University of Alberta

MIME: A Virtual Machine Operating System

by

Anthony Njoroge Mutiso

A thesis
submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree
of Masters of Science

Department of Computing Science

Edmonton, Alberta
Spring 1991

ISBN 0-315-66595-5

We are all captive of
the pictures in our heads -
Our belief that the world we experience
is the world that really exists
......**Walter Lippmann**

All that we see or seem
is but a dream within a dream.
A Dream within a Dream
......**Edgar Allan Poe**

Abstract

The virtual machine paradigm for extending computer user multiplicity has a history that spans over three decades. Over the years, virtual machine operating systems have faded in and out of research focus. A great part of this research has tended to concentrate on large and medium size computer systems. The work presented in this thesis centers on the research and development concerns of virtual machine systems. The emphasis of the work is on virtual machine systems for the personal workstation environment. The improved computational power and cost effectiveness of personal computer systems combined with the lack of adequate computer software that support the research and education of operating systems topics on personal computer systems is the primary motivation for this work.

MIME is an experimental implementation of a virtual machine operating system designed to operate on a workstation-class computer. It is currently being developed on a Sun Microsystems Incorporated Sun-3 workstation. The central component in MIME, as in any virtual machine operating system, is the virtual machine monitor (VMM). The VMM is responsible for the transparent virtualization of all hardware resources in the host architecture. MIME's VMM attempts to project a complete virtual machine interface, but is limited by the hardware to virtual machine interfaces that are smaller than the physical machine interface. This restricts MIME from generating a recursive virtual machine hierarchy, while permitting all operating systems that can execute on the physical machine to execute on a virtual machine. In this thesis we discuss the weaknesses that are present in the current implementation of MIME and suggest hardware changes. These changes will improve the efficiency of the virtual machine operating system. They will also

allow MIME to project a more complete virtual machine interface. The work on MIME is intended as the foundation for further research in operating systems and as a software teaching aid in operating system courses.

Acknowledgements

I would like to acknowledge the great effort, sound support and effective advice that my two supervisors, Dr. Wlodek Dobosiewicz, and Dr. Pawel Gburzynski have shown and given me. Your patience with this work has been recognized and appreciated. To Linda Needham, our librarian, I am enormously in debt for all the time and effort that you spent helping me find my material. To those in Operations, for all the help you all have offered me and the difference you have made during the course of my program, I am graciously grateful. I thank you for making my stay and experience in this department a comfortable and enjoyable one. To Elke, Bob and the many others who took the time to read parts of my work to help proof versions of it, I take immense pleasure in acknowledging your time, support and sincere remarks. Elke, I thank you for those many nights you put up with me. I will always remember. Finally I acknowledge my family, for all that they endured while I went through this program. I love you all!

Without the help of all of you, I would not have been able to accomplish my goals with this work. I thank you all, and ask that you remain, blessed in all your endeavors.

" *May the counsels of effortless accomplishment, walk your individual journeys, a partner in your successes......*"

Contents

1 Introduction 1

1.1 Background 1

1.1.1 Virtual Resources In System Development 2

1.1.2 Virtual Resources In System Research 2

1.1.3 Systems For Education 3

1.1.4 Pseudo-Devices 4

1.1.5 Virtual Machine Operating System, A Definition 5

1.2 History Of Virtual Machine Operating Systems 6

1.2.1 Past To Present 7

1.3 Architectural Requirements For A Virtual Machine Operating System. 7

1.3.1 Formal Models 8

1.3.2 What Is MIME 9

2 Machine Architecture 10

2.1 Motorola MC68020 Microprocessor 10

2.1.1 Exception Processing in a MC68020 10

2.1.2 MC68020 Exception Stack Frames 12

2.1.3 Instruction Set And Addressing Modes 14

2.1.4 Interrupt Levels And Bus Cycles 14

2.2 Sun-3 Architecture 16

2.2.1 VMEbus, Devices And Peripherals 17

2.2.2 Sun-3 Memory Management Unit 18

3 A Virtual Machine Operating System 21

3.1 Hardware Management 21

3.1.1 CPU 22

3.1.2 Virtual Memory Support 23

3.1.3 Peripheral Devices 24

3.2 The Design Of A Virtual Machine Operating System 26

3.2.1 The Virtual Machine Monitor 26

3.2.2 Virtual Devices 28

3.3 Virtual Machines 29

3.4 Virtual Machine Context 30

3.4.1 Overhead In A Virtual Machine 33

4 Virtual Memory In A Virtual Machine 34

4.1 Memory Resources In A Virtual Machine 35

4.2 Direct Address Translation 37

4.3 Hardware Virtualizer 39

4.3.1 Impact And Design 40

4.4 Virtual Memory In MIME 42

4.4.1 Virtual Memory Context Partitions 42

4.4.2 Virtual Devices In Virtual Memory 44

4.4.3 Memory Faults And Paging 44

5 Other Resources In A Virtual Machine Environment 47

5.1 Time Of Day Clocks 48

5.2 Network Support 48

5.2.1 Network Support in MIME 49

5.3 Disk Subsystem 49

5.3.1 Mini-disk Partitions 50

5.3.2 Dynamic Disk Partitions 50

5.4 Serial Devices 53

5.5 Debugging Tools 53

6 Conclusion **55**
6.1 Related Work 56
6.1.1 VM/CMS 56
6.1.2 XINU 57
6.1.3 CPM 58
6.2 Further Work 58
6.2.1 MIME In Education 59
6.2.2 MIME In Further Operating System Research 60
6.2.3 Tomorrow's MIME 61
6.3 Closing Remarks 63

A Exception Vector Table **68**

B Implementations, Details And Specifics **70**
B.1 Implementation Shortfalls In MIME 70
B.1.1 Implementation Platform Details 71
B.2 A Simple Hardware Virtualizer Implementation 71
B.2.1 HV Design 72
B.2.2 HV Operation 74
B.2.3 Summary 75

C Privileged And Trap Instruction Formats **76**

D Virtual Machine Monitor Code Fragments **78**
D.1 VMM Data Structures 78
D.1.1 CPU Context Structure 78
D.1.2 Virtual Machine User Structure 79
D.2 VMM Sample Code Fragments 80

List of Tables

2.1 User Mode Privileged And Trap Instructions. 11

3.2 Machine **CPU** context structure. 22

A.3 Exception Vector Table. 69

C.4 Privileged And Trap Instruction Formats 77

List of Figures

1 Conventional extended machine. 4

2 Virtual machine. 6

3 Dual dimensions of virtual machine environments. 8

4 Basic Exception stack frame 12

5 Sun-3 memory management unit: (Pmeg number translation) . . . 18

6 Sun-3 memory management unit: (Physical address translation) . . 19

7 Recursive *f-maps*. 38

8 A Process exception and VM-*fault*. 39

9 The VMTAB and VMCB. 41

10 Simplified Memory Map. 46

Chapter 1

Introduction

Virtual machine operating systems for large computer facilities have been commonplace experiences for many year. This thesis presents a virtual machine operating system called MIME that is being developed for the personal workstation platform. The primary intention of this work is to provide a workstation environment that is suitable for experimental and developmental operating system work. The system is also intended as a practical teaching platform for operating system principles. The sections to follow in this chapter will provide some background and motivation to the subject of virtual machine operating systems.

1.1 Background

In the past, software and hardware resources had to be dedicated when designing experimental computer systems. This made experimental system design both an expensive and difficult task. As the systems being developed became larger and more complex, sophisticated environments were created to support the design and testing of the new systems. These environments came in several different forms that were based on their principal objective. Some were designed to support efficient debugging of new hardware systems using older or different hardware. Others were designed to provide new features in an older hardware platform so that software development for a new hardware design and the development of the new hardware could take place simultaneously. In effect these design environments created simulators of the new systems. Over the years, the trend has been for these simulators to become larger, more complex and more complete in their simulation of the real system.

1.1.1 Virtual Resources In System Development

As new systems become more complex, the task of systems research and development increases in difficulty. Complex designs require complex support environments, which are expensive. For example, it would be supportive if an experimenter building a new type of file system had an empty disk to use as the test platform. Another, more commonplace, example is the development of an operating system for a new class of machine architecture that is still in the design phase or even the designing of a new and different operating system on the traditional hardware. In both of these cases, the complete emulation of the new or old machine architecture on a traditional machine would allow the development of the new operating system to continue in parallel with the development of the new hardware. This is possible in the traditional machine by providing virtual resources that emulate the functionality of the new machine. In running a simulator on a traditional machine, the developer extends or alters the machine interface to be a replication of the machine interface present in the new machine.

1.1.2 Virtual Resources In System Research

In the academic and research arenas, designing real hands-on experiments or research with machine hardware and software, has been handicapped by several requirements. These requirements range from the need for a complete and private system for experimental use, to, expensive, but sharable resources. Therefore, instead of being shared by many, system resources are found to be limited to a few users. The motivations for the policies that hinder modern system development, research and experimentation are based on installational and administrative issues. These issues cover security, availability, cost and efficiency policies. Security and cost issues have been the major contributors to this under-utilization of system resources for research and development. The connection of a live experimental system to a network of other production level systems could have dire security and reliability consequences. This has forced system developers to create their new systems in a closed, private environment, forcing a one-to-one usage on a hardware resource instead of exploiting the possibility of a one-to-many basis. A classical example is the provision of several dedicated hardware platforms to operating system developers, while experimental level work is being done. The resources are partitioned with a subset of the resources running the experimental system, while the other subset is used in building, evaluating and debugging the experimental system. To offset part of the limitations, system simulators similar to the ones mentioned above, have been used to provide virtual resources that are outside the real environment.

1.1.3 Systems For Education

Use of computer resources in education have been subject to similar limitations as those found in system development and research. This has affected the teaching of subjects like operating system and networks. If one were interested in providing system level education with design and development experience, one would have to provide access to what is or appears to be a real resource. In the past, the practice has been to supply students with complete systems or complex simulators. This has restricted the quality and quantity of the teaching and research in operating systems. To help alleviate this restriction, researchers and educators should be provided with environments that provide, *on demand*, individual resources that the experimenter, student or researcher, can use, while permitting the secure sharing of the systems resources with other users who maybe doing other experimental or general computational work. An alternative solution to the restriction is provision of several replicated systems in which each experimenter or researcher gets their own system[1]. Now some may find it unreasonable to provide each experimenter with their own private lab, disk or workstation while they are working on a new file

[1] Here *system* refers to the collection of resources that an experimenter requires.

system or operating system. The more attractive solution would be an environment in which one can produce complete copies of real and artificial system resources on demand that a user can experiment with, while sharing the real system resources amongst many users. From the system management perspective, one would expect the usual system protection that prevents users from adversely affecting each other or corrupting the general system. This environment would have to support the additional requirement that such a solution be, from a user perspective, a complete and transparent representation of the real system.

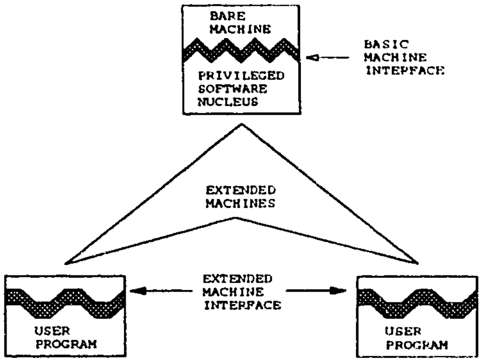

Figure 1: Conventional extended machine.

1.1.4 Pseudo-Devices

As operating systems become more general and complex, so have the general user resource requirements. Complex problems are being computed on general purpose systems resulting in higher dependencies on the interface presented by the multi-purpose operating systems. In these multi-purpose operating systems a central aim has been the support of several users simultaneously. As user needs get more complex, the physical hardware that is in place in the systems is being outpaced! In an effort to provide better systems to support modern complex environments, new ways of improving resource usage have been developed. In modern day computing, a typical user requires many system resources. Most often the resource required by the user greatly exceeds the real available system resources. For example, to improve user productivity the notion of window systems[2] was introduced. Windows are transparent abstractions of an input/output devices like a terminals. In most operating systems the number of real terminal devices on the system are far fewer than the number of windows a user may create. This need for many terminal devices at the user level to support the windowing system has placed a strain on the available real terminal devices in the system environment. The introduction of pseudo-devices that are transparent replications of regular devices in the operating systems has been the instrument in developing these windowing systems. A pseudo-device is seen by the user as a virtual terminal device, and has all the properties of a real terminal device. In like fashion, modern operating systems offer a suite of resources that are realized only on a virtual level. In the previous example of the systems designer with two dedicated machines, it would be desirable if we could let the designer build and test the system in shared environment, but protect others from the designer's follies. Since most system resources in a computer system may be virtualized and shared, it seems a small conceptual step to the next level in virtualization of system resources. Complete hardware virtualization is the next step. An operating system that provides a transparent virtual copy of all the real resources in the architecture: a Virtual Machine Operating System.

1.1.5 Virtual Machine Operating System, A Definition

The single most important notion in a virtual machine operating system is that *every system resource in the machine architecture may be virtualized in one form or another.* It is this notion that permits greater accessibility and usage of system resources by transparent duplication of the resource that is the underlying concept in the following work. The virtual machine operating system is a privileged software nucleus that provides *functional copies* of the complete machine. A major requirement of such a system is that a majority of the machine instructions set in the machine copies execute directly on the real machine. In the chapters to follow, this work will introduce studies done on the topic of Virtual Machine Operating

[2] MIT's X11 window system[26] is one example.

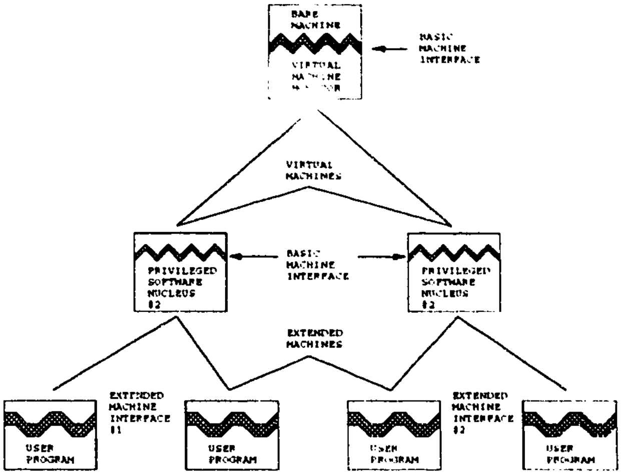

Figure 2: Virtual machine.

Systems, for use in Operating System research and education. This effort in virtual machine systems will have two interesting outcomes. The first being the possibilities that virtual machine operating systems have in higher level operating system research, particularly in distributed operating system environments. The second, the issues surrounding virtual devices and their definition. In the rest of this chapter some history and specification of formal requirements will be examined.

1.2 History Of Virtual Machine Operating Systems

Virtual machines have grown out of a past where there had been an increasing demand for a more complete virtual environment. A software designer who is trying to develop software running on a machine architecture that is yet to be realized in hardware, would traditionally first build a simulator of the new machine architecture on a general purpose machine. The purpose of the simulator being to provide a special-purpose environment on the general purpose machine that would, for most cases appear as the new machine architecture. As the requirements of the simulator become more general, a time sharing system could be developed in the native language of the new machine architecture and executed on the simulator to improve the efficiency of the software design team. In effect the simulator, depending on how complete its implementation, would create a virtual copy of the new machine architecture. Goldberg[12] explains that, while the new machine architecture and the general-purpose machine may be arbitrarily chosen, the instructions that are being simulated may be similar to or different from those of the native architecture. In this fashion a complex simulator can be built that generates a virtual implementation of the yet unrealized machine architecture. Likewise, system designers have begun to develop and use systems that provide virtual copies of limited or artificial system resource so that development and debugging of new tools and software can be done better, faster and more economically.

1.2.1 Past To Present

The mid 60's saw the arrival of early virtual machine systems. In the conclusion of their 1973 paper on virtual machine architectures, Buzen and Gagliardi[5] predicted that virtual machine systems would see rapid growth. This has never really happened. Today we see still few large computer manufacturers supporting virtual machine systems. The rapidly expanded list of applications of virtual machine implementations that they predicted, has never really arrived. The state of virtual machine research is still much the same as it was a decade ago. One aspect of virtual machine research that has improved over the two decades, is the formalization and specifications of virtual machine systems. In the next section we explore some of the formalizations that are relevant to virtual machine systems research.

1.3 Architectural Requirements For A Virtual Machine Operating System.

In this section we would like to examine some basic requirements for virtual machine systems, and introduce the machine architecture that we chose to experiment with. Considering that currently most machine architectures fall under third generation architecture, we will focus only on them.

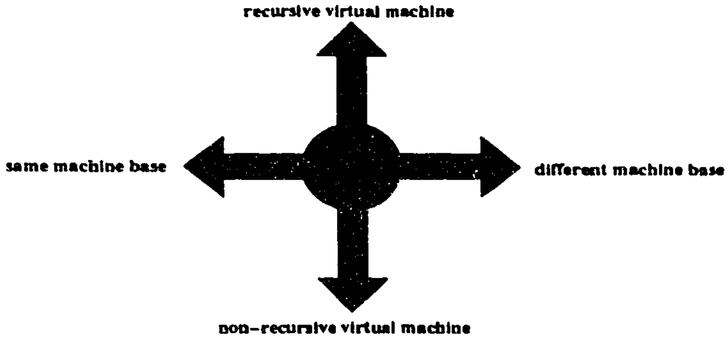

Figure 3: Dual dimensions of virtual machine environments.

1.3.1 Formal Models

In formal models of virtual machine architectures there are two dimensional categories to virtual machine environments, as figure 3 describes. On one dimension we have the equality of machine base being offered by the virtual machine with the real machine versus the equality of the virtual machine to the real machine. The differences on this line may be as great as a different instruction base, like the simulator example in section 1.2, or as small as imperfect virtualization of the complete system[3]. On the other dimension we have the recursive nature of the virtual machine interface, i.e. whether the virtual machine representation of the real machine is complete enough to allow a virtual machine operating system to run in a virtual machine. For the work presented here we are only interested in recursive virtual machine systems that project a same machine interface base. These virtual machine systems are formally described in [4, 9, 10]. Here we define a virtual machine system by the following three chief characteristics:-

1. The execution environment on the virtual machine is equivalent to that on the real machine.

2. The performance of a virtual machine with respect to a real machine is degraded only by the level of resource sharing imposed.

[3] An example of this is the removal of paging facilities in the virtual machines generated [5].

3. No virtual machine can by direct action, or side-effect, affect the state of other virtual machines[4].

From a hardware point of view we also require the machine architecture to be of a third generation architecture[5]. Popek and Goldberg[24] introduce some specific formalization for these machine architectures. On this level the following restrictions are necessary:-

1. The CPU is pre-emptable.
2. Sensitive instructions are protected, and may be executed only in the more privileged mode of operation.
3. Execution of sensitive instructions while the machine is in non-privileged mode will result in a trap and a change of mode.

In this fashion we can then treat the CPU as any other resource in the system that is virtualizable, and all execution sensitive instruction executed in a *non-privileged* mode will cause a machine exception, or trap.

1.3.2 What Is MIME

The work presented here is the description and implementation of a Virtual Machine Operating System called MIME for the Sun-3[6]/MC68020 family of workstations. This effort follows closely the work introduced by Goldberg in early 1970. The work is an on-going effort to produce a complete virtual machine system for research in principles of distributed virtual resource management. The need for a useful virtual machine environment for operating system instruction is also a major foundation of this effort. MIME's current implementation exists as special derivative of the Sprite Network Operating System kernel[23]. In later chapters of this thesis we will describe MIME in greater detail.

[4] Here we do not include the effects of communicating virtual machines via the usual mechanisms.

[5] Third generation architectures are machine architectures that support dual operation modes, i.e. a supervisor mode versus a user mode.

[6] Sun-3 is a trademark of Sun Microsystems Incorporated.

Chapter 2

Machine Architecture

The machine architecture that is being used for the development of MIME is that of a Sun-3 workstation. The Sun-3 is a high-performance family of workstations that are rated at 1.5-4 MIPS. Sun's design strategy has been to use industry standard whenever possible. The 32-bit VMEbus, SCSI and SMD interface are but a few examples. This family of workstations carry an onboard ETHERNET controller and feature a CPU from the Motorola 680*x*0 family of microprocessors.

2.1 Motorola MC68020 Microprocessor

The MC68020 is a full 32-bit micro-processor from Motorola. It implements a full 32-bit data path, 32-bit register with 32-bit addressing [17]. The processor implements three processing states, user/master/interrupt, but the interrupt state maybe considered as a special case of the master state. The following subsections examine the exception handling capabilities of the processor, its instruction set, and interrupt behavior.

2.1.1 Exception Processing in a MC68020

The MC68020 can generate both internal and external exceptions. Internal exceptions are caused by instructions, address errors, tracing and breakpoints. This includes illegal instructions, privilege violations, co-processor protocol violations

Privileged Instructions	*Trap Instructions*
ANDI to SR	TRAP #n
EORI to SR	TRAPcc
cpRESTORE	TRAPV
cpSAVE	cpTRAPcc
MOVE from SR	CHK
MOVE to SR	CHK2
MOVE USP	
MOVEC	
MOVES	
ORI to SP	
RESET	
STOP	

Table 2.1: User Mode Privileged And Trap Instructions.

and trap executions. External exceptions will be generated for interrupts from peripheral devices, bus errors, reset and co-processor errors.
During exception processing the following steps are followed:-

- Save the status register, place the CPU in supervisor mode by setting the S-bit in the processor status register SR. Clear the trace bits, and set the interrupt priority mask for reset and interrupt exceptions.

- Compute the exception vector number.

- Save the processor context on the supervisor stack. Load and fill an exception stack frame on the supervisor stack. If the exception is an interrupt and the status register M-bit is set, it is cleared and the second stack frame operating system created on the interrupt stack.

- Compute an exception vector offset by multiplying the vector number by four and adding it to the contents of the Vector Base Register VBR. The program counter is loaded with this value, and the normal instruction execution cycle continues.

The exception/interrupt routines then execute in master state and return to normal processing by executing the RTE or RTI instructions. This resets the stack and program counter to the values that are in the saved exception stack frame.

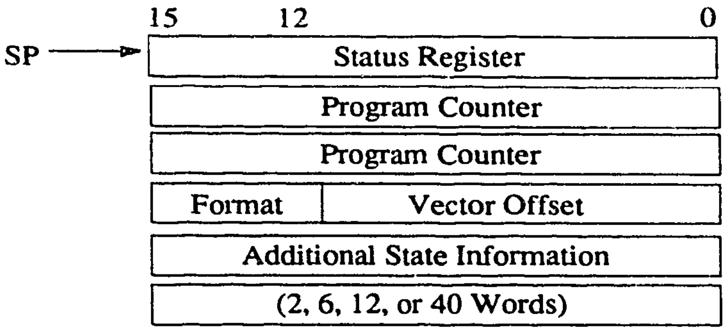

Figure 4: Basic Exception stack frame

2.1.2 MC68020 Exception Stack Frames

Let us examine the details of the MC68020 exception stack frames. The following paragraphs examine the six MC68020 exception stack frames, and describe the various internal and external conditions that generate the exception conditions. Figure 4 shows the basic frame format. Words 0 to 3 are present in all stack frames. The rest of the words in the the frame make up the additional information that is present of the different stack frames.

Normal Four Word Stack Frame.

This frame is placed on the supervisor stack if any of the following events occur: an interrupt, a format error, the execution of TRAP #n instruction, an A-line or F-line emulator trap, a privilege violation, or a co-processor pre-instruction exceptions. The stacked PC will point to the next instruction, RTE or cpRESTORE instruction, next instruction, illegal instruction, A-line, F-line instruction, first word of instruction causing the privilege violation, Op-Word of instruction, respectively. This frame is labeled with the format field value of 0x0.

Throwaway Four Word Stack Frame.

This frame will be on the interrupt stack if an exception is taken while the processor is in master state. It transits the processor to interrupt state. The PC saved on the stack points to the next instruction and is equal to the value on the normal four word or co-processor exceptions stack frame. This frame is marked by a format field value of 0x1.

Normal Six Word Stack Frame.

This frame will be on the supervisor stack if an exception is taken for the CHK, CHK2, cpTRAPcc, TRAPcc, TRAPV instructions, trace, a zero divide arithmetic fault or co-processor post-instruction. In all cases, the PC saved on the stack points to the instruction to be executed next. The address of the instruction that caused the exception is saved in the fifth and sixth words of the frame. This frame is marked with a format field value of 0x2.

Co-processor Mid-Instruction Exception Stack Frame.

This frame will be on the supervisor stack if one of three different co-processor related exceptions occur. The first occurs when a *take mid-instruction exception* occurs during a co-processor instruction. The second occurs when the main processor detects a protocol violation while processing a co-processor instruction. The third occurs when a *null, come again with interrupts allowed* primitive is received, and the processor detects a pending interrupt. For all these exceptions the PC saved on the stack points to the next instruction in the instruction stream. The fifth and sixth words of the stack frame contain the address of the faulting instruction and the following four words contain internal co-processor registers. This frame is marked with a format field value of 0x9.

Short Bus Cycle Fault Stack Frame.

This frame will be on the supervisor stack if a bus cycle fault is detected and the processor determines that it is at an instruction boundary. The PC saved on the stack points to the next instruction to be executed. The frame is sixteen words long and contains several internal saved register values. This frame is marked with a format field value to 0xA.

Long Bus Cycle Fault Stack Frame.

This frame will be on the supervisor stack if a bus cycle fault is detected and the processor determined that it is not at an instruction boundary. The PC saved on the stack points to the address of the instruction that was executing at the time of the fault, which may not be that instruction that caused the bus cycle fault. This frame is forty six words long and is labeled with a format field value of 0xB.

2.1.3 Instruction Set And Addressing Modes

Considering that we are designing virtual machine system for a third generation architecture, the essential machine instructions are those that affect the machine state as described in Section 1.3. Table 2.1 lists all the sensitive machine instructions. Execution of the privileged instructions in a user-mode will cause a privilege violation exception, and alters the CPU state as covered in 2.1.1. While execution of a trap instruction while in user-mode will result in a trap exception with the instruction that caused the trap in the exception stack frame as described in 2.1.1. While the MC68020 supports a wide variety of addressing modes, it is beyond the scope of this work to cover them, and we refer the reader to the MC68020 User's Manual[17, section 2.8][1].

2.1.4 Interrupt Levels And Bus Cycles

The MC68020 supports seven interrupt levels. At instruction boundaries the microprocessor will begin interrupt processing if there is a pending interrupt with a priority greater-than-or-equal-to the interrupt mask, [17, Section 5.2.4.1.3]. Interrupt

[1] CPU addressing modes do not affect the design of the virtual machine operating system.

processing entails entering an interrupt acknowledge cycle and performing the following

1. Set the address space to CPU space.

2. Get the interrupt vector, or provide an internal autovector if none is provided[2].

3. If external logic shows a bus error, take the interrupt to be spurious.

4. Save the exception vector offset, program counter, and status register on the supervisor stack[3].

5. If the M[4] bit in the status register is set, clear it and place a *throwaway* exception stack frame.

6. Begin normal exception processing by loading the program counter with the value contained at the exception vector.

Note that a return form exception instruction RTE or MOVE to SR will clear the interrupt mask level and set it to a lower level. Another thing to note is that interrupt level seven is special: level seven interrupts are non-maskable.
On bus errors the processor follows the usual exception processing steps.

1. Copy the status register.

2. Enter the supervisor state.

3. Disable tracing.

4. Generate an exception vector of *Bus Error*.

5. Stack the vector offset, program counter, status register, non-visible internal registers.

If the exception was taken at an instruction boundary, the exception stack frame will be of the *Short Bus Cycle* variety, or else it will be the *Long Bus Cycle* type, as we saw in section 2.1.1.

[2] See the exception vector table in Appendix A.

[3] During co-processor instruction interrupt further internal state information is saved as well.

[4] The M bit of the status register shows whether the CPU is in Master or Interrupt state.

2.2 Sun-3 Architecture

The Sun-3 architecture that MIME is targeted for features the following major components [2]:-

- MC68020 central processing unit.
- MC68881 floating point co-processor.
- Sun's proprietary paged memory management unit. The Sun-3 MMU is an 8 context memory management unit, that maps 265 megabytes per context.
- A main memory space of 4 to 32 Megabytes.
- Bit-mapped display with virtually addressable video memory.
- IEEE 802.3 ETHERNET network interface controller.
- A software readable programmable boot read only memory (Boot PROM).
- An optional high performance virtual address, direct-map, write-back cache. The cache is application transparent and is 64 Kbytes with 16 byte lines.
- Multiple buses: private bus for the CPU and memory; VMEbus for the system I/O path.

The architecture also features a full 32-bit VME interface, a standard SCSI/SMD disk and tape-drive interface, and direct virtual memory addressing (DVMA) for the I/O systems. Several optional performance enhancing add-ons for floating-point arithmetics and graphics processing are also available. Since we are only interested in the run-of-the-mill Sun-3 hardware we will not concern ourselves over the add-ons and options. The architecture implements the microprocessor's eight address spaces as follows:-

Address Space 0 Function code 000 – Reserved

Address Space 1 Function code 001 – User Data Space

Address Space 2 Function code 010 – User Program Space

Address Space 3 Function code 011 – Memory Management Unit Space

Address Space 4 Function code 100 – Reserved

Address Space 5 Function code 101 – Supervisor Data Space

Address Space 6 Function code 110 – Supervisor Program Space

Address Space 7 Function code 111 – CPU Space

From this list we can see that the MMU may be manipulated only by data moves[5] across address space three.

2.2.1 VMEbus, Devices And Peripherals

The Sun-3 VMEbus is a full 32-bit VMEbus interface that is dual-ported with a master interface for CPU to VMEbus access, and slave interface for VMEbus to main memory access. The VMEbus slave interface allows VMEbus to main memory transfers by treating a range of VMEbus addresses as a range of virtual CPU addresses. The architecture also includes standard devices that provide basic Sun-3 system resources. The following devices can be found on the CPU board:-

- EPROM and EEPROM: they store system configuration and initialization code for booting and diagnostics.
- Time-of-Day Clock: a battery-backed clock.
- Interrupt registers: control the system interrupts.
- Serial Line Interface: two programmable serial I/O channels.
- Keyboard/Mouse Interface: serial interface to the keyboard and the mouse.
- ETHERNET Interface: takes advantage of DVMA to transfer data directly to memory.

[5] This is done by the CPU by the Move address space "MOVES" privileged instruction.

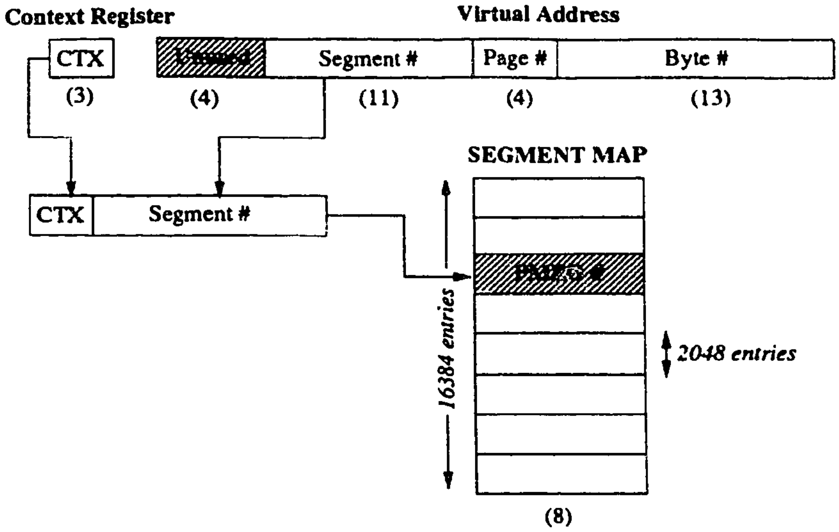

Figure 5: (**Pmeg** number translation) The MMU takes the value of the context register (3 bits) and composes it together with the Segment number (11 bits) that comes from the virtual address. The result of the composition is used to index into one of the 8 Segments groups that are 2048 entries deep. The result of the index is an 8 bit wide **pmeg** value.

On the VMEbus we have the following I/O devices.

- SCSI and SMD Interface: interface to SCSI and SMD disk/tape drives.
- 1/2-inch Tape Interface: interface to 1600/6250 bpi tape drives.
- Second ETHERNET Interface.
- VME-Multibus[6] Adapter for Multibus interface.
- Asynchronous Line Multiplexer: for additional serial devices.

2.2.2 Sun-3 Memory Management Unit

A virtual memory address is composed of an 11 bit segment number, followed by a four bit page number and a 13 bit byte number. The Sun-3 memory management

[6] Multibus is a registered trademark of Multibus Inc.

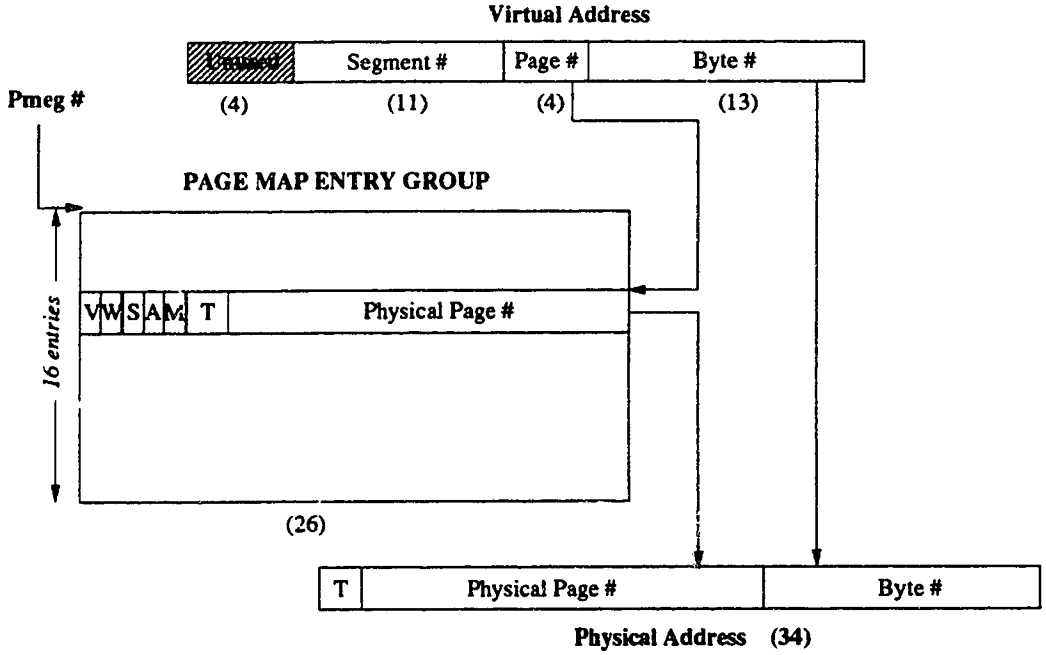

Figure 6: (Physical address translation) To obtain a physical address the MMU uses the page number (4 bits) as an index into a page map entry group or **pmeg**. The top of the **pmeg** is obtained from the previous **pmeg** translation of figure 5. The result of the index is a 26 bit long value whose lower 19 bits define a physical page number. The physical page number is then composed with the byte number (13 bits) from the virtual address and the two bit type field to make a physical address 34 bits). The page entry in the **pmeg** includes three protection bits: the *valid* bit V, the *write* bit W, and the *supervisor* bit S; two statistics bits: the *accessed* bit A, and the *modified* bit M, and one two-bit *type* field T.

unit is a eight-context memory management unit. Each context together with the address segment[7] value point to one of 16,384 segment map entries. The segment map, which may be viewed as 8 contexts of 2048 each, is made of eight bit wide entries that point to a page map entry group, or pmeg. The page map contains 4096 entries that each map an 8 Kbyte page. The page map table is a 26 bit wide table that is indexed by the 8 bit pmeg number composed with the 4 bit page number from the virtual address. The resulting entry is made of a 3 bit protection field (*valid, write, supervisor*), a 2 bit statistic field (*accessed, modified*), a page type field (*main memory, I/O, 16 bit VMEbus, 32 bit VMEbus*), and a 19 bit physical page number. The 34 bit physical address is composed of the 2 bit type field, the 19 bit physical page number, and the 13 bit byte number from the virtual address. The *valid* bit specifies if the page entry is a valid entry read and execute access is permitted. The *write* bit flags allows or disallows write access to the page. The *supervisor* bit restricts access to the page to the supervisor state only. The *access* and *modified* bits are bits that are automatically updated by the MMU hardware as accesses and writes are made to the page. The *type field* provide four physical address spaces, each beginning at physical address 0, one for each MMU resource. Figures 5 and 6 describe the translation that MMU makes when mapping a virtual address to a physical address. The memory management unit is accessed by reading and writing to virtual memory address in address space three. In this fashion, context changes are made just by writing the MMU context register. We will avoid the technical details, manipulation and operation of the Sun-3 proprietary MMU unit.

[7] The address segment value is the 11 most significant bits in a virtual address, bits 17-27.

Chapter 3

A Virtual Machine Operating System

In this section we will propose a Virtual Machine Operating System that is targeted to run on the Sun-3 workstation platform. This virtual machine operating system, which we have christened MIME, is built on a Virtual Machine Monitor (VMM) that is capable of providing Virtual Machines and a collection of device drivers that virtualize system peripheral resources wherever possible. The virtual machine monitor provides the virtual machine interface that was introduced in Section 1.3. This interface appears transparent to any system that runs in virtual machines. You could compare a virtual machine generated by the VMM to a user process on a conventional operating system. The virtual machine will execute for a time quantum before it is preempted and the CPU and hardware resources handed over to another virtual machine running in the environment. In the sections to follow we will present the various requirements and the design of MIME.

3.1 Hardware Management

In Sections 1.3 and 2.2 we explored the formal system requirements for a virtual machine operating system and those offered by the Sun-3 platform. We see that the Sun-3 hardware conforms to our specifications for a virtualizable machine architecture. We will categorize the task of designing the virtual machine monitor into three parts: Managing the CPU, virtual memory issues, peripheral devices and miscellaneous duties like accounting etc.

3.1.1 CPU

CPU Register	**Size** (bits)
Program Counter (**PC**)	32
Processor Status Register (**SR**)	32
14 General Purpose Registers	14 × 32
User Stack Pointer (**USP**)	32
Master Stack Pointer (**MSP,SSP**)	32
Interrupt Stack Pointer (**ISP**)	32
Vector Base Register (**VBR**)	32
Special Status Word (**SSW**)	16
Cache Control Register (**CACR**)	32
Cache Address Register (**CAAR**)	32
Source Function Code Register (**SFC**)	3
Destination Function Code Register (**DFC**)	3

Table 3.2: Machine **CPU** context structure.

The MC68020 is a dual state microprocessor. Our virtual machine monitor can virtualize this resource by restricting the virtual machine processes to the *user state*. This way the virtual machine monitor will not have to interpret the *user state* instructions since they have no effect on the state of the real machine and as such can run directly on the real machine. In Table 3.2 we list all the microprocessor registers that are relevant to describe the CPU context at an instruction boundary[1]. Every virtual machine running under MIME has its CPU state defined by these registers. The C data structures `Cpu_State`, in Appendix D.1.1, adequately describes a machine CPU context. A virtual machine context switch is done by saving the old `Cpu_State` structure and loading the CPU with the new `Cpu_State` structure

[1] More internal CPU registers are necessary to describe the CPU state at mid-instruction.

and performing a return to *user-mode*. The VMM is responsible for changing the virtual memory contexts during the context switch operation.

3.1.2 Virtual Memory Support

Since the machine interface we are aiming to provide supports virtual storage by paging, we will examine the impact of supporting a paged virtual machine interface. As mentioned in Sections 2 and 2.2, the Sun-3 MMU is a virtually addressable co-processor in address space 3. The MMU context register, segment map and page map tables may only be altered by writing to virtual addresses in MMU space. This will cause a privilege violation if the microprocessor is in *user-mode*. As mentioned in the paragraph on the normal four word stack frame in Section 2.1.2, a privilege violation will cause the normal four word stack frame to be placed on top of the supervisor stack. The CPU will enter privilege violation exception processing, taking the exception vector from the VBR. The PC points to the first word of the offending instruction, which here will be a MOVES instruction. If the destination and source function code registers point to MMU address space, then it is known that the user wishes to read or write the address in the MMU address space[2]. If the source or destination register argument to the MOVES instruction is a valid Sun-3 MMU address, the VMM will do the following emulation of the instruction, at exception time.

1. Save the sixteen general purpose registers.

2. Test for validity of the MMU memory space access. Since the Sun-3 MMU addresses are fixed virtual addresses, this is easy to do.

3. Compute the source/destination register argument number. Table C.4 in Appendix C describes the instruction format, in which we can read the register type and number.

[2] In the Sun-3 architecture, the MOVES instruction is used to read and write MMU memory space.

4. Get the data in the argument register. This could be any of the seven general purpose address or data registers. Remember that we have saved the registers and can retrieve their original values at exception fault time.

5. If the instruction is an attempt to alter the MMU state, emulate the change in the simulated MMU structure, for instance the memory management unit context register. If the instruction is an attempt to read MMU space memory, emulate the instruction by reading the virtual machine's simulated MMU structure.

In this model, any attempt to access the MMU for reading or writing will result in an emulation of the request from the virtual MMU data structure. A result of this emulation of the MOVES instruction is that the VMM can keep track of all MMU states for all virtual machines running under it, and emulate changes to these states. Of course, at some point the simulated MMU structure will have to be used in address translation. We will examine this in later sections.

3.1.3 Peripheral Devices

In this work peripheral devices are treated like memory mapped I/O to virtual memory addresses that map the hardware device drivers. The devices also have interrupt service routines that initiate and terminate the I/O operations to the devices. Every virtual machine that is to execute in a virtual machine operating system requires some system devices. At the most trivial level, a machine console and some form of I/O system is needed. Usually these devices are mapped by the hardware's startup code at system boot time. For the Sun-3, all device addresses are virtually mapped, making device management a simple task. A virtual machine operating system manages system devices by creating virtual versions of the resource for every instance of a virtual machine that is present. The virtual copies are multiplexed into the real resource, just as a conventional operating system would multiplex the device resources amongst the running processes. There is, however, a major distinction between the methods a virtual machine operating system uses to multiplex device resources and those used by conventional operating systems. The difference is that the users in a virtual machine operating system, the virtual machines, are not aware that they are running under a VMM. This means that virtual machines cannot make specialized requests for system resources in the same way user processes request system resources in conventional operating system[3]. Since we cannot expect a virtual machine running in the VMM environment to make system call requests whenever it wants to read the disk, a VMM will have to manage the devices by the devious route of trapping requests to the devices and multiplexing the requests on the real devices.

As mentioned previously, the Sun-3 I/O peripheral devices are all mapped into virtual memory. This implies that in the memory map for the Sun-3 a range of virtual addresses are reserved for device drivers and direct virtual memory access. A VMM can protect the devices simply by marking the pages that map the devices as supervisor pages. This means that when a VM accesses such a page, a memory access violation trap will occur, and the VMM can fulfill the I/O request for the VM. The VMM must first determine the cause of the violation and then execute a device request that will produce the desired results.

Several weaknesses exist in this model for virtualization of system resources. The primary weakness is that the Sun-3 machine architecture has several devices that do not lend themselves effectively to virtualization. The Sun-3 framebuffer is one example of such a device. Since there is only one console device per real machine, we should not try to share the resources that make up the console device amongst various virtual machines. Such devices should be allocated to a specific virtual machine and all other virtual machines see an architecture that does not support the device if it cannot be virtualized.

[3] System calls.

3.2 The Design Of A Virtual Machine Operating System

In the preceding sections we have seen hardware specifications for the combination Sun-3/MC68020 architecture[4]. In this section we would like to mention some design specifications for MIME, as an operating system for the management of virtual machines on the described architecture.

MIME consists of a Virtual Machine Monitor and a collection of device drivers that multiplex I/O hardware amongst the various *users*[5]. The Virtual Machine Monitor is responsible for the management of Virtual Machines that can be produced on demand. This entails the maintenance of kernel data structures that define Virtual Machines, schedule execution of the Virtual Machines, and field supervisor service requests. The Virtual device side of MIME is responsible for emulating real and artificial devices that Virtual Machines may require.

3.2.1 The Virtual Machine Monitor

The principal element of the Virtual Machine Monit f MIME, is the system of interrupt and exception handling functions that will manage virtual CPU's by trapping all interrupts, exceptions and traps and passing them on to the appropriate Virtual Machine. By multiplexing the virtual processor data structures onto the real machine, MIME allows multiple virtual machines, each executing entirely in *user-mode*. In this fashion most virtual machine operations will execute efficiently on the real hardware. The following list presents a brief description of MIME's VMM startup step:

- Initialize the machine interrupt vector table. This will cause all hardware generated traps, exceptions, and interrupts to be vectored to the VMM's interrupt routines.

[4] Parts of these specifications may be valid for other chips in the Motorola 68000 family.

[5] Here a *user* is an operating system running in a virtual machine maintained by the VMM.

- Set a machine context for every virtual machine that will execute under the supervision of the VMM. This includes setting up the machine context to that of a machine at boot/startup time, (see Section: 3.4).

- Initialize the scheduling and timing modules that will permit machine context switches and clock events.

- Initialize all machine registers and contexts for the real machine.

- Initialize the virtual memory and paging resources for the VMM. This includes establishing a basic file system service, and swap devices.

- Initialize the network, serial and other device drivers.

The machine interrupt vector table is started by reading the vector base register, VBR, and writing an address pointing to the exception processing function for each of the 256 interrupts vectors, (see Appendix A, Table A.3). During exception processing, the appropriate exception handling routine will be called with an appropriate exception stack frame on the stack.

VMM Exception Handling

This subsection highlights some of the VMM exception processing operations. On receiving an interrupt for a running VM, the VMM attempts to handle the request using the following method:

At exception time the CPU is in *supervisor-mode*, with the supervisor stack as the stack in use. The exception stack frame is on the top of the stack. Based on the type of exception, the VMM can chose to handle the exception completely, or pass on the exception to the faulting VM[6]. Figure 4 describes the format of the basic exception stack frame. If the VMM can safely permit the faulting VM to handle the exception, and the VM is running, the VMM will dummy up a copy of the exception stack frame on the VM's virtual master/supervisor stack, manipulate

[6] The exception type is determined by looking at the format field word in the exception stack frame.

the VM's CPU context structure appropriately, so that the VM will execute its exception handler when it gets the CPU next. The dummy exception frame may have its return address set to the return address on the real exception frame or the next instruction in the VMM's hand' 's. This will allow the VMM to do some post-interpretation if the interrupt calls for it. If the exception cannot be executed safely by the faulting VM, then the VMM will process the exception itself in an attempt to emulate the service that the user VM expects to have access to and was denied, based on privilege. Most of these emulated exceptions are related to protected memory addresses that control devices and resources. For instance, if the exception is related to a bus error or the MMU, the VMM may try to swap in the required page for the faulting VM. In this fashion the VMM can provide machine encapsulation, such that no VM can affect another VM. In Appendix D.2 we have presented some VMM code fragments that show the emulation of privileged instructions. The handling of MMU access is one example of privileged instruction emulation. In Section 3.4, we will examine in more detail the machine context switching operation.

3.2.2 Virtual Devices

In MIME, a virtual device is represented by a collection of kernel data structures. These data structures allow the virtual machine monitor to multiplex the access of the virtual devices or drivers by the virtual machines. In this section we would like to focus on the design aspects of creating virtual devices for a virtual machine environment.

One chief problems with the design of MIME for the Sun-3 workstation environment is the type and quantity of I/O devices available. In chapter 2 we described in detail the quantity and type of devices that are supported by the Sun-3. Of the devices that the architecture supports, the console device does not virtualize well, reducing the list further[7]. Since the Sun-3 architecture supports *virtual* direct memory access, we can view all devices and their drivers as special pages in memory.

[7] The console device does not virtualize well owing to the special relationship between the console keyboard and monitor and between the framebuffer and video drivers.

Indeed the I/O pages will be marked by the VMM as supervisor pages, allowing the monitor to control access to the pages. To virtualize a device, the VMM maintains a spool list of access events requested by executing VMs. As the physical device becomes available, the next access request in the list is copied to the device's control structure and I/O begun. On successful completion of the I/O operation, the results will be copied from the virtual memory buffers that the device accessed to the equivalent addresses in the address space of the virtual machine that requested the I/O operation. This way, the access to a virtual device is treated like a supervisor service request and is implemented as a sequential device operation stream. Every device managed by the VMM will have its own event queue in the monitor. To an executing virtual machine a virtual device access will appear as a synchronous operation, though it is implemented by the VMM as an asynchronous operation. The configuration data for the various devices that VMM in MIME will virtualize is part of a VM's state information in the virtual machine control block. In chapter 4 we will return to the virtual machine control block.

3.3 Virtual Machines

In this section we would like to explore the virtual machine interface offered by MIME, configuration of system resources and the initialization of a virtual machine. In previous sections we described the hardware initialization state that an operating system running on the Sun-3 will expect. For our virtual machine interface to be transparent and flawless, each virtual machine running under the VMM must have an identical initialization. To meet this requirement, MIME will create a copy of the initial system configuration for each virtual machine in the virtual machine's virtual address space (VAS). The following is a list of steps that VMM takes to configure a new virtual machine.

1. Initialize virtual machine pages for the new virtual machine. This includes writing the appropriate addresses of virtual devices and drivers and building an appropriate EPROM structure.

2. Initialize the VMM internal machine context structure that describes the new virtual machine. This context structure describes a virtual machine that is in the default machine boot state. Scheduling tasks are also started. These manage the virtual resources that are expected by the VM. System resources that the new VM needs, like disk space, are also allocated at this point.

3. The executable binary for the new virtual machine is loaded into the VM's VAS. All other VMM virtual machine context data structures are initialized.

4. Once loaded and initialized, the VM is scheduled for CPU time by MIME's scheduler, at some point in the future.

Once all initialization is complete and an executable boot program is loaded, the VM is ready to execute. At some point in the future the VM will be given the CPU. It is expected that the virtual machine will then initialize itself on the virtual machine interface the VMM has projected, just as it would on the physical machine interface. The new VM will be able to determine its boot configuration from the EPROM data that is visible in its address space.

3.4 Virtual Machine Context

In a VM environment, all hardware resources are virtualized. This means that, for every operating system running in the virtual machine environment, we have to provide all the resources it may possibly expect from the architecture. In this section we will look at the requirements in defining a machine context that adequately describes the machine state for a virtual machine.

Considering the machine architecture that was described in Chapter 2, every virtual machine needs to have a CPU context that is defined by the following:

- Processor status word.

- Program counter.

- Sixteen general purpose registers.

- User/Master/Interrupt Stack pointers.
- Memory management unit context.
- Floating point unit context.
- Co-processor unit context.

This machine context provides a sufficient description of the state of the CPU for any Virtual Machine running in the Virtual Machine Environment. During a machine context switch, we set the hardware registers to the values that are stored in the VM's context structure and execute a *return to user mode* procedure. The CPU will then start operating in user mode. This method of virtual machine context switching will impose the following penalties and overheads:

Timing Delays The real time clock as seen by virtual machines will not tick in a smooth incremental fashion. A VM with sensitive timing requirements, like a real-time operating system, would suffer some penalties. Goldberg called this additional time to process an event stream a *stretchout.*

Context Switch Delays A context switch requires the change of the MMU *context.* This means that the MMU instruction cache will get invalidated and the new virtual machine will suffer an added overhead as it goes through some cache misses. This is one processor *overhead* that is mainly incurred by the extra clock ticks.

Delayed I/O A virtual machine that had pending I/O that arrives after the context switch will not receive notification of the I/O until it receives its next CPU slice. This applies to all I/O including network I/O.

We can see here that the machine context switch is the most expensive operation of a VMM, and greatly affects the performance of the system as a whole. The better the VM context switch, the fewer the penalties that the child operating systems will pay for operating in a virtual machine environment. It is also necessary for a virtual machine CPU context structure to store information that describes the states of various devices, pending interrupts and exceptions. The context structure also maintains synchronization data for several devices like the difference between virtual time in a virtual machine and the real time. The VMM keeps the virtual machine context structure as part of a virtual machine control structure called the virtual machine control block (VMCB). The virtual machine control blocks for all the virtual machines running under the VMM appear in a kernel table called the Virtual Machine Table (VMTAB). The VMM goes through the following steps to achieve a machine context change:

1. Update the current VM's VMCB entry with the current state of the processor and peripherals. This includes the MMU state.

2. Obtain a ready VM from the ready queue.

3. Allocate a virtual address space for the new VM, if one does not exist.

4. Prime the processor with the new VM's VMCB entry. In doing so the MMU is updated to reflect a new virtual memory address space. The VMM will also update synchronization and timing routines that will interrupt the new VM once its CPU time quantum is over.

5. Execute a return to *user-mode*. This causes the processor to begin executing instructions from the VM's program space.

A VM has exclusive access to the CPU as long as it does not incur a VM-*fault* or its CPU time slice is not exceeded. On a VM-*fault* the VMM may decide to swap virtual machine contexts as it services the fault for the VM. When an executing VM reaches its CPU time slice limit, a VMM timer will interrupt the VM giving control of the processor to the VMM. The VMM will then begin a virtual machine context change with the next ready VM. The VMM hides the context changes from the executing VM by manipulating all the system resources that can show presence of the VMM. In doing so, the VMM provides a transparent virtual machine interface.

3.4.1 Overhead In A Virtual Machine

Goldberg identified several principal sources of overhead in a virtual machine system that also exist in MIME. The first type of overhead is the maintenance of the status of each virtual processor. This includes the handling of all the visible microprocessor state and reserved memory locations, as we just saw and will see again in Section 4.4. Another source of overhead in a virtual machine system is the support of privileged instructions. This is the instruction emulation described in Section 3.2.1. Support for paging in a VM is a third principal overhead imposed by a VMM. This will also be covered in the next chapter. Other sources of overhead exist, but they depend mostly on the management strategy of the operating system executing in a VM. For instance, the type of functionality an operating system executing in a VM requires for virtual console may call for extra support from the VMM. Another aspect of virtual machines that results in extra overhead, is the optimization of peripheral device operations. In a VM, I/O operations on a virtual device result in a VMM translation that maps the virtual device to a real device. An operating system executing in a virtual machine may attempt to improve the access of the virtual device. Since the VMM translates all access to the virtual device, the optimization will not result in any access performance improvement. The optimization may result in slower disk access, while carrying an extra processing penalty. This aspect of virtual device behavior is a side effect of the VMM virtualization of devices.

Chapter 4

Virtual Memory In A Virtual Machine

In this section we would like to examine the effects of virtual memory management in a VM environment, one difficult aspect of a virtual machine monitor. The requirement that a VMM support transparency of virtual copies of "real" resources makes the task of virtualizing the MMU a complex task. The task is indeed more complex than typical virtual machine management modules in conventional operating systems. In IBM's approach to a virtual machine operating system, we see the development of hardware components to assist in speeding up address translation. In MIME we are not so much in pursuit of an efficient paging method, as of a simple VMM model to experiment with. Since we are using a general definition of a virtual machine system that includes recursion, there is room for confusion when we refer to real resources in a virtual machine. A real resource exists as an extension of the physical hardware. A "real" resource exists as a VMM projected virtual resource in a VM. We will use the quoted "real" when referring to a resource that a VM expects to be operating on the physical hardware, but truly is just a virtual object projected by the VMM. To help define the role that virtual memory management plays, it is useful to make some formal definitions.

4.1 Memory Resources In A Virtual Machine

Virtual memory management in third generation architectures, like the Sun-3, may be viewed as the dynamic mapping of memory resources. The MMU maps memory resources in the user context to a real memory resource by applying a translation based on the current user address map. In a virtual machine environment the translations are m\de up of two major components. The first component defines how a virtual resource in a VM is translated by the operating system running in the VM to a "real" resource in the VM. We call this the process resource model. The second component defines how a "real" resource in a VM is translated by the VMM that is supporting the VM in a "virtual-real" resource in the VMM. We call this the virtual machine resource model. Using the process resource map, this "virtual-real" may be translated into a "real-real" resource in the VMM. In a recursive virtual machine structure, the complete realization of a virtual resource in a VM to a real resource on the physical hardware, may require repeated resource translation. The following paragraphs describe the components of a resource model that describes the translation mechanism of a virtual machine environment.

Process Resource Model

This address translation is Goldberg's [11] *ϕ-map* definition. This *ϕ-map* is defined as a software and hardware operation that translates a set of *process names* $P = \{p_0, p_1, \ldots, p_j\}$ to the set of "real" *resource names* $R = \{r_0, r_1, \ldots, r_i\}$. Effectively, the *$\phi$-map*, MMU here associates process names with "real" resource names. Formally, the *ϕ-map* function is defined as follows:

$$\phi : P \rightarrow R \cup \{e\}$$

such that if $x \in P$, $y \in R$, then

$$\phi(x) = \begin{cases} y & \text{if } y \text{ is the resource name for process name } x. \\ e & \text{if } x \text{ does not have a corresponding resource name.} \end{cases}$$

In the above definition, $\phi(x) = e$ means that the resource does not exist. This event, in a operating system, results in the occurrence of an exception. The ϕ-

map definition describes a *inter-level* resource mapping function, which we call the process resource model.

Virtual Machine Resource Model

The *ϕ-map* resource model describes a translation of virtual resources to real resources both of which are on the same machine. In a virtual machine environment we also need to define a mechanism that will map the set of *virtual resources* $V = (v_0, v_1, \ldots, v_m)$ present in a VM, to the set of *real resources* $R = (r_0, r_1, \ldots, r_n)$ present in the "real" machine. Since recursion is permitted in our model, the "real" machine may well be a VM_i executing at level i under a VMM[1]. Goldberg and Popek [10, 11, 24] developed a formal definition for this mapping mechanism. Goldberg called the mapping an *f-map* and defined it as follows:

$$f : V \rightarrow R \cup \{t\}$$

such that if $y \in V$ and $z \in R$ then

$$f(y) = \begin{cases} z & \text{if } z \text{ is the real name for virtual name } y. \\ t & \text{if } y \text{ does not have a corresponding real name.} \end{cases}$$

The value $f(y) = t$ will cause a VM-*fault*. The *f-map* function associates resources in the VM_i with resources in the "real" machine, VM_{i-1}. VM_{i-1} is the virtual machine running a VMM and executing at level *i - 1*. The *f-map* is an *intra-level* resource mapping function. In a recursive VM environment, a physical resource may require repeated applications of the *f-map* operation to be realized. It is possible to compose several *f-maps*[2] that maps a level n virtual resource to the level 0 resource. For example, figure 7(a) shows the mapping of a level 2 virtual resource, y, to the real resource, z, by the following composition rule[3]:

$$f_1 : V_1 \rightarrow R$$

$$f_{1.1} : V_{1.1} \rightarrow V_1$$

[1] The subscript in the VM name denotes the level in the recursive virtual machine tree that the VM is in.

[2] For the *f-maps* we are using the same naming convention that is used for the VMs.

[3] The mapping $f_1 o f_{1.1}(y)$ is just the composition $f_1(f_{1.1}(y))$ or $f_{1.1} \circ f_1$.

Composed Resource Model

The *ϕ-map* and the *f-map* can be composed to translate a process resource $P = \{p_0, p_1, \ldots, p_j\}$ executing on a virtual machine VM$_x$ $V_x = \{v_{0_x}, v_{1_x}, \ldots, v_{m_x}\}$ to a "real" resource in VM$_{x-1}$. This is done by applying the composed translation $f(\phi(x))$, which can be generalized by the mapping function;

$$f \circ \phi : P \rightarrow R \cup \{t\} \cup \{e\}$$

This composed translation is successful if both component's translations are successful. If the *ϕ-map* fails, the process resource, p_i is not realized by the virtual machine VM$_x$. The VM$_x$ is also handed an exception. Figure 8(a) is an example of this. If the *f-map* fails, the "real" resource in the virtual machine VM$_x$ is not realized by the VMM at level $x - 1$. The VMM receives a VM-*fault*. Figures 8 (b) and (c) are examples of VM-*faults* at a VM level and the real machine level. To support a recursive virtual machine systems we will adopt the simple naming convention of adding level subscripts to the VM and translation function identifiers. In the sections to follow we will look at two devices that fulfill the functionality of the model described here and also examine MIME's implementation.

4.2 Direct Address Translation

Other implementations of virtual machine operating systems and their formalizations introduce extra hardware to simplify the multi-level paging that will occur under the VMM. In IBM's VM/370, [1, 13, 14] we see the introduction of firmware support, referred to as the VM-ASSIST that has the effect of significantly reducing the number of privileged instruction traps the VMM has to manage[4]. For this section we can treat the VM-ASSIST as a DAT management in VM/370, viewed as a blackbox device that intervenes in virtual memory address translation to compose a virtual machine's virtual memory address translation map and that of the virtual machine monitor. The net effects of VM-ASSIST are the single translation map that

[4] Specifics about IBM's VM-ASSIST are scarce and protected under IBM patents.

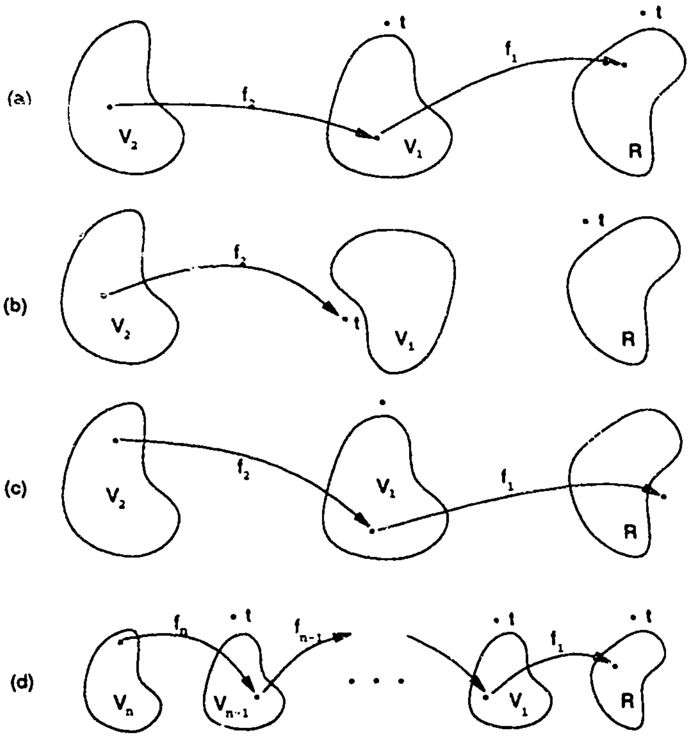

Figure 7: Recursive *f-maps*: The bean shaped areas define the collection of resources for virtual machine V, or real machine R The function f maps virtual resources v to real resources r. The subscripts denote the virtual machine level. The real machine is a virtual machine at level 0. t suggests a trap point for an unrealized resource. The correct translation of a resource in a virtual machine, VM_n, will realize a real resource on the real machine R without causing a VM-*fault*.

allows a virtual machine to page without creating additional VMM sponsored paging, and the reduction of the work a VMM needs to do when managing a privileged instruction fault. The mechanism that is used to compose the address translation maps is proprietary, hence we can only hypothesize about its management. In the following section we shall look at a formalization of a device similar to the DAT described in this section.

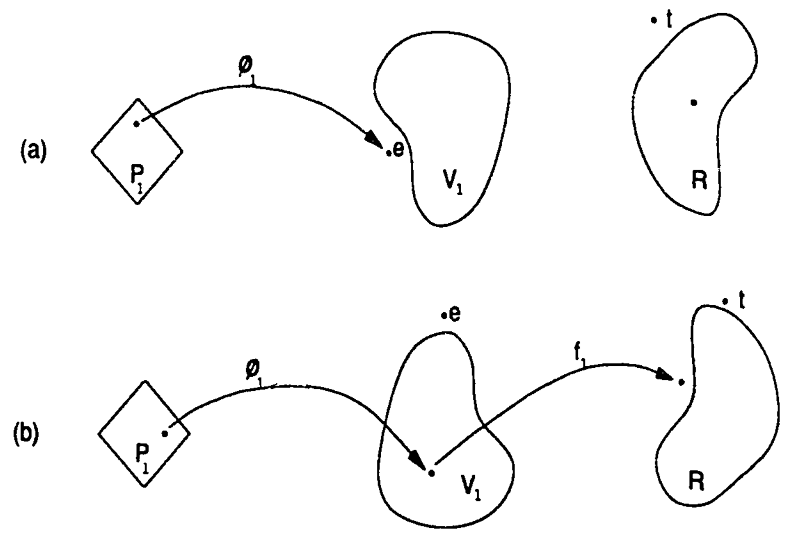

Figure 8: A Process exception and VM-*fault*: (a) describes a *process fault* caused by process P_1 executing in VM_1 when it accesses an invalid address. (b) describes a VM-*fault* caused by VM_1 when it accesses an invalid VMM address. ϕ_1 and f_1 are the address translation functions described in section 4.1.

4.3 Hardware Virtualizer

In the past, various authors [4, 5, 9, 10, 11, 24] have presented motivation for the management of a device Goldberg first called a Hardware Virtualizer, (HV). The chief function of this hardware virtualizer is to map resources efficiently such as the virtual memory of a VM, to real ones on the real hardware. For virtual memory, this function is defined as the composition of *multi-level* paging requirements into a single level operation. The HV allows a VMM to produce a composed[5] translation map for all virtual resources under its control. This results in a faster resource realization process, since at most one level of translation will occur for a resource that is within the boundaries of a virtual machine VM_x. The composition of the address maps may be dynamic or static, based on the mapping algorithm. Later we will look at possible implementations of a HV, but one trivial example can be

[5] [4, 11, 24] present several varying formal specifications that support and develop the hardware virtualizer concept.

contrived for dealing with machine contexts. An *in-bounds* set of registers classifies valid address ranges. If software executing on VM_x generates an address that is in bounds, as seen by the MMU and the *in-bounds* register set, the VMM could then page in the required pages. If the address was not *in-bounds*, the VMM may issue a memory access violation exception to the faulting VM. The *in-bounds* register set would be adjusted as memory conditions change and other virtual machines acquire the CPU.

4.3.1 Impact And Design

The composition of the process resource name model and the virtual machine resource name model provides for a clean and efficient method for resource name translation. We can clearly see that for any VM there will only be one application of the *ϕ-map*, preceded by n applications of an *f-map*. Thus if *f-map* operations are inexpensive in comparison to the *ϕ-map* operation, then the $f(n) \circ \phi$ composition will be easy and inexpensive to implement. Noted that when a process is running directly on the hardware the model permits the usual *ϕ-map* translations. In the model, the f and ϕ functions have different purposes, and since the model does not restrict the form or the inter-relationship of the two functions, they may be implemented in whatever fashion improves the name translation operation. Goldberg points out that this model makes the choice of *f-map* implementation independent of the *ϕ-map* function, which is best modeled after the process model that is in use. For instance, the real hardware may use relocation bounds or paging methods to achieve ϕ, but this will have no effect on the implementation of f. The design of the hardware virtualizer must consider the following points:

1. f store.

2. A mechanism to invoke f.

3. Map composition facilities; dynamic composition versus static composition.

4. VM-fault handlers.

All VMMs manage *f-maps* that describe the relationship between virtual machine resources at two adjacent levels. To do so, the VMM must save the *f-map* data somewhere that is invisible to the virtual machine executing under it. This *f-map* data is accessed only by using the VMM's level number. Of course the level number itself is invisible to the VMM. Figure 9 describes a virtual machine table (VMTAB), which is indexed by the level number. The VMTAB is manipulated by the current VMM to produce a pointer to a virtual machine control block structure, VMCB, which holds all the relevant resource translation details for the *f-maps.* The processor map entry of the VMCB includes processor state information and other coprocessor state information. The I/O map translates I/O resources that are not covered in the virtual memory map. The VMCB structure may also be used to achieve special virtual processor capabilities, like special instructions and operations. Special instruction capabilities may be added by the addition of F-line and A-line exception handling to the VM's VMCB.

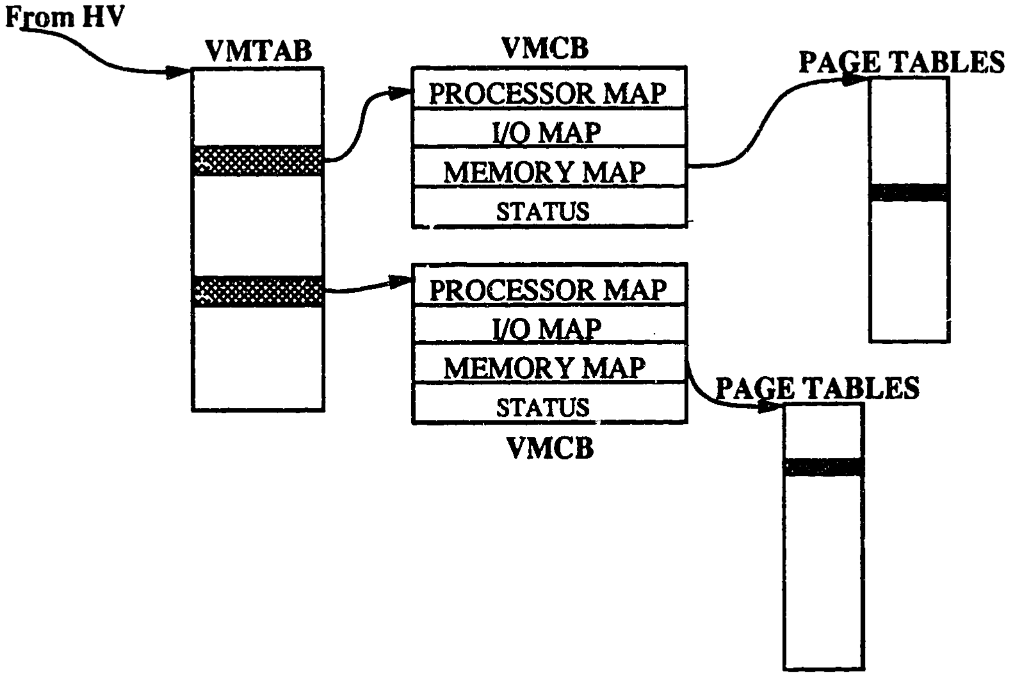

Figure 9: The VMTAB and VMCB.

4.4 Virtual Memory In MIME

Since MIME's target architecture, namely that of the Sun-3, does not support any hardware virtualizer or DAT, the VMM will execute virtual address translation in software. The virtual memory management scheme will provide a simple map composition mechanism in software. The VMM memory management system is similar to the notion introduced by Dobosiewicz et al.[6]. Address translation can take place at two levels: the VMM translation being the top level and the VM translation being the bottom level. MIME's VMM implements dual level translation composition. A virtual memory address, A, generated by a process executing on a virtual machine will first be mapped using the virtual machine's page tables to a virtual machine memory address, A', and then translated using the virtual machine monitor's page tables to a physical address A''. When a virtual machine page faults[6], the following three points classify the existing page status:

1. The page does not exist in the VMM real memory nor does it exist in the VM real memory, but the page is in the virtual address space of the faulting virtual machine. If this is the case, the page is paged in by the VMM.

2. The page does not exist in the VMM real memory, but it does exist in the VM real memory. If this is the case, the address map of the faulting VM is altered to permit access of the page. No physical paging occurs.

3. The page is outside the VAS of the faulting VM. Here the virtual machine is handed a bus error exception that it may do with as it pleases.

In the following subsection we will examine some of the details of MIME's implementation of the software virtual address translation maps.

4.4.1 Virtual Memory Context Partitions

To manage the Sun-3's MMU address space more efficiently MIME places a partition restriction on the virtual memory model. The MMU as mentioned in section

[6] We shall use the terms page fault and memory access violation interchangeably.

2.2.2 is an eight context MMU. The VMM's virtual memory model restricts an executing VM to four of the eight MMU contexts. This permits the VMM to maintain two *ready* VMs in its *ready* queue. The implication of this virtual memory model is that each virtual machine that runs in the VMM will execute with excessive overhead in half of the available virtual memory address space. To extend the virtual memory virtualization to the full range of a real Sun-3, the VMM employs a simple context replacement algorithm. The virtual memory replacement algorithm supports the VMM in projecting a full Sun-3 address space of eight MMU contexts, by replacing one contexts of a faulting VM with the next context the VM is attempting to access. To achieve the replacement strategy, MIME could apply a complex context swapping algorithm that picks the oldest context, but this adds a greater overhead to the VMM[7]. Since one of the principal design goals in implementing a VMM is the simplicity of the system software, MIME's VMM employs a simple tactic in which a faulting VM's address space is victimized for a page. The strategy attempts to replace the current VM's MMU context with the new one being referenced. The argument for supporting such a strategy is as follows:

1. It is unlikely that the current context will be used by an operating system running in the virtual machine next if it is not a kernel context and the operating system is implementing a fair scheduling algorithm.

2. Having two virtual machines in the ready queue is better than having only one, while striking a balance between the extra paging to VAS's in the VMM to meet the expectation of the operating systems running on the virtual machines.

3. Considering the cost of a Sun-3, it is unlikely that the number of VMs will substantially exceed two.

In this fashion, executing VMs may effect a memory management context change but at the price of their most current memory context being victimized and replaced. The replacement of one MMU context with another is a simple process. To effect a

[7] Monitoring access to the MMU contexts, etc.

context change, the VMM need only alter the virtual machine context structure of the requesting VM to reflect a new context value. The new context points to one of the available eight contexts whose pages are not in physical memory, and will have to be paged in on a demand basis. In the next subsections we will deal with issues surrounding demand paging.

4.4.2 Virtual Devices In Virtual Memory

In the Sun-3 architecture, all I/O devices are mapped into virtual memory. The I/O devices are mapped to special pages in virtual memory and are flagged as such in the **pmeg** entry of the page, (see section 2.2.2). To improve on virtual memory access by I/O devices the architecture supports *virtual* direct memory access or DVMA. This allows devices to read and write virtual memory buffers directly, leaving the CPU free of memory transfer cycles. DVMA greatly simplifies virtual I/O device management in a VM environment. No special or complex strategies have to be enacted to manage untrusted channel programs that communicate with special physical memory addresses, as was common to early versions of virtual machine operating systems[5]. An interesting effect of DVMA in a VM is that it becomes a simple virtual memory client in a virtual machine environment. Hence DVMA access to virtual devices in a virtual machine is implemented as DVMA to virtually mapped I/O pages that the VMM administers.

4.4.3 Memory Faults And Paging

As mentioned at the start of this section, MIME employs a two level paging process. The levels are distinguished by who begins the paging process. The VMM or a VM may begin the paging. To describe the paging mechanism let us adopt the example case of MIME's virtual machine monitor supporting two virtual machines, VM$_x$ and VM$_y$. In our example, process[8] $P_{x,i}$, executing on VM$_x$, attempts to access a virtual memory address that is in a page that does not currently exist in the

[8] It does not make much difference to the paging system if the process is a supervisor or user one.

virtual address space of VM_x. This results in a memory access violation that hands control over to the VMM. On examination of VM_x's VMTAB entry, the VMM can determine what action needs to be executed for VM_x to rectify the access. The possible actions fall into two categories, a VM-*fault*, and a VMM-*fault*. The following paragraphs examine these two categories.

VM-fault

A VM-*fault* is defined as the access to a page, by a VM process, that is logically in the virtual address space of the VM, but is not physically in memory. In our example, the violation occurred when process $P_{x,i}$ attempted to read or write a page that was not physically in memory. The VMM handles this fault by executing virtual machine VM_x's *bus error* interrupt handler at a user-level. VM_x may then attempt to page the offending page in from its backing store. In doing so the virtual machine will have to find a free page to use from its pool of pages based on whatever algorithm it is employing for page replacement. All protected references to the physical machine state will be honored with the values currently in the virtual machine's machine state structure, hence the memory address space as seen by VM_x will only reflect the virtual memory address space defined for the virtual machine. Noted that if the address that caused the fault is an invalid address, VM_x's bus error interrupt handler may choose to pass the error directly to the offending process.

VMM-fault

If a page is logically visible to VM_x, as defined by the virtual machine state structures in the VMM, but is not in physical memory, a VMM-*fault* is defined as an access to the page, by a process $P_{x,i}$ of virtual machine VM_x. Here the VMM has to find a free page in memory so that it can move the page from backing store back into physical memory. This type of page fault is handled by the VMM, and as such is invisible from VM_x which was the current virtual machine at the time of the fault. On processing the page fault the memory access instruction is re-executed. This memory fault is also possible for VMM supervisor pages that get swapped in and out of physical memory, and just result in simple paging of the needed pages.

A critical fault that falls in this category it that of an invalid address access by the VMM. The occurrence of this event is fatal to the correct operation of the virtual machine monitor.

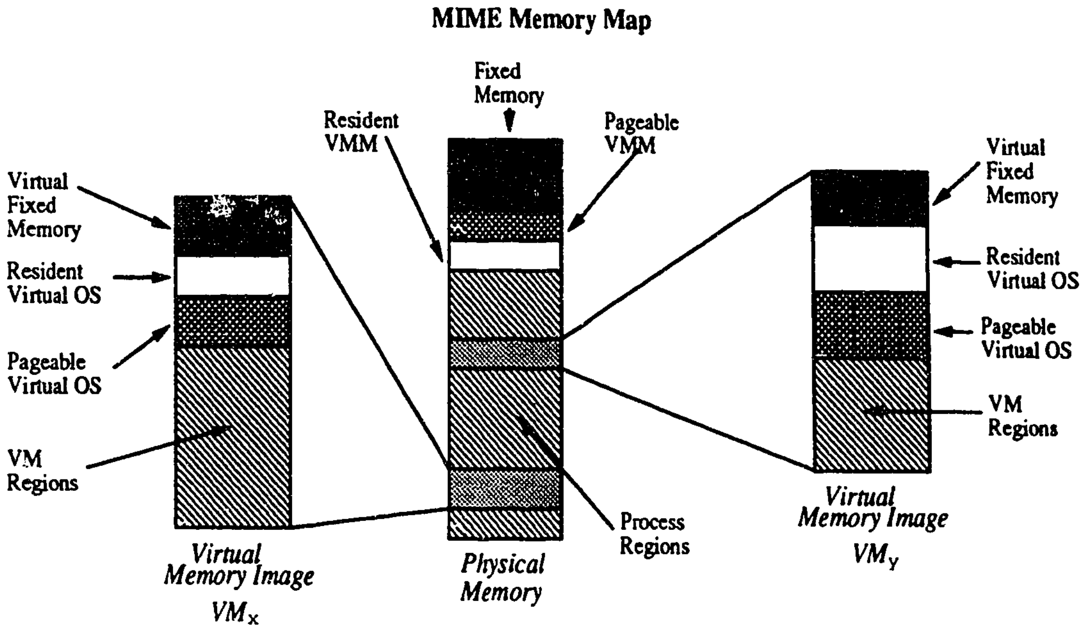

Figure 10: Simplified Memory Map.

In the above cases, paging of virtual memory pages from secondary/backing store to physical memory is done just as it usually is done in traditional operating systems. The page fault causes the virtual memory module of the VMM to victimize a physical page by flushing it to secondary store and bring the required page from secondary store into physical memory. After the page is *faulted-in*, the offending memory access is re-attempted in the same machine context that was current in at the time of the initial fault. Figure 10 provides a simplified view of memory map that MIME uses.

Chapter 5

Other Resources In A Virtual Machine Environment

In this Chapter, we will look at some of the other issues that surround the effective management of a virtual machine operating system. The VMM nucleus software is responsible for the effective virtualization of all hardware resources in the architecture. This includes network devices and drivers, hardware clocks and counters, disk subsystems, and serial ports and drivers. The VMM is also responsible for the generation and provision of any artificial resource that will be exported by any extension of the bare machine interface. All these devices are represented in the VMM as δ contexts. A δ context is either an explicit value or an offset value that when applied to the real hardware value will result in the values to be used for the current virtual machine. Each system resource has its own transformation function that will take a δ context to the real one that the virtual machine expects when it is executing. To maintain virtual machine system transparency the δ context transformation is hidden from the individual virtual machines. In the following subsections we will examine briefly some of the more critical system resources, and how MIME maintains and manipulates the various δ contexts, all of which are part of the a virtual machine's context block.

5.1 Time Of Day Clocks

The Sun-3 supports a time-of-day clock. Virtual time-of-day clocks are implemented by the virtual machine monitor as a monitor data structure that provide each virtual machine its copy of the time-of-day device. These virtual time-of-day clocks could be kept as either absolute values or as relative offsets from the real time-of-day clock that the VMM controls. MIME employs relative offsets for its representation of the virtual clocks. The offsets are represented as signed values in the virtual clock register that is part of a VM's VMCB structure. MIME treats the virtual machine's virtual clock as a *δ clock* to which the real clock value is added to obtain the correct time value for the virtual machine.

5.2 Network Support

VMM manages one physical machine, and several virtual machines From a network perspective, given that we only have one real machine, there exists truly only one real machine identifier. This machine identifier is used by network protocols to address the physical machine. In a virtual machine environment there is some difficulty in multiplexing this identifier amongst all the virtual machines that are running in the system. A simple method that extends the machine network interface of a virtual machine environment would consist of aliasing the physical machine's address to several addresses. There would be a one to one relationship between the addresses and the VM and VMM. This means that the VMM would have to honor more than one name as the network identifier that is reserved for this physical machine. This method of extending the virtual machine interface to the network drivers does present some implementation difficulties, but does offer a short and clear algorithm to manage virtual network devices. The method works especially well with broadcast-based network protocols like ETHERNET[1][16]. The underlying notion in the method is not new and indeed is used in ETHERNET networks to implement network gateway machines.

[1] IEEE Standard 802.3.

5.2.1 Network Support in MIME

MIME's network facility is based on the idea of aliasing several ETHERNET addresses to one physical machine. Every VM running in the VMM has its own singular ETHERNET address. The network driver of the VMM copies all packets that appear on the network into network buffers that are readable by all virtual machines. In this way every VM in the environment will get to see a true and realistic view of the network. All packets that appear on the network are copied by the VMM to the network device driver buffers of all the VMs running. The one weakness of the scheme is the vast amount of copying that will have to take place in the VMM. If there are four VMs running, MIME will have to present a copy of every incoming ETHERNET packet to each of the four copies of the VM ETHERNET drivers. To improve on this bottle-neck MIME implements a network driver mapping strategy that allows for the creation of only one virtual copy of the network driver. This copy is mapped into the address spaces of all the VMs in turn as they are given a CPU time slice. This means that the VMM need not make several copies for the several VMs. To further improve the virtual network drivers, MIME will optionally short-cut outgoing network traffic, if it is addressed to a locally running VM. This short-cut takes the form of copying the outgoing traffic to the input buffers for the local VMs. This has the effect of making the physical machine appear to the outside world somewhat like a gatewayed machine, where there is outgoing traffic if it is addressed to machines out of the sub-net.

5.3 Disk Subsystem

This section deals with the virtualization of disk resources for a virtual machine environment. This is a thorny issue in virtual machine operating system principles. The problem here may be stated as follows: given we have one real machine with several real disks, how can we virtualize the disk resources so that each virtual machine running in the environment can see a complete and consistent view of the disk resources available to it. We cannot just create true virtual copies of the disks and hand one to each VM. The VMs would write all over each others disk space and violate the basic virtual machine system requirement of encapsulation. What is needed is a scheme that virtualizes the disk resources in such a way that each VM sees what appears to be a complete set of disk resources, although smaller-then-real disk resources.

5.3.1 Mini-disk Partitions

One simple solution to the disk resource question, is the partitioning of the disk resources on a static base to satisfy all the VM's running in the VMM. If the total disk resource in a system is 700 Mbytes over all partitions of the physical disk mediums, we can configure our VMM to provide a fixed maximum disk space, say 70 Mbytes, to each VM in the system. This places an upper limit on the maximum number of VMs, and is based on the division of the disk resource into the number of VMs. In this scheme a disk resource request by a VM would result in reads and writes to a virtual disk partition, whose dimensions are statically known. This is the disk virtualization strategy used in IBM's VM/370 virtual machine operating system[1, 14]. The virtual disk partitions are referred to as **mini-disks**. A chief handicap of this scheme is its simplicity. A static sized mini-disk on one VM may fill up to maximum, while one on another VM may remain empty, hence leading to an inefficient usage of expensive and limited resources. The strategy does not allow for optimal use of the available disk space as VM requirements change over time, because of its static nature. The strategy's performance depends on the expertise of the operator who defines the initial system configuration.

5.3.2 Dynamic Disk Partitions

In MIME we decided to go a slightly different route in an attempt to improve on disk space usage and provide greater flexibility to the creation and management of disk resources. The concept employed in MIME's disk virtualization is more complex than the previous mini-disk strategy. A basic component of the idea is the provision of a minimal file system by the VMM. The file system is supported in a *hyper* disk that is an extension of the physical disk. The file objects in this file system constitute the different virtual disk (VD) partitions for the individual virtual machines running under the VMM.

File Objects

The file objects in the minimal disk partition system define a collection of disk blocks that belong to a virtual disk. Typically the disk blocks marked by this file object, reside only on one physical disk partition. It is not difficult to extend the system to support a more complex cross-disk allocation mapping that allows the virtual disks to span across physical disk boundaries. The elements of the file objects are disk block addresses. These disk blocks may be discontinuous on the physical disk, although it is desirable that they be contiguous for performance reasons. A VM disk access for a virtual disk partition is manipulated by the VMM so that the access results in an access within the bounds of the virtual disk that is projected by the file object. File objects are created by the VMM file system module with a size specification. The creation size may be manipulated to realize a near mini-disk strategy, in which the total disk space is divided up statically in advance, with no dynamic disk space available. File objects are allocated in an optimistic fashion, whereby the sum of the disk space presented to all VMs may be larger than the total physical disk space. File objects are added and deleted to virtual disks on demand as VMs grow and shrink their disk requirements.

Disk Operations

Operations on a virtual disk produce two results. They add or delete file objects to a VM's virtual disk. The VMM maintains disk configuration tables that define a VM's current disk geometry. When a VM accesses its virtual disk, the VMM translates the access to the appropriate file object in the *hyper* disk. If the file object does not yet exist, and the VM has not filled its virtual disk, the VMM will attach a new file object[2] to the VM's virtual disk, and continue the access.

In a simple implementation of this scheme the VMM intercepts all disk usage

[2] Free file objects are maintained by the VMM in a free pool.

statistics request by VMs. It returns the usage statistics for the individual VM's virtual disk partition. This can cause some problems. For example, when a virtual machine, VM_x, expects 50 Mbytes of available disk space and gets swapped out, another virtual machine, VM_y, can use up some or all the *physical* disk space, leaving less than 50 Mbytes of available *physical* disk space. Now when VM_x gets to run again, it could attempt an access of its virtual disk that would translate to an invalid *physical* disk location.

VD

Using the dynamic disk partition scheme allows disk resource allocation to be done in a more flexible way as virtual file systems grow and shrink in their respective VM. The VD system has the advantage that the VMM creates the virtual disks from the *hyper* disk, which is larger than the sum of the physical disks. In dividing the *hyper* disk into virtual disks, the VMM runs the risk of being asked to supply disk resources that are outside the physical limits of the component disks. A related disadvantage of this VD system is the problem of choosing an appropriate size for the various virtual disks, such that you provide effective disk resources while minimizing the chances of filling up the *physical* disk often. The current VD system does not provide any recovery mechanism for this condition.

Disk Partitions And MIME

There still are some weakness in MIME's virtualization of disk resources. These weaknesses are common to both the mini-disk strategy and MIME's dynamic disk partition scheme. The most significant failure of both schemes is in the waste of computer resources. This happens when an operating system executing in a VM attempts to access its virtual disk partition optimally by doing some computation, but the VMM re-computes the corrected disk trajectory to access the correct disk partition. Another weakness is the multiple copying of data from the buffers in the VMM address space to equivalent ones in a virtual machine's address space. Finally there is an additional computational burden placed on the VMM to maintain consistent views of the virtual disk geometries under all conditions.

5.4 Serial Devices

In the current implementation of MIME, virtual serial devices are not given much attention. This of serial devices is caused by severe limitations in the machine architecture. The Sun-3's two AMD chips support a maximum of four serial devices. Two of these devices are used in the management of the machine console and keyboard. The other two devices are available for system use. In MIME's VMM, we chose to use the four serial devices to create a console device for each running VM. This implies that the environment exercises an upper limit of four virtual machines with a console device. As a side point, a VM may exist without a serial console if the VM makes special arrangements to have its console device elsewhere, say over a network connection. In effect MIME chooses to virtualize the serial devices only partially, although MIME can create virtual device drivers for the serial lines.

5.5 Debugging Tools

One difficult task in operating system implementation is the debugging of a kernel. Since kernels are special objects, they have special debugging requirements. Kernels are special since they do not have any support environment, unlike processes in a operating system. Some of the special debugging requirements that kernels have, are the need for an effective way to communicate with a kernel that is being debugged, and mechanisms for spying on various kernel events and structures. In earlier chapters, we motivated the need for developing virtual machine operating systems with their help in developing and debugging new conventional and unusual operating system designs. On this point, the greatest asset to an operating system developer is a powerful kernel debugger. MIME, in its simple incarnation, provides a minor but useful interface to operating system kernel debugging. This is possible because MIME maintains complete machine context structures and the ability to disassemble operations, when it attempts to emulate certain instructions. MIME provides that useful environment behind the operating system executing in a virtual machine. Using the environment the VMM has developed, it is conceivable that a specialized kernel debugger can be constructed that takes advantage of these structures and has tools for controlling a host VM. Theses tools may be as conventional as the usual process debuggers, breakpoints and spys, or may have uncommon features like event watchers, and instruction emulators. In this area MIME has not developed much support yet. It is hoped that as the tools and features develop we will see more support for this facility.

Chapter 6

Conclusion

Virtual machines have been with us for over two decades and they will remain for a while longer. They have appeared under several different incarnations and flavors: to improve on architectural limitations, to provide specialized environments, or to offer a full-blown user-tailored computer system.

In the age of window systems, many packages offer user environments that multiply user effectiveness by generating virtual terminal devices. Some devices are special, like a drawing canvas, while others are simply terminal emulators like X11's Xterms[26].

Xerox, in their design and implementation of the Smalltalk-80[8] environment and language, developed a special-service virtual machine that was the operating system and the user environment - an example of a virtual machine that supports a specific programming environment.

IBM offers an implementation of a full-blown virtual machine operating system the - VM/370. As CPU's in micro-computers and workstations become faster and more capable of tasks previously reserved for the mini and mainframe computer arena, we can see this platform is becoming more supportive of virtual resources.

In this chapter, we will examine related efforts and future directions of this work, and conclude with a few closing remarks. We present here some of the work on full-scale virtual machine operating systems.

6.1 Related Work

In the history of virtual machine operating systems, there have been several related research and commercial efforts. Most of the work has been focused on specification, design issues, and implementation. Some work also focused on security and associated issues in virtual machine systems. Some have been purely research efforts, while others have been commercial efforts.

6.1.1 VM/CMS

The most notable effort in commercial Virtual Machine Operating Systems has been IBM's Virtual Machine Facility/370 (VM/370) [27, 14], which appeared in the early seventies[1]. This effort by IBM brought to light the feasibility and usefulness of virtual machine systems. As virtual machine systems became more usable and practical they left the realm of purely academic fancies. IBM recognized the virtual machine system as an effective tool for the distribution of computer resources. Two decades later IBM's VM/370 is still the leading commercial product in this area. There has been little effort at bringing this style of operating system to other hardware platforms, namely the personal workstation. Several studies have been done on the effectiveness and efficiency of the VM/CMS operating system [12, 13, 27]. Most of the work has been focused on improving the efficiency of the operating system, as illustrated in the section 4.2 presentation of the VM-ASSIST feature. The VM/CMS operating system is built around two components: the Control Program (CP) and the Conversation Monitor System (CMS).

Control Program

CP is the virtual machine monitor that is responsible for the management of system resources. It creates virtual machines that support several System/370-compatible operating systems[2]. CP supports anything, albeit not necessarily efficiently. The

[1] IBM has worked with the idea of a virtual machine system as early as the mid sixties on their 360/40,44,67 machines[1, 5].

[2] CP supports DOS/VS OS/MFT, OS/MVT, DOS/VS, OSVSI, SVS, and MVS amongst others.

machine interface provided by CP is that of a System/370 and its derivatives. CP provides only those services required to resolve the perspective differences between the point of view of a virtual machine and of the real system; CP also dispatches virtual machines and manages the real hardware. CP supports features that allow sharing of executable virtual machine code, locking of specific page frames in real memory and the allocation of device and channel resources to special virtual machines on a dedicated basis.

Conversation Monitor System

CMS is a special minimal operating system that depends on CP for its execution. CMS is used typically for interactive program development and personal computing, and supports a single user on a VM/370 virtual machine. Multiple CMS sessions are capable of code sharing. The CMS environment is taken to provide a single user work space on a VM/370. This work space includes language processors and compilers, file access methods, editing, text formatting and debugging capabilities.

6.1.2 XINU

The XINU virtual machine operating system was designed and implemented as a research and teaching facility. It is used principally for teaching operating system courses, providing the students with access to a bare or minimal machine interface. The XINU kernel [3] is one example of a virtual machine system implementation that executes under a general-purpose operating system. The XINU kernel may be viewed as a complex simulator that takes the extended machine interface provided by the host operating system and generates a raw machine interface. The XINU kernel was initially implemented under the Berkeley 4.2 Unix[3][25] operating system running on a set of LSI-11 machines linked by a store-and-forward ring network. The kernel operates a XINU machine simulator with multi-user support. The kernel runs under the native Unix environment as a large single-process thread. Network communication concepts are implemented in the virtual machine environment as

[3] Unix is a trademark of AT&T Bell Laboratories.

XINU machines that talk to each other over RS-232 ports. The RS-232 ports are simulated by using Unix sockets. The ports introduce noise and connection failures to enhance the simulation further. In the version of the XINU kernel examined, the virtual machine interface is not extended to memory management. This was because of the complexity of building and interpreting memory management instructions in a simulator that supports the host machine's native instruction set. XINU's largest weakness as a native language virtual machine implementation is that it executes in a single state. This means that XINU cannot support privileged instruction protection by dual-level instruction sets.

6.1.3 CPM

At the University of South Florida, Kim et al. [15] developed a generation of virtual machine environments on several simple hardware platforms to evaluate concepts in fault-tolerant distributed processes. The system was developed in Concurrent Pascal, and was implemented on a MC68000 architecture. CPM has two components: a Kernel, and a Code Interpreter. The function of the code interpreter is to support a stack-oriented instruction set. The kernel provides the basic mechanisms for concurrent execution of processes and manages low level I/O activities. Network communication facilities are managed by an I/O processor that communicates over one of four serial ports that operate at 9600 baud.

6.2 Further Work

In this Section, we would like to explore several extensions to MIME. As mentioned in the Chapter 1, MIME is a part of a long-term research effort to produce a system that will support operating systems development research and education. These two goals represent most of the motivation behind the work on designing and implementing MIME.

6.2.1 MIME In Education

With a full-featured implementation of a virtual machine operating system like MIME, we can, with relative ease, develop courseware for courses in operating systems, both for graduate and undergraduate level instruction. In most of today's academic facilities we can find many small personal computers and a few large-scale multi-user computing resources. This migration of CPU cycles from the large multi-user base to the personal computer[4] has created some difficulties in the area of operating system education. The following example highlights the problem admirably. An academic instructor teaching a graduate level course on operating system principles would like to give students some hands-on experience with context switching, or process scheduling. At his disposal, he/she has several Sun-3 workstations that are running some proprietary operating system, called XOS. To achieve his aim the instructor develops a toy operating system, ToyOS that allows the students to execute their context switching or process algorithms. All that the students need do is shutdown XOS on the workstation and install their modified versions of ToyOS to test their implementations. This has a few discouraging effects.

1. Bringing down XOS, the student destroys whatever the XOS environment was supporting.
2. The student may have undue access to privileged data and programs, for instance over the network.
3. The version of ToyOS the student runs may be faulty and affect other machines in the networked environment, either accidentally or deliberately.

So the instructor of the course is unable to provide the students with a secure environment in which they can test and develop algorithms for operating system principles. If the instructor had a VMOS for the workstation architecture he/she planned to support for the course, it would then be a simple process to create a virtual machine partition for each student on the VMOS, that would also be

[4] Here a personal computer can be taken as the traditional PC, or a computer workstation, like the Sun-3.

supporting the proprietary XOS. MIME's encapsulation would protect against the three effects that where listed above and also provide a more flexible method of application development and testing.

MIME's principal *raison d'être* in the educational arena is its usefulness as operating system class courseware. Students can develop operating systems that test various issues and algorithms in a clean, simple and safe manner. The operating systems could then be implemented on the real hardware with little or no changes. This presents great possibilities for creative solution and/or management of operating system issues in the academic field. The arguments presented in this section may be extended to other areas of operating system research as we will see next.

6.2.2 MIME In Further Operating System Research

Dobosiewicz el al. [6] introduce a distributed operating system for the *universe* called MESS. The underlying principle in MESS is the virtualization of resources in a completely transparent way. The work on MIME is a founding element in the MESS research project. In developing a research tool like MIME we can design, develop and achieve efficient multiple operating systems, while evaluating concepts and structures in distributed operating systems. It is easy to do rapid proto-typing and testing of new and novel kernel concepts without the fear of interrupting normal day-to-day computing on the hardware base that is being used for the testing. Experimentation with the virtual network described by Dobosiewicz could be done in greater encapsulation and security.

MIME With A Twist

Another side of MIME that will assist operating system research is in the development and experimentation of various methodologies in resource management, sharing and implementation. MIME's support of resources in the Sun-3 architecture should not stop at only the real ones. MIME can be implemented with support for a variety of virtual devices that allow testing of concepts in these virtual devices. Consider for example, the management of a user process based file system in which a user defined process instantiates all file objects a user requires. The process would create a file system out of virtually addressable segments of memory. This is an idea similar to the one used in the implementation of Mutics[22]. Resources, in this environment, could then be represented by virtual address drivers. These drivers would instantiate real and virtual resources by reading and writing virtual addresses to system peripherals with maybe some filtration and emulation. A user in this system would, for instance, get a file printed by copying a range of virtual addresses to another address, where the printing spool device resides. In this operating system example, the uniform view given to the user would be a virtual address. The work elements of the system would be simple memory operations like copy, clear, read, write, refresh. Installation of user drivers would then be managed at the operating system level. This example is just one creative possibilities that MIME would allow.

6.2.3 Tomorrow's MIME

As work on the implementation of MIME draws to an end, we should expect to see work beginning on the definition of a virtual network. As mentioned in the previous section, there is ample motivation for this work. One motivation not mentioned though, is the resulting environment that comes out of having several networked computer resources that are generating VMMs and VNs. This environment will permit dynamic or static configurations of VMM's to create a cluster of virtual machines and resources. This virtual cluster could support a distributed operating system that takes into consideration the various virtual resources it believes it has, and generate a uniform and apparently homogeneous virtual compute base. This would be another feather in the nest for MESS. Universality of the distributed operating system, DOS, could be implemented at this uniform and homogeneous virtual environment by the addition of more virtual compute resources in an indefinite manner.

In tomorrow's MIME, we expect to see efforts directed at improving the performance of MIME with respect to the architecture–VMM combination. Several techniques may be investigated that attempt to improve MIME's performance. These techniques may be categorized as "Policy", "Interface compromise" , and

"Improved or new mechanisms".

Policy

To reduce the overhead and solve some installation management issues, a policy that dedicates resources to a *preferred* VM may be explored. These resources include a percentage of CPU time[5], real I/O devices as opposed to virtual I/O and pages of memory. In alternative VMM systems, the "virtual = real" policy is one interesting variation of the one just mentioned. Other policies that may be investigated are Goldberg's VMM or VM *streamlining.* These policies are based on the manipulation of the VMM of the operating system executing on the VM to take advantage of strengths or weaknesses in both the VMM or the VM. Reducing the amount of paging in a VM by increasing the virtual memory definition is one example that is based on the knowledge that paging in the VMM is far more efficient than paging in the VM.

Interface Compromise

This performance improvement approach involves the changing of the VMM to generate one or more specialized *extended* VM interfaces. The executing operating system could take advantage of this extension to the standard interface to improve performance. An example of this could be the possible extension of MIMEs' interface, in the current architecture, to support special F-line to A-line exception as super instructions that signal a special service request by target virtual machines. This means that the executing operating system in this virtual environment would be incompatible with the real hardware or another VM. The operating system software could possibly determine if it is operating in an extended interface, and if so re-configure itself to take advantage of the extensions offered by the VMM.

[5] Time slice.

Improved Or New Mechanisms

As new and improved hardware architectures for virtual machines arrive, we should be able to improve on MIME's VMM emulation of privileged instructions, hence removing a major performance bottle-neck. IBM showed that it is worth the effort to improve on the architectural design in search of further performance when they introduced the VM-ASSIST firmware changes to the VM/370.

As MIME and its virtual machine interface improves, we can hope to improve execution CPU time delays. The reduction of these values has the effect of reducing the penalties imposed on an operating system executing in a virtual machine. This means we can hope to achieve a better equivalence with conventional operating systems executing directly on the real machine. Better scheduling and memory and resource management methods may come to light with this aim in mind. We can also expect work to be done in the implementation of a hardware virtualizer device to replace the typical memory management unit device in the machine architecture–VMM combination. Given the hardware architecture explored in this work, it is not a difficult task to develop an application-specific integrated circuit (ASIC) to realize an HV as the one formalized in Section 4.3. Appendix B.2 shows a simplistic implementation of an HV in the Sun-3/MC68030 architecture.

6.3 Closing Remarks

As MIME continues to evolve, more and more issues in operating system design and implementation will have to be examined. MIME represents a step in the evolution of operating system development, research and educational tools; it has been, and still is, a large effort in implementation testing and experimentation. This effort at developing a virtual machine operating system for the workstation environment is not trivial, even though the task is well defined on most issues. Since most of the questions on virtual machine systems have been explored in the past, the topic has drawn little fresh work, even though there are still aspects of this art that have not been nailed down. The lessons learned with MIME improved and will continue to improve the understanding of virtual resources. For example, in the development of MIME it was recognized that a consistent access mechanism for system devices in the supported machine environment, made the kernel simpler. In the Sun-3, the VDMA for I/O devices makes the management and operation of device drivers a simple extension of the virtual memory operations. The lessons learned in developing a virtual machine operating system have also helped to highlight various problems related to resource management in a complex systems. For instance, in MIME's current implementation, there is little or no support for system resources that do not partition well, such as a tape driver. An attempt of integrating these classes of device will have to be made so that virtual machine operating systems can present complete machine interfaces.

Virtual machine systems have given us the ability to examine the role devices play in a machine architecture. The lessons learned in resource management for multi-user, time-shared, and distributed systems have benefited from the virtual machine system interface. A great weakness that has limited the role of virtual machine systems in modern experimental scale computing, has been the complexity of developing complete system implementations. The work on MIME has been an attempt to overcome this large obstacle in the way of research and education in virtual resource management.

To conclude this thesis, we would like to state that this work is an attempt at providing a working virtual machine system platform that will permit further efforts in these areas of operating system research and education. As MIME continues to mature and stabilize as a research tool, it is expected that MIME will support a suite of operating systems in an effort to explore various aspects and issues in operating system principles.

Bibliography

[1] R. Adair, R. U. Bayles, L.W. Comeau, and R. J. Creasy. Virtual Machine System for the 360/40. Technical report, IBM Cambridge Scientific Center, 1966. Report No. G320-2007.

[2] M. Arden and A. Bechtolsheim. Sun-3 Architecture: A Sun Technical Report. Technical report, Sun Microsystems, Inc., 1985. Revised August 1986.

[3] J. Bachrach, J. Walleriusa, and J. Paris. A XINU Virtual Machine. In *USENIX Summer '85 Conf.*, pages 384–355, 1985.

[4] G. Belpaire and N. Hus. Formal Properties of Recursive Virtual Machine Architectures. In *Proc. of the 5^{th} Sym. on Operating Systems Principles*, pages 89–96, November 1975.

[5] J. P. Buzen and U. O. Gagliardi. The Evolution of Virtual Machine Architecture. In *Proceedings of National Computer Conference*, pages 291–299, 1973.

[6] W. Dobosiewicz, M. R. Eskicioglu, P. Gburzynski, and A. Mutiso. MESS–A Distributed Operating System for the Universe. In *Second IEEE Workshop on Future Trends of Distributed Computing Systems*, pages 208–214, Cairo, Egypt, September 1990. IEEE Computer Society Press.

[7] F. Douglis and J. Ousterhout. Process migration in the Sprite operating system. In *Proceedings of the 7th International Conference on Distributed Computing Systems*, pages 18–25, Berlin, West Germany, September 1987. IEEE.

[8] Adele Goldberg and David Robson. *Smalltalk-80: The language and it's Implementation.* Addison Wesley Publishing Company, ISBN 0-201-113716 edition, 1980. Xerox Palo Alto Reserach Center.

[9] Robert P. Goldberg. Hardware Requirements for Virtual Machine Systems. In *The Fourth Hawaii International Conference on System Sciences,, Honolulu, Hawaii*, pages 449–451, January 1971.

[10] Robert P. Goldberg. *Architectural Principles for Virtual Computer Systems.* PhD thesis, Division of Engineering and Applied Physics, Harvard University, Cambridge, MA, 1972.

[11] Robert P. Goldberg. Architecture of Virtual Machines. In *Proceedings of National Computer Conference*, pages 309–318, 1973.

[12] Robert P. Goldberg. Survey of Virtual Machine Research. *IEEE Computer*, 6(6):34–45, June 1974.

[13] P. H. Gum. System/370 Extended Architecture: Facilities for Virtual Machines. *IBM Journal of Research and Development*, 27:530–544, November 1983.

[14] IBM Corporation, Department D58, P.O. Box 390, Poughkeepsie, New York 12602. *IBM Virtual Machine Facility/370 Introduction*, IBM Systems Library edition. Order No. GC20-1800.

[15] K. H. Kim. Evolution of a Virtual Machine Supporting Fault-Tolerant Distributed Processes at a Research Laboratory. In *IEEE International Conf. on Data Engineering*, pages 620–628, April 1984.

[16] R. M. Metcalfe and D. R. Boggs. Ethernet: Distributed Packet Switching for Local Computer Networks. *Communications of the ACM*, 19:395–404, July 1976.

[17] Motorola Inc. *MC68020 32-BIT MICROPROCESSOR USER'S MANUAL*, 1984. ISBN 0-13-541467-9 (Prentice Hall).

[18] Motorola Inc. *MC68030 ENHANCED 32-BIT MICROPORCESSOR USER'S MANUAL*, second edition, 1989. ISBN 0-13-566969-3 (Prentice Hall).

[19] M. Nelson, B. Welch, and J. Ousterhout. Caching in the Sprite network file system. *ACM Transactions on Computer Systems*, 6(1):134–154, February 1988.

[20] Michael N. Nelson. Virtual memory for the Sprite operating system. Technical Report UCB/CSD 86/301, Computer Science Division, EECS Department, University of California, Berkeley, June 1986.

[21] Michael Newell Nelson. *Physical Memory Management in a Network Operating System.* PhD thesis, University of California, Berkeley, CA 94720, November 1988. Technical Report UCB/CSD 88/471.

[22] E. I. Organick. *The Multics System: An Examination of Its Structure.* MIT Press, 1972.

[23] John Ousterhout et al. The Sprite network operating system. *IEEE Computer*, 21(2):23-36, February 1988.

[24] G. J. Popek and Robert P. Goldberg. Formal Requirements for Virtualizable Third Generation Architectures. *Communications of the ACM*, 17(7):412–412, July 1974.

[25] D. M. Ritchie and K. Thompson. The unix time-sharing system. *Communications of the ACM*, 17:365-375, July 1974.

[26] R. W. Scheifler and J. Gettys. The X Window System. *ACM Transactions On Graphics*, 5(2):79–109, April, 1987. XACM Scheifler.

[27] L. H. Seawright and R. A. MacKinnon. VM/370-a study of multiplicity and usefulness. *IBM Systems Journal*, 18(1):5–15, 1979.

Appendix A

Exception Vector Table

Vector Number(s)	Vector Offset		Assignments
	Hex	Space	
0	000	SP	Reset: Initial Interrupt Stack Pointer
1	004	SP	Reset: Initial Program Counter
2	008	SD	Bus Error
3	00C	SD	Address Error
4	010	SD	Illegal Instruction
5	014	SD	Zero Divide
6	018	SD	CHK, CHK2 Instruction
7	01C	SD	cpTRAPcc, TRAPcc, TRAPV Instruction
8	020	SD	Privilege Violation
9	024	SD	Trace
10	028	SD	Line 1010 Emulator
11	02C	SD	Line 1111 Emulator
12	030	SD	(Unassigned, Reserved)
13	034	SD	Co-processor Protocol Violation
14	038	SD	Format Error
15	03C	SD	Uninitialized Interrupt
16 Through 23	040 05C	SD SD	(Unassigned, Reserved)
24	060	SD	Spurious Interrupt
25	064	SD	Level 1 Interrupt Auto Vector
26	068	SD	Level 2 Interrupt Auto Vector
27	06C	SD	Level 3 Interrupt Auto Vector
28	070	SD	Level 4 Interrupt Auto Vector
29	074	SD	Level 5 Interrupt Auto Vector
30	078	SD	Level 6 Interrupt Auto Vector
31	07C	SD	Level 7 Interrupt Auto Vector
32 Through 47	080 0BC	SD SD	TRAP #0-15 Instruction Vectors
48 Through 63	0BC 0FC	SD SD	(Unassigned, Reserved)
64 Through 255	100 3FC	SD SD	User Defined Vectors (192)

Table A.3: Exception Vector Table.

Appendix B

Implementations, Details And Specifics

B.1 Implementation Shortfalls In MIME

MIME is still in a developmental and experimental stage. This is because the complexity and effort needed to complete a fully operational version of a virtual machine operating system. At present, MIME's virtual machine monitor only manages a sub-set of its required tasks. The virtual machine monitor does not support scheduling, virtual machine context switching and timing. Virtual memory management is still in its infant stage. Without virtual memory MIME will only support one special virtual machine, as long as it is small enough to fit into the remaining real memory. The virtual device drivers are currently those device drivers defined by the Sun-3 EPROM and are not relocated anywhere else. In the current implementation of MIME, no effort has gone into effective management of instruction tracing. This has the added difficulty of causing the VMM to emulate every sensitive instruction. Optimization of the instruction trace interrupt routines could make this feature more manageable in a VMOS environment.

B.1.1 Implementation Platform Details

The current work on MIME is taking place on a Sun-3/60 with eight megabytes of physical memory and 700+ megabytes of disk storage. Most of MIME's code is written in "C" with a few assembler files. Parts of the VMM code are implementation changes to parts of the Sprite network operating system [23, 20, 21]. MIME's dynamic file systems for virtual disk resources is based on parts of Sprite's file systems, but without the process migration and file caching aspects [19, 7]. MIME is being implemented on a Sun-3/60 that is networked with other production-level workstations. This makes development and debugging of MIME a far more complex task. Once a more useful version of the VMM kernel is established, a stand-alone test environment will be necessary for rigorous testing and debugging.

MIME's chosen hardware platform has several weaknesses. These weaknesses relate to virtualization of resources that do not virtualize well, such as the machine console. The Sun-3 console as mentioned in 2.2 is hardwired to two of the four serial ports, and depends on the framebuffer device. This makes virtualization of the console impossible, and as such MIME does not provide any virtual device support for the console. Since MIME was implemented with portability in mind, we expect to develop the kernel on other workstation based machines as they become available without difficulty.

B.2 A Simple Hardware Virtualizer Implementation

In this section we would like to show a simple implementation of a hardware virtualizer device. This HV implementation will be specified for the Sun-3 hardware platform with a Motorola MC68030[18] microprocessor, but the design is also applicable in general to the MC68020[17] microprocessor. The HV described here follows, principally, the design specifications developed in [11, 9] with added support for the MC68030 CPU. Since the MC68030 microprocessor has an on-chip MMU, the HV will over-achieve the MMU's functionality by providing mainly virtual memory management support for a virtual machine operating system environment. As is mentioned in 4.3 this HV operationally replaces the MMU tasks. This over-implementation is not entirely equivalent and transparent, but does permit a more fluent implementation of the a virtual machine monitor. The difference in operation of the two devices, MMU–HV will require a change to the software that is created for the MC68030–MMU combination, to take full advantage of the MC68030–HV combination.

B.2.1 HV Design

Our implementation of the hardware virtualizer takes the shape of a single ASIC co-processor chip that takes the roll of a paged memory management unit. Although the MC68030 has an on-board MMU, the chip does permit the addition external MMUs. To minimize the effective differences as seen by the rest of the machine (Sun-3), the HV design will follow closely, where feasible, the conventional implementation of an external memory management device. The HV defines its own HV address space, which we arbitrarily choose as address space 4[1]. The chip has the following internal registers[2]:

- VMLEVEL register: This register is updated by the HV firmware to reflect the current virtual machine level number. The value of this register is 0 for the real machine.
- TLA register: A translation *look-aside* register that points to an associative table of virtual addresses and their physical translation. This register's value is different for every virtual machine. The associative translation table, ATT is used by the HV to improve the efficiency of address translation.

Manipulation of the HV is achieved through read and write accesses to the special registers, VMCRTL and VMSTATUS. Both of these registers reside in HV-*space*. By writing specific information to the VMCRTL register, a VMM operation at level

[1] Section 2.2 lists the other supported address spaces.
[2] HV internal registers are not visible to the host CPU.

n can create and destroy translation maps for a virtual machine that will execute at level $n + 1$. The VMSTATUS register provides translation status and error information about the operations of the HV. Since the registers exist in HV-*space*, they are invisible to the user-level processes. In this simple HV implementation, we adopt the following HV operations:-

- CREATE: This HV operation will create a level $VMLEVEL + 1$ translation map for a new virtual machine that will be spawned by the virtual machine monitor executing at level $VMLEVEL$. The instruction will return a virtual machine identifier in the VMSTATUS register.
- DESTROY: This HV operation will remove a translation map for a VM. The virtual machine is specified in the argument of the instruction and must be a valid identifier that originated from a CREATE operation at this level. The VMSTATUS register will show the success of the operation.
- SWITCH: This HV operation is executed for its side effect. It causes the HV to assume a level $n + 1$ resource translation. The instruction requires a virtual machine identifier as an argument. If the null identifier is used, a $n - 1$ level change will be done, except for the special case where the current VMM is at level 0, in which case the operations is a *no-op*. The instruction is used by the VMM to *switch* CPU control over to a child virtual machine. The instruction can also be used by a child virtual machine to preempt itself, forcing the parent VMM to take control of the CPU. This operation also alters the contents of the TLA register.

There is scope for more complex HV operations, like the implementation of special virtual processor features and capabilities, but we will restrict the design to the above operations only. The HV supports an externally read/writable VMCTX register that plays the same role as the CTX register in the Sun-3 MMU. The VMCTX register is duplicated in the VMCB of every virtual machine in the system, hence a VM context change will also result in the correct value in the register. The VMLEVEL internal register may be seen as an index pointer to a list of virtual machine tables. It is used in exception processing to determine which VM's exception handler to cal. and where to being the map lookups. Each VMTAB defines a composed resource translation map that maps a level n virtual resource to the real resource. This is done for all virtual machine that exist as children of the current virtual machine monitor. Since the Sun-3 architecture supports virtual mapped I/O, the only client of the resource translation maps is virtual memory.

B.2.2 HV Operation

The VMTABs makes up the *f-map* database that Goldberg described in [11]. The VMCB entry describes a page table structure that maps a virtual addresses into physical ones. The virtual address translation procedure used by the HV first tries to trace the virtual address in the ATT. If a match is found in the table, the translated physical address is used. If a match is not found, the address is recoded by the HV into a physical address using a similar translation process to the one described in section 2.2.2. The VMCB maintains a processor map that records the viewable CPU state. An entry in the VMTAB structure also classifies the top of the segment table. In this HV design a virtual address is made up of the composed value of the context register, CTX, and the low 27 bits of the virtual address. Figures 5 and 6 in chapter 2 describes how the composed CTX and segment value are used to index a segment table during address translation.

VM-Faults

During address translation, if a VM-*fault* occurs, the HV firmware will reset the VMLEVEL to one less then the current level. It also prepares a VM-*fault* exception structure on the top of the interrupt stack, and places the CPU into interrupt state. This causes the hardware to begin executing an auto-vectored interrupt handler. The fault is processed by the virtual machine executing at this level as an *f-map-exception* in which the virtual machine attempts to demand page in the needed pages. If it is successful, the faulting-child-VM's VMCB is modified to reflect the address space change, and the exception returns successfully. If it fails, the child-VM's VMCB is configured to represent a *bus-error* exception.

B.2.3 Summary

This HV description is designed to minimize the difference between the Sun-3–MMU combination and a Sun-3-HV combination. Except the differences mentioned in the previous paragraphs, the HV supports a paging operation similar to the MMU's one. Owing to the relationship between the VMTAB entries and the HV, this design has one advantage over the MMU design. Paging may be achieved on a single or multiple page scale, depending on the VMM's page replacement policy. This advantage may be used to carry out a page replacement policy like the used by the MMU operating on single pages, or a **pmeg** page replacement policy that pages complete **pmegs** to and from the backing store. Since the HV instructions are based on HV-*space* memory address access, it is easy to add compiler functionality of this device. This hardware virtualizer desciiption is incomplete and requires further work before the device could be used as *plug* replacement for the Sun-3 MMU.

Appendix C

Privileged And Trap Instruction Formats

MIME is required to service *supervisor operations* when ever a VM-*fault* occurs. To do this the VMM uses the data in Table C.4 to determine what *supervisor instruction* was executed by a VM. The VMM disassembles privileged instructions using this table to determine what the *faulting* VM needed. The *supervisor request* is serviced by the VMM by emulation or simulation of the operation. Appendix D shows some VMM functionality to manage some of these operations.

The instructions in Table C.4 define all the instructions that cause a VM-*fault* in the MC68020 microprocessor. The instructions can be divided into two major categories:

1. CPU state queries: This category, a VM requires information pertinent to its machine state. To manage these instructions the VMM will simulate the operation by returning the relevant value that is in the VM's VMCB structure.

2. CPU state changes: In this category a VM is attempting to change the state of the microprocessor. The VMM will simulate the operation by updating the relevant entry in the VM's VMCB structure. The VMM may also have to emulate the operation on the physical hardware, for example, a MMU context change operation.

Instruction Name	Bit Field Numbers															
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ANDI to SR	0	0	0	0	0	0	1	0	0	1	1	1	1	1	0	0
	Word Data															
EORI to SR	0	0	0	0	1	0	1	0	0	1	1	1	1	1	0	0
	Word Data															
cpSAVE	1	1	1	1	Cp-Id			1	0	0	Mode			Register		
cpRESTORE	1	1	1	1	Cp-Id			1	0	1	Mode			Register		
MOVE to SR	0	1	0	0	0	1	1	0	1	1	Mode			Register		
MOVE from SR	0	1	0	0	0	0	0	0	1	1	Mode			Register		
MOVE to USP	0	1	0	0	1	1	1	0	0	1	1	0	0	A-Register		
MOVE from USP	0	1	0	0	1	1	1	0	0	1	1	0	1	A-Register		
MOVEC	0	1	0	0	1	1	1	0	0	1	1	1	1	0	1	dr
	A/D	Reg #			Control Register											
MOVES	0	0	0	0	1	1	1	0	Size		Mode			Register		
	A/D	Reg #			dr	0	0	0	0	0	0	0	0	0	0	0
ORI to SR	0	0	0	0	0	0	0	0	0	1	1	1	1	1	0	0
	Word Data															
RESET	0	1	0	0	1	1	1	0	0	1	1	1	0	0	0	0
STOP	0	1	0	0	1	1	1	0	0	1	1	1	0	0	1	0
	Immediate Data															

Table C 4: Privileged And Trap Instruction Formats.

To reduce the effect the VMM's trap-disassemble-evaluate operation has on the performance of the overall system this part of the VMM has to quick and efficient.

Appendix D

Virtual Machine Monitor Code Fragments

This Appendix section has several data structures that are used to define various virtual machine monitor structures.

D.1 VMM Data Structures

D.1.1 CPU Context Structure

```
#define MACH_NUM_GPRS          16

typedef char    *Address;

typedef struct Cpu_State {
    long        gpr[MACH_NUM_GPRS];          /* General purpose registers
                                              * 16 of them D0-D7/A0-A7    */
    long        pc;                          /* Program counter           */
    long        sr;                          /* Status register           */
    long        usp;                         /* User stack pointer        */
    long        msp;                         /* Master stack pointer      */
    long        isp;                         /* Interrupt stack pointer   */
    long        vbr;                         /* Vector base register      */
    long        cacr;                        /* Cache control register    */
    long        caar;                        /* Cache address register    */
    long        ssw : 16;                    /* Special status word       */
    long        sfc : 3;                     /* Source function code
                                              * register 3-bits           */
    long        dfc : 3;                     /* Destination function code
                                              * register 3-bits           */
    long        fill : 10;                   /* filler to a full word.    */
} Cpu_State;


/*
 * The state structure for the MC68881 Floating point co-processor.
 */

typedef struct Fpu_State {
    long                fpRegs[8][3];        /* 8 fp. registers           */
    long                ctrlRegs[3];         /* 3 fpu control registers   */
    unsigned char       version;             /* Fpu version number        */
    unsigned char       state;               /* Fpu state idle/busy       */
    unsigend short      reserved;            /* Reserved word             */
    unsigned long       internal[(184/4)-1]; /* 184 bytes for fpu state   */
}

/*
 * Simulated Sun-3 MMU structure
 */
typedef struc Mmu_State {
    unsigned char       context;             /* Contex Register           */
    long                *segMapPtr;          /* Pointer to Segment Map    */
    long                *pMapPtr             /* Pointer to Page Map       */
} Mmu_State;


typedef struct VMach_Context {
    Cpu_State   cpuState;                    /* Save area for Cpu state   */
    Address     kernStackStart;              /* Address of start of
                                              * kernel stack.             */
    Fpu_State   fpuState;                    /* Save area for Fpu state.  */
    Mmu_State   mmuState;                    /* Mmu data                  */
} VMach_Context;
```

D.1.2 Virtual Machine User Structure

```
typedef struct VMach_ExcStack {
    short               statusReg;           /* Status register           */
    int                 pc;                  /* Program counter           */
    Mach_VOR            vor;                 /* The vector offset
                                              * register                  */
    union {
        Mach_AddrBusErr addrBusErr;          /* Address or bus error info */
    } tail;
} VMach_ExcStack;

/*
 * The user state for a virtual machine;
 * Synonymous to the user process state for regular operating systems.
 */
```

```
typedef struct VMach_UserState {
    Address             userStackPtr;           /* The user stack pointer     */
    int                 trapRegs[MACH_NUM_GPRS];/* General purpose registers.*/
    Mach_ExcStack       *excStackPtr;           /* The exception stack        */
    Fpu_State           trapFpuState;           /* Internal state of the fpu */
} VMach_UserState;
```

D.2 VMM Sample Code Fragments

In this Section, we would like to provide some sample VMM code fragments to highlight VMM operations managing privileged instruction traps. The following "C" procedure, VMach_Emulate_Prive_Inst, provides MIME's VMM with the ability to emulate sensitive instructions that manipulate the CPU or MMU states. The changes to the CPU or MMU states are reflected in the VMM virtual machine context structures that we show in the previous section. Indeed some instructions do cause changes in the real hardware and are evaluated in an encapsulated form. It is necessary to encapsulate the evaluation of some of the special instructions since the VMM has to protect the rest of the virtual machine environment from being corrupted by the manipulations being made by the current faulting VM. In table C.4 we list all privilege instruction formats. Using an instruction evaluation routine the VMM can determine the type of special operation that was requested by a faulting VM and emulate the operation on the VM's VMCB. The determination of a special instruction is a simple exercise, but is a relatively expensive operation. MIME does not currently support statistics functions to monitor system performance.

```
/*
 * Macros to Make long word mask
 */
#define one(x)          ((x) << 16)
#define two(x, y)       (((x) << 16) + y)

/*
 * Instruction op codes and masks.
 */

/*
 * And immidiate to SR
 */
```

```
#define ANDIW                           one(0001174)
#define ANDIW_MASK                      two(0, 0177777)

/*
 * Exclusive-or immidiate to SP
 */
#define EORIW                           one(0005174)
#define EORIW_MASK                      two(0, 0177777)

/*
 * Co-processor restore function
 */
#define CPRESTORE                       one(0170400)
#define CPRESTORE_MASK                  one(0007077)

/*
 * Co-processor save function
 */
#define CPSAVE                          one(0170500)
#define CPSAVE_MASK                     one(0007077)

/*
 * Move to/from SR
 */
#define MOVEW_TO_SR                     one(0043300)
#define MOVEW_TO_SR_MASK                one(0177700)
#define MOVEW_FRM_SR                    one(0040300)
#define MOVEW_FRM_SR_MASK               one(0177700)

/*
 * Move USP
 */
#define MOVEL_TO_USP                    one(0047140)
#define MOVEL_TO_USP_MASK               one(0177770)
#define MOVEL_FRM_USP                   one(0047150)
#define MOVEL_FRM_USP_MASK              one(0177770)

/*
 * Move control register
 */
#define MOVEC_TO                        one(0047172)
#define MOVEC_TO_MASK                   one(0177777)
#define MOVEC_FRM                       one(0047173)
#define MOVEC_FRM_MASK                  one(0177777)
#define MOVEC_AD_MASK                   two(0, 0100000)
#define MOVEC_REG_MASK                  two(0, 0070000)
#define MOVEC_CREG_MASK                 two(0, 0007777)

/*
 * Moves from/to memory spaces
 */
#define MOVESB_TO                       two(0007000, 0)
#define MOVESB_TO_MASK                  two(0177700, 07777)
```

```
#define MOVESB_FRM                      two(0007000, 04000)
#define MOVESB_FRM_MASK                 two(0177700, 07777)
#define MOVESL_TO                       two(0007200, 0)
#define MOVESL_TO_MASK                  two(0177700, 07777)
#define MOVESL_FRM                      two(0007200, 04000)
#define MOVESL_TO_MASK                  two(0177700, 07777)
#define MOVESW_TO                       two(0007100, 0)
#define MOVESW_TO_MASK                  two(0177700, 07777)
#define MOVESW_FRM                      two(0007100, 04000)
#define MOVESW_FRM_MASK                 two(0177700, 07777)
#define MOVES_REG_MASK                  two(0, 0070000)
#define MOVES_DREG_MASK                 two(0000007, 0)
#define MOVES_AD_MASK                   two(0, 0100000)
#define MOVES_SZ_MASK                   two(0, 0000300)
#define MOVES_DMD_MASK                  two(0000070, 0)

/*
 * Or immediate to SR
 */
#define ORIW                            one(0000174)
#define ORIW_MASK                       two(0, 0177777)

/*
 * Reset microprocessor
 */
#define RESET                           one(0047160)
#define RESET_MASK                      one(0)

/*
 * Halt microprocessor
 */
#define STOP                            one(0047162)
#define STOP_MASK                       two(0, 0177777)


   -----------------------------------------------------------------------------

    Mach_Emulate_Priv_Inst

        Emulate the privileged instruction found at the address of the PC in
        in exception stack frame.
 *      Some instructions are simulated by writing to software copies of the
 *      data structures, e.g. SP, VBR etc. While others are emulated by
 *      interpreting the instructions and executing them in an encapsulated
 *      form. The caller can then modify the user's pc value based on how
 *      the instruction was emulated/executed, and the size of the instr.
 *
 * Results:
 *      True if instruction was emulated by encapsulation and execution.
 *      False if simulated in software structures.
 *
 * Side effects:
 *      Privileged memory may be written.
 *      Returns in instrSize the size (in bytes) of the instruction evaluated.
 *
 * ----------------------------------------------------------------------------
 */
Boolean
VMach_Emulate_Priv_Inst (machUserContex, machUserState, instrSize)
    VMach_UserState     machUserState;
    VMach_Contex        machUserContext;
    short               *instrSize;
{
    Boolean     emulated = FALSE;
    long        instr;
    long        value;
    long        source, dest;
    short       size, mode, ea;

    *instrSize = 4;
    instr = machUserState->excStackPtr->pc;

    switch (instr) {
      case ANDIW:
        value = ANDIW_MASK & instr;
        machUserContex.cpuState.sr &= value;
        emulated = TRUE;
        break;
      case EORIW:
        value = EORIW_MASK & instr;
        machUserContex.cpuState.sr ^= value;
        emulated = TRUE;
        break;
      case CPRESTORE: case CPSAVE:
        VMach_encap_excecute (instr)
        *instrSize = 2;
        break;
      case MOVEW_TO_SR:
        ea = MOVEW_TO_SR_MASK & instr;
        VMach_Copy_From_EffectiveAddr (&machUserContex.cpuState.sr, ea, 2);
        emulated = TRUE;
        *instrSize = 2;
        break;
      case MOVEW_FRM_SR:
        ea = MOVEW_FRM_SR_MASK & instr;
        VMach_Copy_To_EffectiveAddr (machUserContex.cpuState.sr, ea, 2);
        emulated = TRUE;
        *instrSize = 2;
        break;
      case MOVEL_TO_USP:
        source = 8 +(MOVELTO_USP_MASK & instr);
        machUserContex.cpuState.usp = machUserState.trapRegs[8 + source];
        emulated = TRUE;
        *instrSize = 2;
        break;
      case MOVEL_FRM_USP:
        source = MOVELTO_USP_MASK & instr;
        emulated = TRUE;
        *instrSize = 2;
        machUserContex.cpuState.usp = machUserState.trapRegs[8 + source];
break;
      case MOVEC_TO:
        source = (MOVEC_REG_MASK & instr) + ((MOVEC_AD_MASK & instr) * 8);
        switch (MOVEC_CREG & instr) {
          case 0x000:
            machUserState.trapRegs[source] = machUserContext.cpuState.sfc;
            break;
          case 0x001:
            machUserState.trapRegs[source] = machUserContext.cpuState.dfc;
            break;
          case 0x002:
            machUserState.trapRegs[source] = machUserContext.cpuState.cacr;
            break;
          case 0x800:
            machUserState.trapRegs[source] = machUserContext.cpuState.usp;
            break;
          case 0x801:
            machUserState.trapRegs[source] = machUserContext.cpuState.vbr;
            break;
          case 0x802:
            machUserState.trapRegs[source] = machUserContext.cpuState.caar;
            break;
          case 0x803:
            machUserState.trapRegs[source] = machUserContext.cpuState.msp;
            break;
          case 0x804:
            machUserState.trapRegs[source] = machUserContext.cpuState.isp;
            break;
        }
        emulated = TRUE;
        break;
      case MOVEC_FRM:
        source = (MOVEC_REG_MASK & instr) + ((MOVEC_AD_MASK & instr) * 8);
        switch (MOVEC_CREG & instr) {
          case 0x000:
            machUserContext.cpuState.sfc = machUserState.trapRegs[source];
            break;
          case 0x001:
            machUserContext.cpuState.dfc = machUserState.trapRegs[source];
            break;
          case 0x002:
            machUserContext.cpuState.cacr = machUserState.trapRegs[source];
            break;
          case 0x800:
            machUserContext.cpuState.usp = machUserState.trapRegs[source];
            break;
          case 0x801:
            machUserContext.cpuState.vbr = machUserState.trapRegs[source];
            break;
          case 0x802:
            machUserContext.cpuState.caar = machUserState.trapRegs[source];
            break;
          case 0x803:
            machUserContext.cpuState.msp = machUserState.trapRegs[source];
            break;
          case 0x804:
            machUserContext.cpuState.isp = machUserState.trapRegs[source];
            break;
        }
        emulated = TRUE;
        break;
      case MOVESB_TO: case MOVESW_TO: case MOVESL_TO:
        source = (MOVES_REG_MASK & instr) + ((MOVES_AD_MASK & instr) * 8);
        dest = 8 + (MOVES_DREG_MASK & instr);
        size = (MOVES_SZ_MASK & inst);
        mode = (MOVES_DMD_MASK & inst);
        VMach_MmuSpaceRead (machUserState.trapRegs[source],
                            machUserState.trapRegs[dest], mode, size);
        break;
      case MOVESB_FRM: case MOVESW_FRM: case MOVESL_FRM:
        source = (MOVES_REG_MASK & instr) + ((MOVES_AD_MASK & instr) * 8);
        dest = 8 + (MOVES_DREG_MASK & instr);
        size = (MOVES_SZ_MASK & inst);
        mode = (MOVES_DMD_MASK & inst);
        VMach_MmuSpaceWrite (machUserState.trapRegs[source],
                             machUserState.trapRegs[dest], mode);
        break;
      case ORIW:
        value = ORIW_MASK & instr;
        machUserContex.cpuState.sr |= value;
        emulated = TRUE;
        break;
      case RESET:
        VMach_KillMachine (machUserState, machUserContext, RESET_INSTR);
        *instrSize = 2;
        break;
      case STOP:
        value = STOP_MASK & instr;
        machUserContex.cpuState.sr = value;
        VMach_KillMachine (machUserState, machUserContext, STOP_INSTR);
        emulated = TRUE;
        break;
    }
    return emulated;
}
```