search_dir(old_dirp, new_name, &numb, ENTER);
        } else {
                r = search_dir(new_dirp, new_name, &numb, ENTER);
                if (r == OK)
                    (void) search_dir(old_dirp, old_name, (ino_t *) 0, DELETE);
        }
  }
  /* If r is OK, the ctime and mtime of old_dirp and new_dirp have been marked
   * for update in search_dir.
   */

  if (r == OK && odir && !same_pdir) {
        /* Update the .. entry in the directory (still points to old_dirp). */
        numb = new_dirp->i_num;
        (void) unlink_file(old_ip, NIL_INODE, dot2);
        if (search_dir(old_ip, dot2, &numb, ENTER) == OK) {
                /* New link created. */
                new_dirp->i_nlinks++;
                new_dirp->i_dirt = DIRTY;
        }
  }

  /* Release the inodes. */
  put_inode(old_dirp);
  put_inode(old_ip);
  put_inode(new_dirp);
  put_inode(new_ip);
  return(r == SAME ? OK : r);
}

/*===========================================================================*
 *                              truncate                                     *
 *===========================================================================*/
PUBLIC void truncate(rip)
register struct inode *rip;     /* pointer to inode to be truncated */
{
/* Remove all the zones from the inode 'rip' and mark it dirty. */

  register block_t b;
  zone_t z, zone_size, z1;
  off_t position;
  int i, scale, file_type, waspipe, single, nr_indirects;
  struct buf *bp;
  dev_t dev;

  file_type = rip->i_mode & I_TYPE;     /* check to see if file is special */
  if (file_type == I_CHAR_SPECIAL || file_type == I_BLOCK_SPECIAL) return;
  dev = rip->i_dev;             /* device on which inode resides */
  scale = rip->i_sp->s_log_zone_size;
  zone_size = (zone_t) rip->i_sp->s_block_size << scale;
  nr_indirects = rip->i_nindirs;

  /* Pipes can shrink, so adjust size to make sure all zones are removed. */
  waspipe = rip->i_pipe == I_PIPE;      /* TRUE is this was a pipe */
  if (waspipe) rip->i_size = PIPE_SIZE(rip->i_sp->s_block_size);

  /* Step through the file a zone at a time, finding and freeing the zones. */
  for (position = 0; position < rip->i_size; position += zone_size) {
        if ( (b = read_map(rip, position)) != NO_BLOCK) {
                z = (zone_t) b >> scale;
                free_zone(dev, z);
        }
  }

  /* All the data zones have been freed.  Now free the indirect zones. */
  rip->i_dirt = DIRTY;
  if (waspipe) {
        wipe_inode(rip);        /* clear out inode for pipes */
        return;                 /* indirect slots contain file positions */
  }
  single = rip->i_ndzones;
  free_zone(dev, rip->i_zone[single]);  /* single indirect zone */
  if ( (z = rip->i_zone[single+1]) != NO_ZONE) {
        /* Free all the single indirect zones pointed to by the double. */
        b = (block_t) z << scale;
        bp = get_block(dev, b, NORMAL); /* get double indirect zone */
        for (i = 0; i < nr_indirects; i++) {
                z1 = rd_indir(bp, i);
                free_zone(dev, z1);
        }

        /* Now free the double indirect zone itself. */
        put_block(bp, INDIRECT_BLOCK);
        free_zone(dev, z);
  }

  /* Leave zone numbers for de(1) to recover file after an unlink(2).  */
```

```
}

/*===========================================================================*
 *                              remove_dir                                   *
 *===========================================================================*/
PRIVATE int remove_dir(rldirp, rip, dir_name)
struct inode *rldirp;                   /* parent directory */
struct inode *rip;                      /* directory to be removed */
char dir_name[NAME_MAX];                /* name of directory to be removed */
{
  /* A directory file has to be removed. Five conditions have to met:
   *    - The file must be a directory
   *    - The directory must be empty (except for . and ..)
   *    - The final component of the path must not be . or ..
   *    - The directory must not be the root of a mounted file system
   *    - The directory must not be anybody's root/working directory
   */

  int r;
  register struct fproc *rfp;

  /* search_dir checks that rip is a directory too. */
  if ((r = search_dir(rip, "", (ino_t *) 0, IS_EMPTY)) != OK) return r;

  if (strcmp(dir_name, ".") == 0 || strcmp(dir_name, "..") == 0)return(EINVAL);
  if (rip->i_num == ROOT_INODE) return(EBUSY); /* can't remove 'root' */

  for (rfp = &fproc[INIT_PROC_NR + 1]; rfp < &fproc[NR_PROCS]; rfp++)
        if (rfp->fp_workdir == rip || rfp->fp_rootdir == rip) return(EBUSY);
                                /* can't remove anybody's working dir */

  /* Actually try to unlink the file; fails if parent is mode 0 etc. */
  if ((r = unlink_file(rldirp, rip, dir_name)) != OK) return r;

  /* Unlink . and .. from the dir. The super user can link and unlink any dir,
   * so don't make too many assumptions about them.
   */
  (void) unlink_file(rip, NIL_INODE, dot1);
  (void) unlink_file(rip, NIL_INODE, dot2);
  return(OK);
}

/*===========================================================================*
 *                              unlink_file                                  *
 *===========================================================================*/
PRIVATE int unlink_file(dirp, rip, file_name)
struct inode *dirp;             /* parent directory of file */
struct inode *rip;              /* inode of file, may be NIL_INODE too. */
char file_name[NAME_MAX];       /* name of file to be removed */
{
/* Unlink 'file_name'; rip must be the inode of 'file_name' or NIL_INODE. */

  ino_t numb;                   /* inode number */
  int   r;

  /* If rip is not NIL_INODE, it is used to get faster access to the inode. */
  if (rip == NIL_INODE) {
        /* Search for file in directory and try to get its inode. */
        err_code = search_dir(dirp, file_name, &numb, LOOK_UP);
        if (err_code == OK) rip = get_inode(dirp->i_dev, (int) numb);
        if (err_code != OK || rip == NIL_INODE) return(err_code);
  } else {
        dup_inode(rip);         /* inode will be returned with put_inode */
  }

  r = search_dir(dirp, file_name, (ino_t *) 0, DELETE);

  if (r == OK) {
        rip->i_nlinks--;        /* entry deleted from parent's dir */
        rip->i_update |= CTIME;
        rip->i_dirt = DIRTY;
  }

  put_inode(rip);
  return(r);
}
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                              servers/fs/stadir.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/* This file contains the code for performing four system calls relating to
 * status and directories.
 *
 * The entry points into this file are
 *   do_chdir:  perform the CHDIR system call
 *   do_chroot: perform the CHROOT system call
 *   do_stat:   perform the STAT system call
 *   do_fstat:  perform the FSTAT system call
 *   do_fstatfs: perform the FSTATFS system call
 */

#include "fs.h"
#include <sys/stat.h>
#include <sys/statfs.h>
#include <minix/com.h>
#include "file.h"
#include "fproc.h"
#include "inode.h"
#include "param.h"
#include "super.h"

FORWARD _PROTOTYPE( int change, (struct inode **iip, char *name_ptr, int len));
FORWARD _PROTOTYPE( int change_into, (struct inode **iip, struct inode *ip));
FORWARD _PROTOTYPE( int stat_inode, (struct inode *rip, struct filp *fil_ptr,
                        char *user_addr)                                );

/*===========================================================================*
 *                              do_fchdir                                    *
 *===========================================================================*/
PUBLIC int do_fchdir()
{
        /* Change directory on already-opened fd. */
        struct filp *rfilp;

        /* Is the file descriptor valid? */
        if ( (rfilp = get_filp(m_in.fd)) == NIL_FILP) return(err_code);
        return change_into(&fp->fp_workdir, rfilp->filp_ino);
}

/*===========================================================================*
 *                              do_chdir                                     *
 *===========================================================================*/
PUBLIC int do_chdir()
{
/* Change directory.  This function is  also called by MM to simulate a chdir
 * in order to do EXEC, etc.  It also changes the root directory, the uids and
 * gids, and the umask.
 */

  int r;
  register struct fproc *rfp;

  if (who == PM_PROC_NR) {
        rfp = &fproc[m_in.slot1];
        put_inode(fp->fp_rootdir);
        dup_inode(fp->fp_rootdir = rfp->fp_rootdir);
        put_inode(fp->fp_workdir);
        dup_inode(fp->fp_workdir = rfp->fp_workdir);

        /* MM uses access() to check permissions.  To make this work, pretend
         * that the user's real ids are the same as the user's effective ids.
         * FS calls other than access() do not use the real ids, so are not
         * affected.
         */
        fp->fp_realuid =
        fp->fp_effuid = rfp->fp_effuid;
        fp->fp_realgid =
        fp->fp_effgid = rfp->fp_effgid;
        fp->fp_umask = rfp->fp_umask;
        return(OK);
  }

  /* Perform the chdir(name) system call. */
  r = change(&fp->fp_workdir, m_in.name, m_in.name_length);
  return(r);
}

/*===========================================================================*
 *                              do_chroot                                    *
 *===========================================================================*/
PUBLIC int do_chroot()
{
/* Perform the chroot(name) system call. */

  register int r;

  if (!super_user) return(EPERM);       /* only su may chroot() */
  r = change(&fp->fp_rootdir, m_in.name, m_in.name_length);
  return(r);
}
```

```
/*===========================================================================*
 *                              change                                       *
 *===========================================================================*/
PRIVATE int change(iip, name_ptr, len)
struct inode **iip;             /* pointer to the inode pointer for the dir */
char *name_ptr;                 /* pointer to the directory name to change to */
int len;                        /* length of the directory name string */
{
/* Do the actual work for chdir() and chroot(). */
  struct inode *rip;

  /* Try to open the new directory. */
  if (fetch_name(name_ptr, len, M3) != OK) return(err_code);
  if ( (rip = eat_path(user_path)) == NIL_INODE) return(err_code);
  return change_into(iip, rip);
}

/*===========================================================================*
 *                              change_into                                  *
 *===========================================================================*/
PRIVATE int change_into(iip, rip)
struct inode **iip;             /* pointer to the inode pointer for the dir */
struct inode *rip;              /* this is what the inode has to become */
{
  register int r;

  /* It must be a directory and also be searchable. */
  if ( (rip->i_mode & I_TYPE) != I_DIRECTORY)
        r = ENOTDIR;
  else
        r = forbidden(rip, X_BIT);      /* check if dir is searchable */

  /* If error, return inode. */
  if (r != OK) {
        put_inode(rip);
        return(r);
  }

  /* Everything is OK.  Make the change. */
  put_inode(*iip);              /* release the old directory */
  *iip = rip;                   /* acquire the new one */
  return(OK);
}

/*===========================================================================*
 *                              do_stat                                      *
 *===========================================================================*/
PUBLIC int do_stat()
{
/* Perform the stat(name, buf) system call. */

  register struct inode *rip;
  register int r;

  /* Both stat() and fstat() use the same routine to do the real work.  That
   * routine expects an inode, so acquire it temporarily.
   */
  if (fetch_name(m_in.name1, m_in.name1_length, M1) != OK) return(err_code);
  if ( (rip = eat_path(user_path)) == NIL_INODE) return(err_code);
  r = stat_inode(rip, NIL_FILP, m_in.name2);    /* actually do the work.*/
  put_inode(rip);               /* release the inode */
  return(r);
}

/*===========================================================================*
 *                              do_fstat                                     *
 *===========================================================================*/
PUBLIC int do_fstat()
{
/* Perform the fstat(fd, buf) system call. */

  register struct filp *rfilp;

  /* Is the file descriptor valid? */
  if ( (rfilp = get_filp(m_in.fd)) == NIL_FILP) return(err_code);

  return(stat_inode(rfilp->filp_ino, rfilp, m_in.buffer));
}

/*===========================================================================*
 *                              stat_inode                                   *
 *===========================================================================*/
PRIVATE int stat_inode(rip, fil_ptr, user_addr)
register struct inode *rip;     /* pointer to inode to stat */
struct filp *fil_ptr;           /* filp pointer, supplied by 'fstat' */
char *user_addr;                /* user space address where stat buf goes */
{
/* Common code for stat and fstat system calls. */

  struct stat statbuf;
  mode_t mo;
  int r, s;

  /* Update the atime, ctime, and mtime fields in the inode, if need be. */
  if (rip->i_update) update_times(rip);

  /* Fill in the statbuf struct. */
  mo = rip->i_mode & I_TYPE;

  /* true iff special */
  s = (mo == I_CHAR_SPECIAL || mo == I_BLOCK_SPECIAL);

  statbuf.st_dev = rip->i_dev;
  statbuf.st_ino = rip->i_num;
  statbuf.st_mode = rip->i_mode;
  statbuf.st_nlink = rip->i_nlinks;
  statbuf.st_uid = rip->i_uid;
  statbuf.st_gid = rip->i_gid;
  statbuf.st_rdev = (dev_t) (s ? rip->i_zone[0] : NO_DEV);
  statbuf.st_size = rip->i_size;

  if (rip->i_pipe == I_PIPE) {
        statbuf.st_mode &= ~I_REGULAR;  /* wipe out I_REGULAR bit for pipes */
        if (fil_ptr != NIL_FILP && fil_ptr->filp_mode & R_BIT)
                statbuf.st_size -= fil_ptr->filp_pos;
  }

  statbuf.st_atime = rip->i_atime;
  statbuf.st_mtime = rip->i_mtime;
  statbuf.st_ctime = rip->i_ctime;

  /* Copy the struct to user space. */
  r = sys_datacopy(FS_PROC_NR, (vir_bytes) &statbuf,
              who, (vir_bytes) user_addr, (phys_bytes) sizeof(statbuf));
  return(r);
}

/*===========================================================================*
 *                              do_fstatfs                                   *
 *===========================================================================*/
PUBLIC int do_fstatfs()
{
  /* Perform the fstatfs(fd, buf) system call. */
  struct statfs st;
  register struct filp *rfilp;
  int r;

  /* Is the file descriptor valid? */
  if ( (rfilp = get_filp(m_in.fd)) == NIL_FILP) return(err_code);

  st.f_bsize = rfilp->filp_ino->i_sp->s_block_size;

  r = sys_datacopy(FS_PROC_NR, (vir_bytes) &st,
              who, (vir_bytes) m_in.buffer, (phys_bytes) sizeof(st));

   return(r);
}
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                              servers/fs/protect.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/* This file deals with protection in the file system.  It contains the code
 * for four system calls that relate to protection.
 *
 * The entry points into this file are
 *   do_chmod:  perform the CHMOD system call
 *   do_chown:  perform the CHOWN system call
 *   do_umask:  perform the UMASK system call
 *   do_access: perform the ACCESS system call
 *   forbidden: check to see if a given access is allowed on a given inode
 */

#include "fs.h"
#include <unistd.h>
#include <minix/callnr.h>
#include "buf.h"
#include "file.h"
#include "fproc.h"
#include "inode.h"
#include "param.h"
#include "super.h"

```

```
/*===========================================================================*
 *                              do_chmod                                     *
 *===========================================================================*/
PUBLIC int do_chmod()
{
/* Perform the chmod(name, mode) system call. */

  register struct inode *rip;
  register int r;

  /* Temporarily open the file. */
  if (fetch_name(m_in.name, m_in.name_length, M3) != OK) return(err_code);
  if ( (rip = eat_path(user_path)) == NIL_INODE) return(err_code);

  /* Only the owner or the super_user may change the mode of a file.
   * No one may change the mode of a file on a read-only file system.
   */
  if (rip->i_uid != fp->fp_effuid && !super_user)
        r = EPERM;
  else
        r = read_only(rip);

  /* If error, return inode. */
  if (r != OK)  {
        put_inode(rip);
        return(r);
  }

  /* Now make the change. Clear setgid bit if file is not in caller's grp */
  rip->i_mode = (rip->i_mode & ~ALL_MODES) | (m_in.mode & ALL_MODES);
  if (!super_user && rip->i_gid != fp->fp_effgid)rip->i_mode &= ~I_SET_GID_BIT;
  rip->i_update |= CTIME;
  rip->i_dirt = DIRTY;

  put_inode(rip);
  return(OK);
}

/*===========================================================================*
 *                              do_chown                                     *
 *===========================================================================*/
PUBLIC int do_chown()
{
/* Perform the chown(name, owner, group) system call. */

  register struct inode *rip;
  register int r;

  /* Temporarily open the file. */
  if (fetch_name(m_in.name1, m_in.name1_length, M1) != OK) return(err_code);
  if ( (rip = eat_path(user_path)) == NIL_INODE) return(err_code);

  /* Not permitted to change the owner of a file on a read-only file sys. */
  r = read_only(rip);
  if (r == OK) {
        /* FS is R/W.  Whether call is allowed depends on ownership, etc. */
        if (super_user) {
                /* The super user can do anything. */
                rip->i_uid = m_in.owner;        /* others later */
        } else {
                /* Regular users can only change groups of their own files. */
                if (rip->i_uid != fp->fp_effuid) r = EPERM;
                if (rip->i_uid != m_in.owner) r = EPERM;  /* no giving away */
                if (fp->fp_effgid != m_in.group) r = EPERM;
        }
  }
  if (r == OK) {
        rip->i_gid = m_in.group;
        rip->i_mode &= ~(I_SET_UID_BIT | I_SET_GID_BIT);
        rip->i_update |= CTIME;
        rip->i_dirt = DIRTY;
  }

  put_inode(rip);
  return(r);
}

/*===========================================================================*
 *                              do_umask                                     *
 *===========================================================================*/
PUBLIC int do_umask()
{
/* Perform the umask(co_mode) system call. */
  register mode_t r;

  r = ~fp->fp_umask;            /* set 'r' to complement of old mask */
  fp->fp_umask = ~(m_in.co_mode & RWX_MODES);
  return(r);                    /* return complement of old mask */
}

/*===========================================================================*
 *                              do_access                                    *
 *===========================================================================*/
PUBLIC int do_access()
{
/* Perform the access(name, mode) system call. */

  struct inode *rip;
  register int r;

  /* First check to see if the mode is correct. */
  if ( (m_in.mode & ~(R_OK | W_OK | X_OK)) != 0 && m_in.mode != F_OK)
        return(EINVAL);

  /* Temporarily open the file whose access is to be checked. */
  if (fetch_name(m_in.name, m_in.name_length, M3) != OK) return(err_code);
  if ( (rip = eat_path(user_path)) == NIL_INODE) return(err_code);

  /* Now check the permissions. */
  r = forbidden(rip, (mode_t) m_in.mode);
  put_inode(rip);
  return(r);
}

/*===========================================================================*
 *                              forbidden                                    *
 *===========================================================================*/
PUBLIC int forbidden(register struct inode *rip, mode_t access_desired)
{
/* Given a pointer to an inode, 'rip', and the access desired, determine
```

```
 * if the access is allowed, and if not why not.  The routine looks up the
 * caller's uid in the 'fproc' table.  If access is allowed, OK is returned
 * if it is forbidden, EACCES is returned.
 */

  register struct inode *old_rip = rip;
  register struct super_block *sp;
  register mode_t bits, perm_bits;
  int r, shift, test_uid, test_gid, type;

  if (rip->i_mount == I_MOUNT)  /* The inode is mounted on. */
        for (sp = &super_block[1]; sp < &super_block[NR_SUPERS]; sp++)
                if (sp->s_imount == rip) {
                        rip = get_inode(sp->s_dev, ROOT_INODE);
                        break;
                } /* if */

  /* Isolate the relevant rwx bits from the mode. */
  bits = rip->i_mode;
  test_uid = (call_nr == ACCESS ? fp->fp_realuid : fp->fp_effuid);
  test_gid = (call_nr == ACCESS ? fp->fp_realgid : fp->fp_effgid);
  if (test_uid == SU_UID) {
        /* Grant read and write permission.  Grant search permission for
         * directories.  Grant execute permission (for non-directories) if
         * and only if one of the 'X' bits is set.
         */
        if ( (bits & I_TYPE) == I_DIRECTORY ||
             bits & ((X_BIT << 6) | (X_BIT << 3) | X_BIT))
                perm_bits = R_BIT | W_BIT | X_BIT;
        else
                perm_bits = R_BIT | W_BIT;
  } else {
        if (test_uid == rip->i_uid) shift = 6;          /* owner */
        else if (test_gid == rip->i_gid ) shift = 3;    /* group */
        else shift = 0;                                 /* other */
        perm_bits = (bits >> shift) & (R_BIT | W_BIT | X_BIT);
  }

  /* If access desired is not a subset of what is allowed, it is refused. */
  r = OK;
  if ((perm_bits | access_desired) != perm_bits) r = EACCES;

  /* Check to see if someone is trying to write on a file system that is
   * mounted read-only.
   */
  type = rip->i_mode & I_TYPE;
  if (r == OK)
        if (access_desired & W_BIT)
                r = read_only(rip);

  if (rip != old_rip) put_inode(rip);

  return(r);
}

/*===========================================================================*
 *                              read_only                                    *
 *===========================================================================*/
PUBLIC int read_only(ip)
struct inode *ip;               /* ptr to inode whose file sys is to be cked */
```

```
{
/* Check to see if the file system on which the inode 'ip' resides is mounted
 * read only.  If so, return EROFS, else return OK.
 */

  register struct super_block *sp;

  sp = ip->i_sp;
  return(sp->s_rd_only ? EROFS : OK);
}
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                      servers/fs/dmap.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/* This file contains the table with device <-> driver mappings. It also
 * contains some routines to dynamically add and/ or remove device drivers
 * or change mappings.
 */

#include "fs.h"
#include "fproc.h"
#include <string.h>
#include <stdlib.h>
#include <ctype.h>
#include <unistd.h>
#include <minix/com.h>
#include "param.h"

/* Some devices may or may not be there in the next table. */
#define DT(enable, opcl, io, driver, flags) \
  { (enable?(opcl):no_dev), (enable?(io):0), \
        (enable?(driver):0), (flags) },
#define NC(x) (NR_CTRLRS >= (x))

/* The order of the entries here determines the mapping between major device
 * numbers and tasks.  The first entry (major device 0) is not used.  The
 * next entry is major device 1, etc.  Character and block devices can be
 * intermixed at random.  The ordering determines the device numbers in /dev/.
 * Note that FS knows the device number of /dev/ram/ to load the RAM disk.
 * Also note that the major device numbers used in /dev/ are NOT the same as
 * the process numbers of the device drivers.
 */
/*
  Driver enabled     Open/Cls  I/O     Driver #     Flags Device  File
  --------------     --------  ------  -----------  ----- ------  ----
 */
struct dmap dmap[NR_DEVICES];                           /* actual map */
PRIVATE struct dmap init_dmap[] = {
  DT(1, no_dev,   0,       0,           0)              /* 0 = not used   */
  DT(1, gen_opcl, gen_io,  MEM_PROC_NR, 0)              /* 1 = /dev/mem   */
  DT(0, no_dev,   0,       0,           DMAP_MUTABLE)   /* 2 = /dev/fd0   */
  DT(0, no_dev,   0,       0,           DMAP_MUTABLE)   /* 3 = /dev/c0    */
  DT(1, tty_opcl, gen_io,  TTY_PROC_NR, 0)              /* 4 = /dev/tty00 */
  DT(1, ctty_opcl,ctty_io, TTY_PROC_NR, 0)              /* 5 = /dev/tty   */
  DT(0, no_dev,   0,       NONE,        DMAP_MUTABLE)   /* 6 = /dev/lp    */
  DT(1, no_dev,   0,       0,           DMAP_MUTABLE)   /* 7 = /dev/ip    */
  DT(0, no_dev,   0,       NONE,        DMAP_MUTABLE)   /* 8 = /dev/c1    */
  DT(0, 0,        0,       0,           DMAP_MUTABLE)   /* 9 = not used   */
  DT(0, no_dev,   0,       0,           DMAP_MUTABLE)   /*10 = /dev/c2    */
  DT(0, 0,        0,       0,           DMAP_MUTABLE)   /*11 = not used   */
  DT(0, no_dev,   0,       NONE,        DMAP_MUTABLE)   /*12 = /dev/c3    */
  DT(0, no_dev,   0,       NONE,        DMAP_MUTABLE)   /*13 = /dev/audio */
  DT(0, no_dev,   0,       NONE,        DMAP_MUTABLE)   /*14 = /dev/mixer */
  DT(1, gen_opcl, gen_io,  LOG_PROC_NR, 0)              /*15 = /dev/klog  */
  DT(0, no_dev,   0,       NONE,        DMAP_MUTABLE)   /*16 = /dev/random*/
  DT(0, no_dev,   0,       NONE,        DMAP_MUTABLE)   /*17 = /dev/cmos  */
};

/*===========================================================================*
 *                              do_devctl                                    *
 *===========================================================================*/
PUBLIC int do_devctl()
{
  int result;

  switch(m_in.ctl_req) {
  case DEV_MAP:
      /* Try to update device mapping. */
      result = map_driver(m_in.dev_nr, m_in.driver_nr, m_in.dev_style);
      break;
  case DEV_UNMAP:
      result = ENOSYS;
      break;
  default:
      result = EINVAL;
  }
  return(result);
}

/*===========================================================================*
 *                              map_driver                                   *
 *===========================================================================*/
PUBLIC int map_driver(major, proc_nr, style)
int major;                      /* major number of the device */
int proc_nr;                    /* process number of the driver */
int style;                      /* style of the device */
{
/* Set a new device driver mapping in the dmap table. Given that correct
 * arguments are given, this only works if the entry is mutable and the
 * current driver is not busy.
 * Normal error codes are returned so that this function can be used from
 * a system call that tries to dynamically install a new driver.
 */
  struct dmap *dp;

  /* Get pointer to device entry in the dmap table. */
  if (major >= NR_DEVICES) return(ENODEV);
  dp = &dmap[major];

  /* See if updating the entry is allowed. */
  if (! (dp->dmap_flags & DMAP_MUTABLE))  return(EPERM);
  if (dp->dmap_flags & DMAP_BUSY)  return(EBUSY);

  /* Check process number of new driver. */
  if (! isokprocnr(proc_nr))  return(EINVAL);

  /* Try to update the entry. */
  switch (style) {
  case STYLE_DEV:       dp->dmap_opcl = gen_opcl;       break;
  case STYLE_TTY:       dp->dmap_opcl = tty_opcl;       break;
  case STYLE_CLONE:     dp->dmap_opcl = clone_opcl;     break;
  default:              return(EINVAL);
  }
  dp->dmap_io = gen_io;
  dp->dmap_driver = proc_nr;
  return(OK);
}

/*===========================================================================*
 *                              build_dmap                                   *
 *===========================================================================*/
PUBLIC void build_dmap()
{
/* Initialize the table with all device <-> driver mappings. Then, map
 * the boot driver to a controller and update the dmap table to that
 * selection. The boot driver and the controller it handles are set at
 * the boot monitor.
 */
  char driver[16];
  char *controller = "c##";
  int nr, major = -1;
  int i,s;
  struct dmap *dp;

  /* Build table with device <-> driver mappings. */
  for (i=0; i<NR_DEVICES; i++) {
      dp = &dmap[i];
      if (i < sizeof(init_dmap)/sizeof(struct dmap) &&
              init_dmap[i].dmap_opcl != no_dev) {       /* a preset driver */
          dp->dmap_opcl = init_dmap[i].dmap_opcl;
          dp->dmap_io = init_dmap[i].dmap_io;
          dp->dmap_driver = init_dmap[i].dmap_driver;
          dp->dmap_flags = init_dmap[i].dmap_flags;
      } else {                                          /* no default */
          dp->dmap_opcl = no_dev;
          dp->dmap_io = 0;
          dp->dmap_driver = 0;
          dp->dmap_flags = DMAP_MUTABLE;
      }
  }

  /* Get settings of 'controller' and 'driver' at the boot monitor. */
  if ((s = env_get_param("label", driver, sizeof(driver))) != OK)
      panic(__FILE__,"couldn't get boot monitor parameter 'driver'", s);
  if ((s = env_get_param("controller", controller, sizeof(controller))) != OK)
      panic(__FILE__,"couldn't get boot monitor parameter 'controller'", s);

  /* Determine major number to map driver onto. */
  if (controller[0] == 'f' && controller[1] == 'd') {
      major = FLOPPY_MAJOR;
  }
  else if (controller[0] == 'c' && isdigit(controller[1])) {
      if ((nr = (unsigned) atoi(&controller[1])) > NR_CTRLRS)
          panic(__FILE__,"monitor 'controller' maximum 'c#' is", NR_CTRLRS);
      major = CTRLR(nr);
  }
  else {
      panic(__FILE__,"monitor 'controller' syntax is 'c#' of 'fd'", NO_NUM);
  }

  /* Now try to set the actual mapping and report to the user. */
  if ((s=map_driver(major, DRVR_PROC_NR, STYLE_DEV)) != OK)
      panic(__FILE__,"map_driver failed",s);
  printf("Boot medium driver: %s driver mapped onto controller %s.\n",
      driver, controller);
}
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                    servers/fs/device.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/* When a needed block is not in the cache, it must be fetched from the disk.
 * Special character files also require I/O.  The routines for these are here.
 *
 * The entry points in this file are:
 *   dev_open:   FS opens a device
 *   dev_close:  FS closes a device
 *   dev_io:     FS does a read or write on a device
 *   dev_status: FS processes callback request alert
 *   gen_opcl:   generic call to a task to perform an open/close
 *   gen_io:     generic call to a task to perform an I/O operation
 *   no_dev:     open/close processing for devices that don’t exist
 *   tty_opcl:   perform tty-specific processing for open/close
 *   ctty_opcl:  perform controlling-tty-specific processing for open/close
 *   ctty_io:    perform controlling-tty-specific processing for I/O
 *   do_ioctl:   perform the IOCTL system call
 *   do_setsid:  perform the SETSID system call (FS side)
 */

#include "fs.h"
#include <fcntl.h>
#include <minix/callnr.h>
#include <minix/com.h>
#include "file.h"
#include "fproc.h"
#include "inode.h"
#include "param.h"

#define ELEMENTS(a) (sizeof(a)/sizeof((a)[0]))

extern int dmap_size;

/*===========================================================================*
 *                              dev_open                                     *
 *===========================================================================*/
PUBLIC int dev_open(dev, proc, flags)
dev_t dev;                      /* device to open */
int proc;                       /* process to open for */
int flags;                      /* mode bits and flags */
{
  int major, r;
  struct dmap *dp;

  /* Determine the major device number call the device class specific
   * open/close routine.  (This is the only routine that must check the
   * device number for being in range.  All others can trust this check.)
   */
  major = (dev >> MAJOR) & BYTE;
  if (major >= NR_DEVICES) major = 0;
  dp = &dmap[major];
  r = (*dp->dmap_opcl)(DEV_OPEN, dev, proc, flags);
  if (r == SUSPEND) panic(__FILE__,"suspend on open from", dp->dmap_driver);
  return(r);
}

/*===========================================================================*
 *                              dev_close                                    *
 *===========================================================================*/
PUBLIC void dev_close(dev)
dev_t dev;                      /* device to close */
{
  (void) (*dmap[(dev >> MAJOR) & BYTE].dmap_opcl)(DEV_CLOSE, dev, 0, 0);
}

/*===========================================================================*
 *                              dev_status                                      *
 *===========================================================================*/
PUBLIC void dev_status(message *m)
{
        message st;
        int d, get_more = 1;

        for(d = 0; d < NR_DEVICES; d++)
                if (dmap[d].dmap_driver == m->m_source)
                        break;

        if (d >= NR_DEVICES)
                return;

        do {
                int r;
                st.m_type = DEV_STATUS;
                if ((r=sendrec(m->m_source, &st)) != OK)
                        panic(__FILE__,"couldn't sendrec for DEV_STATUS", r);

                switch(st.m_type) {
                        case DEV_REVIVE:
                                revive(st.REP_PROC_NR, st.REP_STATUS);
                                break;
                        case DEV_IO_READY:
                                select_notified(d, st.DEV_MINOR, st.DEV_SEL_OPS);
                                break;
                        default:
                                printf("FS: unrecognized reply %d to DEV_STATUS\n", st.m_type);
                                /* Fall through. */
                        case DEV_NO_STATUS:
                                get_more = 0;
                                break;
                }
        } while(get_more);

        return;
}
```

```
/*===========================================================================*
 *                              dev_io                                       *
 *===========================================================================*/
PUBLIC int dev_io(op, dev, proc, buf, pos, bytes, flags)
int op;                         /* DEV_READ, DEV_WRITE, DEV_IOCTL, etc. */
dev_t dev;                      /* major-minor device number */
int proc;                       /* in whose address space is buf? */
void *buf;                      /* virtual address of the buffer */
off_t pos;                      /* byte position */
int bytes;                      /* how many bytes to transfer */
int flags;                      /* special flags, like O_NONBLOCK */
{
/* Read or write from a device.  The parameter 'dev' tells which one. */
  struct dmap *dp;
  message dev_mess;

  /* Determine task dmap. */
  dp = &dmap[(dev >> MAJOR) & BYTE];

  /* Set up the message passed to task. */
  dev_mess.m_type   = op;
  dev_mess.DEVICE   = (dev >> MINOR) & BYTE;
  dev_mess.POSITION = pos;
  dev_mess.PROC_NR  = proc;
  dev_mess.ADDRESS  = buf;
  dev_mess.COUNT    = bytes;
  dev_mess.TTY_FLAGS = flags;

  /* Call the task. */
  (*dp->dmap_io)(dp->dmap_driver, &dev_mess);

  /* Task has completed.  See if call completed. */
  if (dev_mess.REP_STATUS == SUSPEND) {
        if (flags & O_NONBLOCK) {
                /* Not supposed to block. */
                dev_mess.m_type = CANCEL;
                dev_mess.PROC_NR = proc;
                dev_mess.DEVICE = (dev >> MINOR) & BYTE;
                (*dp->dmap_io)(dp->dmap_driver, &dev_mess);
                if (dev_mess.REP_STATUS == EINTR) dev_mess.REP_STATUS = EAGAIN;
        } else {
                /* Suspend user. */
                suspend(dp->dmap_driver);
                return(SUSPEND);
        }
  }
  return(dev_mess.REP_STATUS);
}

/*===========================================================================*
 *                              gen_opcl                                     *
 *===========================================================================*/
PUBLIC int gen_opcl(op, dev, proc, flags)
int op;                         /* operation, DEV_OPEN or DEV_CLOSE */
dev_t dev;                      /* device to open or close */
int proc;                       /* process to open/close for */
int flags;                      /* mode bits and flags */
{
/* Called from the dmap struct in table.c on opens & closes of special files.*/
  struct dmap *dp;
  message dev_mess;

  /* Determine task dmap. */
  dp = &dmap[(dev >> MAJOR) & BYTE];

  dev_mess.m_type   = op;
  dev_mess.DEVICE   = (dev >> MINOR) & BYTE;
  dev_mess.PROC_NR  = proc;
  dev_mess.COUNT    = flags;

  /* Call the task. */
  (*dp->dmap_io)(dp->dmap_driver, &dev_mess);

  return(dev_mess.REP_STATUS);
}

/*===========================================================================*
 *                              tty_opcl                                     *
 *===========================================================================*/
PUBLIC int tty_opcl(op, dev, proc, flags)
int op;                         /* operation, DEV_OPEN or DEV_CLOSE */
dev_t dev;                      /* device to open or close */
int proc;                       /* process to open/close for */
int flags;                      /* mode bits and flags */
{
/* This procedure is called from the dmap struct on tty open/close. */

  int r;
  register struct fproc *rfp;

  /* Add O_NOCTTY to the flags if this process is not a session leader, or
   * if it already has a controlling tty, or if it is someone elses
   * controlling tty.
   */
  if (!fp->fp_sesldr || fp->fp_tty != 0) {
        flags |= O_NOCTTY;
  } else {
        for (rfp = &fproc[0]; rfp < &fproc[NR_PROCS]; rfp++) {
                if (rfp->fp_tty == dev) flags |= O_NOCTTY;
        }
  }

  r = gen_opcl(op, dev, proc, flags);

  /* Did this call make the tty the controlling tty? */
  if (r == 1) {
        fp->fp_tty = dev;
        r = OK;
  }
  return(r);
}

/*===========================================================================*
 *                              ctty_opcl                                    *
 *===========================================================================*/
PUBLIC int ctty_opcl(op, dev, proc, flags)
int op;                         /* operation, DEV_OPEN or DEV_CLOSE */
dev_t dev;                      /* device to open or close */
int proc;                       /* process to open/close for */
int flags;                      /* mode bits and flags */
```

```
{
/* This procedure is called from the dmap struct in table.c on opening/closing
 * /dev/tty, the magic device that translates to the controlling tty.
 */

  return(fp->fp_tty == 0 ? ENXIO : OK);
}

/*===========================================================================*
 *                              do_setsid                                    *
 *===========================================================================*/
PUBLIC int do_setsid()
{
/* Perform the FS side of the SETSID call, i.e. get rid of the controlling
 * terminal of a process, and make the process a session leader.
 */
  register struct fproc *rfp;

  /* Only MM may do the SETSID call directly. */
  if (who != PM_PROC_NR) return(ENOSYS);

  /* Make the process a session leader with no controlling tty. */
  rfp = &fproc[m_in.slot1];
  rfp->fp_sesldr = TRUE;
  rfp->fp_tty = 0;
  return(OK);
}

/*===========================================================================*
 *                              do_ioctl                                     *
 *===========================================================================*/
PUBLIC int do_ioctl()
{
/* Perform the ioctl(ls_fd, request, argx) system call (uses m2 fmt). */

  struct filp *f;
  register struct inode *rip;
  dev_t dev;

  if ( (f = get_filp(m_in.ls_fd)) == NIL_FILP) return(err_code);
  rip = f->filp_ino;            /* get inode pointer */
  if ( (rip->i_mode & I_TYPE) != I_CHAR_SPECIAL
        && (rip->i_mode & I_TYPE) != I_BLOCK_SPECIAL) return(ENOTTY);
  dev = (dev_t) rip->i_zone[0];

  return(dev_io(DEV_IOCTL, dev, who, m_in.ADDRESS, 0L,
        m_in.REQUEST, f->filp_flags));
}

/*===========================================================================*
 *                              gen_io                                       *
 *===========================================================================*/
PUBLIC void gen_io(task_nr, mess_ptr)
int task_nr;                    /* which task to call */
message *mess_ptr;              /* pointer to message for task */
{
/* All file system I/O ultimately comes down to I/O on major/minor device
 * pairs.  These lead to calls on the following routines via the dmap table.
 */

```

```
  int r, proc_nr;
  message local_m;

  proc_nr = mess_ptr->PROC_NR;
  if (! isokprocnr(proc_nr)) {
      printf("FS: warning, got illegal process number (%d) from %d\n",
          mess_ptr->PROC_NR, mess_ptr->m_source);
      return;
  }

  while ((r = sendrec(task_nr, mess_ptr)) == ELOCKED) {
        /* sendrec() failed to avoid deadlock. The task 'task_nr' is
         * trying to send a REVIVE message for an earlier request.
         * Handle it and go try again.
         */
        if ((r = receive(task_nr, &local_m)) != OK) {
                break;
        }

        /* If we're trying to send a cancel message to a task which has just
         * sent a completion reply, ignore the reply and abort the cancel
         * request. The caller will do the revive for the process.
         */
        if (mess_ptr->m_type == CANCEL && local_m.REP_PROC_NR == proc_nr) {
                return;
        }

        /* Otherwise it should be a REVIVE. */
        if (local_m.m_type != REVIVE) {
                printf(
                "fs: strange device reply from %d, type = %d, proc = %d (1)\n",
                        local_m.m_source,
                        local_m.m_type, local_m.REP_PROC_NR);
                continue;
        }

        revive(local_m.REP_PROC_NR, local_m.REP_STATUS);
  }

  /* The message received may be a reply to this call, or a REVIVE for some
   * other process.
   */
  for (;;) {
        if (r != OK) {
                if (r == EDEADDST) return;      /* give up */
                else panic(__FILE__,"call_task: can't send/receive", r);
        }

        /* Did the process we did the sendrec() for get a result? */
        if (mess_ptr->REP_PROC_NR == proc_nr) {
                break;
        } else if (mess_ptr->m_type == REVIVE) {
                /* Otherwise it should be a REVIVE. */
                revive(mess_ptr->REP_PROC_NR, mess_ptr->REP_STATUS);
        } else {
                printf(
                "fs: strange device reply from %d, type = %d, proc = %d (2)\n",
                        mess_ptr->m_source,
                        mess_ptr->m_type, mess_ptr->REP_PROC_NR);
                return;
        }

        r = receive(task_nr, mess_ptr);
  }
}

/*===========================================================================*
 *                              ctty_io                                      *
 *===========================================================================*/
PUBLIC void ctty_io(task_nr, mess_ptr)
int task_nr;                    /* not used - for compatibility with dmap_t */
message *mess_ptr;              /* pointer to message for task */
{
/* This routine is only called for one device, namely /dev/tty.  Its job
 * is to change the message to use the controlling terminal, instead of the
 * major/minor pair for /dev/tty itself.
 */

  struct dmap *dp;

  if (fp->fp_tty == 0) {
        /* No controlling tty present anymore, return an I/O error. */
        mess_ptr->REP_STATUS = EIO;
  } else {
        /* Substitute the controlling terminal device. */
        dp = &dmap[(fp->fp_tty >> MAJOR) & BYTE];
        mess_ptr->DEVICE = (fp->fp_tty >> MINOR) & BYTE;
        (*dp->dmap_io)(dp->dmap_driver, mess_ptr);
  }
}

/*===========================================================================*
 *                              no_dev                                       *
 *===========================================================================*/
PUBLIC int no_dev(op, dev, proc, flags)
int op;                         /* operation, DEV_OPEN or DEV_CLOSE */
dev_t dev;                      /* device to open or close */
int proc;                       /* process to open/close for */
int flags;                      /* mode bits and flags */
{
/* Called when opening a nonexistent device. */

  return(ENODEV);
}

/*===========================================================================*
 *                              clone_opcl                                   *
 *===========================================================================*/
PUBLIC int clone_opcl(op, dev, proc, flags)
int op;                         /* operation, DEV_OPEN or DEV_CLOSE */
dev_t dev;                      /* device to open or close */
int proc;                       /* process to open/close for */
int flags;                      /* mode bits and flags */
{
/* Some devices need special processing upon open.  Such a device is "cloned",
 * i.e. on a succesful open it is replaced by a new device with a new unique
 * minor device number.  This new device number identifies a new object (such
 * as a new network connection) that has been allocated within a task.
 */
  struct dmap *dp;
  int minor;
  message dev_mess;

  /* Determine task dmap. */
  dp = &dmap[(dev >> MAJOR) & BYTE];
  minor = (dev >> MINOR) & BYTE;

  dev_mess.m_type   = op;
  dev_mess.DEVICE   = minor;
  dev_mess.PROC_NR  = proc;
  dev_mess.COUNT    = flags;

  /* Call the task. */
  (*dp->dmap_io)(dp->dmap_driver, &dev_mess);

  if (op == DEV_OPEN && dev_mess.REP_STATUS >= 0) {
        if (dev_mess.REP_STATUS != minor) {
                /* A new minor device number has been returned.  Create a
                 * temporary device file to hold it.
                 */
                struct inode *ip;

                /* Device number of the new device. */
                dev = (dev & ~(BYTE << MINOR)) | (dev_mess.REP_STATUS << MINOR);

                ip = alloc_inode(root_dev, ALL_MODES | I_CHAR_SPECIAL);
                if (ip == NIL_INODE) {
                        /* Oops, that didn't work.  Undo open. */
                        (void) clone_opcl(DEV_CLOSE, dev, proc, 0);
                        return(err_code);
                }
                ip->i_zone[0] = dev;

                put_inode(fp->fp_filp[m_in.fd]->filp_ino);
                fp->fp_filp[m_in.fd]->filp_ino = ip;
        }
        dev_mess.REP_STATUS = OK;
  }
  return(dev_mess.REP_STATUS);
}
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                       servers/fs/time.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/* This file takes care of those system calls that deal with time.
 *
 * The entry points into this file are
 *   do_utime:          perform the UTIME system call
 *   do_stime:          PM informs FS about STIME system call
 */

#include "fs.h"
#include <minix/callnr.h>
#include <minix/com.h>
```

```
#include "file.h"
#include "fproc.h"
#include "inode.h"
#include "param.h"

/*===========================================================================*
 *                              do_utime                                     *
 *===========================================================================*/
PUBLIC int do_utime()
{
/* Perform the utime(name, timep) system call. */

  register struct inode *rip;
  register int len, r;

  /* Adjust for case of 'timep' being NULL;
   * utime_strlen then holds the actual size: strlen(name)+1.
   */
  len = m_in.utime_length;
  if (len == 0) len = m_in.utime_strlen;

  /* Temporarily open the file. */
  if (fetch_name(m_in.utime_file, len, M1) != OK) return(err_code);
  if ( (rip = eat_path(user_path)) == NIL_INODE) return(err_code);

  /* Only the owner of a file or the super_user can change its time. */
  r = OK;
  if (rip->i_uid != fp->fp_effuid && !super_user) r = EPERM;
  if (m_in.utime_length == 0 && r != OK) r = forbidden(rip, W_BIT);
  if (read_only(rip) != OK) r = EROFS;  /* not even su can touch if R/O */
  if (r == OK) {
        if (m_in.utime_length == 0) {
                rip->i_atime = clock_time();
                rip->i_mtime = rip->i_atime;
        } else {
                rip->i_atime = m_in.utime_actime;
                rip->i_mtime = m_in.utime_modtime;
        }
        rip->i_update = CTIME;  /* discard any stale ATIME and MTIME flags */
        rip->i_dirt = DIRTY;
  }

  put_inode(rip);
  return(r);
}

/*===========================================================================*
 *                              do_stime                                     *
 *===========================================================================*/
PUBLIC int do_stime()
{
/* Perform the stime(tp) system call. */
  boottime = (long) m_in.pm_stime;
  return(OK);
}
```