Finding hot spots in MINIX 3

Jens de Smit
jfdsmit@few.vu.nl

January 7, 2008

1 Abstract

Explains the use of MINIX 3 profiling tools and presents results obtained using these tools.

2 Introduction

MINIX 3 is a relatively new operating system, designed to be flexible, reliable and secure. Unlike its predecessors MINIX 1 and 2, it is not only intended as a teaching tool: it is also to be used as a serious operating system, especially in situations where security, reliability and/or resource limitations are concerned.

To aid in achieving this goal, a set of system profiling tools has been developed [Meurs]. The goal of these tools is to locate places inside MINIX' internals where a lot of time is spent (so-called "hot spots"). This paper explains the capabilities of the profiling tools, demonstrates the use of the statistical profiler and shows how it can be used to find and optimize a hot spot.

3 The profiling tools

There are two different kinds of profiling tools available for MINIX 3: a statistical profiler and a call path profiler. The statistical profiler is meant to obtain an image of the distribution of where in the system program cycles are spent. It does this by periodically checking, using the CMOS timer, which program is executing and to which instruction the program counter is pointing. This data is collected over a period of time specified by the user and, on the user's command, written to a data file. A separate program is available to combine the data in this data file with the binaries of the various parts of the MINIX system and determine which labels (i.e. functions) the samples correspond to. Using this knowledge the program then presents statistical data specifying approximately which functions of which system parts used how many CPU time.

The call path profiler is meant to collect more fine-grained details about program execution. Call path profiling utilizes a special feature of the ACK

(Amsterdam Compiler Kit) compiler that allows for registering function calls. For this to work, it is necessary for all programs that need to be profiled to be recompiled with this special feature enabled, as well as to make some changes to the internals of MINIX itself. The call path profiler can then generate complete statistics on the order of function calls within programs as well as the number of cycles spent in these paths and the number of times each path has occurred. All this administration however does place a burden on the machine causing a two- to threefold performance loss [Meurs]. The focus of the remainder of this paper will be on the statistical profiler.

4 Finding hot spots

4.1 Global hot spot discovery

Profiling is, as said before, meant to find hot spots in the system, places to optimize to increase performance. For hot spots to pop up, it is important to put a load on the system. A simple experiment is to profile the system during a complete recompilation of the system itself. This is done by issuing the command `make fresh hdboot` in the directory */usr/src/tools*. Compiling a large piece of code (such as MINIX) involves reading from and writing to disk, terminal interaction and process creation and destruction: plenty of system work.

Listing 1 (in appendix A) shows the results of such a run. First, a list is given of the "top consumers" of total system time, followed by a breakdown per system process. The results show that a lot of time is spent in labels such as *__send*, *__receiv*, *__sendre*, *_lock_se*, *_phys_co* and *fill_sta*. The first five functions all relate to message passing, the core of MINIX inter-process communication and an integral part of every system call. It is no surprise that these functions are listed as top consumers: they are called very often. The good news is that these are our first hot spots. The bad news is that these functions have been hot spots since MINIX 1 and are already highly optimized assembly routines. The work they do is kept to a bare minimum already. Unless hardware support is available to speed up these operations, there is not much to do to increase performance of message passing.

Fill_sta is a label in the assembly routine *_phys_memset*, which is used to set all bytes in a memory region to a specific value. It is generally used to set all bytes in a region to 0. This is done before memory is granted to a process that has just performed an *exec()* system call: programs in a multiprocessing operating system such as MINIX may not read memory from other programs or processes, so each program starts with a cleared memory area before execution is begun. Because MINIX 3 at present uses static memory allocation (programs receive a preconfigured amount of memory), all a program's memory needs to be cleared at program startup, even if it will not use all of that memory. Other operating systems that use a "paged" memory model do not have this problem because they allocate memory per page at request time instead of at execution

time. Apart from the fact that this routine is also an optimized assembly loop that is hard, if not impossible, to improve, this problem is expected to disappear in the near future because a paged memory implementation is already in the works.

The disappointing conclusion is that finding improvable hot spots is unlikely to happen using this test case. Therefore, a different approach to hot spot discovery is in order.

4.2 Targeted hot spot discovery

Executing a large `make` job such as described in the first paragraph generates a huge amount of system calls and therefore a lot of message passing. Because MINIX message passing is a part of every system call, these functions claim centrality in the final profiling result and overshadow other, perhaps more interesting phenomena that simply occur not as often as the message passing functions. Focusing on a single aspect of the system should make hot spots stand out more, because the hot spot/message passing ratio is higher.

Listing 2 shows the profiling result of a small program that creates a file and writes 50 megabytes of data to that file in chunks of 10 kilobytes at a time using the *write()* function. Its source code is available in listing 3. It shows that 7.8% of system time is spent under the label *_alloc_b* in the `mfs` process. This label is the label of the *alloc_bit()* function, found in */usr/src/servers/mfs/super.c*. This function finds a free bit in the i-node or zone bitmap of a MINIX file system. These bits correspond to unused (and therefore free to use) i-nodes or disk zones. Finding free i-nodes is done when creating a new file, finding free zones is used when data is written to a file and that file grows beyond the number of zones allocated to it. For a more detailed discussion of the MINIX file system, see [Tanenbaum].

It is no surprise that this function gets called often. Although only one i-node is necessary for the file, plenty of zones are required to store the 50 megabytes of data. Typical MINIX zones size is 4 kilobytes, so the required number of zones is $50MB/4KB = 12800$ zones. However, it is still surprising that this function alone is responsible for more than a quarter of the work done by the `mfs`. The reason why this is strange is that the MINIX file server is designed to keep track of the last free zone found and look near that zone on the next lookup, because free zones have a tendency to group together. This is especially the case on a nearly empty file system such as used in this test. Running the same test using 250 MB and 500 MB of data shows that the problem only gets worse with larger amounts of data: 49% and 63% respectively of all work done by the `mfs`.

To provide the look-near-latest-zone functionality described above, *alloc_bit()* has an argument *origin* that specifies the bit number to start searching from. Placing a *printf()* statement in *alloc_bit()* that prints the value of *origin* at the start of the bit finding loop shows that this value does not change over the course of writing the test data file. Placing a *printf()* statement at the end of *alloc_bit()* displaying the difference between *origin* and the actual bit number

returned shows that this difference increases with every call to *alloc_bit()*: apparently the function loops over an increasing range of bits with every subsequent execution, which explains why the relative time spent in *alloc_bit()* increases for larger file sizes.

Apparently, *alloc_bit()* receives an *origin* value that does not correspond to the value of the last free data zone found. A
`grep alloc_bit /usr/src/servers/mfs/*` command shows that it is called from only two places: once from *inode.c* and once from *cache.c*. The only call concerning the zone bitmap is in *cache.c*, so the problem is expected to be found in that file. *Alloc_bit()* is called from the function *alloc_zone()*, which calculates an origin value based on its *zone* argument, which has a meaning similar to *origin* in *alloc_bit()* in that it can be used to specify a search base. In *write.c* it can be seen that *alloc_zone()* decides whether or not to use the last free zone value: if *zone* equals the number of the first data zone of the file system (which is always occupied by the root directory), then *alloc_zone()* uses the last free zone value as the value for *origin*, otherwise the passed zone number is converted to a bit number and the converted value is used as *origin*.

As explained above, the last zone number is *not* used as *origin*, so the value of *zone* passed to *alloc_zone* must be something other than the value of the first data zone. A `grep alloc_zone /usr/src/servers/mfs/*` shows that there are three places from where *alloc_zone()* is called, all three from *write.c*. The first two occurrences relate to allocating indirect zones, which are necessary for storing large files. These calls indeed use a different value as the base zone, namely the value of the first data zone of the file. However, indirect zones are only allocated sporadically, so this does not explain why *alloc_bit()* is called so often with an incorrect value. The third occurrence of a call to *alloc_zone()* uses a value for *zone* which depends on the state of the file for which the zone must be allocated: if the first zone of the file is not allocated, the value of the file system's first data zone is used (which will cause *alloc_zone()* to call *alloc_bit()* with an *origin* value of the last found free data zone). Otherwise, the value of the file's first data zone is used. In other words, the fast lookup technique is only used when the file being written has no first data zone allocated. Apart from special files containing "gaps" which are rarely used, this situation only occurs when the first data zone of an empty file is written. In all other situations, *alloc_bit()* will always be instructed to start searching from the bit number corresponding to the file's first data zone. The larger the file is, the more zones it takes up and the longer *alloc_bit()* has to search. This is definitely a hot spot that can be optimized.

4.3 Optimizing the hot spot

Presumably, this "look near the file's first data zone" behaviour is intended to prevent fragmentation of the file system by writing new data zones of a growing file as near after the old zones as possible. However, as explained above, this will cause increasing search overhead for every zone written beyond the first. This would make a machine running MINIX very unsuitable to operate as for

example a file server or any other type of machine that regularly has to handle writing large amounts of data to disk. By altering *write.c* to always use the value for the file system's first data zone instead of the file's first data zone, this overhead can be reduced significantly. However, this will come at the cost of a higher probability of file system fragmentation, because no effort will be made anymore to keep the zones of a file together.

To solve this problem, the idea of keeping track of the last found free zone can be extended to the file level. By introducing a field *i_zsearch* to the *inode_t* structure as defined in *inode.h*, we can keep track of the last data zone allocated to a specific file. By adding a line to the function *get_inode()* in *inode.c* this field is initialized to the value *NO_ZONE* to signal that no zones have been allocated to this file since it has been opened. The last change is to *write.c*, where the policy of defining the search base is changed. Whenever a new zone is needed for a file, the `mfs` first checks if *i_zsearch* is set. If it is, this value is used as the search base. If it is not set, the `mfs` next checks if the file has a first data zone set. If it has, this value is used as the search base. For large files, this may cause a long search through the zone bitmap, just as in the original situation. However, this will happen only once, because on the next run, *i_zsearch* will be set. If the first zone is not set, *alloc_zone()* is instructted to search from the first data zone of the file system, which will cause *alloc_zone()* to search from the last found data zone. After *alloc_zone()* returns, the returned zone number is stored in the i-node's *i_zsearch* field to help the next lookup. In this situation, the `mfs` will attempt to allocate new zones for a file near the old zones to avoid fragmentation, without the overhead of having to skip over zones that are known to be in use.

Figure 4.3 shows the percentage of samples spent in *alloc_bit()* compared to the total number of samples spent in `mfs`, both for the original file system and an implementation that uses the optimization suggested above. It is clear from this graph that the optimized implementation spends less time in *alloc_bit()* and also does not suffer from increased search overhead for larger files. Figure 4.3 shows the total number of samples required to write different amounts of data to a file. This graph shows that for all amounts of data written, the optimized implementation outperforms the original implementation, and that the difference increases with file size. All shown figures are the average of three profiling runs, executed on MINIX 3.1.3b r3052 running inside QEMU with 256 MB RAM on a Mobile AMD Sempron 3500+ processor.

5 Conclusion

Performance is an important aspect of every serious operating system. If an operating system performs poorly compared to alternatives, it is less likely to become widely adopted. However, finding poorly performing parts of a system is not trivial: even if you are able to determine that an operation takes more time than expected, it is still hard to determine where the problem is just by looking at the code. To help the effort of finding hot spots in MINIX 3, profiling

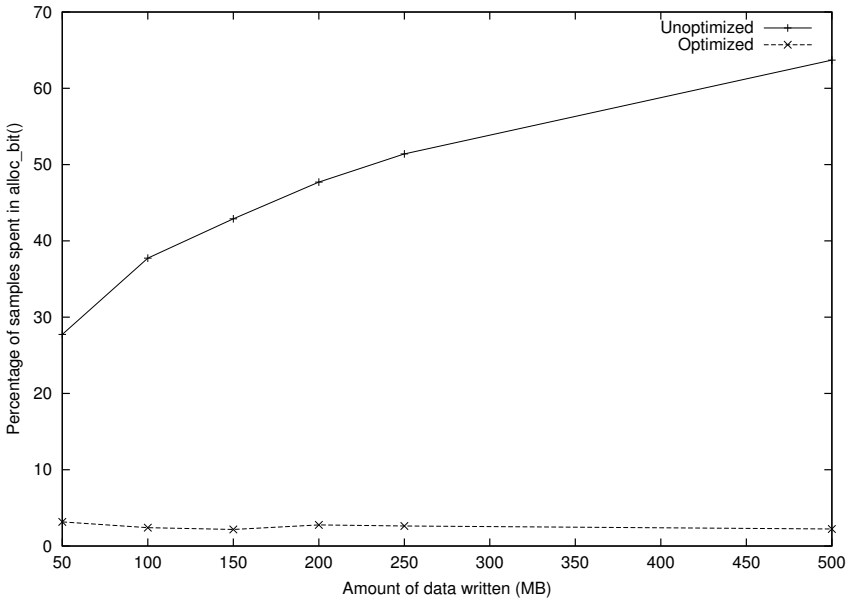

Figure 1: Percentage of `mfs` usage spent by *alloc_bit()*

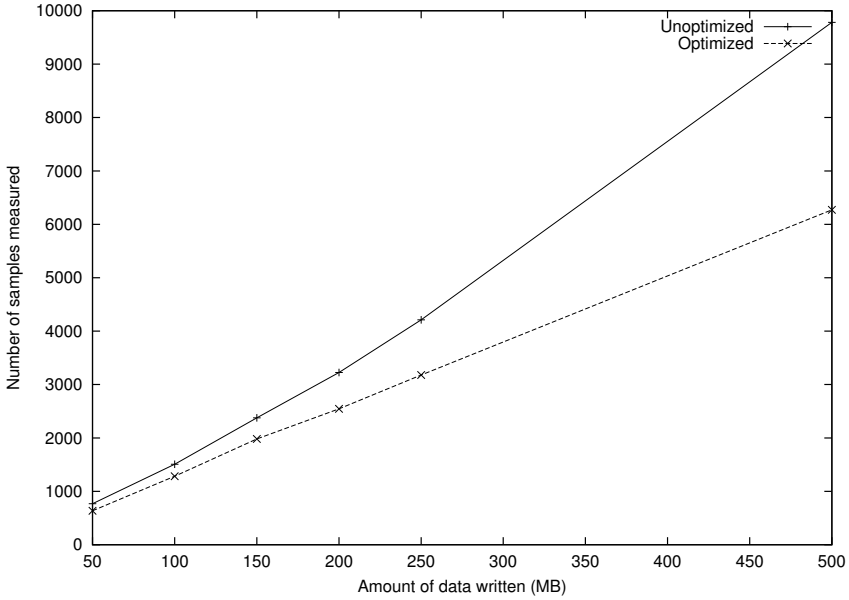

Figure 2: Number of samples needed to write data to a file

tools have been developed that can analyze which parts of the system use up most of the machine's time.

This paper has shown how the statistical profiling tool can be used to find hot spots in parts of the system. It is hard to locate a hot spot by letting the system do a lot at once, but when it is known where to look, for example because of experienced low performance when performing a specific task, the profiler can be used to look into the system to see where all that time is spent. Then, when the source of the problem is found, a skilled programmer can look into that source and take the problem away, such as has been done to the part of the `mfs` described in this paper. However, it is not easy to determine which parts of the system are performing below par. What would be valuable to finding and optimizing hot spots would be to know how various parts of MINIX perform as compared to other operating systems, so that poorly performing parts stick out. A set of automated benchmark tools available for multiple platforms would be very valuable to this end.

References

[Meurs] Rogier Meurs. (August 2006). Building Performance Measurement Tools for the MINIX 3 Operating System. *Minix 3 documentation*. Retrieved January 1 2007, from http://www.minix3.org/doc/meurs_thesis.pdf

[Tanenbaum] Andrew Tanenbaum and Albert Woodhull. Operating Systems, Design and Implementation (3rd edition). Prentice Hall, 2006

6 Appendix A: output listings

Listing 1: statistical profile for "make fresh hdboot"

```
Showing processes and functions using at least 1% time.

================================================================================
Data file: profile.stat.out
================================================================================

  System process ticks:      56910 ( 55%)
    User process ticks:      46044 ( 44%)          Details of system process
       Idle time ticks:       1099 (  1%)          samples, aggregated and
                           ----------  ----         per process, are below.
            Total ticks:     104053 (100%)

--------------------------------------------------------------------------------
Total system process time                                          56910 samples
--------------------------------------------------------------------------------
   clock __receiv ********************************************************  35.3%
  system _phys_co ************************                                 16.1%
  system fill_sta ***********                                               6.7%
  system __receiv **********                                                6.1%
     vfs __sendre ******                                                    3.3%
  system _lock_se *****                                                     2.7%
     vfs __receiv ****                                                      2.2%
     mfs __sendre ****                                                      2.0%
     mfs __receiv ****                                                      2.0%
     vfs   __send ****                                                      1.9%
     mfs   __send ***                                                       1.4%
  system _isokend **                                                        1.2%
             <1% ******************************                             19.1%
--------------------------------------------------------------------------------
total                                                                     100.0%

--------------------------------------------------------------------------------
system                                          37.8% of system process samples
--------------------------------------------------------------------------------
_phys_co ****************************************************************  42.6%
fill_sta ***************************                                      17.6%
__receiv ************************                                         16.2%
_lock_se ***********                                                       7.2%
_isokend *****                                                            3.2%
_do_vdev ****                                                             2.6%
_umap_lo ****                                                             2.0%
_virtual ***                                                              1.8%
_sys_tas ***                                                              1.8%
_do_copy **                                                               1.1%
     <1% ******                                                           3.9%
--------------------------------------------------------------------------------
system                                                                    100.0%

--------------------------------------------------------------------------------
clock                                           35.3% of system process samples
--------------------------------------------------------------------------------
__receiv ********************************************************************* 100.0%
     <1% *                                                                  0.0%
--------------------------------------------------------------------------------
clock                                                                     100.0%

--------------------------------------------------------------------------------
mfs                                             12.0% of system process samples
--------------------------------------------------------------------------------
__sendre ****************************************************************  16.4%
__receiv ****************************************************************  16.3%
  __send ******************************************                       11.8%
_search_ ***************************                                      6.9%
```

```
__notify ********************                                          5.2%
_get_blo *************                                               3.2%
_fs_read ***********                                                 2.6%
_put_blo **********                                                  2.3%
_read_ma *********                                                   2.3%
_parse_p *********                                                   2.2%
   _main ********                                                    1.9%
_get_nam *******                                                     1.7%
_rw_chun *******                                                     1.6%
_get_ino ******                                                      1.5%
_put_ino ******                                                      1.3%
__taskca *****                                                       1.3%
 _rm_lru *****                                                       1.2%
_find_in *****                                                       1.2%
  _conv4 *****                                                       1.1%
 _lookup *****                                                       1.1%
_strncmp *****                                                       1.1%
     <1% ************************************************************   15.8%
-------------------------------------------------------------------------------
mfs                                                                 100.0%

-------------------------------------------------------------------------------
vfs                                          11.1% of system process samples
-------------------------------------------------------------------------------
__sendre ************************************************************  30.0%
__receiv ****************************************                       19.6%
  __send ***********************************                           17.4%
_find_vn ********                                                     3.5%
_read_wr ******                                                      2.7%
     no0 ******                                                      2.7%
_fs_send ****                                                        1.7%
   _main ****                                                        1.5%
_Xlookup ***                                                         1.3%
_get_fil ***                                                         1.2%
_get_wor ***                                                         1.2%
_req_loo ***                                                         1.0%
     <1% *********************************                           16.2%
-------------------------------------------------------------------------------
vfs                                                                 100.0%

-------------------------------------------------------------------------------
pm                                            3.2% of system process samples
-------------------------------------------------------------------------------
__receiv ************************************************************  20.7%
  __send ********************************************************      17.3%
__sendre ********************************************************      17.2%
   _main ************************************************              15.7%
__syscal **********                                                   3.0%
_send_wo *********                                                    2.8%
_pm_isok *******                                                      2.2%
__notify *******                                                      2.1%
__taskca *******                                                      2.0%
_get_wor ******                                                       1.8%
 _adjust *****                                                        1.5%
_real_br ****                                                         1.0%
     <1% ****************************************                     12.7%
-------------------------------------------------------------------------------
pm                                                                  100.0%

-------------------------------------------------------------------------------
processes <1% (not showing functions)         0.6% of system process samples
-------------------------------------------------------------------------------
total                                       100.0%
```

Listing 2: statistical profile for writing 50 MB

```
Showing processes and functions using at least 1% time.

================================================================================
Data file: profile.stat.out
================================================================================

  System process ticks:        786 (100%)
    User process ticks:          3 (  0%)          Details of system process
       Idle time ticks:          0 (  0%)          samples, aggregated and
                          ----------  ----          per process, are below.
            Total ticks:        789 (100%)

--------------------------------------------------------------------------------
Total system process time                                          786 samples
--------------------------------------------------------------------------------
  system __receiv ***********************************************          13.7%
   clock __receiv *********************************************           12.5%
  system fill_sta *********************************                      10.1%
  system _phys_co ******************************                          9.0%
     mfs _alloc_b *************************                               7.8%
     mfs   slword ******************                                      5.6%
  system _lock_se ***************                                         4.3%
  system _do_vdev ***************                                         4.3%
 at_wini __sendre ***********                                             3.2%
     mfs _get_blo *********                                               2.5%
     mfs __receiv *******                                                 1.9%
     mfs __sendre ******                                                  1.5%
     vfs __sendre ******                                                  1.5%
  system _isokend *****                                                   1.4%
  system _umap_lo *****                                                   1.3%
     mfs _rw_scat ****                                                    1.1%
  system _sys_tas ****                                                    1.0%
     vfs   __send ****                                                    1.0%
             <1% ****************************************************    16.3%
--------------------------------------------------------------------------------
total                                                                    100.0%

--------------------------------------------------------------------------------
system                                        48.5% of system process samples
--------------------------------------------------------------------------------
__receiv ************************************************************    28.3%
fill_sta *********************************************                   20.7%
_phys_co *******************************************                     18.6%
_do_vdev ********************                                            8.9%
_lock_se ********************                                            8.9%
_isokend *******                                                         2.9%
_umap_lo ******                                                          2.6%
_sys_tas *****                                                           2.1%
_virtual ***                                                             1.3%
_do_umap ***                                                             1.0%
_verify_ ***                                                             1.0%
     <1% ********                                                        3.7%
--------------------------------------------------------------------------------
system                                                                   100.0%

--------------------------------------------------------------------------------
mfs                                           29.6% of system process samples
--------------------------------------------------------------------------------
_alloc_b ************************************************************    26.2%
  slword *********************************************                   18.9%
_get_blo ********************                                            8.6%
__receiv ***************                                                 6.4%
__sendre ************                                                    5.2%
_rw_scat **********                                                      3.9%
_write_m ********                                                        3.0%
 _rm_lru ********                                                        3.0%
```

```
_put_blo ********                                                          3.0%
_rd_indi ******                                                            2.1%
_cpf_rev ******                                                            2.1%
  __send *****                                                             1.7%
_read_ma *****                                                             1.7%
_rw_chun *****                                                             1.7%
_cpf_new ****                                                              1.3%
_fs_read ****                                                              1.3%
_get_sup ****                                                              1.3%
  _conv4 ****                                                              1.3%
     <1% ******************                                               7.3%
-------------------------------------------------------------------------------
mfs                                                                      100.0%

-------------------------------------------------------------------------------
clock                                              12.5% of system process samples
-------------------------------------------------------------------------------
__receiv ************************************************************** 100.0%
     <1% *                                                                 0.0%
-------------------------------------------------------------------------------
clock                                                                    100.0%

-------------------------------------------------------------------------------
vfs                                                 5.2% of system process samples
-------------------------------------------------------------------------------
__sendre ************************************************************  29.3%
  __send ****************************************                       19.5%
__receiv *********************************                              17.1%
_get_wor ********************                                           9.8%
   _main ***********                                                     4.9%
_get_fil ***********                                                     4.9%
 _ex64lo ******                                                          2.4%
_fs_send ******                                                          2.4%
__taskca ******                                                          2.4%
     no0 ******                                                          2.4%
_req_rea ******                                                          2.4%
_cmp64ul ******                                                          2.4%
     <1% *                                                               0.1%
-------------------------------------------------------------------------------
vfs                                                                      100.0%

-------------------------------------------------------------------------------
at_wini                                             4.2% of system process samples
-------------------------------------------------------------------------------
__sendre ************************************************************  75.8%
_w_trans ******                                                          6.1%
_setup_d ******                                                          6.1%
_sys_saf ***                                                             3.0%
 _rem64u ***                                                             3.0%
__receiv ***                                                             3.0%
__taskca ***                                                             3.0%
     <1% *                                                               0.0%
-------------------------------------------------------------------------------
at_wini                                                                  100.0%

-------------------------------------------------------------------------------
processes <1% (not showing functions)               0.0% of system process samples
-------------------------------------------------------------------------------
total                                             100.0%
```

Listing 3: file writing test program

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <fcntl.h>
#include <errno.h>
#include <string.h>

#define BUF_SIZE (10 * 1024)
#define FILE_SIZE (50 * 1024 * 1024)
#define NUM_RUNS FILE_SIZE / BUF_SIZE
#define OUTFILE_NAME "/usr/tmp/file.out"

/* fill a buffer with repeating lowercase alphabet */
void init_buf(char *buf, size_t buf_size) {
    int i;
    for (i = 0; i < buf_size; i++) {
        buf[i] = (char)(i % 26 + 'a');
    }
}

/* main function */
int main (int argc, char **argv) {
    char buf[BUF_SIZE];
    int i;
    int fd;

    /* open output file */
    fd = open(OUTFILE_NAME, O_WRONLY | O_CREAT | O_TRUNC);
    if (fd < 0 ) {
        perror("open");
        exit(1);
    }

    /* fill buffer with verifiable contents */
    init_buf(buf, BUF_SIZE);

    /* write data to file */
    for (i = 0; i < NUM_RUNS; i++) {
        write(fd, buf, BUF_SIZE);
    }

    return 0;
}
```

7 Appendix B: altered source files

File: /usr/src/servers/mfs/inode.h (complete)

```
/* 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.
 */

#include "queue.h"

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 */
  bit_t i_zsearch; /* where to start search for new zones */

  char i_mountpoint; /* true if mounted on */

  char i_seek; /* set on LSEEK, cleared on READ/WRITE */
  char i_update; /* the ATIME, CTIME, and MTIME bits are here */

  LIST_ENTRY(inode) i_hash;     /* hash list */
  TAILQ_ENTRY(inode) i_unused;  /* free and unused list */

} inode[NR_INODES];

/* list of unused/free inodes */
EXTERN TAILQ_HEAD(unused_inodes_t, inode)  unused_inodes;

/* inode hashtable */
EXTERN LIST_HEAD(inodelist, inode)         hash_inodes[INODE_HASH_SIZE];

EXTERN unsigned int inode_cache_hit;
EXTERN unsigned int inode_cache_miss;

#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_SEEK            0 /* i_seek = NO_SEEK if last op was not SEEK */
#define ISEEK              1 /* i_seek = ISEEK if last op was SEEK */
```

File: /usr/src/servers/mfs/inode.c (altered functions only)

```
/*===========================================================================*
 * 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 the inode in the hash table. If it is not there, get a free inode
 * load it from the disk if it's necessary and put on the hash list
 */
  register struct inode *rip, *xp;
  int hashi;

  hashi = numb & INODE_HASH_MASK;

  /* Search inode in the hash table */
  LIST_FOREACH(rip, &hash_inodes[hashi], i_hash) {
      if (rip->i_num == numb && rip->i_dev == dev) {
          /* If unused, remove it from the unused/free list */
          if (rip->i_count == 0) {
      inode_cache_hit++;
              TAILQ_REMOVE(&unused_inodes, rip, i_unused);
  }
          ++rip->i_count;
          return rip;
      }
  }

  inode_cache_miss++;

  /* Inode is not on the hash, get a free one */
  if (TAILQ_EMPTY(&unused_inodes)) {
      err_code = ENFILE;
      return NIL_INODE;
  }
  rip = TAILQ_FIRST(&unused_inodes);

  /* If not free unhash it */
  if (rip->i_num != 0)
      unhash_inode(rip);

  /* Inode is not unused any more */
  TAILQ_REMOVE(&unused_inodes, rip, i_unused);

  /* Load the inode. */
  rip->i_dev = dev;
  rip->i_num = numb;
  rip->i_count = 1;
  if (dev != NO_DEV) rw_inode(rip, READING); /* get inode from disk */
  rip->i_update = 0; /* all the times are initially up-to-date */
  rip->i_zsearch = NO_ZONE; /* no zones searched for yet */
  if ((rip->i_mode & I_TYPE) == I_NAMED_PIPE)
rip->i_pipe = I_PIPE;
  else
rip->i_pipe = NO_PIPE;
  rip->i_mountpoint= FALSE;

  /* Add to hash */
  addhash_inode(rip);

  return(rip);
}
```

File: /usr/src/servers/mfs/write.c (altered functions only)

```
/*===========================================================================*
 *                              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) {
if (rip->i_zsearch == NO_ZONE) {
/* First search so far, start looking from file's first zone.
 * This tries to prevent file system fragmentation by
 * keeping zones that belong to the same file as close
 * together as possible */
if ( (z = rip->i_zone[0]) == NO_ZONE) {
/* no first zone for file either */
z = rip->i_sp->s_firstdatazone; /* let alloc_zone decide */
}
} else {
/* searched before, start from last find */
z = rip->i_zsearch;
}
if ( (z = alloc_zone(rip->i_dev, z)) == NO_ZONE) return(NIL_BUF);
rip->i_zsearch = z; /* store for next lookup */
if ( (r = write_map(rip, position, z, 0)) != 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);
}
```