Final Year Project

# Porting MINIX to Xen

| | |
|---|---|
| **Name** | Ivan Kelly |
| **ID Number** | 0225991 |
| **Supervisor** | John Sturdy |
| **Course** | Computer Systems |

Submitted in part requirement for final year project course CS4618 to

University of Limerick, Ireland on May 8, 2006

# Acknowledgements

MINIX on Xen would not have been possible without the support of many people. I would like to thank the Xen Team, not only for creating Xen, but for providing support through IRC and email. Without the help of Mark Williamson and Keir Fraser in particular, this project would have never gotten off the ground.

I would also like to thank Andrew S. Tanenbaum whose help in the early days of the project, helped me see what needed to be done.

I would like to thank John Sturdy and Mel Gorman for talking me through many difficult problems during the implementation of MINIX on Xen.

I would like to thank my friends and family for their support and my housemates for bringing me tea, and listening to my ranting.

Lastly, I would like to thank Pili for her love and support, and for her willingness to endure hours of rants regarding GDTs and interrupt handlers.

# Abstract

Virtualisation has received a lot of attention from the I.T. media lately. Paravirtualisation in particular has drawn a lot of attention due to it's high performance. Paravirtualised virtual machines run at near native speeds. Operating systems must be modified to run on paravirtualised platforms.

Developers starting out in the field of paravirtualisation face a steep learning curve. This project hopes to soften that curve, by supplying developers with an insight into the porting of an operating system to a paravirtualised platform. In this case, the MINIX operating system is ported to the Xen platform.

# Table of Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

Virtualisation has seen a sharp raise in popularity lately, especially in the open source community. Projects such as User Mode Linux(1) and Xen(2), have allowed users and administrators alike, to set up whole virtual networks within one single piece of hardware. Linux vendors are starting to see the appeal of this technology with Redhat and Novell both having plans to ship the next generation of their products with virtualisation support built in(3). Likewise, chip makers have seen its potential. AMD(4) and Intel(5) are shipping virtualisation technology in the newest incarnations of their chips.

Virtualisation has been touted as the new 'killer app' for the open source community(6), and open source virtualisation products have started directly challenging the established players in the field such as VMWare. VMWare, seeing the new competition, have released free versions of their workstation and server products.

There are barriers for those wishing to learn about and develop with virtualisation technology. Developing for the proprietary vendors is difficult as the source code is unavailable. Xen was chosen for this project because it is an open source project and community support is substantial.

The documentation for Xen(7), while sufficiently detailed, is targetted primarily at those who have many years of kernel development. The documentation alone is not enough to gain a full understanding of Xen. To understand how Xen works, one must read the source of the operating systems which have been ported to it. This is not an easy task. Linux and Freebsd contain 4,818,291[*] and 1,781,777[†] lines of code respectively. A simpler kernel must be ported

[*]Line count taken on Linux 2.6.16.9 with sloccount.
[†]Line count taken on Freebsd 6.0 with sloccount.

| Type | Speed | Modified OS | Multiple OS | Products |
|---|---|---|---|---|
| Full- | Slow | No | Yes | VMWare(11), Virtual PC(12) |
| Para- | Fast | Yes | Yes | Xen(2), Denali(13) |
| Emulation | Very Slow | No | Yes | qemu(14), Bochs(15) |
| Partitioning | Fast | Yes | No | Solaris Zones(16), UML(1) |
| Hardware | Very Fast | No | Yes | Intel Vanderpool(5), AMD Pacifica(4) |

Table 1.1: Comparison of virtualisation technologies

for ease of understanding of how Xen works. For this, MINIX was chosen. MINIX(8) is an ideal candidate as it was designed with ease of learning in mind. Coupled with its accompanying textbook(9), it is used by many universities to teach operating systems to undergraduates.

There were other reasons which influenced the choice to take on this project.

To date, there had been no microkernels ported to Xen. This was the initial motivation for looking into this field. The choice was between MINIX and GNU Hurd(10). MINIX won out due to its excellent documentation. The fact that MINIX was already used as a tool for teaching kernel development, was also a factor. By porting MINIX to Xen, developing with MINIX would be simplified, as a dedicated machine would not be needed to run it. A MINIX port for Xen would also make it easier to learn about Xen internals, as the MINIX kernel is very small and simple.

## 1.1 What is Virtualisation?

Virtualisation is a technology which allows multiple instances of an operating system to run on a shared set of resources, unaware that it is sharing the resources with others. The operating systems can be all of the same type or of different types. There are many different forms of virtualisation, each with their own advantages and disadvantages.

See table 1.1 for details.

### 1.1.1 State of the Art

There are currently two main players in the field of software virtualisation, VMWare and Xen. VMWare's products have the advantage of offering full virtualisation. As such, operating systems can run unmodified on VMWare's products. This does however come with performance penalties. VMWare also have a para-virtualisation product called VMWare ESX Server(17).

Xen is an open source virtualisation solution. It is an implementation of para-virtualisation and as such, requires operating systems to be modified to run on top of it. These modifications mean that the guest operating system runs very efficiently.

## 1.2 Project Scope

This project details the porting of MINIX to the Xen architecture. This includes the porting of the microkernel, the console driver and the block device driver.

## 1.3 Prerequisites

This documents assumes a prior knowledge of C and some assembly. Also a knowledge of operating systems design and the x86 architecture, while not strictly necessary, will greatly ease the understanding of the following text. To this end, “Operating Systems, Design and Implementation”, by Andrew S. Tanenbaum(9) is recommended.

# Chapter 2

# Background Information

## 2.1 What is Xen?

Xen is a para-virtualising virtual machine monitor for the x86 architecture. Using Xen, one machine can run multiple virtual machines concurrently with very little overhead.

Xen itself is a kernel that runs directly on the hardware. This approach is very different to traditional software virtualisation solutions, which run on top of an operating system which is already running on the hardware.

The advantage of this approach is that Xen can effectively and simply partition resources. It can then guarantee these resources to its virtual machines.

A virtual machine in Xen is called a domain. There are two types of domain, a dom0 and a domU. A single machine can only ever run one dom0, whereas it can run as many domU instances as its resources allow. The dom0 is the control domain that can be used to create and destroy domUs. dom0 is usually the only domain which has access to real hardware. As such, dom0 is responsible for providing backend drivers for virtual block and network devices to the domUs. The domU can then access these resources through very thin frontend drivers.

Xen uses para-virtualisation. An operating system must be modified to run on Xen. This project is to port MINIX to run on Xen.

## 2.2 What is MINIX?

MINIX is a minimalistic UNIX clone, developed by Andrew S. Tanenbaum in 1987. It was developed out of a need for a teaching operating system. Traditionally, AT&T UNIX had been

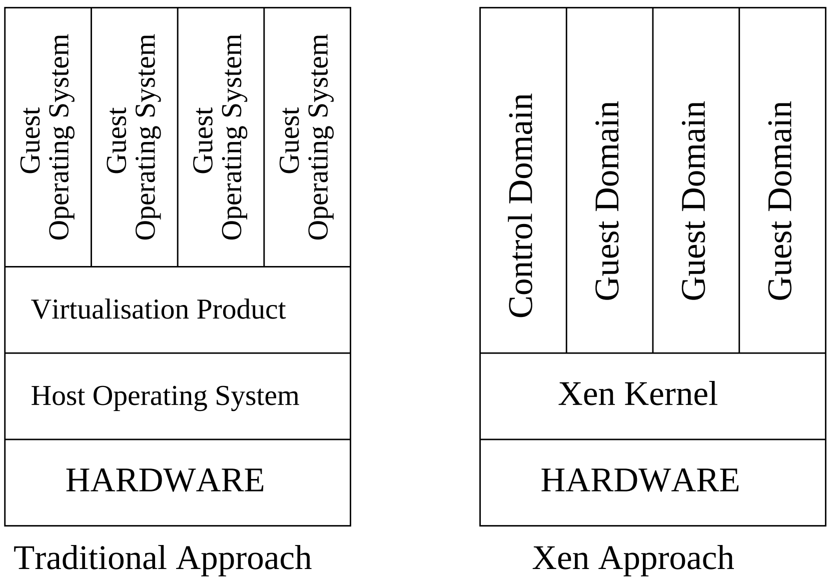

Figure 2.1: Comparison of virtualisation approaches

used, but a change in its license meant that the source code was no longer available for teaching. From then until the release of MINIX, operating systems lecturers could only teach theory, as they had no example code available to them.

MINIX was released alongside a textbook explaining operating system principles and the MINIX implementation(9). The architecture of MINIX is discussed in section 6.1.

# Chapter 3

# Development Environment

## 3.1 Hardware Setup

For development, two computers are used. One was for writing and compiling the code. The other was for testing and running the code. This setup is used so development can continue during a complete system crash.

Both computers are connected to the local area network. There is also a serial cable running between them. This is needed for debugging and for console output from the testing computer.

## 3.2 Software Used

This project used MINIX version 3.1.1 and Xen version 2.0.7.

Programming was done in Emacs. Subversion(18) was used for software configuration management.

Compilation of the Xen modifications was done with gcc. Compilation of MINIX is more difficult as it can only be compiled by the Amsterdam Compiler Kit(19) which will only run from within MINIX. To overcome this, a MINIX installation was set up in qemu(14), and all compilation was done on this virtual machine.

Transferring the source from the development computer to the MINIX installation is another problem, as MINIX does not have an NFS driver. One solution is to copy all files over for each compilation. This is not suitable, as the whole tree needs to be recompiled every time as all the files look like they have been edited. Another solution is to use rsync(20). This was not viable as there was no rsync port for MINIX when development started. However a rsync port for

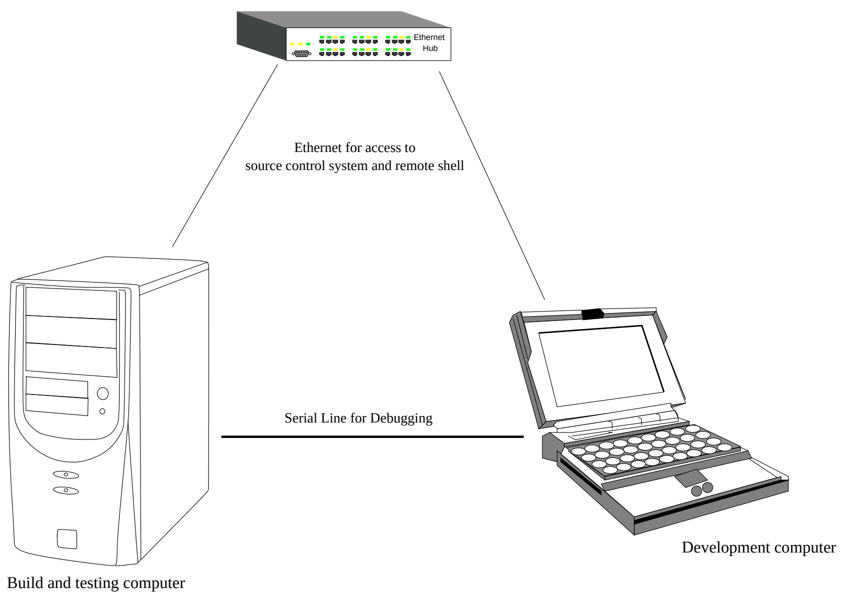
Ethernet Hub
Ethernet for access to source control system and remote shell
Serial Line for Debugging
Build and testing computer
Development computer

Figure 3.1: Hardware setup for project

MINIX has recently been released. Yet another solution would have been to install a subversion client on MINIX, but there are none available.

The current solution is to use a post commit hook for the subversion repository. When a commit occurs, all changes are exported to a CVS(21) repository. MINIX, using CVS, can then update its source tree from the CVS repository and build a kernel image.

## 3.3 Debugging

Debugging of Xen domU kernels takes place over the serial line. Console output also comes over the serial line, so debugging packets have the most significant bit set to distinguish them from console packets(22).

On the development computer, there is a daemon that splits the serial input into two streams and sends them to different TCP ports on the computer. GDB(23) can then connect to one of these TCP ports and debug the kernel.

There are no symbol files for the MINIX kernel, so step through debugging is not possible. It is possible however, to put Xen into a debugging state, and dump memory from any particular domain.

# Chapter 4

# Xen Internals

This chapter will explain the internals of Xen in more detail than the introduction.

Xen can be divided into two parts, its kernel and its applications. Each of these will be discussed separately.

## 4.1 The Kernel

The Xen kernel is a multiboot(24) compliant image, upon which Xen-compatible kernels can run. The Xen kernel acts as a mediation level between the domain kernel and hardware, allowing many domains to coexist.

It presents an interface similar to the x86 architecture. Currently Xen has only been ported to run on x86_32 and x86_64.

### 4.1.1 Privilege Rings

The x86 architecture separates processes requiring differing privileges using a mechanism called privilege rings(25). There are four privilege rings, from ring 0 to ring 3. Ring 0 has the most privileges, and can access all instructions and memory addresses. The operating system usually runs in ring 0. Ring 1 and 2 have less privileges and are used for operating system services. Ring 3 has the least privileges and is used for user applications.

Xen exploits these privilege rings to allow multiple operating systems to run at the same time. Xen itself runs in ring 0. The domain kernels run in ring 1. Any calls which would previously have required ring 0 to operate are replaced with hypercalls to Xen. Applications still run unmodified in ring 3.

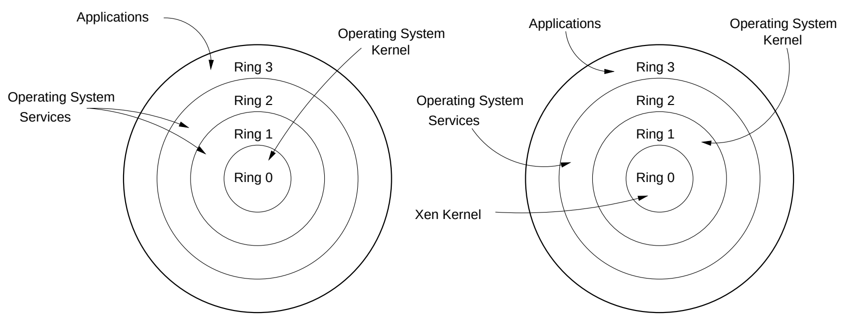

Figure 4.1: Use of Intel privilege rings in Xen

### 4.1.2 Hypercalls

Domains communicate with Xen using software interrupts called hypercalls. For more details on hypercalls, see 6.2.1.

Only ring 1 can make hypercalls, which can present difficulties for those porting a microkernel to Xen. See 6.2.1 for more details.

### 4.1.3 Memory Management

Since domain kernels run in ring 1, all low level memory operations must go through Xen. This allows Xen to partition and manage the memory for these domains.

The operations that Xen manages are those for paging and segmentation. As such, the domain kernel cannot directly edit the page tables, global descriptor tables or local descriptor tables. Doing so could potentially crash other domains on the same system(7). These operations must all be mediated by Xen.

Three different types of addresses are referred to in this document. They are machine addresses, pseudo-physical addresses and virtual addresses.

- Machine addresses are memory addresses on physical memory.

- Pseudo-Physical addresses are addresses in which the first 20 bits specify a page in the page table, and the last 12 indicate an offset.

- Virtual addresses are offsets within a segment. They must be added to the segment base, then looked up in the page table before the machine address can be found.

Xen provides a domain with a list of machine frames during bootstrapping, and it is the domain's responsibility to create the pseudo-physical address space from this.

### 4.1.4 Shared Information

Xen provides each domain with a shared information frame for communication between Xen and the domain. This frame is used for sending event notifications to the domain along with supplying time information to the domain.

### 4.1.5 Control Interface Rings

The control interface rings are placed half way through the shared information frame. There are two rings, the transmitter ring and the receiver ring. Each ring is an array of eight control messages. Xen fills these rings in a round robin fashion. There are also variables associated with each ring that the domain uses to inform Xen that it has handled all requests on the ring and it can start filling it again.

This is covered in more detail in 6.2.4.

## 4.2 The Applications

The Xen applications runs on dom0. It consists of a daemon called Xend, a console application called xm and a number of libraries.

### 4.2.1 Libraries

The two primary C libraries are libxc and libxutil. libxutil is a library of utility functions needed by Xen. libxc is the Xen control library. It communicates with Xen through dom0 hypercalls and does most of the work in creating and destroying domains.

Both libraries have python bindings.

### 4.2.2 Xend and Xm

Xend and xm communicate with Xen through the Xen libraries. Xend is started at boot time and remains running in the background as a daemon until the machine is shut down.

xm is the client for this daemon. It is used for starting and stopping domains and retrieving information about domains.

# Chapter 5

# Modifications to Xen

## 5.1 Motivation for Xen Modifications

Xen is designed to run kernel images that conform to a specific set of criteria. They need to be in an ELF executable format, with extra sections added for verification. The MINIX kernel image has its own format, based on the a.out standard.

MINIX can only be compiled with the Amsterdam Compiler Kit (ACK) (19). The source code is ANSI C, but the assembly is in a modified Intel format only compilable by ACK. ACK has a number of limitations. It can only run on MINIX and only produces a.out executables.

One work around for this is to port MINIX to the GNU tool chain, but this is difficult for a number of reasons. The assembly in MINIX is in Intel syntax. The GNU assembler only takes input in AT&T syntax. To compile the MINIX assembly with the GNU tools a large amount code would need to be modified and this would be a project in itself.

MINIX, as a microkernel(26)[*], runs as an array of processes. All of the initial processes must be loaded at startup by the bootloader. Xen does not allow for this. It expects to bootstrap a macrokernel, where everything is in one executable.

In Xen, domU kernels are loaded by builders. These builders are part of the libxc library. There are two builders available. One is for ELF executables. It is called the Linux builder for historical reasons. The other is for plan9(27). A specialised plan9 builder had to be created because the Plan9 kernel is in an a.out format. The MINIX kernel, as another a.out format, also needs its own builder.

A builder loads the kernel image from dom0's filesystem into domU's allocated memory.

[*]See 6.1

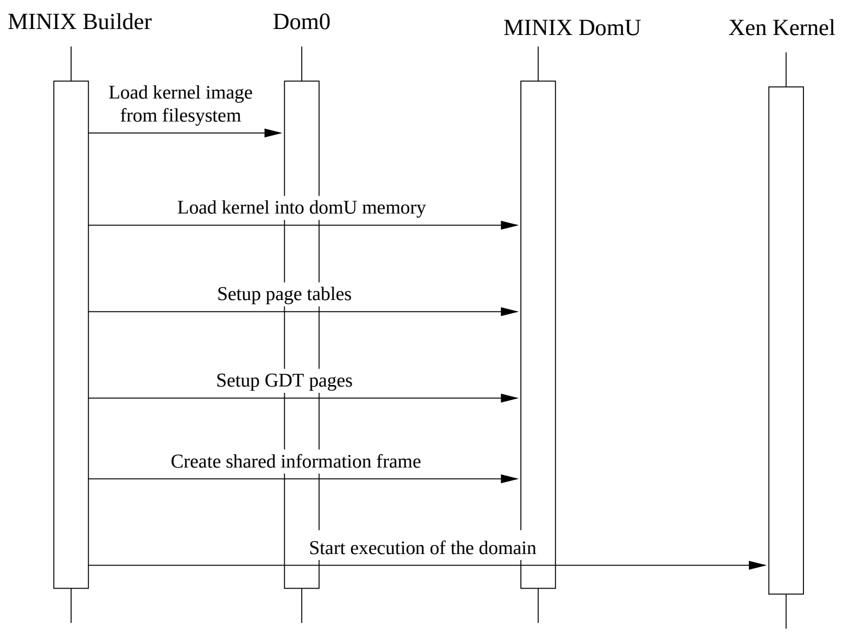
MINIX Builder
Dom0
MINIX DomU
Xen Kernel
Load kernel image from filesystem
Load kernel into domU memory
Setup page tables
Setup GDT pages
Create shared information frame
Start execution of the domain

Figure 5.1: Sequence of MINIX builder actions

The bootstrap page table is then set up. The CPU context is initialised and the domain is booted.

To modify one builder, the whole libxc library must be recompiled. This is not a problem if all kernels are ELF kernels, as modifications are not needed. Problems only arise when you move out of the homogeneous GNU environment. The Xen developers have realised this short coming and recently there has been request for comments on the Xen developers mailing list(28) discussing a new builder architecture. If this is accepted, each builder will be its own executable, and new builders can just be dropped into place.

### 5.1.1 Executeable File Formats

#### Assembler Output (a.out)

a.out is the classic Unix executable file format. There are usually three sections and a header in an a.out executable. The header specifies the size of the sections among other things. The text section contains the machine code instructions of the executable. The data section contains initialised data, while the bss section contains uninitialised data. There may also be a symbols section but this is optional.

#### Executable Linking Format (ELF)

ELF is a newer executable file format that superseded a.out. It is used by most modern UNIX like operating systems. The format is more complex than a.out. Each file contains a header. This header describes the sections in the file, which usually includes a data section, a code section and a bss section.

#### The MINIX Kernel Image

The MINIX kernel image consists of many a.out files concatenated together. The first file must be the microkernel itself followed by the processes that need to be run at startup, such as the device drivers, filesystem and memory manager.

## 5.2 The MINIX Builder

The MINIX builder is based on the builders for Linux and plan9. The entry point for builders is *xc_<imagetype>_build*. See listing 5.1 for an example prototype for a builder function. These functions have python binding to allow calls from Xend.

Listing 5.1: Prototype for builder function.

```
int
xc_minix_build(int xc_handle,
               u32 domid,
               const char *image_name,
               const char *cmdline,
               unsigned int control_evtchn,
               unsigned long flags)
```

### 5.2.1 Loading the Image to Memory

The MINIX kernel is loaded into memory by the function *load_minix_image*. This function loops through all the a.out executables in the kernel image and then loads each one into the domU's allocated memory. The kernel is loaded into the start of the domU's memory space. This makes it easier to apply protection to the kernel's memory pages later on in the setup process.

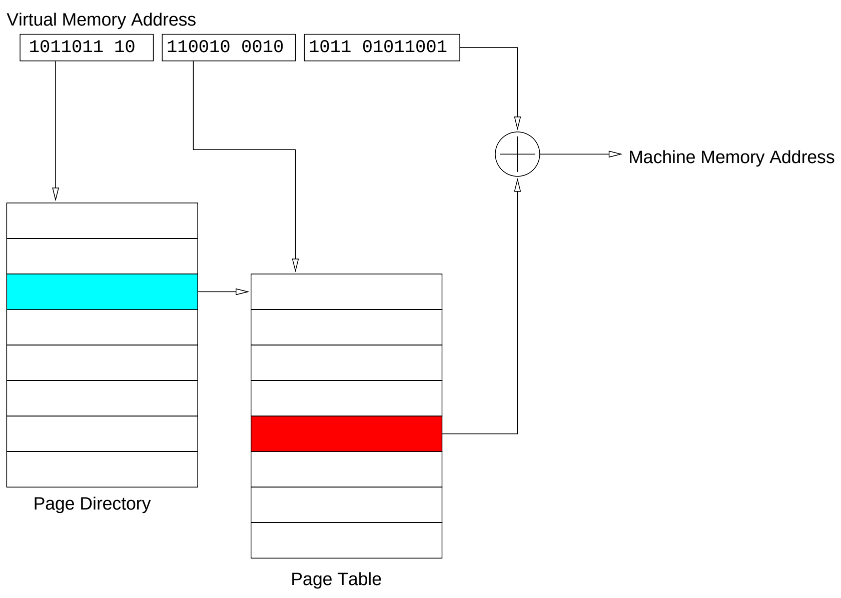

Figure 5.2: Looking up a virtual address in the page table

The header for each process and its position in memory is saved in the start information structure(see 5.2.4),

### 5.2.2 Setting up the Memory Layout

A domU kernel is allocated a number of machine frames on startup. These frames are 4096 bytes in size and correspond to physical memory frames in hardware. These frames must be mapped into a linear memory space by the builder. This mapping is part of a process called paging.

Paging involves a two level mapping scheme. Memory addresses on a 32-bit system are 32 bits long. The first 10 bits work as an index to the page table directory. The page table directory is a list of page tables. The second 10 bits are an index to the page table that is returned from the page table directory. A page table is a list of pages, each 4096 bytes long. Finally, the last 12 bits are an offset in the page returned from the page table.

MINIX's page tables are set up in the *setup_page_table* function. MINIX is different to the other operating systems that have been ported to Xen in that it is completely unaware of paging. This means that all of the allocated memory for the domU must be mapped before the domain starts. The memory space starts at 0x0, as this is where the MINIX kernel expects to be at startup.

MINIX is also different from the other ported operating systems in that it uses segmentation(29).

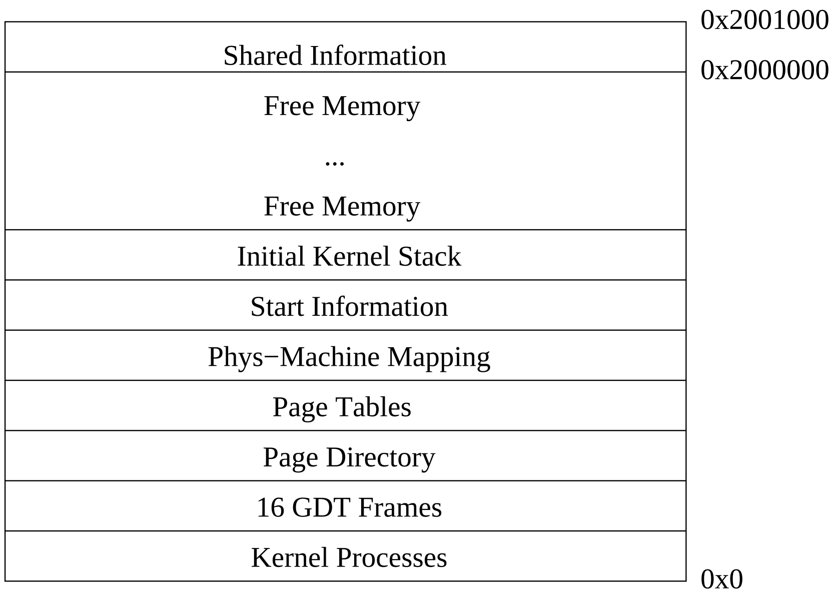

Figure 5.3: Minix domU initial memory layout

Segmentation is not very portable, so most operating systems try to ignore it as best they can. They create a few memory segments that span the whole allocated memory space and use these for all processes. Processes are protected from each other using paging. MINIX allocates five segments for each user space process and three segments for the microkernel. Process segments are defined in the global descriptor table(GDT) and local descriptor tables(LDT). Xen requires that the GDT be placed at the beginning of a page. The GDT therefore, must be created in the builder for MINIX, because MINIX is unaware of the existence of pages.

Xen requires that certain pages are set as read only. These are the pages containing the page directory and page tables, along with the pages for the GDT. Pages also have a protection bit. This bit defines which protection rings may access the page. If set, ring 3 can access the page, otherwise the page can only be accessed by ring 0-2. All kernel pages have this bit unset.

The builder also maps the shared info frame to a high memory address, as specified by the constant FIXED_SHARED_INFO.

Lastly a page is allocated mapping all the pseudo-physical pages to machine frames and the machine to pseudo-physical map in the memory management unit is updated.

### 5.2.3 Global Descriptor Table

Two entries need to be created in the GDT before the domain starts, so that MINIX can enable the new GDT immediately at startup. These entries are for the kernel text segment and kernel data segment. These are both given ring 1 privilege and the maximum limit.

It should be possible to enable this GDT before the domain starts, but this approach failed when attempted. Discussions with the Xen developers shed no light on why it failed, so it was decided that the simplest option would be to initialise the GDT in the builder and enable it immediately on start up.

Xen supplies a default GDT which is used until the new GDT is enabled.

### 5.2.4 Passing Information to the Domain

Some information needs to be passed from the builder to the domain for the domain to operate correctly. MINIX requires more information than most operating systems. As it is a microkernel, information for all the boot processes must be passed in. As the GDT is initialised in the builder, pointers to the GDT must be passed in.

Xen provides a structure, *start_info_t*. This is incorporated into a new structure *minix_start_info_t* alongside another structure *domain_setup_info_t* which contains the MINIX specific startup information. The *start_info_t* structure is at the start of *minix_start_info_t*, and as a result, *minix_start_info_t* can be accessed as a plain *start_info_t* structure.

Listing 5.2: Startup information structures.

```
struct domain_setup_info_t {
        unsigned long msi_vaddr;
        /* virtual address of start_info struct */
        unsigned long fmem_vaddr;
        /* virtual address of free memory */
        unsigned long gdt_vaddr;
        /* virtual address of GDT */
        unsigned long gdt_mfns[NR_GDT_MF];
        /* Machine frame numbers of the GDT */
        process_t processes[PROCESS_MAX];
        image_header_t procheaders[PROCESS_MAX];
};

struct minix_start_info_t {
        start_info_t start_info;
        struct domain_setup_info_t setup_info;
        shared_info_t *shared_info;
        /* vaddr of shared info */
};
```

### 5.2.5 Setting up the CPU Context

The final piece of work the builder does is to set the CPU to a state from which MINIX can boot. The code segment register is set to FLAT_GUEST_CS. The other segment registers are all set to FLAT_GUEST_DS. These constants point to segments which Xen automatically adds to all GDTs.

The interrupt descriptor table is cleared. The hypervisor and failsafe callback instruction pointers, and the general purpose registers are all set to zero. The *esi* register is set to the pseudo-physical address where the start information structure has been placed. The page table base is set to the pseudo-physical address of the page directory table.

The initial stack is placed after the start information structure and extends to the end of that page. It is only used until the GDT is enabled, after which point the MINIX kernel stack is used. The kernel stack is a large area of zeroed memory defined in the data section of the kernel executable.

The domain is started with a DOM0_BUILDDOMAIN hypercall.

# Chapter 6

# Modifications to MINIX

## 6.1 Overview of MINIX Architecture

MINIX(9) is a microkernel. Most of the work that the kernel of an operating system would usually do in ring 0 is moved to ring 1, 2 or 3 (see 4.1.1). Only a minimalistic kernel runs in ring 0. This "micro" kernel takes care of initialisation, context switching, interrupt and exception handling, system calls and interprocess communication.

The other responsibilities of the kernel all take place in separate processes on top of this. These other responsibilities include device drivers, file systems and process and memory management. Processes dealing with devices are called drivers, while the processes dealing with services such as memory management and file systems are called servers.

There are some processes within the the microkernel called the kernel tasks. These share the memory space of the microkernel.

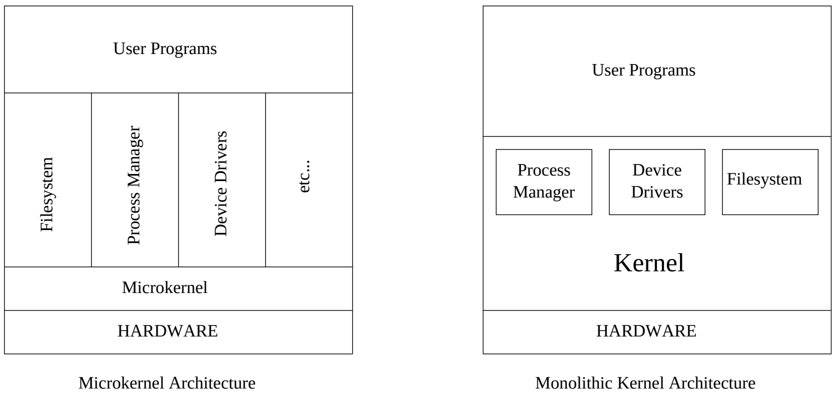

Figure 6.1: Microlithic vs. Monolithic

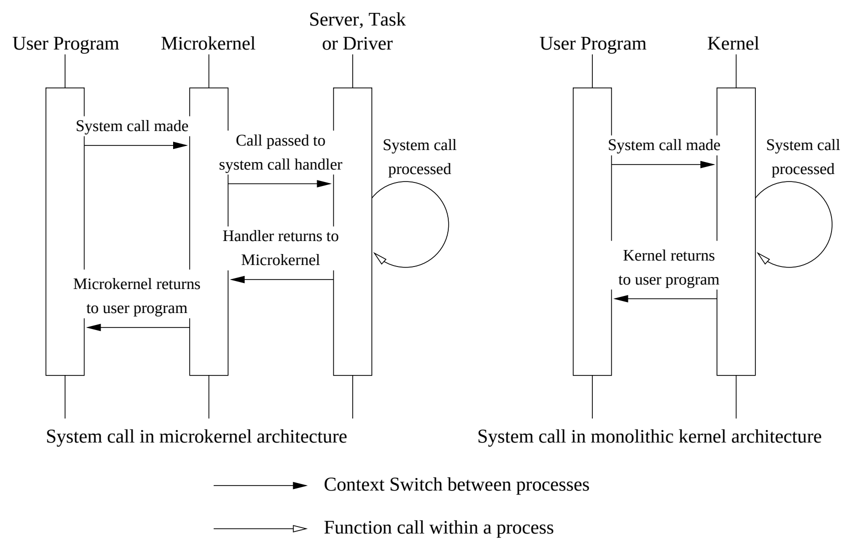
User Program
Microkernel
Server, Task or Driver
System call made
Call passed to system call handler
System call processed
Handler returns to Microkernel
Microkernel returns to user program
System call in microkernel architecture
User Program
Kernel
System call made
System call processed
Kernel returns to user program
System call in monolithic kernel architecture
Context Switch between processes
Function call within a process

Figure 6.2: System calls

This design is very different to the macrokernel approach. In a macrokernel, everything is done within the kernel, and the kernel itself is one large executable file.

### 6.1.1 Advantages and Disadvantages

A microkernel design is much more elegant than a macrokernel design. Separation of responsibilities is much clearer. The design is very modular, so any part of the operating system can be replaced with little trouble.

A microkernel does have the disadvantage of being slower. While all parts of a macrokernel can make system calls directly, the non-kernel parts of a microkernel must make all systems calls through the microkernel. This involves message passing which incurs a time penalty(26).

### 6.1.2 Kernel Tasks, Drivers and Servers

All drivers and servers in the MINIX operating system adhere to the same basic structure. They all have a main execution loop. This takes the form of a *main* function in all tasks and servers except for the four kernel tasks, where it is in the form of a simple function. These functions never return.

Kernel tasks are processes which are part of the MINIX microkernel. As such, they do not have a *main* function. Instead another function is used as its point of entry. These processes are in the kernel for performance reasons, as they have full access to the microkernel's memory space without having to use message passing.

The main execution loop first initialises the task, driver or server. Then it enters a never ending loop of receiving requests and dealing with them.

Listing 6.1: Pseudo Code for MINIX tasks drivers and servers

```
void pseudo_task()
{
  int res;
  message m;

  if (!(ret = init_task())) {
    panic(``Task initialisation failed.'', ret);
  }
  while (TRUE) {
    receive(ANY, &m);

    /* deal with message */
  }
}

int init_task()
{
  /* do task initialisation */
  return 1;
}
```

### 6.1.3 MINIX on Xen

Running on native hardware, the MINIX microkernel will run in ring 0. The kernel tasks run in ring 1. Everything else runs in ring 3.

This changes with Xen, as Xen itself must run in ring 0. Therefore, the microkernel must move to ring 1 and the kernel tasks must move to ring 2.

Porting MINIX to Xen can be split into two tasks. The first is to port the microkernel to run in ring 1. The second is to write the device drivers. While MINIX running natively on hardware uses real devices, on Xen it communicates with the virtual hardware provided by Xen, and therefore new device drivers are required.

## 6.2 Porting the Microkernel

The MINIX microkernel is located under the **kernel/** directory of the source tree. For the assembly components, the same method is used to separate the original minix386 code from the Xen code as is used to separate minix86 from minix386.

For example with **mpx.s**, the entry point of the kernel, there are three conditional preprocessor statements, each including the assembly file for each port depending on what constant is defined.

For the C part, Xen sections are only compiled if the XEN constant[*] is defined.

### 6.2.1 Xen Hypercalls

Xen hypercalls are all defined in **xen.c** except for hypervisor_stack_switch, which is defined in **klibxen.s** as it is a pure assembly subroutine.

All hypercalls make a call to *xen_op* to make the actual hypercall. This assembly subroutine pulls the arguments passed to Xen off the stack, and fires the Xen interrupt with these as the parameters.

Listing 6.2: xen_op subroutine

```
_xen_op:
        mov     eax, 4(esp)
        mov     ebx, 8(esp)
        mov     ecx, 12(esp)
        mov     edx, 16(esp)
        mov     esi, 20(esp)
        int     0x82
        ret
```

It is important to note that any pointers to structures or arrays that are passed as parameters to a hypercall, must be converted from virtual addresses to pseudo-physical addresses. The virt2phys macro, provided by MINIX, is used to do this conversion.

See Appendix A for a description of all the hypercalls and what they do.

Xen hypercalls can only be made from privilege ring 1. This presents a problem as kernel tasks, running in ring 2, need to be able to make hypercalls.

To work around this problem a Xen proxy interrupt handler is created. Any hypercalls required by the kernel tasks check which privilege ring they are running in. If they are running in ring 1, hypercalls can be executed directly. Otherwise the proxy must be used. A *multicall_entry_t* structure is filled with the parameters for the hypercall and the Xen proxy software interrupt is fired.

[*]Defined in **include/minix/config.h**

Listing 6.3: Example hypercall using proxy.

```
PUBLIC int hypervisor_yield()
{
        if (current_ring() != RING1) {
                xen_proxy_op.op = __HYPERVISOR_sched_op;
                xen_proxy_op.args[0] = SCHEDOP_yield;
                xen_proxy_int();
                return xen_proxy_op_ret;
        }
        return xen_op(__HYPERVISOR_sched_op, SCHEDOP_yield);
}
```

The Xen proxy interrupt is trapped by the *xen_proxy* interrupt handler. This handler saves the processor state and calls *do_xen_proxy_op*. *do_xen_proxy_op* makes the hypercall using the parameters specified in the *multicall_entry_t* structure.

Listing 6.4: *do_xen_proxy_op*

```
PUBLIC void do_xen_proxy_op()
{
        xen_proxy_op_ret = xen_op(xen_proxy_op.op,
                                  xen_proxy_op.args[0],
                                  xen_proxy_op.args[1],
                                  xen_proxy_op.args[2],
                                  xen_proxy_op.args[3],
                                  xen_proxy_op.args[4]);
}
```

A more elegant solution would be to modify Xen to allow hypercalls from ring 2. This would, however, requiring modifying the Xen kernel. This is not desirable, as it means all users of MINIX on Xen will have to use a modified Xen kernel as well as modified Xen applications.

### 6.2.2 Initialisation

The first thing that needs to be done when the domain starts is to enable the new GDT (see 5.2.3). Nothing useful can be done without this GDT, as any access to variables will access the wrong memory locations, having potentially disastrous effects. The GDT is enabled by the *xen_init_gdt*[†] function. *hypervisor_set_gdt* is called, with an array of 16 machine frames which constitute the GDT. This array is part of the start information, which is passed to the domain from the builder. See 5.2.4.

[†] **xen.c**

For MINIX to access the start information once the correct GDT is in place, the address must be a virtual address taking the offset of the data segment into account. The value of *esi*(see 5.2.5) is a pseudo-physical address pointing to the start information structure. This needs to be converted to a virtual address before MINIX can use it. *hypervisor_set_gdt* does this conversion. Once it returns, the correct address is placed in the *eax* register. This is later pushed onto the stack for the call to the *xen_cstart* function.

Once the new GDT is in place, the data, stack and extra segment selectors are updated to their correct values and a far jump is made to update the code selector. The new stack is switched in.

*xen_cstart* is called. When *xen_cstart* returns, the flags register is cleared, the kernel stack is saved and the *main* function is called.

### *xen_cstart*

The *xen_cstart* function in start.c creates an environment in which the MINIX main loop can run. Its only parameter is the pointer to the start information structure. This is saved to a global variable. As is a pointer to the shared information frame, for easy access.

Information about the location in memory and size of the kernel process are saved to the kinfo structure. This structure is a global variable and is used later by many functions and macros such as virt2phys.

A call to *xen_prot_init* is made. This function updates the code segment for the kernel, so that the limit doesn't extend past the end of the assembly instructions. Another descriptor is added for an extra segment which extends from memory address 0x0 to the end of memory. This segment is used for copying data between processes. The original MINIX function, *prot_init*, also added a segment descriptor for the microkernel's data segment, which limited the range of memory the microkernel could modify. In MINIX for Xen, this is omitted as the microkernel needs to access the shared information frame which is placed high in the address space.

Originally in MINIX, each process had its own LDT. Limitations imposed by Xen would make giving each process its own LDT very difficult as the LDT must be page aligned(7) and MINIX is unaware of paging. The solution is give each process some descriptors in the GDT. The GDT in MINIX on Xen is much larger than the original GDT used in MINIX. It occupies 16 frames, of 4096 bytes each. Each entry in the GDT is 8 bytes long. Therefore there is enough room for 8192 entries. The default maximum number of processes in MINIX is less than 100.

Each process needs 2 segments, one for text and one for data, and 3 optional segments. So, 500 is the maximum number of segments we could need. There is plenty of room to store all segments descriptors in the GDT. The segment descriptor indexes for each process are allocated in *xen_prot_init* though the actual segment descriptors themselves are not built until the processes are created.

#### *main*

The *main* function does the final piece of the initialisation before the process scheduler starts. All entries in the process table are first zeroed. Then the boot processes, the kernel tasks, the servers and the drivers are initialised. Which processes to initialise are specified by the *image* array in **table.c**. The process table entries for each of these processes is populated using information from the header, passed in with the start information. Their segments are then initialised by the *alloc_segments* function.

The *alloc_segments* function has not changed much from its original MINIX implementation except that GDT indexes are passed to the *init_seg* functions instead of a pointer to the entries. This is because, in Xen, the GDT is read-only and can only be updated through the *update_descriptors* hypercall.

Once all boot processes have been initialised, a banner message is printed, and scheduling begins.

*main* is also responsible for calling *init_events* which initialises events and interrupts for Xen.

### 6.2.3 Events, Interrupts & the Control Interface

While all events in minix386 could be handled by the IDT, in Xen, different types of events must be handled in different ways.

#### Exceptions and Software Interrupts

Interrupts from the CPU are handled in a similar fashion to how they are handled in minix386. There is a table of interrupt traps in **evtchn.c**. These traps are passed to Xen through the *hypervisor_set_trap_table* hypercall.

The same traps can be used for MINIX on Xen as are used for minix386.

The structure of an interrupt handler is uniform. When an exception or software interrupt occurs, the current values of the *eip*, *cs* and *flags* registers are pushed onto the stack. If the current process is running in ring 2 or 3 the *esp* and *ss* registers are pushed also and replaced by the values saved by *hypervisor_switch_stack*. The code of the specified interrupt trap is entered at this point(30).

The trap first saves the CPU's current state by jumping to the *save* subroutine. This subroutine manipulates the stack so that when the calling function returns, the restart or restart1 subroutine will be entered. Once the CPU state has been saved, the trap runs its own specific code and returns.

When the trap returns, it jumps to the restart or restart1 subroutine, depending on whether the interrupt occurred while the CPU was running in ring 1 or not. This function restores the CPU state from the stack and restarts the execution of the the interrupted process.

### Interrupts from Xen

All hardware events in Xen come through the event channels. There are three types of event that go through the event channel.

- Virtual Interrupts
- Physical Interrupts
- Interdomain communication

Of these, only virtual interrupts and interdomain communication are discussed here. MINIX will not be run as a dom0 kernel, so handling physical interrupts is unnecessary.

On initialisation, the hypervisor and failsafe callbacks are set. This is done in *init_events*. The structure of these callbacks is very similar to that of MINIX's interrupt handlers. The save subroutine is called, events are disabled, *do_hypervisor_callback* is called, events are reenabled and the callback returns.

Theoretically, events in Xen can be reentrant. On real x86 hardware, when an interrupt occurs interrupts are disabled until *iretd* is called. Xen does not automatically disable events when it sends an interrupt. This means that the hypervisor callback must do all disabling and reenabling of events. Events are disabled by setting shared_info->vcpu_data[0].evtchn_upcall_mask to non-zero, and reenabled by setting this to zero. This can only be done when the kernel address space is being used, so disabling must occur after the call to *save*, and reenabling must

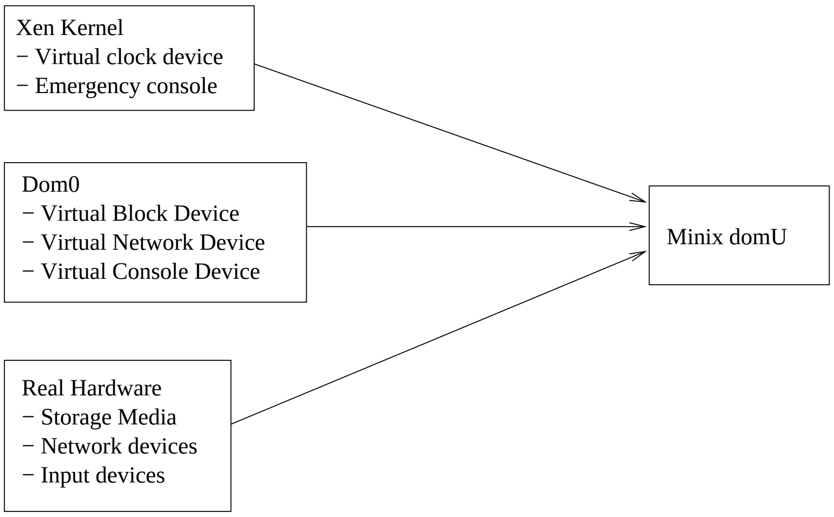

Figure 6.3: Sources of interrupts

occur before the callback returns. It is during the call to *save* and the call to *return*[‡] that reentrance may occur. Reenterring callbacks can overflow the kernel stack, and cause the system to crash.

Three arrays control what happens when an event occurs. These are *evtchn_to_irq*, *virq_to_irq* and *handlers*. When a task wishes to register a handler for a virtual irq or event channel, it must first bind that virtual irq or event channel to an irq. These irqs are internal to the MINIX on Xen port and have no relevance to Xen itself. Once the task has bound the virtual irq or event channel, it will receive an irq which it can use to register their handler.

Finally it must enable the irq to allow events to start occurring.

Events are dependant on the *vcpu_data* structure and the evtchn variables in the shared information. Each event has a event channel associated with it. For an event to occur.

- *shared_info->vcpu_data[0].evtchn_upcall_pending* must be 0.
- *shared_info->vcpu_data[0].evtchn_upcall_mask* must be 0.
- The *n*th bit of *shared_info->evtchn_pending[32]* must be 0, where n is the number of the event channel for that event.
- The *n*th bit of *shared_info->evtchn_mask[32]* must be 0.
- The ($n >> 5$) bit of *shared_info->evtchn_pending_sel* must be 0.

[‡]Invoked by ret. See 6.2.3

Listing 6.5: Interface for registering irq handlers.

```
/* from evtchn.c */
PRIVATE unsigned int evtchn_to_irq[NR_EVENT_CHANNELS];
PRIVATE unsigned int virq_to_irq[NR_VIRQS];

PRIVATE struct irq_handler_t handlers[NR_IRQS];

PUBLIC unsigned int bind_evtchn_to_irq(unsigned int evtchn);
PUBLIC unsigned int bind_virq_to_irq(unsigned int virq);

PUBLIC unsigned int add_irq_handler(unsigned int irq,
                                    void (*handler)
                                    (unsigned int,
                                    struct stackframe_s *))
PUBLIC unsigned int enable_irq_handler(unsigned int irq)
```

Once all these conditions are met, an event can occur. When an event occurs, *hypervisor_callback* calls the function *do_hypervisor_callback*. *do_hypervisor_callback* will determine the irq of the event which occurred, lookup the corresponding handler for this irq in the handlers array and then execute it.

#### Interrupts from Dom0

Interaction between dom0 and domUs takes place on the control interface. The control interface is a special event channel, over which a domU can communicate with its virtual console, virtual block devices or virtual network device. In MINIX on Xen, the control interface has become a new kernel task. See 6.2.4 for details.

### 6.2.4 Kernel Tasks

Kernel tasks are processes that are part of the MINIX microkernel. They do not have a main function. Their point of entry is a normal function which is specified in the images array in table.c. They run in privilege ring 2.

#### Clock Task

The clock task requires very little modification to work with Xen. The initialisation function has to be replaced so that the handler is registered with the event channel. Once this is done, the clock works perfectly.

### Control Interface Task

While Xen uses the event channels to notify the domU kernel when an event has occurred, no data can be attached to this event. Only the event channel on which the event occurred can be determined.

Therefore another mechanism is needed to transfer actual data to and from the virtual device causing the event. For this the domain controller is used. The control interface is used for communication with the domain controller.

The control interface is a structure at a half frame offset[§] from the start of the shared information frame. This structure consists of two transaction rings and four counters. One transaction ring is for receiving control interface messages and the other is for transmitting control interface messages. These each have two counters associated with them, to count the number of requests and responses made on the ring.

Control messages are structures that can hold up to 60 bytes of data. They also have a type field and a subtype field. There is a new MINIX interprocess communication message type to facilitate passing of these messages between processes, as no other message type is large enough[¶].

Listing 6.6: Control interface structure.

```
typedef struct {
        control_msg_t tx_ring[CONTROL_RING_SIZE];        /*
0: guest -> controller */
        control_msg_t rx_ring[CONTROL_RING_SIZE];        /*
512: controller -> guest */
        CONTROL_RING_IDX tx_req_prod, tx_resp_prod;
/* 1024, 1028 */
        CONTROL_RING_IDX rx_req_prod, rx_resp_prod;
/* 1032, 1036 */
} PACKED control_if_t;          /* 1040 bytes */
```

In MINIX, the control interface is a new kernel task. When MINIX is initialising, the control interface task registers an interrupt handler for the domain controller event channel which was passed in from the builder through the start information structure. The task then loops continuously, waiting for messages from other processes.

When an event occurs on the domain controller event channel, the interrupt handler, *ctrl_if_interrupt*

[§]2048 bytes

[¶]see **include/minix/ipc.h**

is called. This function simply sends a notification message to the control interface task which will be received by it as a HARD_INT message. It is important to note that the interrupt handler will run in privilege ring 1, while the control interface task runs in privilege ring 2. This means that any real work must be done within the task, as to do it from the interrupt would mean that any messages sent or received would have an unpredictable source due to the source being taken as the process pointed to by *proc_ptr* which could be any process when an event occurs.

Messages to the control interface can have seven types. They are:

- HARD_INT
- CTRLIF_REG_HND
- CTRLIF_UNREG_HND
- CTRLIF_SEND_BLOCK
- CTRLIF_SEND_NOBLOCK
- CTRLIF_SEND_RESPONSE
- CTRLIF_NOP

**HARD_INT** messages are used to notify the control interface that an event needs to be dealt with. When a message of this type occurs, the control interface task tries to send all outgoing control interface messages waiting to be sent and passes any received messages to their registered handler if one exists.

**CTRLIF_REG_HND & CTRLIF_UNREG_HND** messages register and unregister handlers for different types of control interface messages. A handler is simply a process number. When a control interface message is received, it is forwarded to the process indicated by this process number.

**CTRLIF_SEND_BLOCK, CTRLIF_SEND_NOBLOCK & CTRLIF_SEND_RESPONSE** messages are all used to send control interface messages to the domain controller. CTRLIF_SEND_NOBLOCK will try to send a message on the transmitter ring but will return if it is not possible. CTRLIF_SEND_BLOCK will do the same but block until the message sends. CTRLIF_SEND_RESPONSE puts a message on the receiver ring to simulate a message from the domain controller.

**CTRLIF_NOP** messages are used for debugging.

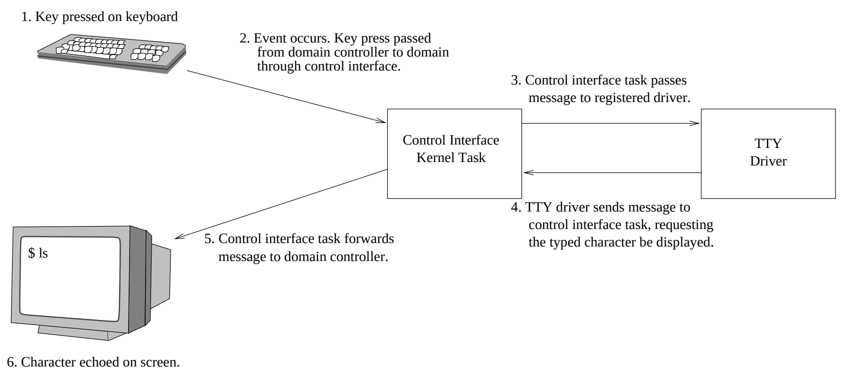
1. Key pressed on keyboard
2. Event occurs. Key press passed from domain controller to domain through control interface.
3. Control interface task passes message to registered driver.
Control Interface Kernel Task
TTY Driver
4. TTY driver sends message to control interface task, requesting the typed character be displayed.
5. Control interface task forwards message to domain controller.
$ ls
6. Character echoed on screen.

Figure 6.4: Handling a keystroke

## 6.3 Console Driver

### 6.3.1 Xen Virtual Console

Xen provides each domU with a virtual console. Users in the dom0 can access this console using the command *xm console <domid>*. To the user, the console will look like a telnet session.

This console can be accessed by the kernel through the control interface transaction rings. When the user pressed a key, a domain controller event occurs, and a control interface message with type CMSG_CONSOLE is placed on the receiving ring. This message will contain the key or keys input.

To write to the console, a control message with a type of CMSG_CONSOLE must be placed on the transmission ring and a notification is sent to the domain controller event channel. When the domain controller receives this notification it will check to see if there are any new messages on the transmission ring, and if there are, it will send them on to the desired destination.

### 6.3.2 The MINIX TTY Driver

The tty driver in MINIX contains drivers for three types of terminal, hardware consoles, rs232 serial consoles and pseudo-terminals. The main driver function is *main* in **drivers/tty/tty.c**. This function calls the initialisation function and begins listening for messages from other processes.

The initialisation function creates a number of terminals depending on the values of the constants NR_CONS, NR_RS_LINES and NR_PTYS. These constants also decide how many terminals of each type are created.

The hardware specific drivers for each terminal type are in separate files. The hardware console, rs232 serial console and pseudo terminal drivers are in **console.c**, **rs232.c** and **pty.c** respectively. The hardware console driver also contains **keyboard.c**, as the input is separate to the display in the case of a hardware console.

Each of these terminal drivers define a set of functions for reading and writing data to the terminal. When a terminal is being initialised, the initialisation function of its terminal type is called. The reading and writing functions for that terminal type are assigned to function pointers in the tty structure for that terminal. From this point onward, all terminals are treated equally by the tty driver.

#### Xen Terminal Driver

For the Xen virtual console, a new hardware specific terminal driver is needed. This is contained in the file **xencons.c**. A new constant is also created, NR_XEN_CONS. The driver itself consists of an initialisation function, a set of read and write functions and an interrupt handler function.

The initialisation function, *xencons_init*, registers the tty process as the handler for CMSG_CONSOLE messages. It also initialises two queues, the input buffer and the output buffer. These are used by the reading and writing functions to buffer data so that a call to the control interface does not need to be made every time a character is printed or a key is pressed.

The interrupt handler function is called whenever the tty process receives a HARD_INT message from the control interface.

Modifications are needed in the rest of the driver, to allow for the hardware console to be turned off. Traditionally, a computer would always have had a hardware console, so turning it off would be pointless but with Xen there is no hardware console, so it must be turned off. To do this, the console and keyboard drivers are wrapped in preprocessor statements, which will only include the code if NR_CONS is greater than zero.

### 6.3.3 Debugging Console

The virtual console supplied by Xen is not the only place that output can be sent. There is also a debugging console that can be written to with a simple hypervisor call. This is used within the microkernel only, as other processes are unable to make hypervisor calls.

The hypervisor call for writing to the debug console is *hypervisor_console_write*. It is only ever called by the *kputc_xen* function in **kernel/utility.c**, which in turn is only called by *xen_kprintf*.

## 6.4 Block Device Driver

### 6.4.1 MINIX Block Devices

As with all other drivers in MINIX, a block device driver is its own independent process. However block device drivers differ from other drivers because they have no message receiving loop as other drivers do. Instead, a *driver* structure, containing pointers to the driver's hardware specific functions, is passed to the *driver_task* library call, which will run the event loop.

This allows for device independent code, such as that for buffering, to be shared among all the block device drivers.

### 6.4.2 Xen Block Interface

The Xen virtual block device interface uses two channels of communication for transactions between domU and the backend driver.

It uses the control interface to communicate with the domain controller. These transactions are used to connect or disconnect to the virtual block device interface or to query the status of a virtual block device. It is also used to setup the infrastructure to allow the second mode of communication.

The second mode of communication consists of a shared memory frame and a event channel. The shared memory frame is specified when the virtual block device interface is connected. This shared frame contains a communication ring, much like those in the control interface. Requests and responses are both placed on a single ring however. The event channel is set by a block interface query to the control interface.

A request contains an operation to be performed, the device to use, the first sector to use and a list of frames and sections.

No actual data is transferred over the rings, as a DMA like mechanism is used. When the domU wants to read or write some memory, a request structure is created, specifying where on disk to read or write and the memory location which is the destination or source of the data to be transferred. This allows requests to be batched, and is much faster than actually passing all the data over the block interface directly.

The request is then put onto the communication ring and a notification is sent to the virtual block device event channel. When the request has been fulfilled or an error has occurred, a response is put onto the communication ring. An event then occurs on the virtual block device event channel.

### 6.4.3 Current Status

As of the time of writing, the block device driver is still in development.

Chapter 7

# Recommendations for Further Development

## 7.1 Port to Xen 3.0

This project focuses on porting MINIX to the Xen hypervisor, version 2.0.7. During the course of the project, Xen 3.0 was released. Version 3.0 has significant differences to version 2.0.7, especially in regard to communication with the virtual block device and virtual network device.

There has recently been a discussion on the Linux kernel mailing list(31) regarding a new *Virtual Management Interface* (VMI)(32) to standardize the interface which all hypervisors expose. It has been proposed by VMWare, and responses have been mixed, but if VMI is accepted by the Xen development team, then VMI will be another target for a MINIX port.

## 7.2 Replace Custom Builder with ELF Bootloader

While using a custom builder for MINIX works well, it is not an elegant solution. It requires that the Xen libraries be patched, which means for each new version of Xen a new patch will possibly be needed. Also, unless the patch is accepted by the Xen developers, users will not be able to use a binary distribution of Xen to run MINIX.

A better solution would be to create a minimal ELF bootloader that would chain load MINIX into memory and initiate execution. With this, MINIX would be capable of running on a vanilla Xen installation.

# Chapter 8

# Conclusion

Throughout this document I have hoped to effectively illustrate the process of porting an operating system to run upon the Xen hypervisor. In conjunction with the Xen interface documentation and the Xen source code, this report should help others understand how Xen works and the tasks required to port an operating system to run on a Xen system.

Because of its size and simplicity, MINIX was a excellent choice for this project. The source code was well commented, and the most complicated parts were well explained by its associated text book, *Operating Systems: Design and Implementation*(9). The book would however benefit from a chapter or section explaining the boot loading code, as this initially presented a very difficult learning curve due to the code's complexity and lack of documentation.

While the MINIX kernel itself did present an excellent kernel to work with, it was not without its own frustrating quirks, without which development would have been simplified hugely. The fact that MINIX is an a.out kernel and can only be compiled with the Amsterdam Compiler Kit resulted in much work that otherwise could have been avoided.

More application support on MINIX would also have eased development considerably. The lack of basic tools like rsync made the build process far more complicated than it needed to be.

Porting a microkernel also raised problems with Xen that would not have occurred with a monolithic kernel. Monolithic kernels perform all their privileged instructions in kernel mode. Microkernels do some privileged instructions from user mode. This presented a problem because Xen only allows calls from kernel mode. I have outlined a possible solution to this in section 6.2.1.

Overall, MINIX was very pleasant to work with. With the kernel[*] itself at just 7104 lines of

[*]The device drivers are not part of the kernel in a microkernel.

code, understanding its inner working was much simpler than that of monolithic kernels such as Linux or freebsd, each of which contain millions of lines of code. With the Xen extensions, MINIX only grew to 9435[†] lines.

[†]16692 LOC including tty and block device driver.

# Appendix A

# Xen Hypercalls

```
int hypervisor_console_write (char *string, int length)
```

Write *length* characters of the string *string* to the emergency console. See 6.3.3.

```
int hypervisor_set_gdt (unsigned long *framelist, int entries)
```

Set the array of frames pointed to by *framelist* as the GDT. *framelist* has a pseudo-physical address, as this operation must be performed before virtual addresses can be used. See 6.2.2.

```
int hypervisor_update_descriptor (unsigned long index,
                                  struct segdesc_s *segdp)
```

Update a segment descriptor in the GDT. *index* specifies the entry to update. *segdp* points to a segment descriptor structure which will be used to update the descriptor. See 6.2.2

```
int hypervisor_set_trap_table (trap_info_t *traps)
```

Copy the contents of *traps* into the interrupt descriptor table for the domain. See 6.2.3.

```
int hypervisor_set_callbacks (unsigned long event_selector,
                              unsigned long event_address,
                              unsigned long failsafe_selector,
                              unsigned long failsafe_address)
```

Set *event_address* and *failsafe_address* as the entry points for the hypervisor and failsafe callbacks respectively. *event_selector* and *failsafe_selector* specify the code segments which should be loaded when the callbacks occur. See 6.2.3.

```
int hypervisor_event_channel_op (evtchn_op_t *t)
```

Send operation *t* to the event channel. Operations include those to bind, close and send data to event channels. See 6.2.3.

```
int hypervisor_xen_version ()
```

Request the hypervisor's version. Used to force a hypervisor callback.

```
int hypervisor_shutdown ()
```

Shutdown the calling domain.

```
int hypervisor_yield ()
```

Yield the processor to another domain. Used when there is no work to be done for the calling domain.

Appendix B

# Installing MINIX on Xen

## B.1 Patching the Xen Libraries

### B.1.1 Downloading and Applying the Patch

Download the newest patch from https://svn.skynet.ie/~ikelly/MinixOnXen/patches/. Extract the Xen 2.0.7 sources and apply the patch.

```
# tar zxf xen-2.0.7-src.tgz
# cd xen-2.0/
# cat ../minix_builder-0604112115.patch | patch -p1
patching file tools/libxc/Makefile
patching file tools/libxc/minixa.out.h
patching file tools/libxc/minix.h
patching file tools/libxc/xc.h
patching file tools/libxc/xc_minix_build.c
patching file tools/python/setup.py
patching file tools/python/xen/lowlevel/xc/xc.c
patching file tools/python/xen/xend/XendDomainInfo.py
patching file tools/xfrd/Makefile
```

### B.1.2 Compiling Xen with the Patch

To compile and install Xen with the new patch is only a matter of running the following.

```
# make install-tools
```

This assumes you already have Xen installed from this source tree. If not you need to do so.

```
# make world
# make install
```

Xend will need to be restarted for the changed to take effect.

```
# /etc/init.d/xend restart
```

## B.2 Downloading MINIX for Xen

A snapshot of MINIX for Xen is available at http://svn.skynet.ie/~ikelly/MinixOnXen/snapshots/.

There are two subversion repositories. One at secure.bleurgh.com and one at svn.skynet.ie. secure.bleurgh.com is the most current. svn.skynet.ie is a mirror. To checkout from the repositories, do the following.

```
# svn co https://secure.bleurgh.com/svn/minix/trunk
# svn co https://svn.skynet.ie/svn/MinixOnXen/minix
```

You can then copy the code into your MINIX install for compilation with using. MINIX does not have a subversion client yet.

There is also a cvs repository that is updated whenever a commit is made to the svn server at secure.bleurgh.com. MINIX has a cvs client, so it can checkout directly from this. To checkout from cvs do the following.

```
# cvs -d :pserver:anoncvs:@bleurgh.com:/var/lib/cvs login
# cvs -d :pserver:anoncvs:@bleurgh.com:/var/lib/cvs co minix
```

## B.3 Compiling MINIX

MINIX can only be compiled by MINIX. For this reason you will have to have MINIX either running on hardware or in an emulator to compile MINIX for Xen.

Once you have downloaded and extracted the MINIX for Xen sources, you need to install the some new headers. The easiest way to do this is just to install all the headers again.

```
# cd <minixforxen>/includes
# make install
```

**WARNING:** If you have preexisting changes in the MINIX includes that you wish to keep, you will have to install the headers manually. This requires that you copy the **xen/** subdirectory of the includes directory to **/usr/include**. You will also need modify **/usr/include/minix/com.h** and **/usr/include/minix/config.h** to match the Xen versions.

You must move some of the original MINIX kernel source out of the way and move the new versions in. The directories which you need to move are the kernel directory and the tty driver.

```
# cd /usr/src
# mv kernel kernel.old; mv drivers/tty drivers/tty.old
```

Then move the new directories into place. You may need to run mkdep on the directories. The compilation will fail if .depend files are missing.

```
# cp -r <minixforxen>/kernel /usr/src
# cp -r <minixforxen>/drivers/tty /usr/src/drivers
# cp -r <minixforxen>/drivers/xenvbd /usr/src/drivers
# cd /usr/src
# mkdep kernel; mkdep drivers/tty; mkdep drivers/xenvbd
```

You are now ready to compile the kernel.

```
# cd /usr/src/tools
# make image
```

The result will be a file **image** in **/usr/src/tools**. Copy this to your Xen host, and run it using the following configuration.

```
# Kernel image file.
kernel = "/etc/xen/minix"

# The domain build function. Default is 'linux'.
builder='minix'

# Initial memory allocation (in megabytes) for the new domain.
memory = 20

# A name for your domain. All domains must have different names.
name = "MinixTester"

# Number of network interfaces. Default is 1.
nics=0

extra="""rootdev=896
ramimagedev=896
ramsize=0
processor=686
bus=xen
video=xen
chrome=color
memory=930000:a00000
label=XEN
controller=c0
image=xenimage
"""
```

# Bibliography

[1] “The User-mode Linux Kernel Home Page.” http://user-mode-linux.sourceforge.net/.

[2] “The Xen virtual machine monitor.” http://www.cl.cam.ac.uk/Research/SRG/netos/xen/.

[3] S. Shankland, “Novell follows Red Hat with Xen announcement.” http://news.zdnet.co.uk/software/linuxunix/0,39020390,39259033,00.htm.

[4] “AMD’s Virtualization Solutions.” http://enterprise.amd.com/Solutions/Consolidation/virtualization.aspx.

[5] “Intel Virtualization Technology.” http://www.intel.com/technology/computing/vptech/.

[6] “Xen virtualization quickly becoming open source ’killer app’.” http://searchopensource.techtarget.com/originalContent/0,289142,sid39_gci1152219,00.html.

[7] The Xen Team, “Xen Interface Manual.” http://www.cl.cam.ac.uk/Research/SRG/netos/xen/readmes-2.0/interface/interface.html.

[8] “The MINIX 3 Operating System.” http://www.minix3.org/.

[9] A. S. Tanenbaum and A. S. Woodhull, *Operating Systems, Design and Implementation*. Prentice Hall, 2006.

[10] “The GNU Hurd.” http://www.gnu.org/software/hurd/hurd.html.

[11] “VMWare.” http://www.vmware.com.

[12] "Microsoft Virtual PC." http://www.microsoft.com/windows/virtualpc/default.mspx.

[13] "Denali: Lightweight virtual machines for distributed and networked systems." http://denali.cs.washington.edu/.

[14] "QEMU." http://www.qemu.org/.

[15] "Bochs IA-32 Emulator Project." http://bochs.sourceforge.net/.

[16] "Solaris Zones." http://www.sun.com/bigadmin/content/zones/.

[17] "VMware ESX Server." http://www.vmware.com/products/esx/.

[18] "Subversion." http://subversion.tigris.org/.

[19] "The Amsterdam Compiler Kit." http://tack.sourceforge.net/.

[20] "RSync." http://samba.anu.edu.au/rsync/.

[21] "Concurrent Versions System." http://www.nongnu.org/cvs/.

[22] A. Ho, "Pervasive Debugging." http://xenbits.xensource.com/xen-2.0.hg?cmd=file;filenode=52bcf66776cf12be2222793203fbe8c46b3a9486;file=docs/pdb.txt.

[23] "GDB: The GNU Project Debugger." http://www.gnu.org/software/gdb/.

[24] "Multiboot Specification Manual." http://www.gnu.org/software/grub/manual/multiboot/.

[25] *Intel Architecture Software Developer's Manual Volume 3: System Programming*, p. 112. Intel Corporation.

[26] "Microkernel Wikipedia Article." http://en.wikipedia.org/wiki/Microkernel.

[27] "Plan 9 from Bell Labs." http://cm.bell-labs.com/plan9/.

[28] J. Levon, "domU Builder RFC." http://lists.xensource.com/archives/html/xen-devel/2006-02/msg00987.html.

[29] *Intel Architecture Software Developer's Manual Volume 3: System Programming*, pp. 63–79. Intel Corporation.

[30] *Intel Architecture Software Developer's Manual Volume 3: System Programming*, pp. 141–198. Intel Corporation.

[31] Z. Amsden, "VMI RFC on the Linux kernel mailing list." http://lkml.org/lkml/2006/3/13/140.

[32] "Virtual Machine Interface (VMI) Specifications." http://www.vmware.com/interfaces/vmi_specs.html.