Kernel Initialization in an Operating System: Design and Implementation

A Dissertation

Submitted in partial fulfillment of the requirement for the award of the degree of

MASTER OF ENGINEERING
(COMPUTER TECHNOLOGY & APPLICATIONS)

By

AMIT KHANCHI
College Roll No. 07/CTA/03
Delhi University Roll No. 3007

Under the guidance of
Dr. Goldie Gabrani

Department Of Computer Engineering
Delhi College Of Engineering, New Delhi-110042
(University of Delhi)

July-2005

ACKNOWLEDGEMENT

It is a great pleasure to have the opportunity to extent my heartiest felt gratitude to everybody who helped me throughout the course of this project.

I would like to express my heartiest felt regards to **Dr. Goldie Gabrani**, Assistant Professor, Department of Computer Engineering for the constant motivation and support during the duration of this project. It is my privilege and owner to have worked under the supervision. Her invaluable guidance and helpful discussions in every stage of this thesis really helped me in materializing this project. It is indeed difficult to put her contribution in few words.

I would also like to take this opportunity to present my sincere regards to my teachers, viz. Professor D. Roy Choudhury, Dr S. K. Saxena, Mr. Rajeev Kumar and Mrs. Rajni Jindal for their support and encouragement.

I am thankful to my friends and classmates for their unconditional support and motivation during this project.

Amit Khanchi
M.E. (Computer Technology & Applications)
College Roll No. 07/CTA/03
Delhi University Roll No. 3007

CERTIFICATE

This is to certify that the Dissertation entitled "**Kernel Initialization in an Operating System: Design and Implementation**", submitted by **Amit Khanchi** in the partial fulfillment of the requirement for the award of degree of **Master of Engineering** in **Computer Technology and Applications** from **Computer Engineering Department** of **Delhi College of Engineering**, **Delhi,** is an account of his work carried out under my guidance and supervision.

Professor D. Roy. Choudhary
Head of Department,
Department of Computer Engineering,
Delhi College of Engineering,
Delhi – 110042.

Dr. Goldie Gabrani
Assistant Professor,
Department of Computer Engineering,
Delhi College of Engineering,
Delhi – 110042.

ABSTRACT

In this dissertation, design and implementation of kernel initialization phase to support virtual addressing with non-paging in an operating system has been proposed and implemented. This design is for Minix operating system, as it does not support virtual addressing. The underlying architecture is Intel 80x86, from 80386 onwards.

To support virtual addressing in an operating system, paging unit of the underlying architecture must be enabled. Before enabling the paging unit, the initialization of page directory, page tables and memory handling data structures is required during booting of the system. A suitable mechanism is needed to perform this initialization phase. This dissertation is focusing on the design and implementation of this mechanism. The design does not support the paging i.e. swapping of pages and hence it is for virtual addressing with non-paging. Further, a method is proposed and implemented to increase the physical memory supported in Minix from 16 MB to the full available RAM.

Contents

CHAPTER 1 PROBLEM DEFINITION AND INTRODUCTION

1.1 Literature survey

Before having problem introduction, first we summarize a few basic terms and ideas related to operating systems in this section.

1.1.1 Operating System

An operating system is software that is used to provide a *convenient* and *efficient* environment to the user by acting as an intermediary between a user of the computer and the computer hardware [9]. It provides the *convenient* environment by hiding all the complex details of the working of a computer system at hardware level from user. To provide *efficiency*, it allocates different resources of a computer system such as processors, memories, and I/O devices in an ordered and controlled fashion among the various programs competing for them.

At the center of an operating system is the *kernel*, which provides the most basic computing functions [24]. These functions include process scheduling, memory management, inter-process communication, interrupt handling etc. Besides the kernel, an operating system provides other basic services needed to operate the computer, including:

File systems - structure in which information is stored on the computer.
Device drivers - provide the interfaces to each of the hardware devices.
User interfaces - a way for users to run programs and access the file system.
System services - include processes that mount file systems, start network, etc.

1.1.2 Types of operating systems

On the basis of the modes of operation, operating systems can be divided into three categories:

1. Monolithic operating system
2. Microkernel based operating system
3. Modular kernel based operating system

Brief description of each of these operating systems is given below.

Monolithic operating system is written as a collection of procedures, each of which can call any of the other ones whenever it needs to [4]. Each procedure in the system has a well-defined interface in terms of parameters and results, and each one is free to call any other one, if the latter provides some useful computation that the former needs. It uses two modes of operation: Kernel Mode and User Mode. Virtually all operating system code is executed in Kernel Mode that results into a huge monolithic program. The drawbacks of this type of operating system are that first, it is difficult to replace or adapt operating system components without Shutdown and even recompilation & reinstallation and second, it is not ideal from the point of view of openness, Software Engineering, Reliability or Maintainability.

Microkernel based operating system are primarily organized into two parts. The first part consists of a collection of modules for managing the hardware but can equally well be executed in user mode. Second part of the operating system consists of a small Microkernel containing only the code that must execute in the kernel mode [4]. Microkernel also contains the code to pass the System Calls to calls on the appropriate user level operating system modules and to return their result. The main advantages of this type of operating system are that it is easy to replace a module without recompilation or reinstallation of entire system and this approach lends itself well to extending a uniprocessor operating system to distributed computers. But the drawback is that Microkernel has extra communication and thus a slight performance loss.

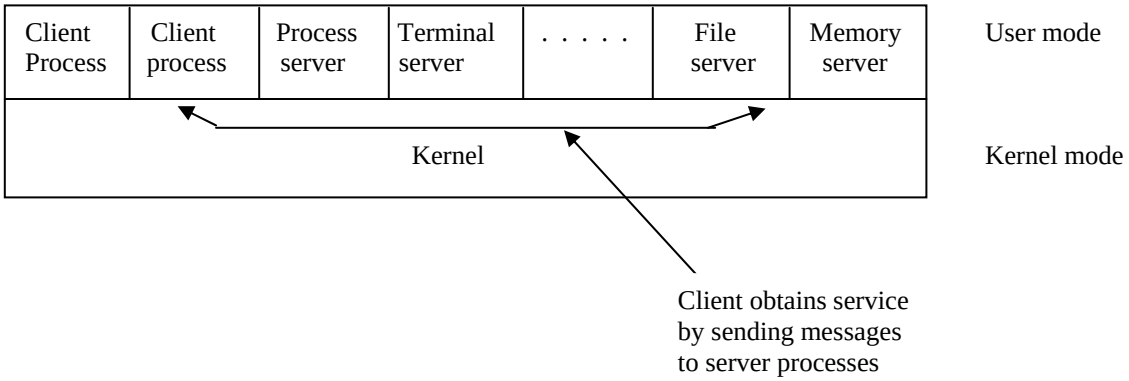


Figure 1.1: Microkernel based approach

In this model, shown in Figure 1.1, all the kernel does is handling the communication between clients and servers. By splitting the operating system up into parts, each of which only handles one facet of the system, such as file service, process service, terminal service, or memory service, each part becomes small and manageable. Furthermore, because all the servers run as user-mode processes, and not in kernel mode, they do not have direct access to the hardware. As a consequence, if a bug in the file server is triggered, the file service may crash, but this will not usually bring the whole machine down.

Another advantage of the client-server model is its adaptability to use in distributed systems (see Figure 1.2). If a client communicates with a server by sending it messages, the client need not know whether the message is handled locally in its own machine, or whether it was sent across a network to a server on a remote machine. As far as the client is concerned, the same thing happens in both cases: a request was sent and a reply came back.

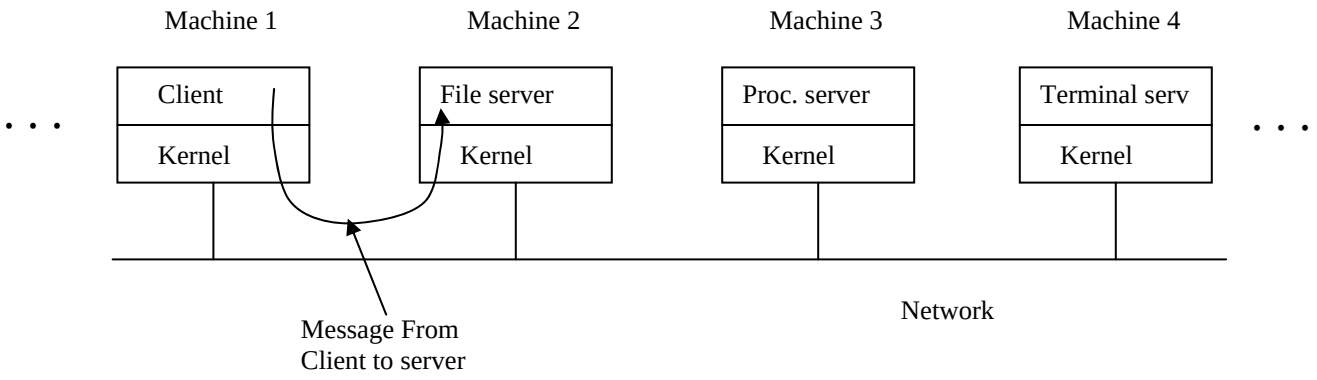


Figure 1.2: Microkernel based approach extended to distributed systems

Modular kernel based operating system uses kernel modules that can be loaded and unloaded into the kernel upon demand. They extend the functionality of the kernel without the need to reboot the system [2]. For example, one type of module is the device driver, which allows the kernel to access hardware connected to the system. Without modules, we would have to build monolithic kernels and add new functionality directly into the kernel image. Besides having larger kernels, this has the disadvantage of requiring us to rebuild and reboot the kernel every time we want new functionality. Modular based operating system are different from the Microkernel based operating in the sense that in former still the major part is executed in kernel mode. This model is used in implementing Linux kernel.

1.1.3 Booting: Overview

The boot process details are architecture-specific, so we shall focus our attention on the IBM PC/AT32 architecture. Due to old design and backward compatibility, the PC firmware boots the operating system in an old-fashioned manner. This process [7] can be separated into the following five logical stages:

1. BIOS selects the boot device.

2. BIOS loads the bootsector, also known as bootblock, from the boot device which contains bootstrap program.
3. Bootstrap program loads a larger program, which then loads the operating system itself.
4. Low-level initialization is performed by assembly code.
5. High-level C initialization.

1.1.4 Booting: BIOS POST

1. The power supply starts the clock generator and asserts #POWERGOOD signal on the bus.
2. CPU #RESET line is asserted (CPU now in real 8086 mode).
3. Set registers *ds, es, fs, gs, ss* equal to zero, *cs*=0xFFFF0000, *eip*=0x0000FFF0 (ROM BIOS POST code).
4. All POST checks are performed with interrupts disabled.
5. IVT (Interrupt Vector Table) initialized at address 0.
6. The BIOS Bootstrap Loader function is invoked via int 0x19, with register *dl* containing the boot device 'drive number'. This loads track 0, sector 1 at physical address 0x7C00 (0x07C0:0000).

1.1.5 Booting: bootsector

When the computer is turned on, the hardware reads the first sector of the first track of the boot disk into memory and executes the code it finds there. The details vary depending upon whether the boot disk is a diskette or a hard disk. On a diskette this sector contains the bootstrap program. It is very small, since it has to fit in one sector. The bootstrap loads a larger program, which then loads the operating system itself.

In contrast, hard disks require an intermediate step. A hard disk is divided into partitions, and the first sector of a hard disk contains a small program and the disk's partition table.

Collectively these are called the *master boot record*. The program part is executed to read the partition table and to select the active partition. The active partition has a bootstrap on its first sector, which is then loaded and executed to find and start a copy of a larger program in the partition, exactly as is done when booting from a diskette.

In either case, this program loads the complete operating system in memory. The parts include the kernel, the memory manager, the file system, and init, the first user process. This startup process is not a trivial operation. Operations that are in the realms of the disk task and the file system must be performed by boot before these parts of the system are active. Once the loading operation is complete the kernel starts running.

1.1.6 Kernel Initialization in Unix based systems for Intel 80x86

When kernel starts running it goes through an initialization phase which setup the proper environment for its smooth and efficient functioning. This phase is architecture dependent, as different hardware resources are initialized here. We will concentrate on Intel 80x86 based architecture in this dissertation. During kernel initialization phase, the kernel initializes the global descriptor table, locate boot parameters and setup kernel segment registers and kernel stack by using assembly routines. It may initialize global page directory and page tables, if virtual addressing is to be used. Kernel also initializes protected mode descriptors for various interrupts, tasks, servers and user processes. It initializes interrupt controller, detects available physical memory and initialize various memory handling data structures.

Kernel also initializes task table for various tasks and starts them. When all these tasks have run and initialized themselves, the first user process generally named *init* will be executed [3]. Then init forks off a child process for each login terminal which then prints a login message on that terminal. Then login name is verified using a program usually named *login.* After a successful login, the *login* program executes the user's shell. The shell waits for commands to be typed and then forks of a new process for each command.

1.2 Problem Description

For enabling paging circuitry in an operating system, the first, and the most obvious step is selection of a suitable hardware platform and operating system to achieve our target. Owing to time consideration, and differences in structure of various type of operating systems as explained in previous sections, a generalized code which could work for all the different architecture is beyond comprehension.

Also owing to various licensing restriction and factors related to cost effectiveness, it is necessary to select a generalized hardware platform with an open source operating system for this project.

Intel based 80x86 is the most widely used architecture in used in the computing world, so as a stepping stone selecting this as a target achitecture for use in this project will be suitable. The details of the this architecture is also widely published and hence could easily be referenced for the purpouse of this project.

There are numerous open source operating system available, which could be modified freely without much restrictions. It includes Linux which is widely available and is considered an industry standard today. But, as it already have a paging architeture, it is not prudent to replace this architecture with the paging architecture of our own. It will be more like reinventing the wheel. A similar logic applies for most other open source operating system.

Also, present day Linux is a modular kernel based operating system designed with help and contributions of hundreds of volunteers over a time of more then a decade. To make any change to its kernel would need to recompile the whole operating system each time -which is a much massive process in comparison to working with a micro kernel based operating system. Also, as Linux is a widely used operating system, it contains a lot of modules which are not exactly needed for a project designed for research or study purpouses. This all contributes to a massive code size, managing and modifying which is a much more time taking and error prone process with no real advantage, if aims of this project is considered.

Similar problem, stands for most other monolithic kernel based or modular kernel based operating system.

Inspite of these limitations Linux, although can't be considered as a target operating system, is still a very good study for the purpose of implementation of this project. During the course of this project, the paging circuitry of Linux provides very valuable insight in design and implementation of virtual addressing in a Unix based environment.

Considering these factors, it would be ideal to choose Minix, an open source operating system by Andrew S. Tanenbaum, designed to teach students about the fundamentals of operating systems. Minix is a stripped down version of Unix without any paging architecture of its own. It is widely available and worldover popular between students and academecians.

Also, Minix is a micro kernel based operating system and contains very concise basic code needed for creating a Unix lookalike operating system. In comparison to monolithic kernel based or modular kernel based operaing system, it is much easier to make and track changes to a Minix kernel. Also, as it contains only the basic modules needed for running and using the operating system, hence it is much easier to manage or distribute a Minix based code.

For the purpose of maintaining the simplicity of code, Minix doesn't use the paging unit of 80x86 based Intel architecture. The aims of this project includes taking benefit of this paging unit. The word 'Paging' is nowadays used interchangably with vitrual memory management with swapping of pages, which is the most widely used paging method in most major operating systems like Linux and Microsoft based Windows platform. This project aims to enable virtual addressing by dividing physical memory in *page frames* (fixed sized blocks of physical memory) and using contiguous linear address space mapped on non-contiguous page frames for user processes without any swapping of pages. This in itself is not paging, as paging refers to swapping of pages between physical memory and the swap area done by a program called pager. This is actually virtual memory with non-paging.

Owing to the examplary complexity connected with implementation of paging in a Minix like architecture this project targets creating the basic structure needed to implement paging and kernel initialization for this purpose, i.e., *kernel initialization for virtual addressing with non-paging* .

To enable a paging circuitry in Minix for virtual addressing would not only be a step forward, but would also be an ideal tribute to an operating system desiged for sole purpouse of helping students understand the nuianses of an operating systems.

1.3 Dissertation organization

The organization of the rest of this dissertation is as follows.

Chapter 2: Provides Intel 80x86 architecture details of its memory management unit related to segmentation, paging and protection.

Chapter 3: Explains the Minix architecture and its booting in detail. Minix memory layout is also discussed here.

Chapter 4: Provides complete system initialization of the Minix. It includes the various routines explained in the order of their use. Then the various data types and data structures are discussed for their role in the system initialization phase.

Chapter 5: Explains various design issues along with the design of system initialization phase to support virtual addressing with non-paging. All the routines involved in the initialization phase are discussed in detail from the implementation point of view. Also, a mechanism is provided to increase the RAM size supported.

Chapter 6: Presents the conclusion over the various aspects of the proposed design and discusses the further enhancements as future work.
References
Source code of selected files

CHAPTER 2 TARGET ARCHITECTURE DESCRIPTION

2.1 Introduction

The Intel 80x86 are advanced 32-bit microprocessor optimized for multitasking operating systems and designed for applications needing very high performance. The 32-bit registers and data paths support 32-bit addresses and data types. The processor can address up to four gigabytes of physical memory and 64 terabytes (2^{46} bytes) of virtual memory. The on-chip memory-management facilities include address translation registers, advanced multitasking hardware, a protection mechanism, and paged virtual memory. Special debugging registers provide data and code breakpoints even in ROM-based software.

2.2 Processing Modes Of 80x86

The 80x86 has three processing modes:

1. Protected Mode.
2. Real-Address Mode.
3. Virtual 8086 Mode.

Protected mode is the natural 32-bit environment of the 80x86 processor. In this mode all instructions and features are available.

Real-address mode (often called just "real mode") is the mode of the processor immediately after RESET. In real mode the 80x86 appears to programmers as a fast 8086 with some new instructions. Most applications of the 80x86 will use real mode for initialization only.

Virtual 8086 mode (also called V86 mode) is a dynamic mode in the sense that the processor can switch repeatedly and rapidly between V86 mode and protected mode. The CPU enters V86 mode from protected mode to execute an 8086 program, then leaves V86 mode and enters protected mode to continue executing a native 80x86 program.

2.3 Software Model

Software model [7] of 80x86 is shown in the Figure 2.1. It consists of various registers. These registers are described below.

2.4 Systems Registers

The registers designed for use by systems programmers fall into these classes:

EFLAGS
Memory-Management Registers
Control Registers
Debug Registers
Test Registers

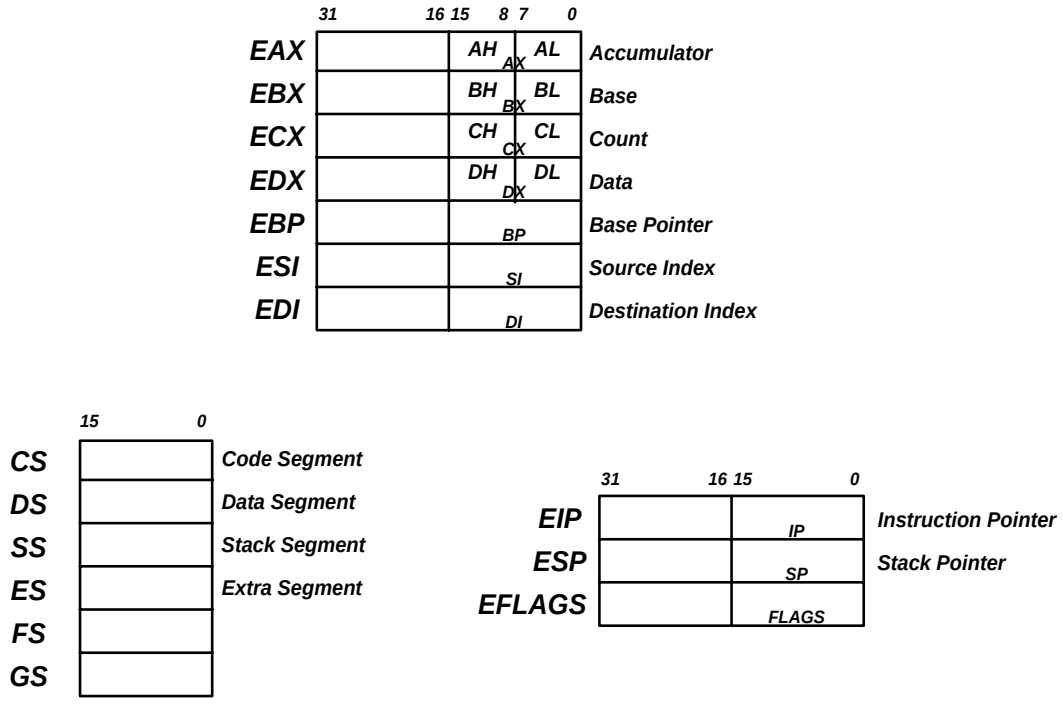


Figure 2.1: Software model of 80x86

2.4.1 Systems Flags

The systems flags of the EFLAGS register control I/O, maskable interrupts, debugging, task switching, and enabling of virtual 8086 execution in a protected, multitasking environment. These flags are highlighted in Figure 2.2.

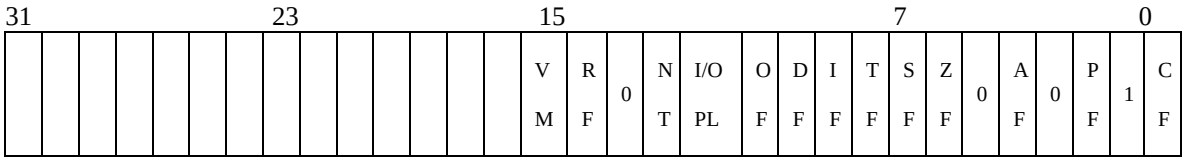


Figure 2.2: system flags

IF (Interrupt-Enable Flag, bit 9)

Setting IF allows the CPU to recognize external (maskable) interrupt requests. Clearing IF disables these interrupts. IF has no effect on either exceptions or nonmaskable external interrupts.

NT (Nested Task, bit 14)

The processor uses the nested task flag to control chaining of interrupted and called tasks. NT influences the operation of the IRET instruction.

RF (Resume Flag, bit 16)

The RF flag temporarily disables debug exceptions so that an instruction can be restarted after a debug exception without immediately causing another debug exception.

TF (Trap Flag, bit 8)

Setting TF puts the processor into single-step mode for debugging. In this mode, the CPU automatically generates an exception after each instruction, allowing a program to be inspected as it executes each instruction. Single-stepping is just one of several debugging features of the 80x86.

VM (Virtual 8086 Mode, bit 17)

When set, the VM flag indicates that the task is executing an 8086 program.

2.4.2 Memory-Management Registers

Four registers of the 80x86 locate the data structures that control segmented memory management:

GDTR Global Descriptor Table Register

LDTR Local Descriptor Table Register

These registers point to the segment descriptor tables GDT and LDT.

IDTR Interrupt Descriptor Table Register

This register points to a table of entry points for interrupt handlers (the IDT).

TR Task Register

This register points to the information needed by the processor to define the current task.

2.4.3 Control Registers:

Figure 2.3 shows the format of the 80x86 control registers CR0, CR2, and CR3.

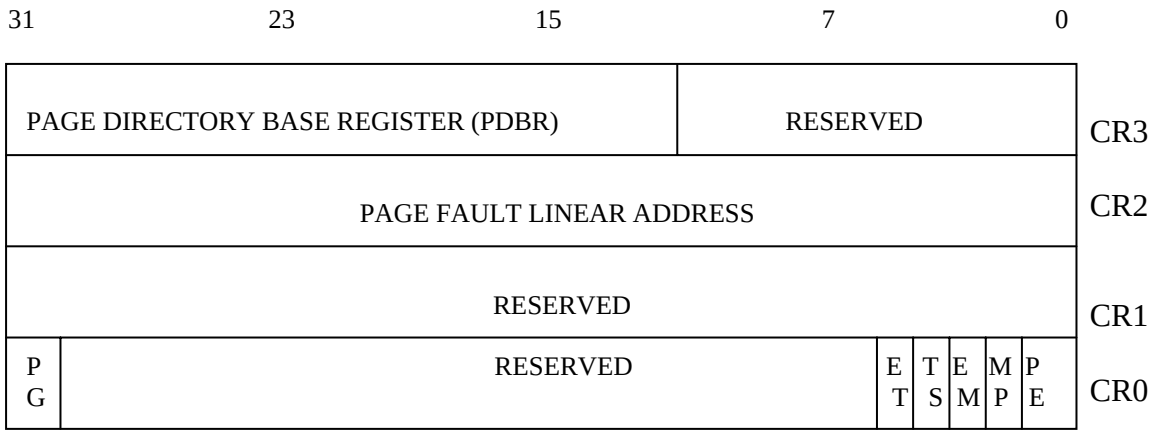


Figure 2.3: control registers

CR0 contains system control flags, which control or indicate conditions that apply to the system as a whole, not to an individual task.

EM (Emulation, bit 2) : EM indicates whether coprocessor functions are to be emulated.
ET (Extension Type, bit 4):ET indicates the type of coprocessor present in the system (80287 or 80387).
MP (Math Present, bit 1) : MP controls the function of the WAIT instruction, which is used to coordinate a coprocessor.
PE (Protection Enable, bit 0): Setting PE causes the processor to begin executing in protected mode. Resetting PE returns to real-address mode.
PG (Paging, bit 31) : PG indicates whether the processor uses page tables to translate linear addresses into physical addresses.
TS (Task Switched, bit 3) :The processor sets TS with every task switch and tests TS when interpreting coprocessor instructions.

CR2 is used for handling page faults when PG is set. The processor stores in CR2 the linear address that triggers the fault. CR3 is used when PG is set. CR3 enables the processor to locate the page table directory for the current task.

2.4.4 Debug Register:

The debug registers bring advanced debugging abilities to the 80x86, including data breakpoints and the ability to set instruction breakpoints without modifying code segments.

2.4.5 Test Registers

The test registers are not a standard part of the 80x86 architecture. They are provided solely to enable confidence testing of the translation look-aside buffer (TLB), the cache used for storing information from page tables.

2.5 Memory Management

The 80x86 transforms logical addresses (i.e., addresses as viewed by programmers) into physical address (i.e., actual addresses in physical memory) in two steps:

- Segment translation, in which a logical address (consisting of a segment selector and segment offset) are converted to a linear address.
- Page translation, in which a linear address is converted to a physical address. This step is optional, at the discretion of systems-software designers.

These translations are performed in a way that is not visible to applications programmers. Figure 2.4 illustrates the two translations at a high level of abstraction.

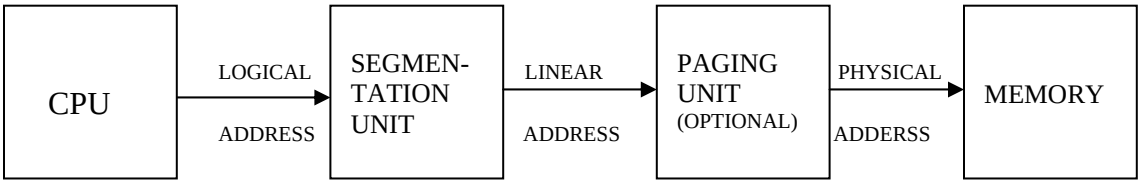


Figure 2.4: Address transformation in 80x86

In reality, the addressing mechanism also includes memory protection features. For the sake of simplicity, however, the subject of protection is taken up in another section.

LOGICAL ADDRESS:

CPU generates a logical address of 48 bits. Format of logical address is shown in Figure 2.5.

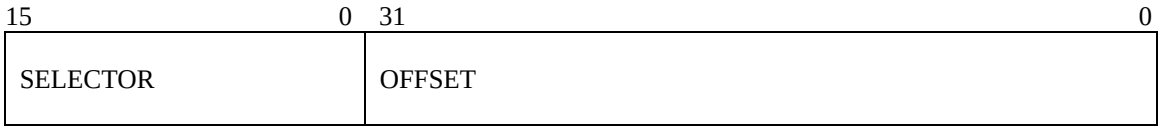


Figure 2.5: Format of logical address in 80x86

2.5.1 Segmentation Unit

Figure 2.6 shows in more detail how the segmentation unit of 80x86 converts a logical address into a linear address.

To perform this translation, the processor uses the following data structures:

- Descriptors
- Descriptor tables
- Selectors
- Segment Registers

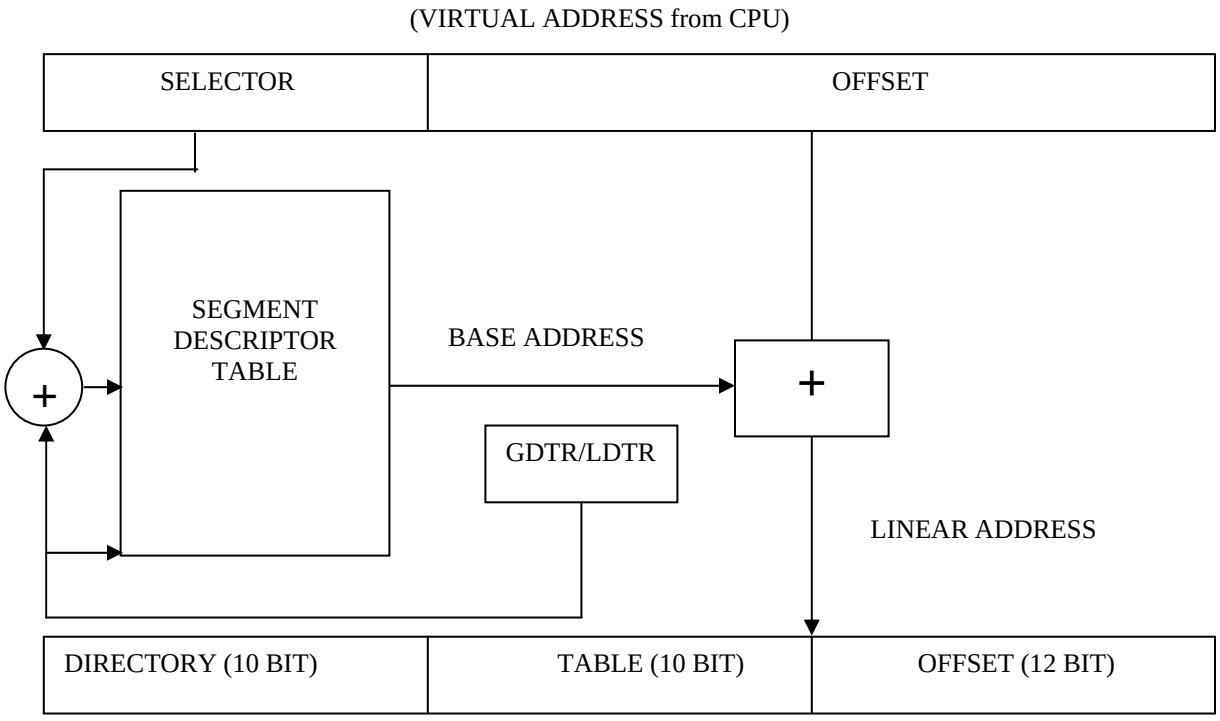


Figure 2.6: Segmentation Unit

Descriptors

The segment descriptor [6] provides the processor with the data it needs to map a logical address into a linear address. Descriptors are created by compilers, linkers, loaders, or the

operating system, not by applications programmers. Figure 2.7 illustrates the two general descriptor formats. All types of segment descriptors take one of these formats.

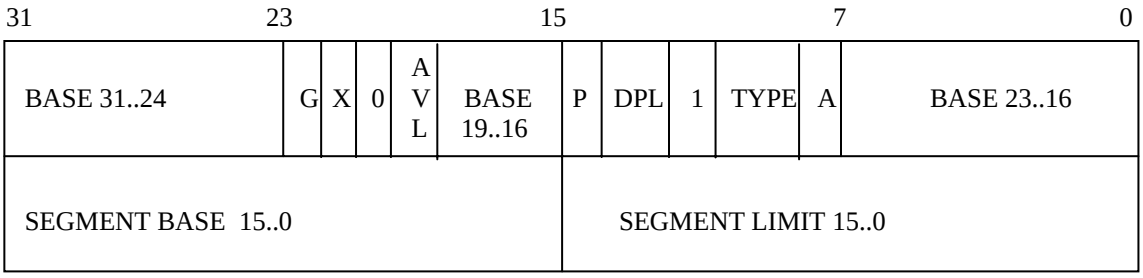
Segment-descriptor fields

BASE: Defines the location of the segment within the 4 gigabyte linear address space. The processor concatenates the three fragments of the base address to form a single 32-bit value.

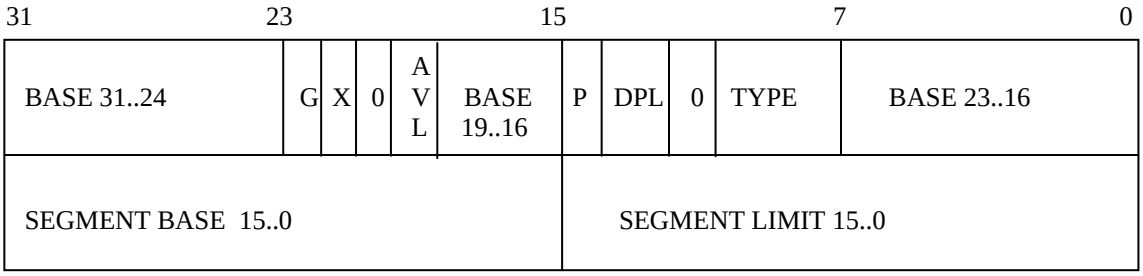
LIMIT: Defines the size of the segment. When the processor concatenates the two parts of the limit field, a 20-bit value results. The processor interprets the limit field in one of two ways, depending on the setting of the granularity bit:

1. In units of one byte, to define a limit of up to 1 megabyte.
2. In units of 4 Kilobytes, to define a limit of up to 4 gigabytes. The limit is shifted left by 12 bits when loaded, and low-order one-bits are inserted.

Granularity bit: Specifies the units with which the LIMIT field is interpreted. When the bit is clear, the limit is interpreted in units of one byte; when set, the limit is interpreted in units of 4 Kilobytes.
TYPE: Distinguishes between various kinds of descriptors.
DPL (Descriptor Privilege Level): Used by the protection mechanism.



Descriptors used for applications code and data segments



Descriptors used for special system segments

A - Accessed
AVL - Available for use by systems programmers
DPL - Descriptor privilege level
G - Granularity
P - Segment present

Figure 2.7: Different descriptor formats

Segment-Present bit: If this bit is zero, the descriptor is not valid for use in address transformation; the processor will signal an exception when a selector for the descriptor is loaded into a segment register.

Accessed bit: The processor sets this bit when the segment is accessed; i.e., a selector for the descriptor is loaded into a segment register or used by a selector test instruction. Operating systems that implement virtual memory at the segment level may, by periodically testing and clearing this bit, monitor frequency of segment usage.

Descriptor Tables

Segment descriptors are stored in either of two kinds of descriptor table:

- The global descriptor table (GDT)
- A local descriptor table (LDT)

A descriptor table is simply a memory array of 8-byte entries that contain descriptors, as shown in Figure 2.8. A descriptor table is variable in length and may contain up to 8192 (2^{13}) descriptors. The first entry of the GDT (INDEX=0) is not used by the processor, however.

The processor locates the GDT and the current LDT in memory by means of the GDTR and LDTR registers. These registers store the base addresses of the tables in the linear address space and store the segment limits. The instructions LGDT and SGDT give access to the GDTR; the instructions LLDT and SLDT give access to the LDTR.

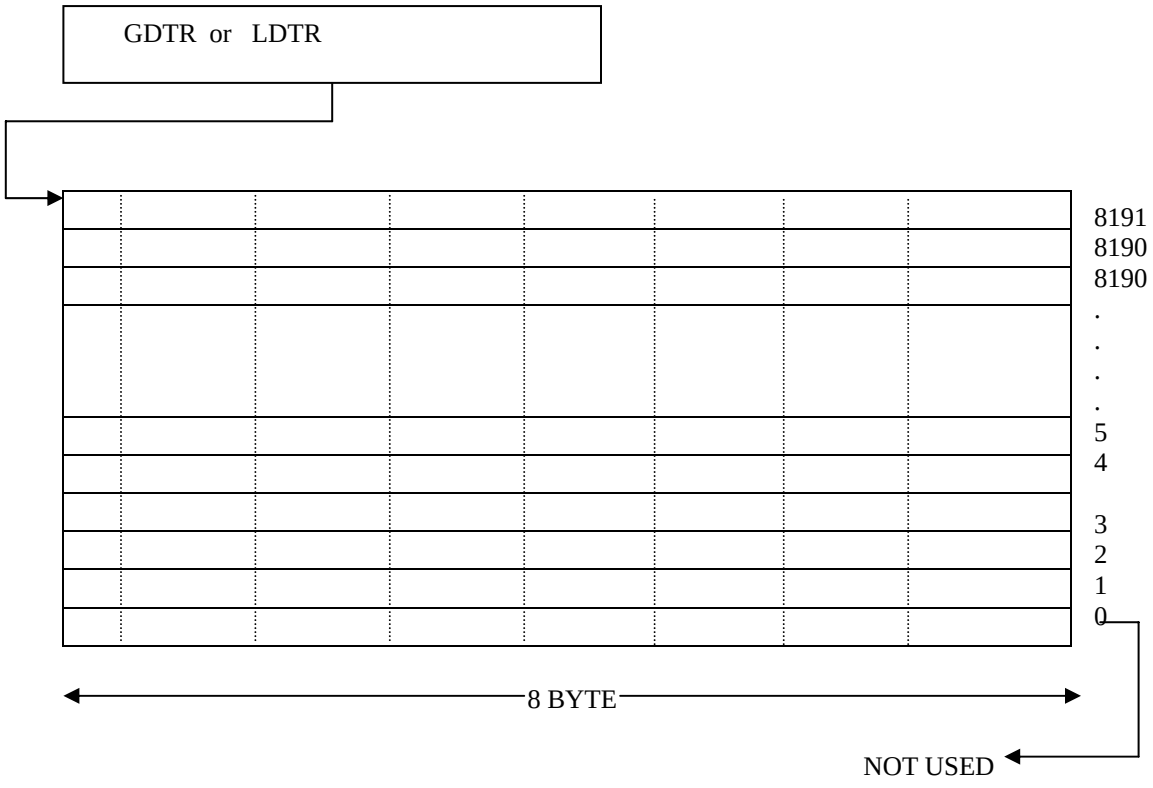
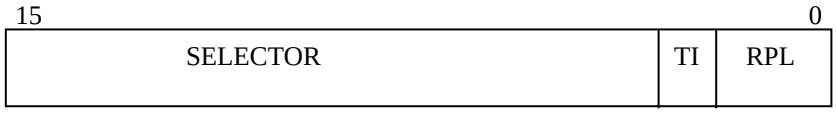


Figure 2.8: Format of logical address in 80x86

Selectors:

The selector portion of a logical address identifies a descriptor by specifying a descriptor table and indexing a descriptor within that table. Selectors may be visible to applications programs as a field within a pointer variable, but the values of selectors are usually assigned (fixed up) by linkers or linking loaders. Figure 2.9 shows the format of a selector.



TI - TABLE INDICATOR 0=Global, 1=Local
RPL - REQUESTOR'S PRIVILEGE LEVEL

Figure 2.9: Selector Format

Index: Selects one of 8192 descriptors in a descriptor table. The processor simply multiplies this index value by 8 (the length of a descriptor), and adds the result to the base address of the descriptor table in order to access the appropriate segment descriptor in the table.

Table Indicator: Specifies to which descriptor table the selector refers. A zero indicates the GDT; a one indicates the current LDT.

Requested Privilege Level: Used by the protection mechanism.

Segment Registers

The 80x86 stores information from descriptors in segment registers, thereby avoiding the need to consult a descriptor table every time it accesses memory. Every segment register has a "visible" portion and an "invisible" portion, as Figure 2.10 illustrates. The visible portions of these segment address registers are manipulated by programs as if they were simply 16-bit registers. The invisible portions are manipulated by the processor.

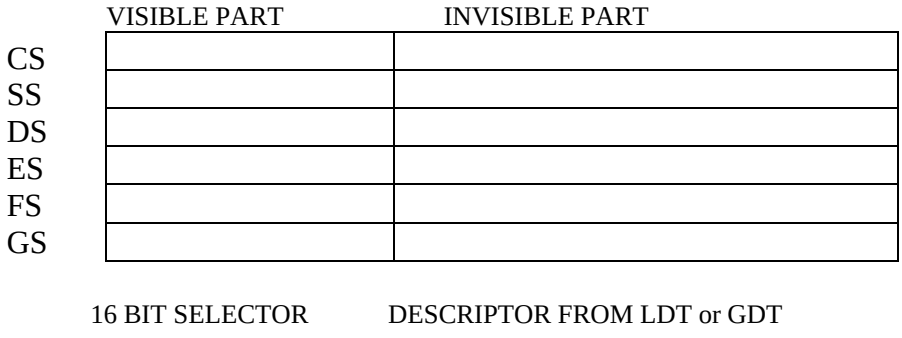


Figure 2.10: Segment Registers

When a program loads the visible part of the segment register with a 16-bit selector. The processor automatically fetches the base address, limit, type, and other information from a descriptor table and loads them into the invisible part of the segment register.

2.5.2 Paging Unit

In the second phase of address transformation, the 80x86 transforms a linear address into a physical address. This phase of address transformation implements the basic features needed for page-oriented virtual-memory systems and page-level protection. The page-translation step is optional. Page translation is in effect only when the PG bit of CR0 is set. This bit is typically set by the operating system during software initialization. The PG bit must be set if the operating system is to implement multiple virtual 8086 tasks, page-oriented protection, or page-oriented virtual memory.

Figure 2.11 shows how the processor converts the DIR, PAGE, and OFFSET fields of a linear address into the physical address by consulting two levels of page tables. The addressing mechanism uses the DIR field as an index into a page directory, uses the PAGE field as an index into the page table determined by the page directory, and uses the OFFSET field to address a byte within the page determined by the page table.

Page Frame

A page frame is a 4K-byte unit of contiguous addresses of physical memory. Pages begin on byte boundaries and are fixed in size.

Page Tables and Directory

A page table is simply an array of 32-bit page specifiers. A page table is itself a page, and therefore contains 4 Kilobytes of memory or at most 1K 32-bit entries.

Two levels of tables are used to address a page of memory. At the higher level is a page

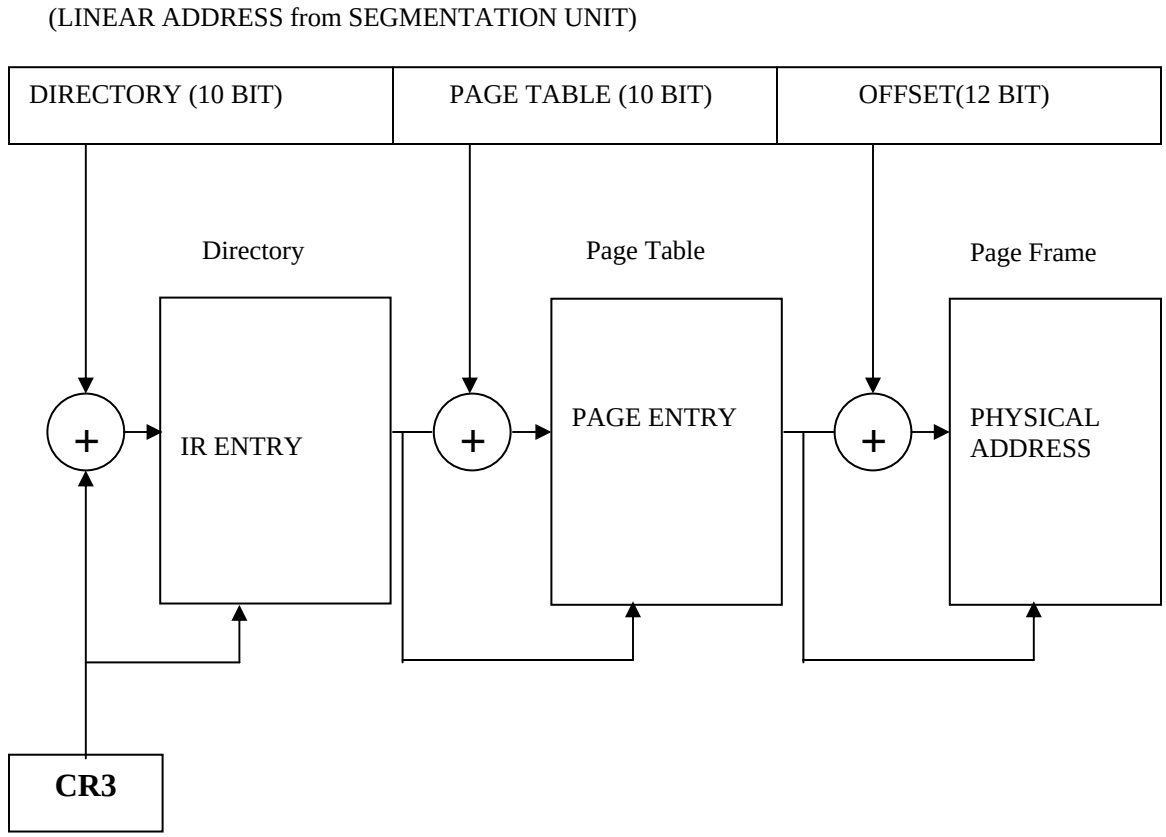


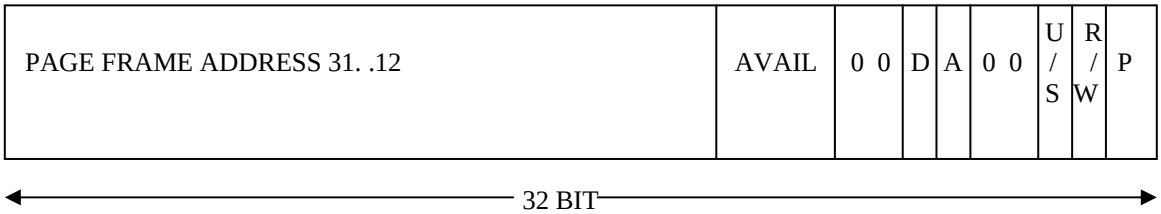
Figure 2.11: Paging Unit

directory. The page directory addresses up to 1K page tables of the second level. A page table of the second level addresses up to 1K pages. All the tables addressed by one page directory, therefore, can address 1M pages (2^{20}). Because each page contains 4K bytes (2^{12} bytes), the tables of one page directory can span the entire physical address space of the 80x86 (2^{20} times $2^{12} = 2^{32}$).

The physical address of the current page directory is stored in the CPU register CR3, also called the page directory base register (PDBR). Memory management software has the option of using one page directory for all tasks, one page directory for each task, or some combination of the two.

Page-Table Entries

Entries in either level of page tables have the same format. Figure 2.12 illustrates this format.



P - PRESENT
R/W - READ/WRITE
U/S - USER/SUPERVISOR
D - DIRTY
AVAIL - AVAILABLE FOR SYSTEMS PROGRAMMER USE

Figure 2.12:Page Table Entry Format

Page Frame Address

The page frame address specifies the physical starting address of a page. Because pages are located on 4K boundaries, the low-order 12 bits are always zero. In a page directory, the page frame address is the address of a page table. In a second-level page table, the page frame address is the address of the page frame that contains the desired memory operand.

Present Bit

The Present bit indicates whether a page table entry can be used in address translation. P=1 indicates that the entry can be used. When P=0 in either level of page tables, the entry is not valid for address translation, and the rest of the entry is available for software use; none of the other bits in the entry is tested by the hardware. If P=0 in either level of page tables when an attempt is made to use a page-table entry for address translation, the processor signals a page exception. In software systems that support paged virtual memory, the page-not-present exception handler can bring the required page into physical memory. The instruction that caused the exception can then be re-executed.

Note that there is no present bit for the page directory itself. The page directory may be not-present while the associated task is suspended, but the operating system must ensure that the page directory indicated by the CR3 image in the TSS is present in physical memory before the task is dispatched.

Accessed and Dirty Bits

These bits provide data about page usage in both levels of the page tables. With the exception of the dirty bit in a page directory entry, these bits are set by the hardware; however, the processor does not clear any of these bits. The processor sets the corresponding accessed bits in both levels of page tables to one before a read or write operation to a page.

The processor sets the dirty bit in the second-level page table to one before a write to an address covered by that page table entry. The dirty bit in directory entries is undefined.
An operating system that supports paged virtual memory can use these bits to determine what pages to eliminate from physical memory when the demand for memory exceeds the physical memory available. The operating system is responsible for testing and clearing these bits.

Read/Write and User/Supervisor Bits

These bits are not used for address translation, but are used for page-level protection, which the processor performs at the same time as address translation.

Page Translation Cache

For greatest efficiency in address translation, the processor stores the most recently used page-table data in an on-chip cache. Only if the necessary paging information is not in the cache must both levels of page tables be referenced. The existence of the page-translation cache is invisible to applications programmers but not to systems programmers; operating-system programmers must flush the cache whenever the page tables are changed. The page-translation cache can be flushed by either of two methods:

1. By reloading CR3 with a MOV instruction; for example:
   MOV CR3, EAX
2. By performing a task switch to a TSS that has a different CR3 image than the current TSS.

2.6 Protection

The purpose of the protection features of the 80x86 is to help detect and identify bugs. The 80x86 supports sophisticated applications that may consist of hundreds or thousands of program modules. In such applications, the question is how bugs can be found and eliminated as quickly as possible and how their damage can be tightly confined. To help debug applications faster and make them more robust in production, the 80x86 contains mechanisms to verify memory accesses and instruction execution for conformance to protection [8] criteria. These mechanisms may be used or ignored, according to system design objectives.

Protection in the 80x86 has five aspects:

1. Type checking
2. Limit checking
3. Restriction of addressable domain
4. Restriction of procedure entry points
5. Restriction of instruction set

The protection hardware of the 80x86 is an integral part of the memory management hardware. Protection applies both to segment translation and to page translation. Each reference to memory is checked by the hardware to verify that it satisfies the protection criteria. All these checks are made before the memory cycle is started; any violation prevents that cycle from starting and results in an exception. Since the checks are performed

concurrently with address formation, there is no performance penalty. Invalid attempts to access memory result in an exception.

2.6.1 Type Checking

The descriptor contains a TYPE field. This TYPE field has two functions:

1. It distinguishes among different descriptor formats.
2. It specifies the intended usage of a segment.

Besides the descriptors for data and executable segments commonly used by applications programs, the 80x86 has descriptors for special segments used by the operating system and for gates. Table 2.1 lists all the types defined for system segments and gates.
Type checking can be used to detect programming errors that would attempt to use segments in ways not intended by the programmer. The processor examines type information on two kinds of occasions:

1. When a selector of a descriptor is loaded into a segment register. Certain segment registers can contain only certain descriptor types; for example:
   - The CS register can be loaded only with a selector of an executable segment.
   - Selectors of executable segments that are not readable cannot be loaded into data-segment registers.
   - Only selectors of writable data segments can be loaded into SS.

2. When an instruction refers (implicitly or explicitly) to a segment register. Certain segments can be used by instructions only in certain predefined ways; for example:
   - No instruction may write into an executable segment.
   - No instruction may write into a data segment if the writable bit is not set.
   - No instruction may read an executable segment unless the readable bit is set.

Table 2.1:System and Gate Descriptor Types

Code	Type of Segment or Gate
0	Reserved
1	Available 286 TSS
2	LDT
3	Busy 286 TSS
4	Call Gate
5	Task Gate
6	286 Interrupt Gate
7	286 Trap Gate
8	reserved
9	Available 386 TSS
A	Reserved
B	Busy 386 TSS
C	386 Call Gate
D	reserved
E	386 Interrupt Gate
F	386 Trap Gate

2.6.2 Restricting access to data

To address operands in memory, an 80x86 program must load the selector of a data segment into a data-segment register (DS, ES, FS, GS, SS). The processor automatically evaluates access to a data segment by comparing privilege levels. The evaluation is performed at the time a selector for the descriptor of the target segment is loaded into the data-segment register. As Figure 2.13 shows, three different privilege levels enter into this type of privilege check:

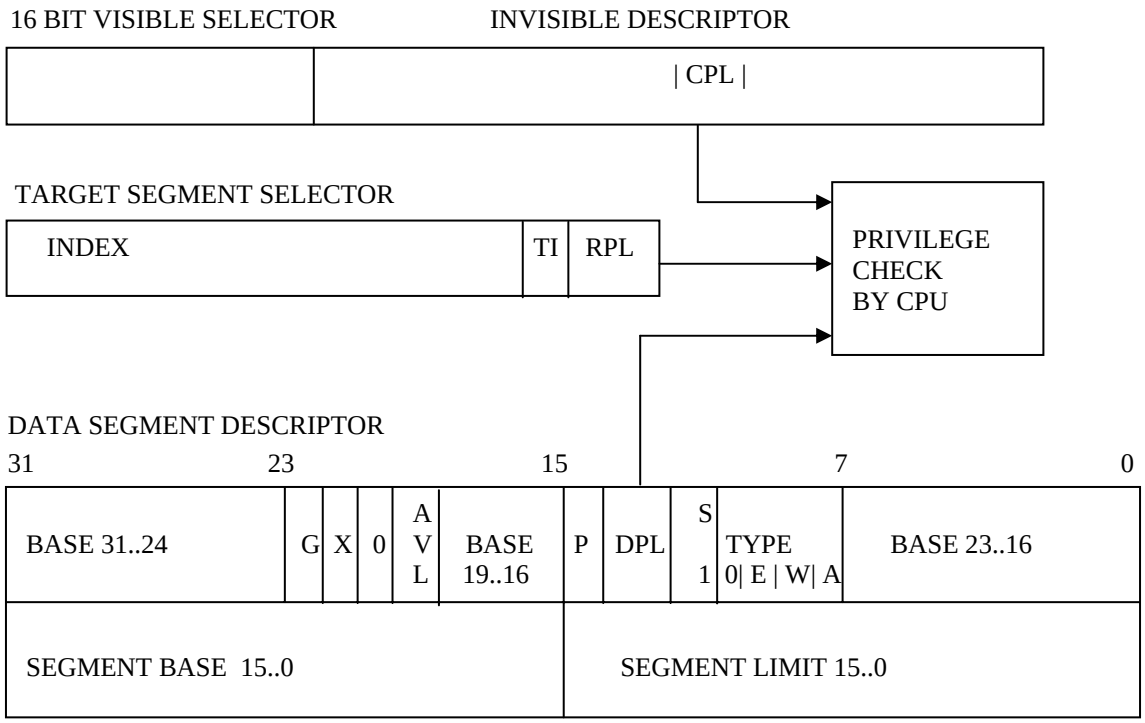
1. The CPL (current privilege level).
2. The RPL (requestor's privilege level) of the selector used to specify the target segment.
3. The DPL of the descriptor of the target segment.

Instructions may load a data-segment register (and subsequently use the target segment) only if the DPL of the target segment is numerically greater than or equal to the maximum of the CPL and the selector's RPL. In other words, a procedure can only access data that is at the same or less privileged level.

The addressable domain of a task varies as CPL changes. When CPL is zero, data segments at all privilege levels are accessible; when CPL is one, only data segments at privilege levels one through three are accessible; when CPL is three, only data segments at privilege level three are accessible. This property of the 80x86 can be used, for example, to prevent applications procedures from reading or changing tables of the operating system.

The type fields of data and executable segment descriptors include bits, which further define the purpose of the segment:

- The writable bit in a data-segment descriptor specifies whether instructions can write into the segment.
- The readable bit in an executable-segment descriptor specifies whether instructions are allowed to read from the segment (for example, to access constants that are stored with instructions).



CPL - Current privilege level
RPL - Requestor's privilege level
DPL - Descriptor privilege level

Figure 2.13: Privilege check for data access

2.6.3 Limit Checking

The limit field of a segment descriptor is used by the processor to prevent programs from addressing outside the segment. The processor's interpretation of the limit depends on the setting of the G (granularity) bit. For data segments, the processor's interpretation of the limit depends also on the E-bit (expansion-direction bit) and the B-bit (big bit). When G=0, the actual limit is the value of the 20-bit limit field as it appears in the descriptor. When G=1, the processor appends 12 low-order one-bits to the value in the limit field.

The limit field of descriptors for descriptor tables is used by the processor to prevent programs from selecting a table entry outside the descriptor table. The limit of a descriptor table identifies the last valid byte of the last descriptor in the table. Since each descriptor is eight bytes long, the limit value is N * 8 - 1 for a table that can contain up to N descriptors.

2.6.4 Privilege Levels

The concept of privilege is implemented by assigning a value from zero to three to key objects recognized by the processor. This value is called the privilege level. The value zero represents the greatest privilege, the value three represents the least privilege. Different privilege levels are shown in the Figure 2.14.

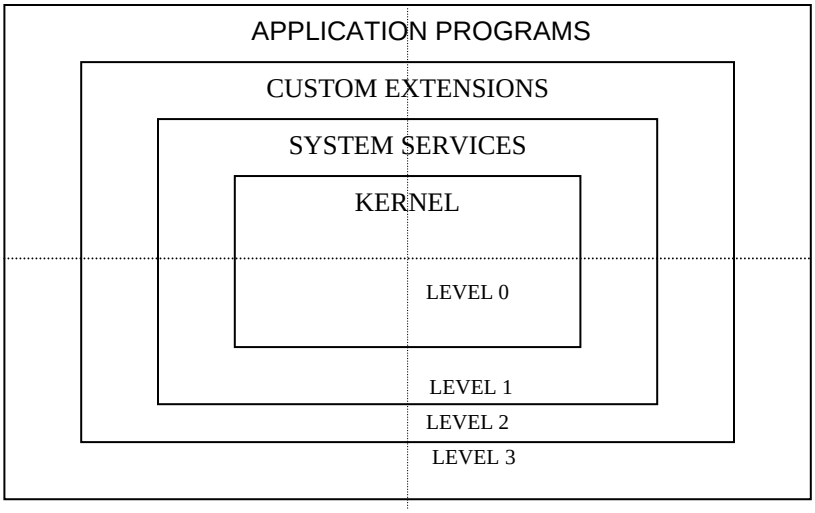


Figure 2.14: Levels Of Privilege

The following processor-recognized objects contain privilege levels:

- Descriptors contain a field called the descriptor privilege level (DPL).
- Selectors contain a field called the requestor's privilege level (RPL). The RPL is intended to represent the privilege level of the procedure that originates a selector.
- An internal processor register records the current privilege level (CPL). Normally the CPL is equal to the DPL of the segment that the processor is currently executing. CPL changes as control is transferred to segments with differing DPLs.

The processor automatically evaluates the right of a procedure to access another segment by comparing the CPL to one or more other privilege levels. The evaluation is performed at the

time the selector of a descriptor is loaded into a segment register. The criteria used for evaluating access to data differs from that for evaluating transfers of control to executable segments; therefore, the two types of access are considered separately in the following sections.

2.6.5 Gate Descriptors Guard Procedure Entry Points

To provide protection for control transfers among executable segments at different privilege levels, the 80x86 uses gate descriptors. There are four kinds of gate descriptors:

- Call gates
- Trap gates
- Interrupt gates
- Task gates

A call gate has two primary functions:

1. To define an entry point of a procedure.
2. To specify the privilege level of the entry point.

Call gate descriptors are used by call and jump instructions in the same manner as code segment descriptors. When the hardware recognizes that the destination selector refers to a gate descriptor, the operation of the instruction is expanded as determined by the contents of the call gate.

The selector and offset fields of a gate form a pointer to the entry point of a procedure. A call gate guarantees that all transitions to another segment go to a valid entry point, rather than possibly into the middle of a procedure (or worse, into the middle of an instruction). The far pointer operand of the control transfer instruction does not point to the segment and offset of the target instruction; rather, the selector part of the pointer selects a gate, and the offset is not used. Gates can be used for control transfers to numerically smaller privilege levels or to the same privilege level (though they are not necessary for transfers to the same level). Only CALL instructions can use gates to transfer to smaller privilege levels. A gate may be used by a JMP instruction only to transfer to an executable segment with the same privilege level or to a conforming segment.

CHAPTER 3 MINIX: ARCHITECTURE AND BOOTING

Minix stands for Mini-Unix. It was developed by Andrew S. Tanenbaum. It is compatible with Unix from the user's point of view, but completely different on the inside. This system avoids the licensing restrictions and hence, could be freely studied, modified and distributed for non-profit usage. It is written in C programming language.

3.1 Minix Architecture

Minix operating system [1] is a micro kernel based operating system. Architecture of the Minix is shown in the Figure 3.1. Unlike Unix, whose kernel is a monolithic program not split up into modules, Minix itself is a collection of processes that communicate with each other and with user processes using a single interprocess communication primitive---message passing. This design gives a more modular and flexible structure [27], making it easy, for example, to

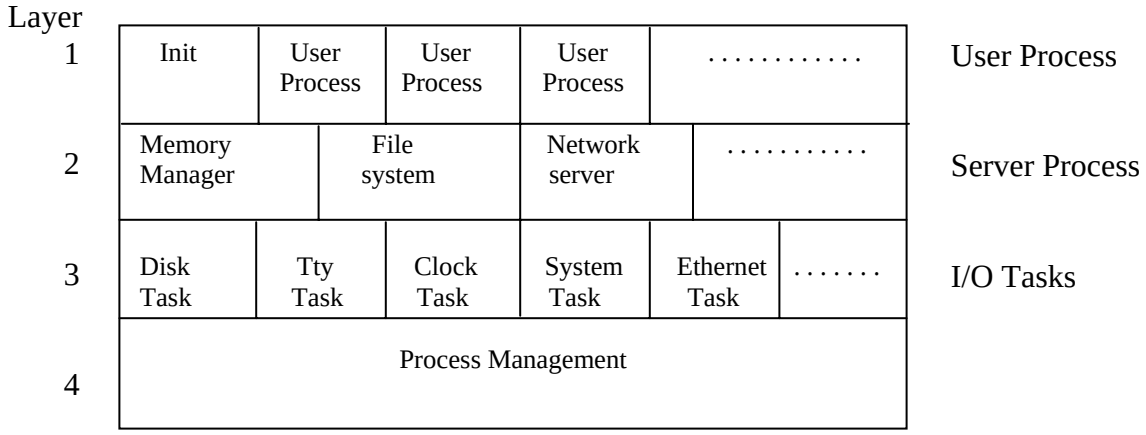


Figure 3.1: Minix Architecture

replace the entire file system by a completely different one, without having even to recompile the kernel.

The bottom layer catches all interrupts and traps, does scheduling, and provides higher layers with a model of independent sequential processes that communicate using messages. The code in this layer has two major functions. The first is catching the traps and interrupts, saving

and restoring registers, scheduling, and the general nuts and bolts of actually making the process abstraction provided to the higher layers work. The second is handling the mechanics of messages; checking for legal destinations, locating send and receive buffers in physical memory, and copying bytes from sender to receiver. That part of the layer dealing with the lowest level of interrupt handling is written in assembly language. The rest of the layer and all of the higher layers are written in C.

Layer 2 contains the I/O processes, one per device type. To distinguish them from ordinary user processes, we will call them *tasks*, but the differences between tasks and processes are minimal. In many systems the I/O tasks are called *device drivers*; we will use the terms "task" and "device driver" interchangeably. A task is needed for each device type, including disks, printers, terminals, network interfaces, and clocks. If other I/O devices are present, a task is needed for each one of those, too. One task, the system task, is a little different, since it does not correspond to any I/O device. All of the tasks in layer 2 and all the code in layer 1 are linked together into a single binary program called the *kernel*. Some of the tasks share common subroutines, but otherwise they are independent from one another, are scheduled independently, and communicate using messages. Intel processors starting with the 286 assign one of four levels of privilege to each process. Although the tasks and the kernel are compiled together, when the kernel and the, interrupt handlers are executing, they are accorded more privileges than the tasks. Thus the true kernel code can access any part of memory and any processor register--essentially, the kernel can execute any instruction using data from anywhere in the system.

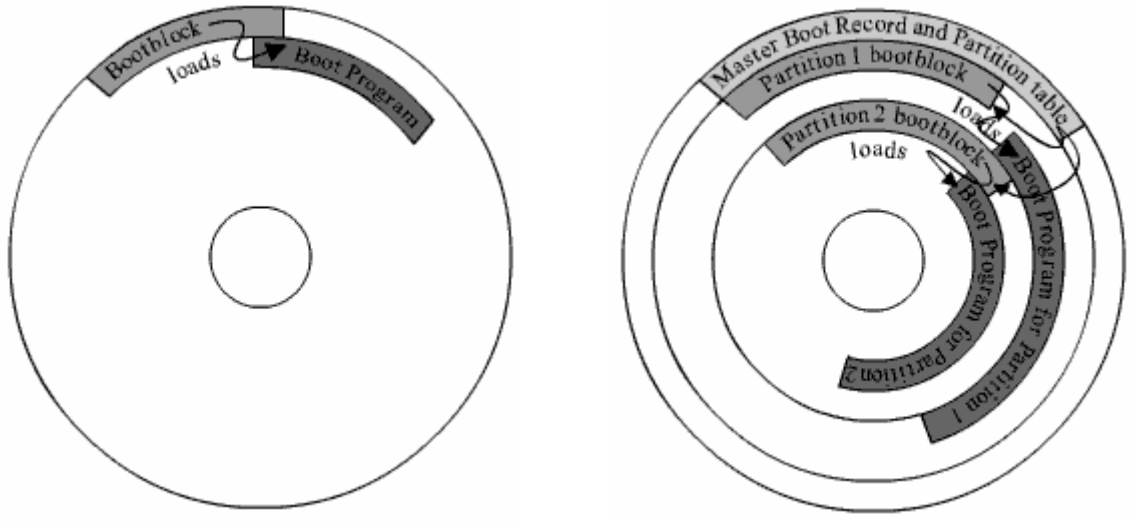
Tasks cannot execute all machine level instructions, nor can they access all CPU registers or all parts of memory. They can, however, access memory regions belonging to less-privileged processes, in order to perform I/O for them. One task, the system task does not do I/O in the normal sense but exists in order to provide services. Such as copying between different memory regions, for processes which are not allowed to do such things for themselves. On machines, which do not provide different privilege levels, such as older Intel processors, these restrictions cannot be enforced, of course.

Layer 3 contains processes that provide useful services to the user processes. These server processes run at a less privileged level than the kernel and tasks and cannot access I/O ports directly. They also cannot access memory outside the segments allotted to them, The *Memory Manager* (MM) carries out all the Minix system calls that involve memory management, such as FORK, EXEC, and BRK. The *File System* (FS) carries out all the file system calls, such as READ, MOUNT, and CHDIR. In MlNIX the resource management is largely in the kernel (layers 1 and 2), and system call interpretation is in layer 3. The file system has been designed as a file "server" and can be moved to a remote machine with almost no changes. This also holds for the memory manager, although remote memory servers are not as useful as remote file servers.

Finally, layer 4 contains the entire user processes-shells, editors, compilers, and user-written *a.out* programs. A running system usually has some processes that are started when the system is booted and which run forever. For example, a *daemon* is a background process that executes periodically or always waits for some event, such as a packet arrival from the network. In a sense a daemon is a server that is started independently and runs as a user process. However, unlike me servers installed in privileged slots, such programs can not get the special treatment from the kernel that the memory and file server processes receive.

3.2 Bootstrapping Minix

Figure 3.2 shows how Minix is loaded into memory. It is, of course, loaded from a disk**.** Figure below shows how diskettes and partitioned disks are laid out. When the system is started, the hardware (actually, a program in ROM) reads the first sector of the boot disk and executes the code found there. On an unpartitioned Minix diskette the first sector is a *bootblock* which loads the boot program, as in Figure 3.2(a). Hard disks are partitioned, and the program on the first sector reads the partition table, which is



(a) **(b)**

Figure 3.2: Disk structure used for bootstrapping. (a) Unpartitioned disk. The first sector is the bootblock. (b) Partitioned disk. The first sector is the master boot record.

also in the first sector, and loads and executes the first sector of the active partition, as shown in Figure 3.2(b). (Normally one and only one partition is marked active). A Minix partition has the same structure as an unpartitioned Minix diskette, with a bootblock that loads the boot program.

The actual situation can be a little more complicated than the Figure shows, because a partition may contain subpartitions. In this case the first sector of the partition will be another master boot record containing the partition table for the subpartitions. Eventually, however, control will be passed to a boot sector, the first sector on a device that is not further subdivided. On a diskette the first sector is always a boot sector. Minix does allow a form of partitioning of a diskette, but only the first partition may be booted; there is no separate master boot record, and subpartitions are not possible. This makes it possible for partitioned and nonpartitioned diskettes to be mounted in exactly the same way. The main use for a partitioned floppy disk is that it provides a convenient way to divide an installation disk into a

root image to be copied to a RAM disk and a mounted portion that can be dismounted when no longer needed, in order to free the diskette drive for continuing the installation process.

The Minix boot sector is modified at the time it is written to the disk by patching in the sector numbers needed to find a program called *boot* on its partition or subpartition. This patching is necessary because previous to loading the operating system there is no way to use the directory and file names to find a file. A special program called *installboot* is used to do the patching and writing of the boot sector. *Boot* is the secondary loader for Minix. It can do more than just load the operating system however, as it is a *monitor program* that allows the user to change, set, and save various parameters. Boot looks in the second sector of its partition to find a set of parameters to use. Minix, like standard Unix, reserves the first 1 K block of every disk device as a *bootblock***,** but only one 512-byte sector is loaded by the ROM boot loader or the master boot sector, so 512 bytes are available for saving settings. These control the boot operation, and are also passed to the operating system itself. The default settings present a menu with one choice, to start Minix, but the settings can be modified to present a more complex menu allowing other operating systems to be started (by loading and executing boot sectors from other partitions), or to start Minix with various options. The default settings can also be modified to bypass the menu and start Minix immediately.

Boot is not a part of the operating system, but it is smart enough to use the file system data structures to find the actual operating system image. By default, *boot* looks for a file called /*minix*, or, if there is a /*minix*/ directory, for the newest file within it, but the boot parameters can be changed to look for a file with any name. This degree of flexibility is unusual, and most operating systems have a predefined file name for the system image. But, Minix is an unusual operating system that encourages users to modify it and create experimental new versions.

The Minix image loaded by *boot* is nothing more than a concatenation of the individual files produced by the compiler when the kernel, memory manager, fil0e system, and init programs are compiled. Each of these includes a short header of the type defined in *include/a.out.h*, and from the information in the header of each part, *boot* determines how much space to reserve

for uninitialized data after loading the executable code and the initialized data for each part, so the next part can be loaded at the proper address. The *_sizes* array located at the beginning of the kernel's data segment and defined at the end of the assembly language file *mpx386.s* also receives a copy of this information so the kernel itself can have access to the locations and sizes of all the modules loaded by boot. The regions of memory available for loading the bootsector, *boot* itself, and Minix will depend upon the hardware. Also, some machine architectures may require adjustment of internal addresses within executable code to correct them for the actual address where a program is loaded. The segmented architecture of Intel processors makes this unnecessary. Once the loading is complete, control passes to the executable code of the kernel.

3.3 Memory Layout Of Minix

When Minix is compiled, all the source code files in *src/kernel/*, *src/mm/*, and *src/fs/* are compiled to object files. All the object files in *src/kemel/* are linked to form a single executable program, kernel. The object files in *src/md* are also linked together to form a single executable program, *mm*. The same holds for *fs*. Extensions can be added by adding additional servers, for instance network s u p port is added by modifying *include/minix/config.h* to enable compilation of the files in *src/inet/* to form *inet*. Another executable program, *init*, is built in *src/tools/*. The program installboot (whose source is in *src/boot/*) adds names to each of these programs, pads each one out so that its length is a multiple of the disk sector size (to make it easier to load the parts independently), and concatenates them onto a single file. This new file is the binary of the operating system and can be copied onto the root directory or the */minix/* directory of a floppy disk or hard disk partition. Later, the boot monitor program can load and execute the operating system.

Figure 3.3 shows the layout of memory after the concatenated programs are separated and loaded. Details, of course, depend upon the system configuration. The example in the Figure is for a Minix system configured to take advantage of a computer equipped with several megabytes of memory. This makes it possible to deallocate a large number of file system buffers, but the resulting large file system does not fit in the lower range of memory, below

640K. If the number of buffers is reduced drastically it is possible to make the entire system fit into less than 640K of memory, with room for a few user processes as well.

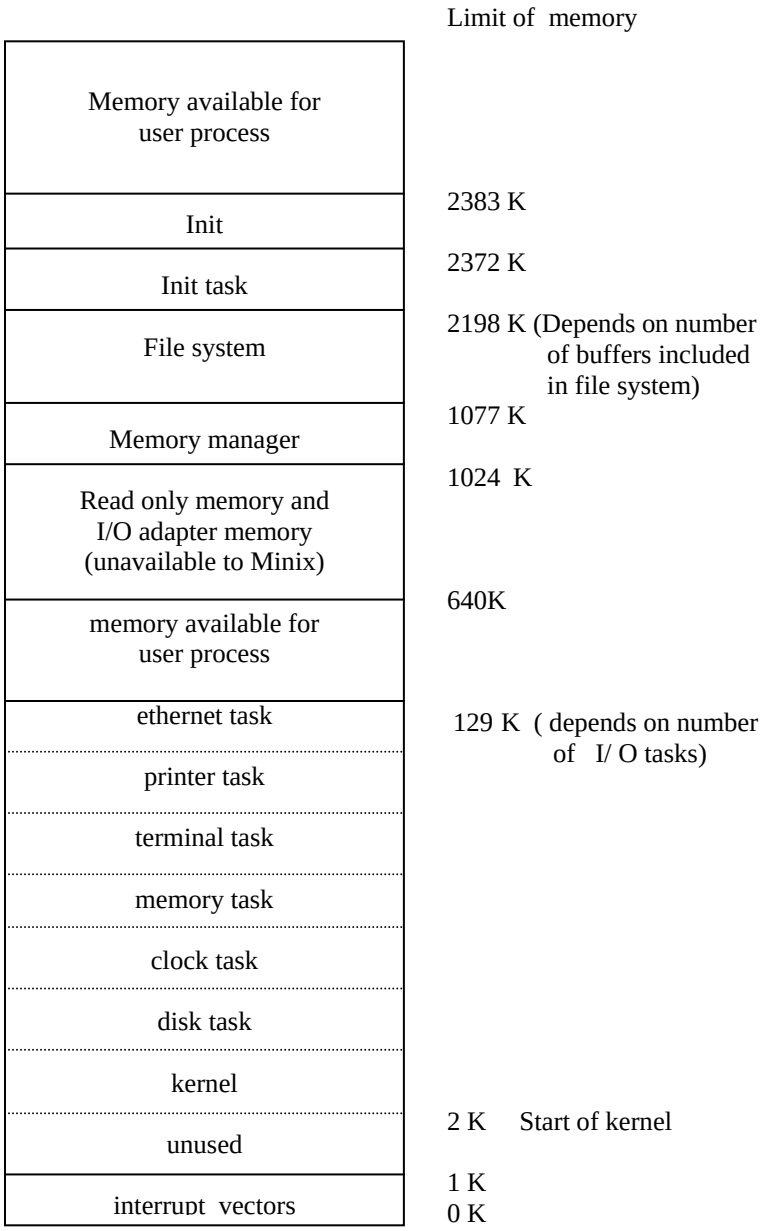


Figure 3.3: Memory layout after Minix has been loaded from the disk into memory. The four (or five, with network support) independently compiled and linked parts are clearly distinct. -The sizes are approximate, depending on the configuration.

CHAPTER 4 KERNEL INITIALIZATION IN MINIX

Minix for IBM PC-type machines can be compiled in 16-bit mode if compatibility with older processor chips is required, or in 32-bit mode for better performance on 80386+ processors. The same C source code is used and the compiler generates the appropriate output depending upon whether the compiler itself is the 16-bit or 32-bit version of the compiler. A macro defined by the compiler itself determines the definition of the *_WORD_SIZE* macro in *include/minix/config.h*. The first part of Minix to execute is written in assembly language, and different source code files must be used for the 16-bit or 32-bit compiler. The 32-bit version of the initialization code is in *mpx386.s*. The alternative, for 16-bit systems, is in *mpx88.s*. Both of these also include assembly language support for other low-level kernel operations. The selection is made automatically in *mpx.s*. In the following discussion we will use mpx386.s since we are going to use Intel 80x86 architecture to support virtual memory.

4.1 System Initialization

The startup of Minix involves several transfers of control between the assembly language routines in *mpx386.s* and routines written in C and found in the files *start.c* and *main.c*. We will describe these routines in the order that they are executed, even though that involves jumping from one file to another.

4.1.1 Low level initialization: mpx386.s File

Once the bootstrap process has loaded the operating system into memory, control is transferred to the label Minix (in *mpx386.s*). The first instruction is a jump over a few bytes of data: this includes the boot monitor flags, used by the boot monitor to identify various characteristics of the kernel, most importantly, whether it is a 16-bit or 32-bit system. The boot monitor always starts in 16-bit mode, but switches the CPU to 32-bit mode if necessary. This happens before control passes to Minix. The monitor also sets up a stack. There is a

substantial amount of work to be done by the assembly language code, setting up a stack frame to provide the proper environment for code compiled by the C compiler, copying tables used by the processor to define memory segments, and setting up various processor registers. As soon as this work is complete, the initialization process continues by calling the C function *cstart*. Note that it is referred to as *cstart* in the assembly language code. This is because all functions compiled by the C compiler have an underscore prepended to their names in the symbol tables, and the linker looks for such names when separately compiled modules are linked. Since the assembler does not add underscores, the writer of an assembly language program must explicitly add one in order for the linker to be able to find a corresponding name in the object file com-piled by the C compiler. *cstart* calls another routine to initialize the Global Descriptor Table, the central data structure used by Intel 32-bit processors to oversee memory protection, and the Interrupt Descriptor Table, used to select the code to be executed for each possible interrupt type. Upon returning from *cstart* the *lgdt* and *lidt* instructions make these tables effective by loading the dedicated registers by which they are addressed. The following instruction,

jmpf CS-SELECT0R:csinit

looks at first glance like a no-operation, since it transfers control to exactly where control would be if there were a series of *nop* instructions in its place. But this is an important part of the initialization process. This jump forces use of the structures just initialized. After some more manipulation of the processor registers, Minix terminates with a jump (not a call) to the kernel's main entry point (in *main.c*). At this point the initialization code in mpx386.s is complete. The rest of the file contains code to start or restart a task or process, interrupt handlers, and other support routines that had to be written in assembly language for efficiency.

4.1.2 High level initialization

We will now look at the top-level C initialization functions. The general strategy is to do as much as possible using high-level C code. There are already two versions of the mpx code, as

we have seen, and anything that can be off-loaded to C code eliminates two chunks of assembler code. Almost the first thing done by *cstart* (in *start.c*) is to set up the CPU's protection mechanisms and the interrupt tables, by calling *prot-init*. Then it does such things as copying the boot parameters to the kernel's part of memory and converting them into numeric values. It also determines the type of video display, size of memory, machine type, processor operating mode (real or protected), and whether a return to the boot monitor is possible. All information is stored in appropriate global variables, for access when needed by any part of the kernel code.

Main (in *main.c*), completes initialization and then starts normal execution of the system. It configures the interrupt control hardware by calling *intr-init*. This is done here because it can not be done until the machine type is known, and the procedure is in a separate file because it is so dependent upon the hardware. The parameter (1) in the call tells *intr-init* that it is initializing for Minix. With a parameter (0) it can be called to reinitialize the hardware to the original state. The call to *intr-init* also takes two steps to insure that any interrupts that occur before initialization is complete have no effect. First a byte is written to each interrupt controller chip that inhibits response to external input. Then all entries in the table used to access device-specific interrupt handlers are filled in with the address of a routine that will harmlessly print a message if a spurious interrupt is received. Later these table entries will be replaced, one by one, with pointers to the handler routines, as each of the I/O tasks runs its own initialization routine. Each task then will reset a bit in the interrupt controller chip to enable its own interrupt input.

Mem_init is called next. It initializes an array that defines the location and size of each chunk of memory available in the system. As with the initialization of the interrupt hardware, the details are hardware dependent and isolation of *mem_init* (in *misc.c)* as a function in a separate file keeps main itself free of code that is not portable to different hardware.

The largest part of *rnain*'s code is devoted to setup of the process table, so that when the first tasks and processes are scheduled, their memory maps and registers will be set correctly. All

slots in the process table are marked as free, and the *pproc_addr* array that speeds access to the process table is initialized.

The largest part of *main*, initializes the process table with the necessary information to run the tasks, servers, and *init*. All of these processes must be present at startup time and none of them will terminate during normal operation.

The tasks, of course, are all compiled into the same file as the kernel, and the information about their stack requirements is in the *tasktab* array defined in *table.c*. Since tasks are compiled into the kernel and can call code and access data located anywhere in the kernel's space, the size of an individual task is not meaningful, and the size field for each of them is filled with the sizes for the kernel itself. The array *sizes* contains the text and data sizes in clicks of the kernel, memory manager, file system, and *init*. This information is patched into the kernel's data area by *boot* before the kernel starts executing and appears to the kernel as if the compiler had provided it. The first two elements of *sizes* are the kernel's text and data sizes; the next two are the memory manager's, and so on. If any of the four programs does not use separate I and D space, the text size is 0 and the text and data are lumped together as data. Assigning *sizeindex* a value of zero for each of the tasks assures that the zeroth element of *sizes* will be accessed for all of the tasks.

The design of the original IBM PC placed read-only memory at the top of the usable range of memory, which is limited to 1 MB on an 8088 CPU. Modern PC-compatible machines always have more memory than the original PC, but for compatibility they still have read-only memory at the same addresses as the older machines. Thus, the read-write memory is discontinuous, with a block of ROM between the lower 640 KB and the upper range above 1 MB. The boot monitor loads the servers and *init* into the memory range above the ROM if possible. This is primarily for the benefit of the file system, so a very large block cache can be used without bumping into the read-only memory.

Two entries in the process table correspond to processes that do not need to be scheduled in the ordinary way. These are the IDLE and HARDWARE processes. IDLE is a do-nothing

loop that is executed when there is nothing else ready to run, and the HARDWARE process exists for bookkeeping purposes---it is credited with the time used while servicing an interrupt. All other processes are put on the appropriate queues by calling *lock-ready*. The function called, *lock-ready*, sets a lock variable, *switching*, before modifying the queues and then removes the lock when the queue bas been modified.

The last step in initializing each slot in the process table is to call *alloc_segments*. This procedure is part of the system task, but of course no tasks are running yet, and it is called as an ordinary procedure. It is a machine-dependent routine that sets into the proper fields the locations, sizes, and permission levels for the memory segments used by each process. For older Intel processors that do not support protected mode, it defines only the segment locations, It would have to be rewritten to handle a processor type with a different method of allocating memory.

Once the process table is initialized for all the tasks, the servers, and *init*, the system is almost ready to roll. The variable *bill-ptr* tells which process gets billed for processor time; it needs to have an initial value set, and IDLE is an appropriate choice. Later on it may be changed by the next function called, *lock-pick-proc*. All of the tasks are now ready to run and *bill-ptr* will be changed when a user process runs. *Lock-pick-proc*'s other job is to make the variable *proc-ptr* point to the entry in the process table for the next process to be run. This selection is made by examining the task, server, and user process queues, in that order. In this case, the result is to point *proc-ptr* to the entry point for the console task, which is always the first one to be started.

Finally, *main* has run its course. In many C programs main is a loop, but in the Minix kernel its job is done once the initialization is complete. The call to *restart* starts the first task. Control will never return to *main*.

Restart is an assembly language routine in *mpx386.s*. In fact, restart is not a complete function; it is an intermediate entry point in a larger procedure. *_restart* causes a context switch, so the process pointed to by *proc-ptr* will run. When *_restart* has executed for the first time we can say that Minix is running-it is executing a process. *_Restart* is executed again and

again as tasks, servers, and user processes are given their opportunities to run and then are suspended, either to wait for input or to give other processes their turns.

The task queued first (the one using slot 0 of the process table, that is, the one with the most negative number) is always the console task, so other tasks can use it to report progress or problems as they start. It runs until it blocks trying to receive a message. Then the next task will run until it, too, blocks trying to receive a message. Eventually, all the tasks will be blocked, so the memory manager and file system can run. Upon running for the first time, each of these will do some initialization, but both of them will eventually block, also. Finally *init* will fork off a *getty* process for each terminal. These processes will block until input is typed at some terminal, at which point the first user can log in.

We have now traced the startup of Minix through three files, two written in C and one in assembly language. The assembly language file, mpx386.s, contains additional code used in handling interrupts. However, before we go on let us wrap up with a brief description of the remaining routines in the two C files. The other procedures in *start.c* are *k_atoi*, which converts a string to an integer, and *k_getenv*, which is used to find entries in the kernel's environment, which is a copy of the boot parameters. These are both simplified versions of standard library functions which are rewritten here in order to keep the kernel simple. The only remaining procedure in *main.c* is *panic*. It is called when the system has discovered a condition that makes it impossible to continue. Typical panic conditions are a critical disk block being unreadable, an inconsistent internal state being detected, or one part of the system calling another part with invalid parameters. The calls to *printf* here are actually calls to the kernel routine *printk*, so the kernel can print on the console even if normal interprocess communication is disrupted.

4.2 Data structures involved in system initialization

In this section we are presenting some important data structures used in Minix for system initialization.

4.2.1 Minix Data Types

Minix allocates memory in terms of clicks. For Intel chipset a click is defined by a constant named *CLICK_SIZE* and is equal to 256. And another constant *CLICK_SHIFT* denotes the bits needed to represent click size and is defined as follows:

```
#define CLICK_SHIFT 8          /* log2 of CLICK_SIZE */
```

Minix defined the following data types for handling memory-related operations.

```
typedef unsigned int vir_clicks;      /*  virtual addresses and lengths in clicks */
typedef unsigned long phys_bytes;  /*  physical addresses and lengths in bytes */
typedef unsigned int phys_clicks;   /*  physical addresses and lengths in clicks */
typedef unsigned int vir_bytes;       /*  virtual addresses and lengths in bytes */
typedef unsigned reg_t;               /* machine register */
```

Minix uses a structure *mem_map* to represent a memory segment such as text or data segment. This structure contains three fields as shown below. The first field *mem_vir* contains the virtual address of the memory segment. The second field *mem_phys* contains the physical address and third field *mem_len* contains the length of the memory segment. All the three fields are measured in terms of clicks rather then in terms of bytes. All segments must start on a click boundary and occupy an integral number of clicks.

```
struct mem_map {
        vir_clicks mem_vir;             /* virtual address */
        phys_clicks mem_phys;    /* physical address */
        vir_clicks mem_len;             /* length */
};
```

To represent a physical memory chunk Minix uses a structure *memory* as given below. It contains the base address and size of the chunk in terms of clicks.

```
struct memory {
        phys_clicks base;
        phys_clicks size;
};
```

All available memory chunks are stored in an array named *mem* as defined below.

```
struct memory mem[NR_MEMS];
```

In the above declaration *NR_MEMS* is a constant to represent maximum number of chunks. For Intel chipset its value is 3.

mem[0] contains the low memory chunk that can be up to 1 Mb. *mem[1]* contains the extended memory. This usually starts at 1 Mb. *mem[2]* is used to contain the another chunk that may be present just below 16 Mb, reserved under DOS for shadowing ROM.

4.2.2 Data Structures for Segmentation Unit

Intel 80x86 based architecture have a segmentation unit as explained in chapter 2. Segmentation unit uses descriptor tables to store information related to various segments of processes. Minix defines the format for each entry in a descriptor table in a structure named *segdesc_s* and is given below.

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

Minix defines gate descriptor for various types of gates in a structure named *gatedesc_s*. gate descriptors are stored in interrupt descriptor table *IDT*.

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

The segment descriptors of all the segments in a system are stored in a global descriptor table explained already in chapter 2. Minix defines global descriptor table in an array named *gdt*.

```
PUBLIC struct segdesc_s gdt[GDT_SIZE];
```

GDT_SIZE is defined as follows:

```
#define GDT_SIZE (FIRST_LDT_INDEX + NR_TASKS + NR_PROCS)
```

NR_TASKS and *NR_PROCS* are explained in detail in the next section and value of *FIRST_LDT_INDEX*. is 14. In Minix entries up to first *FIRST_LDT_INDEX* in *GDT* are fixed. Global descriptors are prescribed by the BIOS and are used as it is in Minix. The fixed global descriptors used by Minix are given below.

```
#define GDT_INDEX          1    /* GDT descriptor */
#define IDT_INDEX          2    /* IDT descriptor */
#define DS_INDEX           3    /* kernel DS */
#define ES_INDEX           4    /* kernel ES (386: flag 4 Gb at startup) */
#define SS_INDEX           5    /* kernel SS (386: monitor SS at startup) */
#define CS_INDEX           6    /* kernel CS */
#define MON_CS_INDEX       7    /* temp for BIOS (386: monitor CS at startup) */
#define TSS_INDEX          8    /* kernel TSS */
#define DS_286_INDEX       9    /* scratch 16-bit source segment */
#define ES_286_INDEX       10   /* scratch 16-bit destination segment */
#define VIDEO_INDEX        11   /* video memory segment */
#define DP_ETH0_INDEX      12   /* Western Digital Etherplus buffer */
#define DP_ETH1_INDEX      13   /* Western Digital Etherplus buffer */
#define FIRST_LDT_INDEX    14   /* rest of descriptors are LDT's */
```

The gate descriptors for all the exceptions and interrupts are stored in an array *idt* and is defined as follows.

```
struct gatedesc_s idt[IDT_SIZE];
```

All the exceptions and interrupts handled by Minix are stored in a gate table. Each entry type in this table is defined by the following structure called *gate_table_s*. It contains a pointer to the exception/interrupt handling routine, vector number of the exception/interrupt and privilege level of this gate.

```
static struct gate_table_s {
        _PROTOTYPE( void (*gate), (void) );
        unsigned char vec_nr;
        unsigned char privilege;
 }
```

The gate table is defined by an array named *gate_table* as given below. The gate table is used by the Kernel to setup *idt* table entries for exceptions and interrupts during system initialization.

```
gate_table[] = {
        divide_error, DIVIDE_VECTOR, INTR_PRIVILEGE,
        single_step_exception, DEBUG_VECTOR, INTR_PRIVILEGE,
        nmi, NMI_VECTOR, INTR_PRIVILEGE,
        breakpoint_exception, BREAKPOINT_VECTOR, USER_PRIVILEGE,
        overflow, OVERFLOW_VECTOR, USER_PRIVILEGE,
        bounds_check, BOUNDS_VECTOR, INTR_PRIVILEGE,
        inval_opcode, INVAL_OP_VECTOR, INTR_PRIVILEGE,
        copr_not_available, COPROC_NOT_VECTOR, INTR_PRIVILEGE,
        double_fault, DOUBLE_FAULT_VECTOR, INTR_PRIVILEGE,
        copr_seg_overrun, COPROC_SEG_VECTOR, INTR_PRIVILEGE,
        inval_tss, INVAL_TSS_VECTOR, INTR_PRIVILEGE,
        segment_not_present, SEG_NOT_VECTOR, INTR_PRIVILEGE,
        stack_exception, STACK_FAULT_VECTOR, INTR_PRIVILEGE,
        general_protection, PROTECTION_VECTOR, INTR_PRIVILEGE,
        page_fault, PAGE_FAULT_VECTOR, INTR_PRIVILEGE,
        copr_error, COPROC_ERR_VECTOR, INTR_PRIVILEGE,
};
```

Vector numbers for these exceptions and interrupts are defined in <*kernel/const.h*>. Minix uses three kind of privilege levels as given below.

```
#define INTR_PRIVILEGE    0
#define TASK_PRIVILEGE    1
#define USER_PRIVILEGE    3
```

All constants related to protected mode in Minix are defined in a file <*kernel/protect.h*>.

4.2.3 Data Structures for process management

To implement the process model, the operating system maintains a table (an array of structures), called the process table, with one entry per process. This entry contains information about the process' state, its program counter, stack pointer, memory allocation, the status of its open files, its accounting and scheduling information, and everything else about the process that must be saved when the process is switched from running to ready state so that it can be restarted later as if it had never been stopped.

In Minix the process management, memory management, and file management are each handled by separate modules within the system, so the process table is partitioned, with each module maintaining the fields that it needs. In this section we will discuss process table managed by the kernel for process management. This process table is defined by an array named *proc* of type *struct proc*.

Each process has an entry in the process table. Each entry have various fields related to process management and is defined by a structure named *proc* as given below.

```
struct proc {

struct stackframe_s  p_reg;                 /* process' registers saved in stack frame */

#if (CHIP == INTEL)
        reg_t  p_ldt_sel;                   /* selector in gdt giving ldt base and limit */
        struct segdesc_s  p_ldt[2] ;        /* local descriptors for code and data */
                                            /* 2 is LDT_SIZE */
#endif /* (CHIP == INTEL) */

reg_t *p_stguard;                           /* stack guard word */
int p_nr;                                   /* number of this process (for fast access) */
int p_int_blocked ;                         /* nonzero if int msg blocked by busy task */
int p_int_held;                             /* nonzero if int msg held by busy syscall */
struct proc *p_nextheld ;                   /* next in chain of held-up int processes */
int p_flags;                                /* P_SLOT_FREE, SENDING, RECEIVING, etc. */
struct mem_map p-map[NR-SEGS];              /* memory map */
pid_t p_pid;                                /* process id passed in from MM */

clock_t user_time;                          /* user time in ticks */
clock_t sys_time;                           /* sys time in ticks */
clockt child_utime;                         /* cumulative user time of children */
clock_t child_stime;                        /* cumulative sys time of children */
clock_t p_alarm;                            /* time of next alarm in tick s or 0 */

struct proc *p_callerq;                     /* head of 1ist of procs wishing to send */
struct proc *p_sendlink;                    /* link to next proc wishing to send */
message *p_messbuf;                         /* pointer to message buffer */
int p_getfrom;                              /* from whom does process want to receive? */
int p_sendto;

struct proc *p_nextready;                   /* pointer to next ready process */
sigset_t p_pending;                         /* bit map for pending signals */
unsigned p_pendcount ;                      /* count of pending and unfinished signals */
char p_name [16] ;                          /* name o f the process */

};

struct proc proc[NR_TASKS + NR_PROCS] /* process table */
```

Guard word for task stacks is defined as follows.

```
#define STACK_GUARD ((reg_t) (sizeof(reg_t) == 2 ? 0xBEEF : 0xDEADBEEF))
```

The field *p_map* in *proc* structure above is used to represent memory map of process. Each process in Minix has three segments namely text, data and stack segment. The field *p_map* points to these segments. Constants related to these segments are defined as follows:

```
#define NR_SEGS         3
#define T               0
#define D               1
#define S               2
```

There is a field *p_flags* in process structure *proc* to denote the process status. A process is runnable if and only if *p_flags* = 0. The possible flags are given below.

```
#define P_SLOT_FREE         001     /* set when slot is not in use */
#define NO_MAP              002     /* keeps unmapped forked child from running */
#define SENDING             004     /* set when process blocked trying to send */
#define RECEIVING           010     /* set when process blocked trying to recv */
#define PENDING             020     /* set when inform() of signal pending */
#define SIG_PENDING         040     /* keeps to-be-signalled proc from running */
#define P_STOP              0100    /* set when process is being traced */
```

The total number of the slots in the process table is the sum of two constants *NR_PROCS* and *NR_TASKS.* The constant *NR_TASKS* is used to represent maximum number of user processes and its default value is 32. The constant *NR_TASKS* denotes the total number of tasks running in kernel and is defined as follows:

```
#define NR_TASKS        (9 + ENABLE_WINI + ENABLE_SCSI + ENABLE_CDROM
                        + ENABLE_NETWORKING + 2 * ENABLE_AUDIO)
```

Here ENABLE_*XXX* have a value 0 or 1 depending upon whether the task *XXX* is disabled or enabled respectively.

Process numbers of some important processes have been defined as constants and are given below.

```
#define MM_PROC_NR          0               /* process number of memory manager */
#define FS_PROC_NR          1               /* process number of file system */
#define INET_PROC_NR        2               /* process number of the TCP/IP server */
#define INIT_PROC_NR    (INET_PROC_NR + ENABLE_NETWORKING)
                                            /* init -- the process that goes multiuser */
#define LOW_USER        (INET_PROC_NR + ENABLE_NETWORKING)
                                            /* first user not part of operating system */
```

Addresses of some important process slot in the process table have been defined as constant. These process table addresses are given below:

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

4.2.4 Data Structures for task initialization

Minix defines a number of tasks and each task is identified by its task number or task name. Task numbers are negative numbers starting from –NR_TASKS upto –1. The tasks numbers are assigned to the tasks in <*minix/com.h*> in the manner shown below.

```
#define TTY             (DL_ETH - 1)
                        /* terminal I/O class */

#define DL_ETH          (CDROM - ENABLE_NETWORKING)
                        /* networking task */

#define CDROM           (AUDIO - ENABLE_CDROM)
                        /* cd-rom device task */

#define AUDIO           (MIXER - ENABLE_AUDIO)


#define MIXER           (SCSI - ENABLE_AUDIO)
                        /* audio & mixer device tasks */

#define SCSI            (WINCHESTER - ENABLE_SCSI)
                        /* scsi device task */

#define WINCHESTER      (SYN_ALRM_TASK - ENABLE_WINI)
                        /* winchester (hard) disk class */

#define SYN_ALRM_TASK   -8
                        /* task to send CLOCK_INT messages */

#define IDLE            -7
                        /* task to run when there's nothing to run */

#define PRINTER         -6      /* printer I/O class */

#define FLOPPY          -5      /* floppy disk class */

#define MEM             -4
                        /* /dev/ram, /dev/(k)mem and /dev/null class */
```

```
#define CLOCK          -3          /* clock class */

#define SYSTASK        -2          /* internal functions */

#define HARDWARE       -1
                                   /* used as source on interrupt generated msgs*/
```

Each task has its own stack for use. The stack size of each task is calculated as shown below.

```
#define          SMALL_STACK(128 * sizeof(char *))
#define          TTY_STACK     (3 * SMALL_STACK)
#define          SYN_ALRM_STACK      SMALL_STACK
#define          DP8390_STACK(SMALL_STACK * ENABLE_NETWORKING)
#define          IDLE_STACK    ((3+3+4) * sizeof(char *))          /* 3 intr, 3 temps, 4 db */
 #define         PRINTER_STACK         SMALL_STACK
#define          WINCH_STACK(2 * SMALL_STACK * ENABLE_WINI)
#define          SCSI_STACK    (2 * SMALL_STACK * ENABLE_SCSI)
#define          CDROM_STACK           (4 * SMALL_STACK * ENABLE_CDROM)
#define          AUDIO_STACK (4 * SMALL_STACK * ENABLE_AUDIO)
#define          MIXER_STACK (4 * SMALL_STACK * ENABLE_AUDIO)
#define          FLOP_STACK    (3 * SMALL_STACK)
#define          MEM_STACK     SMALL_STACK
#define          CLOCK_STACKSMALL_STACK
#define          SYS_STACK     SMALL_STACK
#define          HARDWARE_STACK     0                    /* dummy task, uses kernel stack */

#define          TOT_STACK_SPACE               (TTY_STACK + DP8390_STACK + SCSI_STACK +
                 SYN_ALRM_STACK + IDLE_STACK + HARDWARE_STACK + PRINTER_STACK +
                 WINCH_STACK + FLOP_STACK + MEM_STACK + CLOCK_STACK + SYS_STACK +
                 CDROM_STACK + AUDIO_STACK + MIXER_STACK)
```

All these tasks are collectively grouped in a task table. Each task has an entry in the task table. The entry type is defined by the following structure called *tasktab*. It contains a pointer to the startup routine of the task, stack size and name of the task.

```
struct tasktab {
        task_t *initial_pc;
        int stksize;
        char name[8];
};
```

The task table is defined by an array named *tasktab* as given below. The task table is used by the Kernel to setup proc table entries for tasks and servers during system initialization. The tty_task should always be first so that other tasks can use printf if their initialization has problems.

```
struct tasktab tasktab[] = {
        { tty_task,             TTY_STACK,      "TTY"           },
#if ENABLE_NETWORKING
        { dp8390_task,          DP8390_STACK,   "DP8390"        },
#endif
#if ENABLE_CDROM
        { cdrom_task,           CDROM_STACK,    "CDROM"         },
#endif
#if ENABLE_AUDIO
        { audio_task,           AUDIO_STACK,    "AUDIO"         },
        { mixer_task,           MIXER_STACK,    "MIXER"         },
#endif
#if ENABLE_SCSI
        { scsi_task,            SCSI_STACK,     "SCSI"          },
#endif
#if ENABLE_WINI
        { winchester_task,      WINCH_STACK,    "WINCH"         },
#endif
        { syn_alrm_task,        SYN_ALRM_STACK, "SYN_AL"        },
        { idle_task,            IDLE_STACK,     "IDLE"          },
        { printer_task,         PRINTER_STACK,  "PRINTER"       },
        { floppy_task,          FLOP_STACK,     "FLOPPY"        },
        { mem_task,             MEM_STACK,      "MEMORY"        },
        { clock_task,           CLOCK_STACK,    "CLOCK"         },
        { sys_task,             SYS_STACK,      "SYS"           },
        { 0,                    HARDWARE_STACK, "HARDWAR"       },
        { 0,                    0,              "MM"            },
        { 0,                    0,              "FS"            },
#if ENABLE_NETWORKING
        { 0,                    0,              "INET"          },
#endif
        { 0,                    0,              "INIT"          },
};
```

Stack space for all the task stacks is allocated by the following declaration.

```
PUBLIC char *t_stack[TOT_STACK_SPACE / sizeof(char *)];
```

CHAPTER 5 DESIGN AND IMPLEMENTATION

To support virtual addressing with non-paging in Minix we have to change the kernel initialization phase. There are numerous issues related to virtual addressing which have to be dealt. In this chapter we will focus on those issues and present the design for kernel initialization to support virtual addressing with non-paging.

5.1 Design issues

Minix uses click size of 256 and is used to represent memory segments. Since we are going to enable paging unit memory will be used in terms of page frames. As we know that page frame size for Intel 80x86 based architecture is 4096 bytes, so we have to increase the click size to 4096 [35].

When allocating memory in terms of pages, it is possible to allocate memory to the user processes either in chunks of 4K or 4M sizes. Both have there own pros and cons. 4K chunk size being smaller in size results in much better allocation of physical memory with lesser internal fragmentation [26] if memory losses due to unused spaces are considered. The use of 4K chunk size is very good for practical purposes but is also much harder to manage. This may result in multiple user processes sharing a single page table for mapping. This has a grater chance to cause overlap of page entries during design and testing phases. This is why for purpose of this project, we have chosen 4M chunk sizes for allocation. When using a 4M page size, each user process will have individual page tables allocated without any fear of unintentional overlap during design phases. Also, with the increasing requirements of modern world programs and availability of larger and cheaper physical memories it is much more convenient to allocate and manage a single 4M chunk instead of 4K chunks.

With this design the memory is allocated to user processes from linear address space. The size of this address space is also worth considering. In this design its size is managed through a constant named *VM_SIZE_CLICKS* with a default size of *0x80000* (2G in clicks). The size is

restricted to 2G only, because this design doesn't support any paging, which practically means that number of user processes are still very much dependent on the size of actual physical memory. 2G being much more then the physical memory present in target machines is more then appropriate for all practical design purposes.

In standard Minix memory manager allocates physical address space to the processes as the paging unit is disabled. Memory manager stores the memory map of each process in its process table. This memory map contains the information about the physical address space of text, data, and stack segment of the process. The kernel in its own process table also copies this memory map. But once paging unit is enabled, memory manager will allocate linear addresses to the processes. Now its up to kernel to map this linear address space onto physical address space and fill up the page table entries accordingly. With Kernel in complete control of memory mapping it would be much more appropriate to relinquish duplication of memory maps from memory manager and letting it consult kernel each time a memory map for a process is needed by it. It results in memory manager process table not holding the memory map of processes.

5.2 Design

In this section we will discuss the overall design of the system initialization phase to support virtual addressing with non-paging in Minix. It includes the modified system initialization phase and description of the updated routines.

5.2.1 System Initialization to support virtual addressing

In Minix memory is represented in terms of chunks as we already discussed in previous section. But, in this design not only the physical memory but also linear address space is represented in terms of these chunks. These chunks are stored in an array named *chunk_table,* which is defined as follows.

```
Struct memory chunk_table[NR_CHUNKS];
```

The structure *memory* is defined already in section 4.2.1 and *NR_CHUNKS* is a constant whose value is16.

System initialization to support virtual addressing is same as in standard Minix up to *mem_init( )* routine. *mem_init( )* is initializing *chunk_table* array now instead of *size* array that defines the base and size of each memory chunk available. The values are stored in terms of clicks. Two chunks without any gap between them would get merged to form a larger size chunk. To manage these chunks, various functions [28] have been defined and are listed below.

chunk_init( ) -This routine initializes the *chunk_table* with zero entries.

chunk_add( ) - This routine is used to add a chunk in the *chunk_table*. It also merges two chunks if they are contiguous to form a larger chunk.

chunk_del( ) - This routine deletes the specified chunk from the *chunk_table*.

chunk_find( ) - This routine finds a specified size chunk from the *chunk_table*.

After initializing memory chunks the routine *main( )* would find out where the different parts of the Minix image are loaded. To perform this task *main( )* will invoke a routine named *calc_maps( ).* The results will be stored in an array *image_maps*, defined as follows:

```
struct image_map image_maps[MAP_NR];
```

MAP_NR denotes maximum number of executables in an image. Executables includes kernel, servers and first user process *init( ).* The constant *MAP_NR* has a value 16. The structure *image_map* is used to represent an image. It contains memory map of the image, size of *bss* (an area used by process to store static data), size of stack, starting address of the executable code in the image and a valid flag. Its definition is given below.

```
struct image_map
{
        struct mem_map im_map[NR_SEGS];
        phys_clicks im_bss_clicks;
        phys_clicks im_stack_clicks;
        phys_bytes im_initial_pc;
        int im_valid;
};
```

Later the information stored in *image_maps* array can be remapped in a virtual address space , before it is stored in the process structure. The *calc_maps( )* also calculates which part of the memory should not be used for user processes by deleting the memory chunks from chunk table corresponding to these image maps.

main.c would now call another routine *vm_init( )* for virtual memory initialization. In virtual memory management, pages having linear address space are mapped on page frames having physical address space. We need to maintain the status of page frames to find the free page frames for this mapping [31]. The status information for each page frame is stored in a table *rlmem_table*. The entry at index x in this table would contain the reference count of page frame x. An entry of 0 indicates page frame is free. *vm_init( )* firstly invokes routine *rlmem_init( )* to create and initialize this page frame status table. *rlmem_init( )* also removes all physical memory chunks present in *chunk_table* while noting the status of page frames corresponding to these chunks in *rlmem_table*.

After initializing *rlmem_table* through rlmem_*init( ), vm_init( )* then creates a page global directory and page tables to map linear address space from 0 to *hi_mem* (size of RAM) on the identical physical address space. This identical mapping helps the kernel to access RAM directly. After setting the tables for two-level paging to map RAM, it enables the paging unit through an assembly routine *vm_enable( ).*

After that, It would calculate the base of linear address space to be used by servers and processes. Now this linear address space starting from *paging_base* and having range of *VM_SIZE_CLICKS* (2 GB), as shown in Figure 5.1, is added to *chunk_table* so that servers and user processes can treat it as normal memory [29]. In this stage this is the only chunk

present in *chunk_table* since all the physical chunks had been removed already as discussed above.

At this step control would return back to *main( ) and* would then setup entries in the process table for tasks by invoking another routine *init_task( ).*

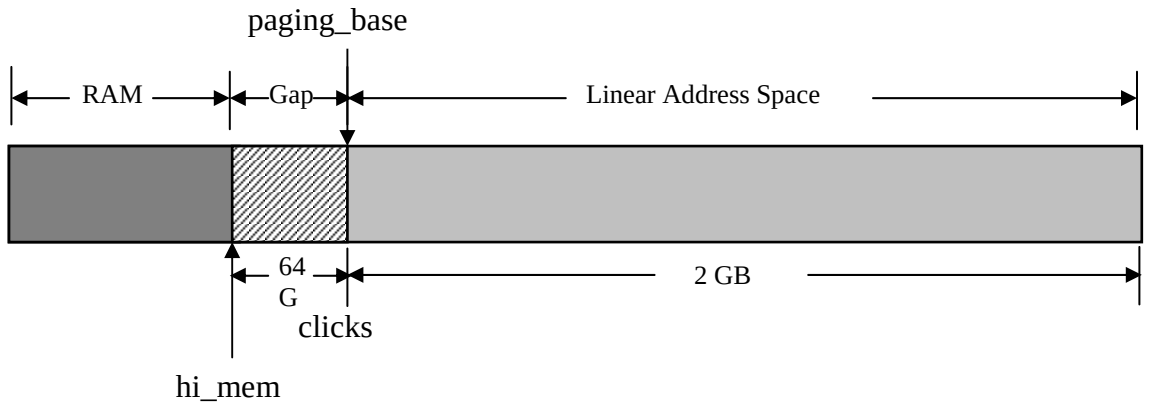


Figure 5.1 : Virtual Addressing

init_task( ) would set up the initial program counter field, stack pointer for the task, stack guard and segment map of the task. Task uses the segment map of kernel since tasks are compiled into the kernel and can call code and access data located anywhere in the kernel's space. The memory map of task is in terms of physical address space.

After setting up entries of tasks in process table, *main( )* would set up entries for the servers such as MM, FS and process *init*. For each of them, it will re-map their image part on the virtual address space [37] by invoking a routine *vm_map_server( ).*

Vm_map_server( ) maps provided image on the virtual address space by extracting sufficient address space from *chunk_table* and updates the corresponding entries in page global directory and page tables to map this linear address space on the physical address of the image. It then increase the *paging_base* by the image size plus 4 MB (gap between successive servers or processes).

Then this remapped image is used by *main.c* to set up memory map of the server or process. In this way, linear address space is used for all servers and user processes. *Main.c* then set up the local segment descriptor table entries for these servers and processes in process table by calling a routine *alloc_segments()* already present in standard Minix.

Then it follows the same step by calling *lock_pick_proc()* as standard Minix does.

5.2.2 Description of routines involved in kernel initialization

In the above section we discussed the system initialization of Minix to support virtual addressing with non-paging. Now we will discuss in detail the major routines involved in system initialization.

5.2.2.1 The calc_maps( ) routine

The function of calc_maps( ) routine is to find out where the different parts of the Minix image are loaded. It stores the results in an array *image_maps*, explained in section 5.2. The prototype is

void calc_maps()

Kernel, servers and *init* process are concatenated to form a Minix image. The boot monitor loads this image during booting. The image maps of these subparts are stored at location starting from *IMAGE_HDRS* (normally 0x600).

The *calc_maps( )* routine reads these headers one by one and calculates memory map for text, data and stack segments. It stores the memory map for each subpart in *image_maps* array at the corresponding entry. This means entry at index 0 in *image_maps* is for kernel, index 1 is for MM server and so on. In this routine we have to take care that the image of servers are loaded in extended memory instead of just behind kernel image. The routine also removes the entry of physical memory, where these images are loaded, from the chunk table. This makes that part of physical memory unavailable for user processes.

5.2.2.2 The rlmem_init( ) routine

This routine is used to create and initialize a table *rlmem_table*, which contains the reference count for each page frame. A reference count of 0 indicates that page frame is free. Thus, the size of table depends on the total number of page frames i.e. RAM size. So, *rlmem_init( )* calculates the highest address of physical memory by scanning through *chunk_table*, and store this value in a variable named *hi_mem*. Its value is equal to the sum of the base address and the size of the last chunk and is represented in terms of clicks. It represents the total number of page frames actually. The prototype of this routine is

void rlmem_init()

Now, a memory chunk of size *hi_mem* bytes or (hi_mem/4096) clicks is removed from the chunk table to store *rlmem_table*. The starting address of the table will be stored in a variable *rlmem_table_base* and its size in variable *rlmem_table_size*.

Initially all entries in the *rlmem_table* are set to 1 to represent that page frame is allocated. Now *rlmem_init( )* will read *chunk_table* and set entries in *rlmem_table* to 0 for page frames corresponding to free memory chunks available. Also, all the chunks would be removed from the chunk table. The *rlmem_table* finally denotes the status of each page frame and *chunk_table* contains no chunk.

5.2.2.3 The vm_init( ) routine

This routine is used to initialize virtual memory as described below. The prototype is

void vm_init()

It invokes rlmem_init() to cretae *rlmem_table* and to remove chunks from chunk table as explained above. *vm_init()* will now get a free page frame through routine rlmem_getpage( ) for creating page global directory.

rlmem_getpage() actually scans through rlmem_table to find a zero entry in the table. it then returns the index of that entry which is actually the free page frame. Physical address of the Page global directory will be stored in a variable *page_base.* After clearing this page frame with the help of an assembly routine *phys_clr_page(),* the routine will make entries in the page global directory for linear address range of 0 to *hi_mem* through a loop structure given below.

```
hi_addr= hi_mem << CLICK_SHIFT;
        for (dir_addr= 0; dir_addr<hi_addr; dir_addr += 4*1024*1024)
        {
                dir_pointer= rlmem_getpage();
                phys_clr_page(dir_pointer);          /* No pages */
                map_dir(dir_addr, dir_pointer);
        }
```

In each cycle of the above loop, it first allocate a page frame for a new page table through *rlmem_getpage()* routine. It clear this page frame through an assembly routine *phys_clr_page().* It updates the corresponding entry in the page global directory by invoking another routine *map_dir().* The first argument to *map_dir()* is the linear address and the second argument is the physical address. At the end of each cycle, loop index *dir_addr* is increased by a constant 4*1024*1024, which is the size of linear address interval spanned by a single page table.

Each entry in a page global directory or a page table takes 4 bytes. The first 20 bits contains the physical address of a page table or a page frame in terms of clicks and remaining bits are available for protection flags. These flag bits along with other constants and macros related to virtual addressing are shown below.

```
#define VM_PRESENT              1
#define VM_WRITE                2
#define VM_USER                 4
#define VM_INMEM                0x200
#define VM_INMEM_N_PRESENT      (VM_INMEM | VM_PRESENT)
#define VM_IM_RW_PRES           (VM_INMEM | VM_WRITE | VM_PRESENT)

#define VM_ADDRMASK             0xfffff000
#define VM_DIRMASK              0x003fffff
#define VM_PAGEMASK             0x00000fff
```

```
#define VM_PAGESIZE             0x1000
#define VM_DIRSIZE              0x400000
#define VM_PAGESHIFT            12              /* 2log VM_PAGESIZE */
#define VM_DIRSHIFT             22              /* 2log VM_DIRSIZE */

#define vm_addr_to_page(a)      ((a >> VM_PAGESHIFT) & 0x3ff)
#define vm_addr_to_dir(a)       ((a >> VM_DIRSHIFT) & 0x3ff)
```

The position of these flag bits in the 32-bit entry is shown in the Figure-5.2

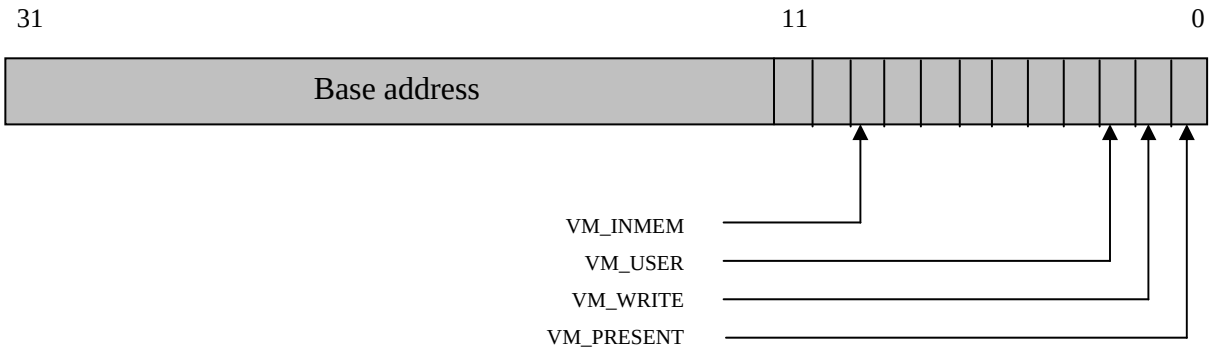


Figure 5.2: Format of page table entry

After that, the page tables created in the loop above, would be initialized in such a way that kernel can address the RAM having size *hi_mem* by linear address identical to physical ones. In other words, a page having linear address x will be mapped to a page frame having physical address x. This is done through the following loop.

```
for (page_addr= 0; page_addr<hi_addr; page_addr += CLICK_SIZE)
                    map_page(page_addr, page_addr);
```

In this loop, it updates the corresponding entry in the page table by invoking another routine *map_page().* The first argument to *map_page()* is the linear address and the second argument is the physical address. Here both the arguments have the same value for identical mapping as explained above. At the end of each cycle, loop index *page_addr* is increased by a constant CLICK_SIZE, which is the size of linear address interval spanned by a single page frame mapped in page table entry.

Then it calculates the base of linear address space to be used by servers and processes as follows.

```
virt_base= (hi_mem + 0x1000000) & ~0x3fffff;
paging_base= virt_base;
```

This base will be on 4 MB boundary and approximately 16*1024*1024 clicks above *hi_mem*. This base will be stored in a variable *paging_base*.

Now this linear address space starting from *paging_base* and having range of *VM_SIZE_CLICKS* (2 GB) is added to *chunk_table* so that servers and user processes can treat it as normal memory. In this stage this is the only chunk present in *chunk_table* since all the physical chunks had been removed already as discussed above.

5.2.2.4 The init_task( ) routine

This routine is used to initialize process table entry of the task. The prototype of this routine is given below.

*void init_task (int t, task_t *initial_pc,int stack_size, char *name, u32_t msw, reg_t *task_stackp)*

t denotes the task number defined in task table for well known tasks. *Initial_pc* is the task entry point, *stack_size* is the size of stack used by the task defined in task table, *name* is the name of task, msw is the machine status word (msw is actually control register cr0 on a 386 and above), and *task_stackp* is pointer to the task stack.

It puts the *STACK_GUARD* word at the starting of the stack pointed by *task_stackp* to protect it from overlapping by the stack of another task. *task_stackp* is incremented by stack_size so that now *task_stackp* can point to the stack of the next task. Then it sets the various fields of *stackframe,* which is used during context switching.

Then task text and data segments are set equal to that of kernel and are taken from first entry in *image_maps* table. Stack segment is set at the ending boundary of kernel's data segment

with zero length. Then it invokes a routine *alloc_segments()* to set the local descriptor table, defined as a field in each process table entry, for the code and data segments. After that it copies the name of task in the *name* field in the process table entry.

5.2.2.5 The vm_map_server( ) routine

This routine is used to re-map the given image from physical address space to the linear address space. The prototype is as follows.

phys_clicks vm_map_server(text_base, text_clicks, data_clicks, bss_clicks, heap_clicks)

it first calculates the total clicks for image by adding the *text_clicks, data_clicks, bss_clicks* and *heap_clicks*.

Then it allocates linear address space starting from *vm_base* which is equal to *paging_base* (initialized by vm_init() as explained above). To allocate this linear address space it first allocates page tables and updates corresponding entries in the page global directory through the following loop structure.

```
for (i= 0; i<tot_clicks; i+= (VM_DIRSIZE/CLICK_SIZE))
{
        dir_base= rlmem_getpage();
        phys_clr_page(dir_base);
        map_dir(vm_base+ (i << CLICK_SHIFT), dir_base);
}
```

The entries in the page tables, created in the above loop, are now made in such a way that these tables map the allocated linear address space onto the physical address space of already loaded image as shown below.

```
for (i= 0; i<text_clicks+data_clicks; i++)
{
        map_page(virt_base, text_base << CLICK_SHIFT);
        text_base++;
        virt_base += VM_PAGESIZE;
}
for (i= 0; i<bss_clicks; i++)
{
        bss_base= rlmem_getpage();
        phys_clr_page(bss_base);
```

```
            map_page(virt_base, bss_base);
            virt_base += VM_PAGESIZE;
}
virt_base += heap_clicks << CLICK_SHIFT;
```

map_dir( ) and *map_page( )* are already explained in section 5.3.3. In the second loop structure above, page frame is allocated for bss segment in each cycle as the loaded image contains only the text and data segments. The physical address of this page frame is then used in mapping by *map_page( ).*At the end, *Virt_base* pointer is shifted ahead by *heap_size* to accommodate heap in the linear address space allocation.

The *virt_base* pointer is then shifted by a value 4*1024*1024 to maintain gap between linear address spaces of successive processes. This chunk of allocated linear address space including gap is deleted from the *chunk_table*. The routine then returns the base of allocated linear address space in terms of clicks to the calling routine.

5.2.2.6 The main( ) routine

The routine *main( )* initializes the system and starts the ball rolling by setting up the *proc* table, interrupt vectors, and scheduling each task to run to initialize itself. It executes only once during system startup. The prototype is

void main( )

It first initializes the interrupt controller by invoking a routine *intr_init( ) like standard Minix.* It then initializes the console through a routine *scr_init( )* to be able to generate console output now. It then initialize memory table *chunk_table* with physical memory chunks by invoking a routine *mem_init( ).*

After that it calculates where the different parts of the Minix image are loaded by invoking *calc_maps( )* routine. The results are stored in *image_maps[ ]*. *calc_maps( )* also calculates which part of the memory should not be used for user processes. After *calc_map( ), main( )*

invokes *vm_init( )* routine to allocate all remaining physical memory for virtual memory and put the new (linear) address space in the *chunk_table*.

It then clears the process table and set up mappings for *proc_addr( )* and *proc_number( )* macros using a loop structure given below.

```
for (rp = BEG_PROC_ADDR, t = -NR_TASKS; rp < END_PROC_ADDR; ++rp, ++t) {
        rp->p_flags = P_SLOT_FREE;
        rp->p_status = 0;
        rp->p_nr = t;
        (pproc_addr + NR_TASKS)[t] = rp;          /* proc ptr from number */
        rp->p_name[0]= 0;
        rp->p_flags |= NO_MAP;
 }
```

After that, it set up *proc* table entries for tasks. The stacks of the kernel tasks are initialized to an array in data space. To initialize *proc* table entry for tasks it invokes the routine *init_task()*.

It then initializes the servers and user process *init* using *image_maps* array. The bss and stack areas of the servers will be allocated in virtual memory. The kernel is in low memory, the rest if loaded is in extended memory. To re-map the image of servers and process *init* from physical address space to linear address space (virtual memory) it uses the routine *vm_map_server()*. This remapped image is now used to make entries in the *proc* table.

It then calls *load_pick_proc()* to choose the next process to be scheduled. After that, it invokes *restart()* routine to call the scheduler for context switching.

5.3 Additional Enhancement

Standard Minix supports RAM up to the size of 16 MB. But as memory is allocated to user processes now in terms of 4M chunks and no swapping of pages is provided, this limits the maximum number of processes in the system due to less RAM available. It will be better if size of physical memory available to Minix can be increased.

To increase the size of RAM supported, we changed the memory detection code in the boot monitor program. In standard Minix, it can detect up to 64MB memory using 88h

memory detection scheme. Now, it uses the combination of `e801h` and `88h` memory detection schemes to support full RAM available.

First it uses `e801h` detection scheme which returns a 32-bit memory size. If this method fails, then it go for the old method `88h`, which returns 0-64MB memory. The two methods are given below.

e801h method

```
big_ext:
        mov     ax, 0xE801      ! Code for get memory size for >64M
        int     0x15            ! ax = mem < 16M per 1K, bx = mem > 16M per 64K
        jc      small_ext
        cmp     ax, 15*1024     ! No hole below 16M please, we want 15M
        jb      small_ext
        xchg    ax, bx          ! Now bx = mem < 16M, and ax = mem > 16M
        mov     dx, 64
        mul     dx              ! dx:ax = ax * 64, because ax was in 64K units
        add     ax, bx          ! Add mem < 16M (in 1K units)
        adc     dx, 0           ! dx:ax = extended memory in K
        jmp     got_ext
```

e88h method

```
small_ext:
        xor     dx, dx
        movb    ah, 0x88 ! Code for get extended memory size
        clc                     ! Carry will stay clear if call exists
        int     0x15            ! Returns size (in K) in ax for AT's
        jnc     got_ext

no_ext:
        xor     ax, ax          ! Error, no extended memory
        xor     dx, dx
got_ext:
        ret
```

Also standard Minix kernel supports up to 16M RAM because no more than 16M can be addressed in 286 mode as shown below. So, *mem_init()* routine of standard Minix is slightly modified to support full RAM available, which is passed by boot monitor program to kernel during system initialization.

```
/*Note that no more than 16M can be addressed in 286 mode, so make
sure that the highest memory address fits in a short when counted
in clicks.*/
```

```
ext_clicks = k_to_click((u32_t) ext_memsize);
max_clicks = USHRT_MAX - (EM_BASE >> CLICK_SHIFT);
mem[1].size = MIN(ext_clicks, max_clicks);
mem[1].base = EM_BASE >> CLICK_SHIFT;
```

Now to support full RAM size, *ext_memsize* is not fitted in a short but taken as it is directly as shown below.

```
mem[1].size =  k_to_click((u32_t) ext_memsize);
mem[1].base = EM_BASE >> CLICK_SHIFT;
```

CHAPTER 6 CONCLUSIONS AND FUTURE WORK

6.1 Conclusions

In this dissertation a design is proposed for virtual addressing with non-paging implementation for Minix operating system. During the course of this project, emphasis is given on the kernel initialization phase for achieving this enhancement in the Minix operating system. System initialization phase of standard Minix has been studied thoroughly and this design is proposed to support virtual addressing without affecting the primary design of standard Minix.

This project results in a Minix operating system design in which servers and user processes use the linear address space instead of physical address space as before. The physical memory allocated to these processes in this implementation of the proposed design will always be in multiples of click (page) size as already explained.

The design, as proposed in this dissertation, is also successfully implemented, installed and tested on the target machines in a ready to use stage. This complete implementation consists of several changes in the standard Minix distribution, presenting all of which on this dissertation will diffuse the focus from the primary changes which are the building blocks of this proposed design.

To keep the focus of this dissertation as much on the primary problem taken up in this project some of these changes which are not directly related with the key goals in the design for virtual addressing with non-paging implementation for Minix operating system are not included in this dissertation. Instead the focus of this dissertation is to present these primary changes in an elaborate manner.

This design is proposed keeping Intel based 80x86 architecture, starting from 80386 onwards as target architecture. The target machines on which this design is implemented and tested are Pentium II and Pentium III based machines. The primary target machine is an Intel Pentium II

based system with 64MB of physical memory available. The design is also tested for a multiple boot operation with a standard Windows 98 distribution and a Red Hat Linux 9.0 distribution on this target machine.

The standard Minix distribution supports only upto 16 MB of the physical memory. This design also proposes to make necessary changes to support the total available physical memory of the system. This aspect of this design has also been tested on target machines and in all the cases the proposed system successfully managed to utilize the total available physical memory of the system. The availability of more physical memory results in the increase in the maximum number of processes supported simultaneously by Minix. Also, large size applications could now be executed on Minix with some changes in Minix file system to support large file size.

6.2 Future work

This design has been proposed keeping Intel based 80x86 architecture in view. In future, this design could be further extended to other architectures. The basic framework needed to extend this design to other architectures would be same, but some small changes would be needed to incorporate the idiosyncrasies of the particular target architecture in question.

Also, this design created necessary outlines for extending it to support virtual addressing with paging. For paging implementation, a pager would be needed, which could swap pages between physical memory and swap area. To support swap partition, Minix file system would also have to be changed. A page replacement policy will need to be required for the implementation of pager. Also, the design issues like working set model, thrashing and global versus local allocation policies are needed to be considered for a paging system. After these changes, Minix will support complete virtual memory management, i.e. virtual addressing with paging.

REFERENCES

[1] Andrew S. Tanenbaum, Albert S. Woodhull: Operating Systems Design and Implementation, Second Edition, Pearson Education, 2004.

[2] Daniel P. Bovet, Marco Cesati: Understanding the Linux Kernel, Second Edition, O'Reilly, 2004.

[3] Maurice J. Bach: The Design of the Unix Operating System, PHI, 1986.

[4] Andrew S. Tanenbaum, Marteen V. Steen: Distributed Systems Principles and Paradigm, Pearson Education.

[5] Richard Stones, Neil Matthew: Beginning Linux Programming, WROX Publishers.

[6] James S. Antonakos: The Pentium Microprocessor, Pearson Education.

[7] Barry B. Brey: Programming the 80286, 80386, 80486, and Pentium-Based Personal Computer, PHI.

[8] Douglas V. Hall: Microprocessors And Interfacing Programming And Hardware, Second Edition, TATA McGRAW HILL.

[9] Abraham Silberschatz, Peter Baer Galvin: Operating System Concepts, Fifth Edition, Addison-Wesley, 1999.

[10] http://www.online.ee/~andre/i80386/: Intel 80386 Programmer's Reference 1986.

[11] http://www.cs.vu.nl/minix/ : Minix home Page.

[12] B. W. Kernighan, D. M. Ritchie: The C Programming Language, Second edition.

[13] http://www.linuxdoc.org/guides.html : Linux Kernel Internals.

[14] http://linuxassembly.org : Linux Assembly HOWTO.

[15] http://world.std.com/~bochs: home page of Bochs, an 80386 emulator.

[16] http://linuxfromscratch.org/ : home page of LFS project.

[17] http://minixfromscratch.org/ : home page of MFS project.

[18] P.J. Denning: “Virtual Memory”, Computing Surveys, Vol. 2, Sept. 1970.

[19] D. Lewine: POSIX Programmer’s Guide, O’Reilly & Associates, 1991.

[20] IEEE: Information Technology- Portable Operating System Interface (POSIX), part 1: System Application Program Interface (API) [C Language], New York: Institute of Electrical & Electronics Engineers, Inc., 1990.

[21] K. Thompson: “Unix Implementation”, Bell System Technical Journal, Vol. 57, July-Aug. 1978.

[22] W.R. Stevens: Advanced Programming in the Unix Environment, Addison-Wesley, 1992.

[23] U. Vahalia: Unix Internals-The New Frontiers, Prentice Hall, 1996.

[24] W. Stallings: Operating Systems, Second Edition, Prentice Hall, 1995.

[25] K. Li & P. Hudak: “Memory Coherence in Shared Virtual Memory Systems”, ACM Transactions on Computer Systems, Vol. 7, Nov. 1989.

[26] E. Abrossimov, M. Rozier, M. Shapiro: ”Generic virtual memory management for operating system kernels”, Proceedings of the twelfth ACM symposium on Operating systems principles, 1989.

[27] Hermann Hartig , Michael Hohmuth , Jochen Liedtke , Sebastian Schönberg: ”The performance of µ-kernel-based systems”, ACM SIGOPS Operating Systems Review, v.31 n.5, Dec. 1997.

[28] R. Rashid, A. Tevanian,Jr., M. Young, D. Golub, R. Baron, D. Black, .J. Bolosky, J. Chew : “Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures”, IEEE Transactions on Computers, Vol_ 37, No_ 8, August 1988.

[29] R. Fitzgerald and R. F. Rashid: "The integration of virtual memory management and interprocess communication in Accent," ACM Transactions on Computer Systems., vol. 4, May 1986.

[30] Corbato, F. J.: “A Paging Experiment with the Multics System”, Project MAC, MAC-M-384, July 1968.

[31] G. Oppenheimer, N. Weizer: “Resource management for a medium scale time-sharing operating system”, Communications of the ACM, v.11 n.5, May 1968.

[32] Andrew W. Appel and Kai Li: “Virtual Memory Primitives for User Programs”, CS-TR-276-90, Department of Computer Science, Princeton University.

[33] Philip Koopman, John DeVale: "The Exception Handling Effectiveness of POSIX Operating Systems", IEEE Transactions on Software Engineering, Volume 26, September 2000.

[34] J. Goodenough: "Exception Handling: Issues and a Proposed Notation", Communications of the ACM, vol. 18, Dec. 1975.

[35] TE Anderson, HM Levy, BN Bershad, ED Lazowska: “The Interaction of Architecture and Operating System Design” - ASPLOS, 1991 - portal.acm.org.

[36] Paul R. Wilson and Sheetal V. Kakkad: “Pointer Swizzling at Page Fault Time: Efficiently and Compatibly Supporting Huge Address Spaces on Standard Hardware”, Proceedings of 1992 Intel Workshop on Object Orientation in Operating Systems, Paris, France, Sept. 24-25, 1992, (IEEE Press).

[37] RW Carr, JL Hennessy: “WSClock - A Simple and Effective Algorithm for Virtual Memory Management”- SOSP, 1981 - portal.acm.org.

SOURCE CODE
OF
SELECTED FILES

++

vm386.h

++

```
#ifndef VM386_H
#define VM386_H

#define VM_PRESENT          1
#define VM_WRITE  2
#define VM_USER             4
#define VM_INMEM 0x200
#define VM_INMEM_N_PRESENT          (VM_INMEM | VM_PRESENT)
#define VM_IM_RW_PRES       (VM_INMEM | VM_WRITE | VM_PRESENT)

#define VM_ADDRMASK         0xfffff000
#define VM_DIRMASK          0x003fffff
#define VM_PAGEMASK         0x00000fff

#define VM_PAGESIZE         0x1000
#define VM_DIRSIZE          0x400000
#define VM_PAGESHIFT        12          /* 2log VM_PAGESIZE */
#define VM_DIRSHIFT         22          /* 2log VM_DIRSIZE */

#define vm_addr_to_page(a)  ((a >> VM_PAGESHIFT) & 0x3ff)
#define vm_addr_to_dir(a)   ((a >> VM_DIRSHIFT) & 0x3ff)

#endif /* VM386_H */
```

++

main.c

++

```
/* This file contains the main program of MINIX.  The routine main()
 * initializes the system and starts the ball rolling by setting up the proc
 * table, interrupt vectors, and scheduling each task to run to initialize
 * itself.
 *
 * The entries into this file are:
 *   main:                  MINIX main program
 *   panic:                 abort MINIX due to a fatal error
 */

#include "kernel.h"
#include <a.out.h>
#include <signal.h>
#include <unistd.h>
#include <minix/callnr.h>
#include <minix/com.h>
#include <minix/minlib.h>
#include "assert.h"
INIT_ASSERT
#include "mq.h"
#include "proc.h"
#include "protect.h"
#include "../version/version.h"    /* Tells the O.S. release and version. */

#define MAP_NR                16        /* Assume no more than MAP_NR different
                                         * executables in an image.
                                         */
#define IMAGE_HDRS            0x600L    /* The struct exec header of the parts of the
                                         * image are stored at absolute address
                                         * IMAGE_HDRS.
                                         */

struct image_map
{
        struct mem_map im_map[NR_SEGS];
        phys_clicks im_bss_clicks;
        phys_clicks im_stack_clicks;
        phys_bytes im_initial_pc;
        int im_valid;
};

struct image_map image_maps[MAP_NR];

FORWARD _PROTOTYPE( void calc_maps, (void) );
FORWARD _PROTOTYPE( void remap_image, (struct proc *pp, int index) );
FORWARD _PROTOTYPE( void create_env, (struct proc *pp)                    );
#if DEBUG
FORWARD _PROTOTYPE( void k_printenv, (void) );
#endif
FORWARD _PROTOTYPE( void init_task, (int t, task_t *initial_pc,
        int stack_size, char *name, u32_t msw, reg_t *task_stackp)        );


/*===========================================================================*
 *                              main                                         *
 *===========================================================================*/
PUBLIC void main()
{
/* Start the ball rolling. */

  register struct proc *rp;
  register int t;
```

```
int imageindex;
reg_t ktsb;                             /* kernel task stack base */
reg_t msw;
struct tasktab *ttp;

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

scr_init(0);                    /* We initialize the console before the tty driver
                                 * to be able to generate console output now.
                                 */

/* Display the Minix startup banner. */
printf("Minix %s.%s  Copyright 1997 Prentice-Hall, Inc.\n\n",
                                                        OS_RELEASE, OS_VERSION);


mem_init();                     /* Make list of available memory chunks */

/* Calculate where the different parts of the image are loaded. The results
 * are stored in image_maps[]. Later this information can be remapped in
 * a virtual address space, before it is stored in the process structure.
 * calc_maps also calculates which part of the memory should not be used for
 * user processes.
 */
calc_maps();

vm_init();              /* Allocate all remaining physical memory for virtual memory
                         * and put the new address space in the table */
assert(vm_hard_check_stats() ? 1: (vm_dump(), 0));


/* Clear the process table.
 * Set up mappings for proc_addr() and proc_number() macros.
 */
for (rp = BEG_PROC_ADDR, t = -NR_TASKS; rp < END_PROC_ADDR; ++rp, ++t) {
        rp->p_flags = P_SLOT_FREE;
        rp->p_status = 0;
        rp->p_nr = t;                   /* proc number from ptr */
        (pproc_addr + NR_TASKS)[t] = rp;        /* proc ptr from number */
        rp->p_name[0]= 0;
        rp->p_flags |= NO_MAP;
}

/* Set up proc table entries for tasks and servers.  The stacks of the
 * kernel tasks are initialized to an array in data space.  The bss and
 * stack areas of the servers will be allocated in virtual memory.  The
 * kernel is in low memory, the rest if loaded in extended memory.
 */

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

/* Setup a dummy proc_ptr. */
proc_ptr= cproc_addr(IDLE);
proc_ptr->p_priority= PPRI_IDLE;

/* Fetch the msw. This value of msw will be stored in process structure and
 * reloaded during a context switch, so make sure that all changes to msw
 * after this point are done on the msw field in the stackframe
 * (pp->p_reg.msw). Note, msw is actually cr0 on a 386 and above.
 */
msw= get_msw();

/* Initialize the tasks. */
for (ttp= tasktab; ttp->tt_name != NULL; ttp++)
{
        t= ttp->tt_proc_no;     /* fixed process slot?  (well known task) */
        if (t == 0)
        {
```

```
                        for (t= -NR_TASKS; t < 0; t++)
                        {
                                if (proc_addr(t)->p_flags & P_SLOT_FREE)
                                        break;
                        }
                        if (t >= 0)
                        {
                                printf("no space for task '%s'\n", ttp->tt_name);
                                continue;
                        }
                }
                assert(proc_addr(t)->p_flags & P_SLOT_FREE);

                init_task(t, ttp->tt_initial_pc, ttp->tt_stksize, ttp->tt_name,
                        msw, &ktsb);
        }

        /* Initialize the servers and processes 1 (init), 2 (inet?), ... */
        for (imageindex= 1; imageindex < MAP_NR && image_maps[imageindex].im_valid;
                imageindex++)
        {
                t= imageindex - 1;
                rp= proc_addr(t);
                assert(isokprocn(t) && rp->p_flags & P_SLOT_FREE);

                rp->p_reg.sf_psw = INIT_PSW;
                rp->p_reg.sf_msw = msw;
                if (processor >= 486) {
                        /* Enable the 486 alignment check. */
                        rp->p_reg.sf_msw |= CR0_AM;
                }
                rp->p_flags = 0;
                rp->p_priority = t < LOW_USER ? PPRI_SERVER : PPRI_USER;

                /* Remap this part of the image into a virtual
                 * address space.
                 */
                remap_image(rp, imageindex);

                /* Let's setup the segment map */
                rp->p_map[SEG_T]= image_maps[imageindex].im_map[SEG_T];
                rp->p_map[SEG_D]= image_maps[imageindex].im_map[SEG_D];
                rp->p_map[SEG_S]= image_maps[imageindex].im_map[SEG_S];

                rp->p_reg.sf_pc = (reg_t) image_maps[imageindex].im_initial_pc;
                rp->p_reg.sf_sp = (rp->p_map[SEG_S].mem_vir +
                                                rp->p_map[SEG_S].mem_len) << CLICK_SHIFT;
                rp->p_reg.sf_sp -= sizeof(reg_t);

                /* Set inital schedule quantum for init, etc. */
                rp->p_schedquant = DEF_SCHEDQUANT;

                /* Set the name of MM, and FS.  Set the rest to #1, #2, ... */
                switch (t) {
                case MM_PROC_NR:
                        strcpy(rp->p_name, "MM");
                        break;
                case FS_PROC_NR:
                        strcpy(rp->p_name, "FS");
                        break;
                default:
                        rp->p_name[0]= '#';
                        strcpy(rp->p_name+1, itoa((t - INIT_PROC_NR) + 1));
                }
                create_env(rp);

                lock_ready(rp);

                alloc_segments(rp);
        }
```

```
  assert(vm_hard_check_stats());

  lock_pick_proc();

  /* Unmap page zero. */
  vm_map_zp(FALSE);

  /* Now go to the assembly code to start running the current process. */
  restart();
}


/*===========================================================================*
 *                                init_task                                  *
 *===========================================================================*/
PRIVATE void init_task(t, initial_pc, stack_size, name, msw, task_stackp)
int t;
task_t *initial_pc;
int stack_size;
char *name;
u32_t msw;
reg_t *task_stackp;
{
        struct proc *rp;

        rp= proc_addr(t);

        assert(rp == cproc_addr(HARDWARE) || stack_size >= 4096);
        if (stack_size > 0) {
                rp->p_stguard = (reg_t *) *task_stackp;
                *rp->p_stguard = STACK_GUARD;
        }
        *task_stackp += stack_size;
        rp->p_reg.sf_sp = *task_stackp;

        rp->p_reg.sf_pc = (reg_t) initial_pc;
        rp->p_reg.sf_psw = INIT_TASK_PSW;
        rp->p_reg.sf_msw = msw;
        rp->p_flags = 0;

        if (rp != cproc_addr(IDLE))
                rp->p_priority= PPRI_TASK;
        rp->p_status |= P_ST_VM_INCORE;

        /* Setup the kernel segment map */
        if (!image_maps[0].im_valid)
                panic("invalid map for task ", t);

        /* Text and data maps are already setup */
        rp->p_map[SEG_T]= image_maps[0].im_map[SEG_T];
        rp->p_map[SEG_D]= image_maps[0].im_map[SEG_D];

        /* We don't have a stack, the stack is in the data segment.
         * we also don't have to map any bss since that is part of the
         * image.
         */
        assert(image_maps[0].im_bss_clicks == 0);
        assert(image_maps[0].im_stack_clicks == 0);

        rp->p_map[SEG_S]= rp->p_map[SEG_D];
        rp->p_map[SEG_S].mem_vir += rp->p_map[SEG_S].mem_len;
        rp->p_map[SEG_S].mem_len= 0;

#if VMDEXT_KERNEL_LINMAP
        if (rp->p_map[SEG_D].mem_phys == 0)
        {
                /* We got a linearly mapped kernel data segment. */
                rp->p_map[SEG_S].mem_vir= 0xFFFFFFFF >> CLICK_SHIFT;
        }
```

```
#endif /* VMDEXT_KERNEL_LINMAP */

	/* IDLE, HARDWARE neveready */
	if (!isidlehardware(t)) lock_ready(rp);

	alloc_segments(rp);
	strcpy(rp->p_name, name);
}


/*===========================================================================*
 *                                panic                                      *
 *===========================================================================*/
PUBLIC void panic(s,n)
_CONST char *s;
int n;
{
/* The system has run aground of a fatal error.  Terminate execution.
 * If the panic originated in MM or FS, the string will be empty and the
 * file system already synched.  If the panic originates in the kernel, we are
 * kind of stuck.
 */
 stacktrace(0);
 if (s != NULL && *s != 0) {
	printf("\nKernel panic: %s",s);
	if (n != NO_NUM) printf(" %d", n);
	printf("\n");
 }
 wreboot(RBT_PANIC);
}

#if DEBUG
PRIVATE void k_printenv()
{
 register char *namep;
 register char *envp;
 extern char k_environ[];

 for (envp = k_environ; *envp != 0;) {
	printf("'%s'\n", envp);
	while (*envp++ != 0)
		;
 }
}
#endif /* DEBUG */


/*===========================================================================*
 *                                calc_maps                                  *
 *===========================================================================*/
PRIVATE void calc_maps()
{
/* Calculate the maps for different kernel parts. */
	int i;
	phys_clicks image_base_clicks, text_base_clicks, data_base_clicks;
	struct exec e_hdr;
	phys_clicks text_clicks, data_clicks, bss_clicks, stack_clicks;
	phys_bytes a_text, a_data, a_bss, a_total, a_entry, a_stack, a_tmp;
	int a_flags;
	struct mem_map *mapp;

	for (i= 0; i<MAP_NR; i++)
		image_maps[i].im_valid= FALSE;

	for (i= 0; i<MAP_NR; i++)
	{
		/* Image headers are placed at IMAGE_HDRS, (normally 0x600)
		 * by the bootstrap loader.
		 */
		phys_copy(IMAGE_HDRS + i * A_MINHDR, vir2phys(&e_hdr),
```

```
                                                (phys_bytes) A_MINHDR);
if (BADMAG(e_hdr))
        break;                          /* We are through. */
a_flags= e_hdr.a_flags;
/* We ignore the cpu field. */
a_text= e_hdr.a_text;
a_data= e_hdr.a_data;
a_bss= e_hdr.a_bss;
a_entry= e_hdr.a_entry;
a_total= e_hdr.a_total;
/* a_syms is ignored! */

/* Where does the text start?  The first (kernel) segment
 * starts at kernel_code_base, this includes UZP. Later parts
 * start at image_base.
 */
if (i == 0)
{
        assert((kernel_code_base & (CLICK_SIZE-1)) == 0);
        image_base_clicks= kernel_code_base >> CLICK_SHIFT;
        if (a_flags & A_UZP)
        {
                /* kernel_code_base is the start of the segment,
                 * we are looking for the start of the actual
                 * image.
                 */
                image_base_clicks++;
        }
}

a_stack = a_total - a_data - a_bss;
if (!(a_flags & A_SEP))
{
        a_stack -= a_text;
}

/* A page aligned program has a header at the front. */
if (a_flags & A_PAL)
{
        a_text += e_hdr.a_hdrlen;
}

/* Treat a common I&D program as having no text. */
if (!(a_flags & A_SEP))
{
        a_data += a_text;
        a_text= 0;
}

/* Let's fixup a_text, a_data, ... by rounding them to
 * click boundaries.  Subtract what is added to a_data
 * from a_bss, etc.
 */
a_text= (a_text + CLICK_SIZE-1) & ~(CLICK_SIZE-1);
a_tmp= a_data;
a_data= (a_data + CLICK_SIZE-1) & ~(CLICK_SIZE-1);
a_bss -= (a_data - a_tmp);
a_tmp= a_bss;
a_bss= (a_bss + CLICK_SIZE-1) & ~(CLICK_SIZE-1);
a_stack -= (a_bss - a_tmp);
a_stack= (a_stack + CLICK_SIZE-1) & ~(CLICK_SIZE-1);

/* Now that we got some reasonable sizes, we can allocate
 * some memory to this part of the image.
 */
text_clicks= (a_text >> CLICK_SHIFT);
data_clicks= (a_data >> CLICK_SHIFT);
bss_clicks= (a_bss >> CLICK_SHIFT);
stack_clicks= (a_stack >> CLICK_SHIFT);
```

```
                        /* Maybe we should check if the total number of clicks does
                         * not overflow.
                         */
                        text_base_clicks= image_base_clicks;
                        data_base_clicks= text_base_clicks + text_clicks;

                        mapp= image_maps[i].im_map;
                        mapp[SEG_T].mem_phys= text_base_clicks;
                        mapp[SEG_T].mem_vir= 0;
                        mapp[SEG_T].mem_len= text_clicks;
                        mapp[SEG_D].mem_phys= data_base_clicks;
                        mapp[SEG_D].mem_vir= 0;
                        mapp[SEG_D].mem_len= data_clicks;

                        if (i == 0)
                        {
                                /* The kernel bss and stack are not virtual, but are
                                 * created by the bootstrap loader.  Add them to the
                                 * data segment.
                                 */
                                mapp[SEG_D].mem_len += bss_clicks + stack_clicks;
                                bss_clicks= stack_clicks= 0;
                        }
                        image_maps[i].im_bss_clicks= bss_clicks;
                        image_maps[i].im_stack_clicks= stack_clicks;
                        image_base_clicks= mapp[SEG_D].mem_phys + mapp[SEG_D].mem_vir +
                                mapp[SEG_D].mem_len;

                        /* Delete the memory for the process from the list of memory
                         * chunks.  Also remove memory in front of the kernel.
                         */
                        if (i == 0) chunk_del(0, text_base_clicks);
                        chunk_del(text_base_clicks, image_base_clicks-text_base_clicks);

                        /* Adjust for UZP. */
                        if (a_flags & A_UZP)
                        {
                                /* Assume CLICK_SIZE == page size */
                                mapp[SEG_T].mem_phys--;
                                mapp[SEG_T].mem_vir++;
                                mapp[SEG_D].mem_phys--;
                                mapp[SEG_D].mem_vir++;
                        }

                        image_maps[i].im_initial_pc= a_entry;
                        image_maps[i].im_valid= TRUE;

#if DEBUG
                        printf(
"map[%d]: (p,v,l) (0x%x, 0x%x, 0x%x) (0x%x, 0x%x, 0x%x) b=0x%x, s=0x%x\n",
                                i,
                                mapp[SEG_T].mem_phys, mapp[SEG_T].mem_vir,
                                                                mapp[SEG_T].mem_len,
                                mapp[SEG_D].mem_phys, mapp[SEG_D].mem_vir,
                                                                mapp[SEG_D].mem_len,
                                image_maps[i].im_bss_clicks,
                                image_maps[i].im_stack_clicks);
#endif

#if WORD_SIZE == 4
                        if (i == 0) {
                                /* MM, FS, etc. are loaded in extended memory. */
                                image_base_clicks= 0x100000 >> CLICK_SHIFT;
                        }
#endif
                }
                if (i == MAP_NR)
                        printf("calc_maps: warning to many image parts (>%d)\n", i-1);
}
```

```
/*============================================================================*
 *                                    remap_image                             *
 *============================================================================*/
PRIVATE void remap_image(pp, index)
struct proc *pp;
int index;
{
        phys_clicks t_phys, t_vir, t_len, d_phys, d_vir, d_len;
        phys_clicks bss_clicks, stack_clicks, vm_base_clicks;
        int sep, uzp;

        t_phys= image_maps[index].im_map[SEG_T].mem_phys;
        t_vir= image_maps[index].im_map[SEG_T].mem_vir;
        t_len= image_maps[index].im_map[SEG_T].mem_len;
        d_phys= image_maps[index].im_map[SEG_D].mem_phys;
        d_vir= image_maps[index].im_map[SEG_D].mem_vir;
        d_len= image_maps[index].im_map[SEG_D].mem_len;

        bss_clicks= image_maps[index].im_bss_clicks;
        stack_clicks= image_maps[index].im_stack_clicks;


        if (stack_clicks < 512)
                stack_clicks= 512;

        sep= (t_phys != d_phys);
        assert(!sep || d_phys == t_phys + t_len);       /* ! */
        uzp= (t_vir != 0);
        assert(!uzp || (t_vir == 1 && d_vir == 1));

        vm_base_clicks=vm_map_server(t_phys, (t_vir + t_len, d_vir + d_len – bss_clicks – stack_clicks,     bss_clicks, stack_clicks)

        /* Now fill in the text segment map.
         */

        image_maps[index].im_map[SEG_T].mem_phys= vm_base_clicks;
        image_maps[index].im_map[SEG_T].mem_vir= t_vir;
        image_maps[index].im_map[SEG_T].mem_len= t_len;

        /* Next fill in the map of the data segment. */
        if (sep)
        {
                vm_base_clicks= (image_maps[index].im_map[SEG_T].mem_phys +
                        image_maps[index].im_map[SEG_T].mem_vir +
                        image_maps[index].im_map[SEG_T].mem_len) << CLICK_SHIFT;
        }

        image_maps[index].im_map[SEG_D].mem_phys= vm_base_clicks ;
        image_maps[index].im_map[SEG_D].mem_vir= d_vir;
        image_maps[index].im_map[SEG_D].mem_len= d_len;

        /* And we have preallocate the bss. */
        vm_base_clicks=(image_maps[index].im_map[SEG_D].mem_phys + image_maps[index].im_map[SEG_D].mem_vir +
                                                                image_maps[index].im_map[SEG_D].mem_len;

        image_maps[index].im_map[SEG_S].mem_phys= image_maps[index].im_map[SEG_D].mem_phys;

        image_maps[index].im_map[SEG_S].mem_vir= image_maps[index].im_map[SEG_D].mem_vir +
                image_maps[index].im_map[SEG_D].mem_len + stack_clicks;

        image_maps[index].im_map[SEG_S].mem_len= 0;

}


/*============================================================================*
 *                                    create_env                              *
 *============================================================================*/
PRIVATE void create_env(pp)
```

```
struct proc *pp;
{
	vir_bytes envs, envv;
	vir_bytes args, argv;
	int len, argc, envc, args_size, envs_size;
	char *p;
	char *name;
	char *ptr;
	vir_bytes sp;
	phys_bytes phys_env;
	u32_t base, top;
	vir_clicks clicks;

	/* Program name. */
	name= pp->p_name;

	/* First, calculate the position of the argument and environment
	 * vectors.
	 */
	len= strlen(name)+1;		/* Program name */
	args_size= len;
	argc= 1;

	envs_size= 0;
	envc= 0;
	p= k_environ;
	while (p[0] != '\0')
	{
		len= strlen(p)+1;
		envs_size += len;
		envc++;
		p += len;
	}

	/* Make space for strings, vectors and argument count. */
	sp= pp->p_reg.sf_sp;
	sp -= envs_size + args_size;
	sp -= (envc + 1 + argc + 1) * sizeof(char *);
	sp -= sizeof(char *);
	sp &= ~(sizeof(char *) -1);

	/* Map stack pages */
	top= (pp->p_map[SEG_S].mem_phys + pp->p_map[SEG_S].mem_vir +
					pp->p_map[SEG_S].mem_len) << CLICK_SHIFT;
	base= top - (pp->p_reg.sf_sp - sp);
	base &= ~(CLICK_SIZE-1);
	clicks= (top-base) >> CLICK_SHIFT;
	pp->p_map[SEG_S].mem_vir -= clicks;
	pp->p_map[SEG_S].mem_len += clicks;
	vm_make_clear(pp, base, top-base);

	/* Addresses of vectors and strings. */
	argv= sp + sizeof(char *);
	envv= argv + (argc+1) * sizeof(char *);
	args= envv + (envc+1) * sizeof(char *);
	envs= args + args_size;

	/* Set argc. */
	ptr= (char *) argc;
	phys_env= umap(pp, SEG_D, sp, sizeof(char *));
	assert(phys_env != 0);
	phys_copy(vir2phys(&ptr), phys_env, sizeof(char *));

	/* Copy the arguments (only the program name at the moment). */
	ptr= (char *) args;
	phys_env= umap(pp, SEG_D, argv, sizeof(char *));
	assert(phys_env != 0);
	phys_copy(vir2phys(&ptr), phys_env, sizeof(char *));
	argv += sizeof(char *);
```

```
	len= strlen(name)+1;
	phys_env= umap(pp, SEG_D, args, sizeof(char *));
	assert(phys_env != 0);
	phys_copy(vir2phys(name), phys_env, len);
	args += len;

	/* Put the terminating NULL */
	ptr= NULL;
	phys_env= umap(pp, SEG_D, argv, sizeof(char *));
	assert(phys_env != 0);
	phys_copy(vir2phys(&ptr), phys_env, sizeof(char *));
	argv += sizeof(char *);

	assert(args == envs);
	assert(argv == envv);

	/* Copy the environment */
	p= k_environ;
	while (p[0] != '\0')
	{
		len= strlen(p)+1;

		/* Set envv[n]. */
		ptr= (char *) envs;
		phys_env= umap(pp, SEG_D, envv, sizeof(char *));
		assert(phys_env != 0);
		phys_copy(vir2phys(&ptr), phys_env, sizeof(char *));
		envv += sizeof(char *);

		/* Copy the string */
		phys_env= umap(pp, SEG_D, envs, len);
		assert(phys_env != 0);
		phys_copy(vir2phys(p), phys_env, len);
		envs += len;

		p += len;
	}

	/* Add a terminating NULL for the environment pointer array */
	ptr= NULL;
	phys_env= umap(pp, SEG_D, envv, sizeof(char *));
	assert(phys_env != 0);
	phys_copy(vir2phys(&ptr), phys_env, sizeof(char *));
	envv += sizeof(char *);

	assert(envv == args - args_size);
	assert((pp->p_reg.sf_sp - envs) < sizeof(char *));

	/* Commit the new stack. */
	pp->p_reg.sf_sp= sp;

	assert(vm_hard_check_stats());
}
```

++

vm386.c

Virtual memory routines for the 386

++

```
#include "kernel.h"
#include <signal.h>
#include <minix/com.h>
#include "assert.h"
#include "glo.h"
#include "proc.h"
#include "vm386.h"

PRIVATE phys_bytes rlmem_table_base, rlmem_table_size;
PRIVATE phys_clicks free_mem, hi_mem;
/* PRIVATE */ phys_bytes page_base;
PRIVATE phys_bytes virt_base, paging_base;

FORWARD _PROTOTYPE( void rlmem_init, (void)                             );
FORWARD _PROTOTYPE( int rlmem_free, (phys_bytes page_addr)              );
FORWARD _PROTOTYPE( phys_bytes rlmem_getpage, (void)                    );
FORWARD _PROTOTYPE( void map_dir, (phys_bytes vm_addr,
                                phys_bytes real_addr)                   );
FORWARD _PROTOTYPE( void map_page, (phys_bytes vm_addr,
                                phys_bytes real_addr)                   );
#if DEBUG
FORWARD _PROTOTYPE( void dump_mem, (void)                               );
#endif
FORWARD _PROTOTYPE( int check_user_fault, (phys_bytes addr)             );

#ifdef phys_zero_scan
FORWARD _PROTOTYPE (phys_bytes my_scan, (phys_bytes addr));
#endif

#define VM_SIZE_CLICKS       0x80000            /* 2G in clicks */

/*===========================================================================*
 *                                vm_init                                    *
 *===========================================================================*/

PUBLIC void vm_init()
{
        phys_bytes hi_addr, dir_addr, dir_pointer, page_addr;

        rlmem_init();

        page_base= rlmem_getpage();
        phys_clr_page(page_base);       /* No page directories */

        hi_addr= hi_mem << CLICK_SHIFT;
        for (dir_addr= 0; dir_addr<hi_addr; dir_addr += 4*1024*1024)
        {
                dir_pointer= rlmem_getpage();
                phys_clr_page(dir_pointer);     /* No pages */
                map_dir(dir_addr, dir_pointer);
        }
        for (page_addr= 0; page_addr<hi_addr; page_addr += CLICK_SIZE)
                map_page(page_addr, page_addr);

#if DEBUG
 { printW(); printf("enabling paging\r\n"); }
#endif

        vm_enable(page_base);           /* This is what it's all about */

#if DEBUG
```

```
 { printW(); printf("paging enabled\r\n"); }
#endif

	/* calculate virt_mem */
	virt_base= (hi_mem + 0x1000000) & ~0x3fffff;
	paging_base= virt_base;

	chunk_add (virt_base >> CLICK_SHIFT, VM_SIZE_CLICKS, MEM_LOW);
	/* Normal... memory */

	vm_not_alloc= free_mem;		/* Reset vm_not_alloc to memory now available */
}

/*===========================================================================*
 *                                rlmem_init                                 *
 *===========================================================================*/


PRIVATE void rlmem_init()
{
	phys_clicks rlmem_table_clicks, chk_size, chk_base, click_ptr;
	phys_bytes phys_ptr;
	int i;

	if (CLICK_SIZE != 4096)
		panic ("Wrong click size", CLICK_SIZE);
		/* It is essencial that CLICK_SIZE is equal to the size of
		 * one page */

	hi_mem= 0;
	for (i=0; i<CHUNK_NR; i++)
	{
		if (!chunk_table[i].chk_size)
			break;
		if (chunk_table[i].chk_base + chunk_table[i].chk_size >
			hi_mem)
			hi_mem= chunk_table[i].chk_base +
				chunk_table[i].chk_size;
	}
#if DEBUG || 1
 { printW(); printf("hi_mem= %d clicks\r\n", hi_mem); }
#endif

	rlmem_table_clicks= (hi_mem >> CLICK_SHIFT)+1;
#if DEBUG || 1
 { printW(); printf("rlmem_table_clicks= %d clicks\r\n", rlmem_table_clicks); }
#endif
	/* At least one non allocated entry in the table */

	/* Find a place for the table */
	chk_base= 0;
	for (i=0; i<CHUNK_NR; i++)
	{
#if DEBUG || 1
 { printW(); printf("chunk_table[%d]: size= %d, base= %d, mode= %d\r\n",
	i, chunk_table[i].chk_size, chunk_table[i].chk_base,
	chunk_table[i].chk_mode); }
#endif
		chk_size= chunk_table[i].chk_size;
		if (!chk_size)
			break;
		if (chk_size < rlmem_table_clicks)
			continue;
		if (chunk_table[i].chk_mode != MEM_LOW &&
			chunk_table[i].chk_mode != MEM_EXT)
			continue;
		chk_base= chunk_table[i].chk_base;
		chunk_del(chk_base, rlmem_table_clicks);
		break;
```

```
		}
		if (!chk_base)
			panic("Unable to find a place for the memory allocation table",
				NO_NUM);

		rlmem_table_base= chk_base << CLICK_SHIFT;
		rlmem_table_size= rlmem_table_clicks << CLICK_SHIFT;
		for (phys_ptr= rlmem_table_base; phys_ptr<rlmem_table_base+
			rlmem_table_size; phys_ptr++)
		{
			put_phys_byte(phys_ptr, 1);	/* Click is allocated */
		}
#if DEBUG
 { printW(); dump_mem(); }
#endif

		free_mem= 0;
		/* Free all available mem */
		while (chunk_table[0].chk_size)
		{
			chk_base= chunk_table[0].chk_base;
			chk_size= chunk_table[0].chk_size;
			if (chunk_table[0].chk_mode == MEM_LOW ||
				chunk_table[0].chk_mode == MEM_EXT)
			{
				for (click_ptr= chk_base; click_ptr<chk_base+chk_size;
					click_ptr++)
					rlmem_free(click_ptr << CLICK_SHIFT);
			}
#if DEBUG
 else { printf("Chunk at %d clicks of size %d clicks and mode %d not used\r\n",
		chk_base, chk_size, chunk_table[0].chk_mode); }
#endif
			chunk_del(chk_base, chk_size);
		}
#if DEBUG
 { printW(); dump_mem(); }
#endif
#if DEBUG
 { printW(); printf("Total free pages: %d\r\n", free_mem); }
#endif
		vm_not_alloc= free_mem;
}

/*===========================================================================*
 *                                rlmem_free                                 *
 *===========================================================================*/

PRIVATE int rlmem_free(page_addr)
phys_bytes page_addr;
{
		phys_bytes entry_addr;
		int link_count;

		if (page_addr & (CLICK_SIZE-1))
			panic("rlmem_free on strange address: ", page_addr);

		entry_addr= rlmem_table_base + (page_addr >> CLICK_SHIFT);
		link_count= get_phys_byte(entry_addr);

		if (!link_count)
			panic("freeing unuse rlmem page", NO_NUM);

		link_count--;
		if (!link_count)
			free_mem++;
		put_phys_byte(entry_addr, link_count);
		return link_count;
}
```

```
/*===========================================================================*
*                                rlmem_getpage                              *
*===========================================================================*/

PRIVATE phys_bytes rlmem_getpage()
{
        phys_bytes phys_ptr;

#if DEBUG & 256
 { printW(); printf("in rlmem_getpage()\r\n"); dump_mem(); }
#endif

        if (!free_mem)
                panic("Out of pages", NO_NUM);
        free_mem--;

        phys_ptr= phys_zero_scan(rlmem_table_base, rlmem_table_size);

#if DEBUG & 256
 { printW(); printf("phys_ptr= 0x%x, rlmem_table_base= 0x%x, phys_zero_scan= 0x%x\r\n",
        phys_ptr, rlmem_table_base, (phys_zero_scan)(rlmem_table_base,
                rlmem_table_size)); }
#endif
assert (!get_phys_byte(phys_ptr));

        put_phys_byte(phys_ptr, 1);
#if DEBUG & 256
 { printW(); dump_mem(); }
#endif
        return (phys_ptr-rlmem_table_base) << CLICK_SHIFT;
}

/*===========================================================================*
*                                map_dir                                    *
*===========================================================================*/

PRIVATE void map_dir(vm_addr, real_addr)
phys_bytes vm_addr;
phys_bytes real_addr;
{
        u32_t dir_ent;
        phys_bytes ent_addr;

#if DEBUG & 256
 { printW(); printf("map_dir(0x%x, 0x%x) called\r\n", vm_addr, real_addr); }
#endif

        if (vm_addr & VM_DIRMASK)
                panic("Invalid directory base: ", vm_addr);
        if (real_addr & VM_PAGEMASK)
                panic("Invalid directory addr: ", real_addr);

        dir_ent= real_addr | VM_INMEM_N_PRESENT | VM_WRITE | VM_USER;
        ent_addr= page_base+ vm_addr_to_dir(vm_addr)*4;

        put_phys_dword(ent_addr, dir_ent);
}

/*===========================================================================*
*                                map_page                                   *
*===========================================================================*/

PRIVATE void map_page(vm_addr, real_addr)
phys_bytes vm_addr;
phys_bytes real_addr;
{
        u32_t dir_ent, page_ent;
        phys_bytes ent_addr;
```

```
#if DEBUG & 256
 { printW(); printf("map_page(0x%x, 0x%x) called\r\n", vm_addr, real_addr); }
#endif

	if (vm_addr & VM_PAGEMASK)
		panic("Invalid page base: ", vm_addr);
	if (real_addr & VM_PAGEMASK)
		panic("Invalid page addr: ", real_addr);

	ent_addr= page_base+ vm_addr_to_dir(vm_addr)*4;
	dir_ent= get_phys_dword(ent_addr);

	if ((dir_ent & VM_INMEM_N_PRESENT) != VM_INMEM_N_PRESENT)
		panic("Page directory not present for page: ", vm_addr);

	ent_addr= (dir_ent & VM_ADDRMASK) + vm_addr_to_page(vm_addr)*4;

	page_ent= real_addr | VM_INMEM_N_PRESENT | VM_WRITE | VM_USER;
	put_phys_dword(ent_addr, page_ent);
}

/*===========================================================================*
 *                                vm_page_fault                              *
 *===========================================================================*/


PUBLIC void vm_page_fault(err, addr)
u32_t err;
phys_bytes addr;
{
	phys_bytes dir_ent_addr, dir_addr, page_ent_addr, page_addr;
	u32_t dir_ent, page_ent;
	phys_clicks page_no;
	int linkC;

#if DEBUG & 256
 { printW(); printf("process %d got a page fault at 0x%x due to a %s.\n",
	proc_number(proc_ptr), addr, (err & 1) ? "protection violation" :
	"not present page");
	printf("The process was running in %s mode and issuing a %s.\n",
	(err & 4) ? "user" : "supervisor", (err & 2) ? "write" : "read"); }
	printf("pc= 0x%x\n", proc_ptr->p_reg.pc);
#endif
assert (addr >= paging_base);

	/* First we gather as much information as possible */
	dir_ent_addr= page_base + vm_addr_to_dir(addr)*4;
	dir_ent= get_phys_dword(dir_ent_addr);
	if (dir_ent & VM_PRESENT)
	{
		page_ent_addr= (dir_ent & VM_ADDRMASK) +
			vm_addr_to_page(addr)*4;
		page_ent= get_phys_dword(page_ent_addr);
	}

	if (proc_number(proc_ptr) < 0)
	{
#if DEBUG
 if (vm_cp_mess)
 { printW(); printf("Page fault in vm_cp_mess (vm_cp_mess= %d)\n", vm_cp_mess); }
#endif
		if (!vm_cp_mess && proc_number(proc_ptr) != SYSTASK &&
			proc_number(proc_ptr) != TTY &&
			proc_number(proc_ptr) != MEM)
			panic("Kernel task causes page fault",
				proc_number(proc_ptr));
		/* No further checks should be necessary */
	}
```

```
		else if(!vm_cp_mess) /* User process causing page fault */
		{
			if (check_user_fault(addr) != OK)
				return;
		}

		if (!dir_ent)
		{
#if DEBUG & 256
 { printW(); printf("Page directory not present (allocating one)\n"); }
#endif
			dir_addr= rlmem_getpage();
			phys_clr_page(dir_addr);		/* No pages */
			map_dir(addr & ~VM_DIRMASK, dir_addr);
#if DEBUG & 256
 { printW(); printf("Allocation of directory done\n"); }
#endif
			vm_reload();
			return;
		}
		if ((dir_ent & VM_INMEM_N_PRESENT) != VM_INMEM_N_PRESENT)
		{
			printf("Strange dir ent: 0x%x\n", dir_ent);
			panic("Inconsistent paging system", NO_NUM);
		}
		if (!page_ent)
		{
#if DEBUG & 256
 { printW(); printf("Page not present (allocating one)\n"); }
#endif
			page_addr= rlmem_getpage();
			phys_clr_page(page_addr);	/* Empty page*/
			map_page(addr & VM_ADDRMASK, page_addr);
#if DEBUG & 256
 { printW(); printf("Allocation of page done\n"); }
#endif
			vm_reload();
			return;
		}
		if (!(page_ent & VM_PRESENT))		/* Copy on access */
		{
assert ((page_ent & (VM_INMEM|VM_WRITE)) == (VM_INMEM|VM_WRITE));
			page_no= page_ent >> VM_PAGESHIFT;
			linkC= get_phys_byte(rlmem_table_base+page_no);
			if (linkC != 1)
			{
				page_addr= rlmem_getpage();
				phys_copy(page_ent & VM_ADDRMASK, page_addr,
					VM_PAGESIZE);
				page_ent= (page_ent & VM_PAGEMASK) | page_addr;
				put_phys_byte(rlmem_table_base+page_no, linkC-1);
			}
			page_ent |= VM_PRESENT;
			put_phys_dword(page_ent_addr, page_ent);
			vm_reload();
			return;
		}
		printf("Strange page ent: 0x%x\n", page_ent);
		panic("Inconsistent paging system", NO_NUM);
}

#if DEBUG

/*===========================================================================*
 *                                dump_mem                                   *
 *===========================================================================*/

PRIVATE void dump_mem()
{
```

```
	int i;
	phys_bytes phys_ptr;

	printf("\r\nDumping rlmem_table\r\n");
	for (i=0, phys_ptr= rlmem_table_base; i<256; i++, phys_ptr++)
		printf("%d ", get_phys_byte(phys_ptr));
	printf("\r\n");
}
#endif

/*===========================================================================*
 *                                vm_unmap                                   *
 *===========================================================================*/

PUBLIC void vm_unmap(addr, vm_size, alloc_size)
phys_bytes addr;
phys_bytes vm_size;
phys_clicks alloc_size;
{
	phys_bytes top;
	phys_bytes dir_ent_addr, page_ent_addr;
	u32_t dir_ent, page_ent;
	int i, link_count;

#if DEBUG & 256
 { printW(); printf("freeing 0x%x at 0x%x\r\n", vm_size, addr); }
#endif
assert(!(addr & VM_DIRMASK));		/* aligned on a 4M boundary */

	top= addr+vm_size;
	while(addr<top)
	{
		dir_ent_addr= page_base+vm_addr_to_dir(addr)*4;
		dir_ent= get_phys_dword(dir_ent_addr);
		if (!dir_ent)		/* not mapped */
		{
			addr += VM_DIRSIZE;
			continue;
		}
		if ((dir_ent & VM_INMEM_N_PRESENT) != (VM_INMEM_N_PRESENT))
			panic("Dir not present", NO_NUM);

		put_phys_dword(dir_ent_addr, (u32_t)0);
		page_ent_addr= dir_ent & VM_ADDRMASK;
		for (i= 0; i<1024; i++, page_ent_addr += 4)
		{
			page_ent= get_phys_dword(page_ent_addr);
			if (!page_ent)
				continue;
			if (!(page_ent & VM_INMEM))
				panic("Page not in memory", NO_NUM);
			link_count= rlmem_free(page_ent & VM_ADDRMASK);
			if (link_count && !(page_ent & VM_WRITE))
			/* Compansating for read only pages */
				vm_not_alloc--;
		}
		rlmem_free(dir_ent & VM_ADDRMASK);
		addr += VM_DIRSIZE;
	}
#if DEBUG & 256
 { printW(); printf("vm_not_alloc += %d + %d\n", alloc_size,
	((vm_size+VM_DIRSIZE-1) >> VM_DIRSHIFT)); }
#endif
	vm_not_alloc += alloc_size + ((vm_size+VM_DIRSIZE-1) >> VM_DIRSHIFT);
	vm_u_reload();
}
```

```
/*===========================================================================*
 *                                vm_fork                                    *
 *===========================================================================*/

PUBLIC void vm_fork(parent, c_base_clicks)
struct proc *parent;
phys_clicks c_base_clicks;
{
        int dirs;     /* Number of directories */
        phys_bytes p_base, p_data_base, p_dir_ent_addr, p_page_ent_addr;
        phys_bytes c_base, c_page_ent_addr, c_dir_ent_addr;
        phys_bytes vir_addr;
        phys_clicks p_base_clicks, p_top_clicks, page_no;
        u32_t p_dir_ent, p_page_ent;
        int i, j, linkC;
        int traced;

#if DEBUG & 256
 { printW(); printf("In vm_fork()\n"); }
#endif
        p_base_clicks= parent->p_map[T].mem_phys;
        p_base= p_base_clicks << CLICK_SHIFT;
        p_data_base= (parent->p_map[D].mem_phys) << CLICK_SHIFT;
        p_top_clicks= parent->p_map[S].mem_phys + parent->p_map[S].mem_vir +
                                                        parent->p_map[S].mem_len;

assert (!(p_base & VM_DIRMASK));
assert (p_base >= paging_base);
        dirs= ((p_top_clicks-p_base_clicks-1) >> (VM_DIRSHIFT-CLICK_SHIFT))+1;

#if DEBUG & 256
 { printW(); printf("vm_not_alloc -= %d\n", dirs); }
#endif
        vm_not_alloc -= dirs;

        c_base= c_base_clicks << CLICK_SHIFT;
assert (!(c_base & VM_DIRMASK));
assert (c_base >= paging_base);

        p_dir_ent_addr= page_base + vm_addr_to_dir(p_base)*4;
        c_dir_ent_addr= page_base + vm_addr_to_dir(c_base)*4;

        traced= !!(parent->p_status & P_ST_TRACED);

        for (i= 0; i<dirs; i++, p_dir_ent_addr += 4, c_dir_ent_addr += 4)
        {
if (get_phys_dword(c_dir_ent_addr))
 {
        printf("c_dir_ent_addr= 0x%x, get_phys_dword(...)= 0x%x\n",
                c_dir_ent_addr, get_phys_dword(c_dir_ent_addr));
assert (!get_phys_dword(c_dir_ent_addr));
 }
                p_dir_ent= get_phys_dword(p_dir_ent_addr);
                if (!p_dir_ent)
                {
#if DEBUG || 1
 { printW(); printf("p_dir %d is empty\n", i); }
#endif
                        continue;
                }
assert ((p_dir_ent & VM_INMEM_N_PRESENT) == VM_INMEM_N_PRESENT);
                p_page_ent_addr= p_dir_ent & VM_ADDRMASK;
                for (j= 0; j<1024; j++, p_page_ent_addr += 4)
                {
                        p_page_ent= get_phys_dword(p_page_ent_addr);
                        if (!p_page_ent)
                        {
#if DEBUG & 256
```

```
 { printW(); printf("p_page %d of dir %d is empty\n", j, i); }
#endif
                                        continue;
                                }
assert(p_page_ent & VM_INMEM);
                                if ((p_page_ent & VM_IM_RW_PRES) == VM_IM_RW_PRES)
                                                                        /* ordinary page */
                                {
                                        vir_addr= p_base + (i << VM_DIRSHIFT) +
                                                (j<<VM_PAGESHIFT);
                                        if (!traced && vir_addr<p_data_base)
                                                /* Text page */
                                        {
#if DEBUG & 256
 { printW(); printf("i= %d, j= %d, page will be read only", i, j); }
#endif
                                                p_page_ent &= ~VM_WRITE;
#if DEBUG & 256
 { printW(); printf("vm_not_alloc++\n"); }
#endif
                                                vm_not_alloc++;
                                        }
                                        else    /* Data page */
                                        {
#if DEBUG & 256
 { printW(); printf("i= %d, j= %d, page will be unmapped\n", i, j); }
#endif
                                                p_page_ent &= ~VM_PRESENT;
                                        }
                                        page_no= p_page_ent >> VM_PAGESHIFT;
assert(get_phys_byte(rlmem_table_base+page_no) == 1);
                                        put_phys_byte(rlmem_table_base+page_no, 2);
                                        put_phys_dword(p_page_ent_addr, p_page_ent);
                                        continue;
                                }
                                /* Check if page is copy on access or read only */
                                /* It can't be both and INMEM has allready been
                                 * checked */
assert(p_page_ent & (VM_WRITE | VM_PRESENT));

#if DEBUG & 256
 { printW(); printf("i= %d, j= %d, page is read only or unmapped\n", i, j); }
#endif
                                if (p_page_ent & VM_PRESENT) /* Read only page */
                                {
#if DEBUG & 256
 { printW(); printf("vm_not_alloc++\n"); }
#endif
                                        vm_not_alloc++;
                                }
                                page_no= p_page_ent >> VM_PAGESHIFT;
                                linkC= get_phys_byte(rlmem_table_base+page_no);
                                put_phys_byte(rlmem_table_base+page_no, linkC+1);
                        }
                        /* Allocate a new page dir, and copy dir */
                        c_page_ent_addr= rlmem_getpage();
                        phys_copy(p_dir_ent & VM_ADDRMASK, c_page_ent_addr,
                                VM_PAGESIZE);

                        /* Map dir */
                        map_dir(c_base+ (i<<VM_DIRSHIFT), c_page_ent_addr);
                }
#if DEBUG & 256
 { printW(); printf("vm_fork() done\n"); }
#endif
                vm_u_reload();
}
```

```
/*===========================================================================*
 *                              vm_map_server                                *
 *===========================================================================*/


PUBLIC phys_clicks vm_map_server(text_base, text_clicks, data_clicks,
        bss_clicks, heap_clicks)
phys_clicks text_base;
phys_clicks text_clicks;
phys_clicks data_clicks;
phys_clicks bss_clicks;
phys_clicks heap_clicks;
{
        phys_bytes vm_base;
        phys_bytes bss_base, dir_base;
        phys_clicks vm_base_clicks, tot_clicks;
        int i;

        vm_base= virt_base;
        tot_clicks= text_clicks + data_clicks + bss_clicks + heap_clicks;
        vm_not_alloc -= bss_clicks;     /* text and data segment are part of
                                                * the loaded image */
        for (i= 0; i<tot_clicks; i+= (VM_DIRSIZE/CLICK_SIZE))
        {
                dir_base= rlmem_getpage();
                vm_not_alloc--;
                phys_clr_page(dir_base);
                map_dir(vm_base+ (i << CLICK_SHIFT), dir_base);
        }

        for (i= 0; i<text_clicks+data_clicks; i++)
        {
                map_page(virt_base, text_base << CLICK_SHIFT);
                text_base++;
                virt_base += VM_PAGESIZE;
        }
        for (i= 0; i<bss_clicks; i++)
        {
                bss_base= rlmem_getpage();
                phys_clr_page(bss_base);
                map_page(virt_base, bss_base);
                virt_base += VM_PAGESIZE;
        }
        virt_base += heap_clicks << CLICK_SHIFT;

        /* Calculate new virt_base */
        virt_base= (virt_base + 0x400000) & ~0x3fffff;
/*      paging_base= virt_base; */
        vm_base_clicks= vm_base >> CLICK_SHIFT;
        chunk_del(vm_base_clicks, (virt_base >> CLICK_SHIFT)-vm_base_clicks);
        vm_u_reload();
        return vm_base_clicks;
}

/*===========================================================================*
 *                              vm_check_unmapped                            *
 *===========================================================================*/


PUBLIC void vm_check_unmapped(base, top)
phys_bytes base;
phys_bytes top;
{
        phys_bytes ptr, dir_ent_addr;
        u32_t dir_ent;

assert(!(base & VM_DIRMASK));           /* aligned on a 4M boundary */
```

```
	for (ptr= base; ptr<top; ptr += VM_DIRSIZE)
	{
		dir_ent_addr= page_base+vm_addr_to_dir(ptr)*4;
		dir_ent= get_phys_dword(dir_ent_addr);
#if DEBUG || 1
 if (dir_ent)
 {
	printW(); printf("check_unmapped failed, base= 0x%x, top= 0x%x, ptr= 0x%x, dir_ent_addr= 0x%x, dir_ent= 0x%x\n",
		base, top, ptr, dir_ent_addr, dir_ent);
 }
#endif
assert (!dir_ent);	/* not mapped */
	}
}

/*===========================================================================*
 *                                vm_dump                                    *
 *===========================================================================*/

PUBLIC void vm_dump()
{
	printf("\r\nvm_dump:\r\n\r\n");
	printf("free memory pages: %d (= %dK), not reserved mem: %d (= %dK)\r\n",
		free_mem, ((free_mem << CLICK_SHIFT) + 512) >> 10,
		vm_not_alloc, ((vm_not_alloc << CLICK_SHIFT) + 512) >> 10);
}

/*===========================================================================*
 *                                check_user_fault                           *
 *===========================================================================*/

PRIVATE int check_user_fault(addr)
phys_bytes addr;
{
	vir_bytes sp;
	phys_clicks sp_click, delta_clicks;
	phys_bytes data_base;

	data_base= (proc_ptr->p_map[D].mem_phys) << CLICK_SHIFT;
	if (addr<data_base)	/* Text segment */
	{
#if DEBUG || 1
 { printW(); printf("Page fault in Text segment at 0x%x\n", addr); }
#endif
		if (addr<(proc_ptr->p_map[T].mem_phys +
				proc_ptr->p_map[T].mem_vir) << CLICK_SHIFT)
		{
			cause_sig(proc_number(proc_ptr), SIGSEGV);
			return ERROR;
		}
assert (addr < ((proc_ptr->p_map[T].mem_phys+proc_ptr->p_map[T].mem_vir +
					proc_ptr->p_map[T].mem_len) << CLICK_SHIFT));
	}
	else if (addr < ((proc_ptr->p_map[D].mem_phys +
		proc_ptr->p_map[D].mem_vir + proc_ptr->p_map[D].mem_len)
						<< CLICK_SHIFT)) /* Data segment */
	{
#if DEBUG & 256
 { printW(); printf("Page fault in Data segment at 0x%x\n", addr); }
#endif
		if (addr<proc_ptr->p_map[D].mem_phys +
					proc_ptr->p_map[D].mem_vir << CLICK_SHIFT)
		{
			cause_sig(proc_number(proc_ptr), SIGSEGV);
			return ERROR;
		}
	}
	else if (addr >= ((proc_ptr->p_map[S].mem_phys +
					proc_ptr->p_map[S].mem_vir) << CLICK_SHIFT))
```

```
                                                                        /* Stack segment */
                {
#if DEBUG & 256
 { printW(); printf("Page fault in Stack segment\n"); }
#endif
assert (addr < ((proc_ptr->p_map[S].mem_phys+
                proc_ptr->p_map[S].mem_vir + proc_ptr->p_map[S].mem_len) <<
                                                                                CLICK_SHIFT));
                }
                else                                                    /* Growing stack */
                {
#if DEBUG
 { printW(); printf("Page fault in Heap segment\n"); }
#endif
                        sp= proc_ptr->p_reg.sp;
                        sp_click= (sp >> CLICK_SHIFT)-1;
                        /* One click extra to avoid problems with pushad */
                        if (sp_click < proc_ptr->p_map[S].mem_vir)
                        {               /* Growing stack */
                                if (sp_click < proc_ptr->p_map[D].mem_vir +
                                                proc_ptr->p_map[D].mem_len +
                                                STACK_SAFETY_CLICKS)
                                {
#if DEBUG
 { printW(); printf("calling cause_sig\n"); }
#endif
                                        cause_sig(proc_number(proc_ptr), SIGSTKFLT);
                                        return ERROR;
                                }
                                delta_clicks= proc_ptr->p_map[S].mem_vir - sp_click;
                                if (vm_not_alloc < delta_clicks)
                                {
                                        if (proc_number(proc_ptr) <= INIT_PROC_NR)
                                        { /* Let the OS procede */
                                                printf("Warning: allocating memory for %d but out of memory\n",
proc_number(proc_ptr));
                                        }
                                        else
                                        {
                                                cause_sig(proc_number(proc_ptr),
                                                        SIGSTKFLT);
                                                return ERROR;
                                        }
                                }
                                proc_ptr->p_map[S].mem_len += delta_clicks;
                                proc_ptr->p_map[S].mem_vir -= delta_clicks;
assert(proc_ptr->p_map[S].mem_vir == sp_click);
#if DEBUG & 256
 { printW(); printf("vm_not_alloc -= %d\n", delta_clicks); }
#endif
                                vm_not_alloc -= delta_clicks;
                        }
                        /* recheck page fault */
                        if (addr >= ((proc_ptr->p_map[S].mem_phys +
                                        proc_ptr->p_map[S].mem_vir) << CLICK_SHIFT))
                                        /* Stack segment */
                        {
#if DEBUG & 256
 { printW(); printf("Page fault in enlarged Stack segment\n"); }
#endif
assert (addr < ((proc_ptr->p_map[S].mem_phys+proc_ptr->p_map[S].mem_vir+
                                        proc_ptr->p_map[S].mem_len) << CLICK_SHIFT));
                        }
                        else            /* Signal process */
                        {
#if DEBUG
  { printW(); printf("calling cause_sig for %d, addr= 0x%x pc= 0x%x\n",
                proc_number(proc_ptr), addr, proc_ptr->p_reg.pc); }
#endif
                                cause_sig(proc_number(proc_ptr), SIGSEGV);
```

```
                    return ERROR;
                }
            }
            return OK;
}
```

++

vm386.s

This file contains assembler routines for the 386 virtual memory management

++

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

#define CR0_PG                          0x80000000

! Sections
.sect .text; .sect .rom; .sect .data; .sect .bss

! Exported routines
.sect .text
.define _put_phys_byte
.define _get_phys_byte
.define _put_phys_dword
.define _get_phys_dword
.define _phys_zero_scan
.define _vm_enable
.define _vm_reload
.define _vm_u_reload
.define _phys_clr_page

.sect .text

! void put_phys_byte (phys_bytes phys_addr, int byte);

_put_phys_byte:
        push    ebx
        push    es
        mov     ax,FLAT_DS_SELECTOR
        mov     es, ax
        mov     eax, 4+12(sp)           ! byte
        mov     ebx, 0+12(sp)           ! phys_addr
        eseg
        movb    (ebx), al
        pop     es
        pop     ebx
        ret

! int get_phys_byte (phys_bytes phys_addr);

_get_phys_byte:
        push    ebx
        push    es
        mov     ax, FLAT_DS_SELECTOR
        mov     es, ax
        mov     ebx, 0+12(sp)           ! phys_addr
        eseg
        movzxb  eax, (ebx)
        pop     es
        pop     ebx
        ret

! void put_phys_dword (phys_bytes phys_addr, u32_t dword);

_put_phys_dword:
        push    ebx
        push    es
        mov     ax,FLAT_DS_SELECTOR
        mov     es, ax
        mov     eax, 4+12(sp)           ! dword
```

```
	mov	ebx, 0+12(sp)		! phys_addr
	eseg
	mov	(ebx), eax
	pop	es
	pop	ebx
	ret

! u32_t get_phys_dword (phys_bytes phys_addr);

_get_phys_dword:
	push	ebx
	push	es
	mov	ax, FLAT_DS_SELECTOR
	mov	es, ax
	mov	ebx, 0+12(sp)		! phys_addr
	eseg
	mov	eax, (ebx)
	pop	es
	pop	ebx
	ret

! phys_bytes phys_zero_scan (phys_bytes table_base, phys_bytes table_size);

_phys_zero_scan:
	push	edi
	push	ecx
	push	es
	mov	edi, 0+16(sp)			! table_base
	mov	ecx, 4+16(sp)			! table_size
	mov	ax, FLAT_DS_SELECTOR
	mov	es, ax
	movb	al, 0
	cld				!clear direction flag
	repne
	scasb				! Search 0 byte in [ES:EDI]
	mov	eax, edi
	dec	eax
	pop	es
	pop	ecx
	pop	edi
	ret

! void vm_enable(phys_bytes page_base)

_vm_enable:
	mov	eax, 0+4(sp)		! page_base
	mov	cr3, eax

	mov	eax,cr0
	or	eax,CR0_PG
	mov	cr0,eax
	ret

! void vm_reload(void)

_vm_reload:
	mov	eax, cr3
	mov	cr3, eax
	ret

_vm_u_reload:
	int	VMRELOAD_VECTOR
	ret

! void phys_clr_page (phys_bytes addr);

_phys_clr_page:
	push	edi
	push	ecx
	push	es
```

```
mov     ax, FLAT_DS_SELECTOR
mov     es, ax
mov     edi, 0+16(sp)                   ! addr
mov     eax, 0
mov     ecx, CLICK_SIZE/4
cld
rep
stos
pop     es
pop     ecx
pop     edi
ret
```

++

system.c

++

```
/* This task handles the interface between file system and kernel as well
as
 * between memory manager and kernel.  System services are obtained by
sending
 * sys_task() a message specifying what is needed.  To make life easier for
 * MM and FS, a library is provided with routines whose names are of the
 * form sys_xxx, e.g. sys_xit sends the SYS_XIT message to sys_task.  The
 * message types and parameters are:
 *
 *   SYS_FORK      informs kernel that a process has forked
 *   SYS_GETMAP    allows MM to get a process' memory map
 *   SYS_EXEC      sets program counter and stack pointer after EXEC
 *   SYS_XIT       informs kernel that a process has exited
 *   SYS_GETSP     caller wants to read out some process' stack pointer
 *   SYS_TIMES     caller wants to get accounting times for a process
 *   SYS_ABORT     MM or FS cannot go on; abort MINIX
 *   SYS_FRESH     start with a fresh process image during EXEC (68000
only)
 *   SYS_SENDSIG send a signal to a process (POSIX style)
 *   SYS_SIGRETURN complete POSIX-style signalling
 *   SYS_KILL      cause a signal to be sent via MM
 *   SYS_ENDSIG    finish up after SYS_KILL-type signal
 *   SYS_COPY      request a block of data to be copied between processes
 *   SYS_VCOPY   request a series of data blocks to be copied between procs
 *   SYS_GBOOT     copies the boot parameters to a process
 *   SYS_MEM       returns the next free chunk of physical memory
 *   SYS_UMAP      compute the physical address for a given virtual address
 *   SYS_TRACE     request a trace operation
 #if (CHIP == INTEL) && VIRT_MEM
 *   SYS_ADJMAP  allows MM to changed a map for a brk or a signal
 *   SYS_EXECMAP allows MM to install a new map during to exec system call
 *   SYS_UNMAP     release allocated pages for a process, obsolete
 #else
 *   SYS_NEWMAP    allows MM to set up a process memory map, obsolete
 #endif

 * Message types and parameters:
 *
 *    m_type       PROC1     PROC2      PID     MEM_PTR
 * ------------------------------------------------------
 * | SYS_FORK   | parent  |  child  |   pid   |         |
 * |------------+---------+---------+---------+---------|
 * | SYS_EXEC   | proc nr | traced  | new sp  |         |
 * |------------+---------+---------+---------+---------|
 * | SYS_XIT    | parent  | exitee  |         |         |
 * |------------+---------+---------+---------+---------|
 * | SYS_GETSP  | proc nr |         |         |         |
 * |------------+---------+---------+---------+---------|
 * | SYS_TIMES  | proc nr |         | buf ptr |         |
 * |------------+---------+---------+---------+---------|
 * | SYS_ABORT  |         |         |         |         |
```

```
* |------------+---------+---------+---------+---------|
* | SYS_FRESH  | proc nr | data_cl |         |         |
* |------------+---------+---------+---------+---------|
* | SYS_GBOOT  | proc nr |         |         | bootptr |
* |------------+---------+---------+---------+---------|
*
*    m_type       PROC      m1_i2     m1_i3    MEM_PTR
* ------------------------------------------------------
* | SYS_GETMAP | proc nr |         |         | map ptr |
* |------------+---------+---------+---------+---------|
* | SYS_ADJMAP | proc nr |dt_clcks |  sp     | map_ptr |
* |------------+---------+---------+---------+---------
* | SYS_EXECMAP| proc nr |         |         | map_ptr |
* |------------+---------+---------+---------+---------
#else
* | SYS_NEWMAP | proc nr |         |         | map ptr |
* |------------+---------+---------+---------+---------|
#endif

* ------------------------------------------------------
*
*    m_type          m1_i1     m1_i2     m1_i3       m1_p1
* ---------------+---------+---------+---------+-------------
* | SYS_VCOPY    |  src p  |  dst p  | vec siz | vc addr     |
* |--------------+---------+---------+---------+-------------|
* | SYS_SENDSIG  | proc nr |         |         | smp         |
* |--------------+---------+---------+---------+-------------|
* | SYS_SIGRETURN | proc nr |         |         | scp         |
* |--------------+---------+---------+---------+-------------|
* | SYS_ENDSIG   | proc nr |         |         |             |
* ------------------------------------------------------------
*
*    m_type       m2_i1     m2_i2     m2_l1     m2_l2
* ------------------------------------------------------
* | SYS_TRACE  | proc_nr | request |  addr   |  data   |
* ------------------------------------------------------
*
*
*    m_type       m6_i1     m6_i2     m6_i3     m6_f1
* ------------------------------------------------------
* | SYS_KILL   | proc_nr  |  sig    |         |         |
* ------------------------------------------------------
*
*
*    m_type    m5_c1   m5_i1    m5_l1   m5_c2   m5_i2    m5_l2   m5_l3
* ------------------------------------------------------------------------
* | SYS_COPY |src seg|src proc|src vir|dst seg|dst proc|dst vir| byte ct |
* ------------------------------------------------------------------------
* | SYS_UMAP |  seg  |proc nr |vir adr|       |        |       | byte ct |
* ------------------------------------------------------------------------
*
*
*    m_type      m1_i1      m1_i2      m1_i3
* |------------+----------+----------+----------
* | SYS_MEM    | mem base | mem size | tot mem |
* ---------------------------------------------
*
```

```
 * In addition to the main sys_task() entry point, there are 5 other minor
 * entry points:
 *   cause_sig:   take action to cause a signal to occur, sooner or later
 *   inform:      tell MM about pending signals
 *   numap: umap D segment starting from process number instead of pointer
 *   umap:  compute the physical address for a given virtual address
 *   alloc_segments: allocate segments for 8088 or higher processor
 */
```

```
#include "kernel.h"
#include <signal.h>
#include <unistd.h>
#include <sys/sigcontext.h>
#include <sys/ptrace.h>
#include <minix/boot.h>
#include <minix/callnr.h>
#include <minix/com.h>
#include "proc.h"
#if (CHIP == INTEL)
#include "protect.h"
#if VIRT_MEM
#include "vm386.h"
#endif
#endif

/* PSW masks. */
#define IF_MASK 0x00000200
#define IOPL_MASK 0x003000

PRIVATE message m;

FORWARD _PROTOTYPE( int do_abort, (message *m_ptr) );
FORWARD _PROTOTYPE( int do_copy, (message *m_ptr) );
FORWARD _PROTOTYPE( int do_exec, (message *m_ptr) );
FORWARD _PROTOTYPE( int do_fork, (message *m_ptr) );
FORWARD _PROTOTYPE( int do_gboot, (message *m_ptr) );
FORWARD _PROTOTYPE( int do_getmap, (message *m_ptr) );
FORWARD _PROTOTYPE( int do_getsp, (message *m_ptr) );
FORWARD _PROTOTYPE( int do_kill, (message *m_ptr) );
FORWARD _PROTOTYPE( int do_mem, (message *m_ptr) );
FORWARD _PROTOTYPE( int do_sendsig, (message *m_ptr) );
FORWARD _PROTOTYPE( int do_sigreturn, (message *m_ptr) );
FORWARD _PROTOTYPE( int do_endsig, (message *m_ptr) );
FORWARD _PROTOTYPE( int do_times, (message *m_ptr) );
FORWARD _PROTOTYPE( int do_trace, (message *m_ptr) );
FORWARD _PROTOTYPE( int do_umap, (message *m_ptr) );
FORWARD _PROTOTYPE( int do_xit, (message *m_ptr) );
FORWARD _PROTOTYPE( int do_vcopy, (message *m_ptr) );
FORWARD _PROTOTYPE( int do_getmap, (message *m_ptr) );
#if (CHIP == INTEL) && VIRT_MEM
FORWARD _PROTOTYPE( int do_adjmap, (message *m_ptr) );
FORWARD _PROTOTYPE( int do_execmap, (message *m_ptr) );
FORWARD _PROTOTYPE( int do_unmap, (message *m_ptr) );
#else
FORWARD _PROTOTYPE( int do_newmap, (message *m_ptr) );
#endif

#if (SHADOWING == 1)
FORWARD _PROTOTYPE( int do_fresh, (message *m_ptr) );
#endif

/*===========================================================================*
 *                                sys_task                                   *
 *===========================================================================*/
PUBLIC void sys_task()
{
/* Main entry point of sys_task.  Get the message and dispatch on type. */
```

```
  register int r;

  while (TRUE) {
        receive(ANY, &m);

        switch (m.m_type) {   /* which system call */
          case SYS_FORK:   r = do_fork(&m);       break;
#if CHIP == INTEL && VIRT_MEM
          case SYS_ADJMAP:        r = do_adjmap(&m);   break;
          case SYS_EXECMAP:       r = do_execmap(&m);break;
#else
          case SYS_NEWMAP:        r = do_newmap(&m); break;
#endif

          case SYS_GETMAP:        r = do_getmap(&m);   break;
          case SYS_EXEC:   r = do_exec(&m);       break;
          case SYS_XIT:       r = do_xit(&m);                 break;
          case SYS_GETSP:  r = do_getsp(&m);      break;
          case SYS_TIMES:  r = do_times(&m);      break;
          case SYS_ABORT:r = do_abort(&m);      break;
#if (SHADOWING == 1)
          case SYS_FRESH: r = do_fresh(&m);      break;
#endif
          case SYS_SENDSIG:       r = do_sendsig(&m);  break;
          case SYS_SIGRETURN: r = do_sigreturn(&m);       break;
          case SYS_KILL:    r = do_kill(&m);        break;
          case SYS_ENDSIG:        r = do_endsig(&m);   break;
          case SYS_COPY:   r = do_copy(&m);       break;
        case SYS_VCOPY:         r = do_vcopy(&m);    break;
          case SYS_GBOOT:         r = do_gboot(&m);    break;
          case SYS_MEM:    r = do_mem(&m);                 break;
          case SYS_UMAP:  r = do_umap(&m);      break;
          case SYS_TRACE: r = do_trace(&m);       break;
          default:                r = E_BAD_FCN;
          panic("SYSTASK got invalid request: ", m.m_type);
        }

        m.m_type = r;                   /* 'r' reports status of call */
        send(m.m_source, &m);           /* send reply to caller */
  }
}


/*===========================================================================*
 *                                do_fork                                    *
 *===========================================================================*/
PRIVATE int do_fork(m_ptr)
register message *m_ptr;        /* pointer to request message */
{
/* Handle sys_fork().  m_ptr->PROC1 has forked.  The child is m_ptr->PROC2. */

#if (CHIP == INTEL)
  reg_t old_ldt_sel;
  int old_flags;
#endif
  register struct proc *rpc;
  struct proc *rpp;
  phys_clicks child_base;

  if (!isoksusern(m_ptr->PROC1) || !isoksusern(m_ptr->PROC2))
        return(E_BAD_PROC);
  rpp = proc_addr(m_ptr->PROC1);
  rpc = proc_addr(m_ptr->PROC2);
child_base= (phys_clicks)m_ptr->m1_p1;

#if CHIP == INTEL && VIRT_MEM
  /* On a vm system we have to check available memory first. */
  if (vm_not_alloc < rpp->p_map[T].mem_len + rpp->p_map[D].mem_len +
        rpp->p_map[S].mem_len)
  {
```

```
                return ENOMEM;      /* Bad luck */
  }
#endif

  /* Copy parent 'proc' struct to child. */
#if (CHIP == INTEL)
  old_ldt_sel = rpc->p_ldt_sel;     /* stop this being obliterated by copy */
  *rpc = *rpp;                                  /* copy 'proc' struct */
  rpc->p_ldt_sel = old_ldt_sel;
  rpc->p_map[T].mem_phys = child_base;
  if (rpc->p_map[T].mem_len)        /* Separate I&D */
                rpc->p_map[D].mem_phys = rpc->p_map[T].mem_phys +
                                rpc->p_map[T].mem_vir + rpc->p_map[T].mem_len;
  else
                rpc->p_map[D].mem_phys = child_base;
  rpc->p_map[S].mem_phys = rpc->p_map[D].mem_phys;
  alloc_segments(rpc);
  #if CHIP == INTEL && VIRT_MEM         /* Make pages shared or copy on access */
    vm_fork(rpp, child_base);
    vm_not_alloc -= rpc->p_map[T].mem_len + rpc->p_map[D].mem_len + rpc->
                p_map[S].mem_len;
    old_flags = rpc->p_flags;           /* save the previous value of the flags */
  rpc->p_flags &= ~NO_MAP;
  if (old_flags != 0 && rpc->p_flags == 0) lock_ready(rpc);
#else
    /* HACK because structure copy is or was slow. */
    phys_copy( (phys_bytes)rpp, (phys_bytes)proc_addr(m_ptr->PROC2),
       (phys_bytes)sizeof(struct proc));
#endif

  rpc->p_nr = m_ptr->PROC2;     /* this was obliterated by copy */

#if (SHADOWING == 0)
  rpc->p_flags |= NO_MAP;           /* inhibit the process from running */
#endif

  rpc->p_flags &= ~(PENDING | SIG_PENDING | P_STOP);

  /* Only 1 in group should have PENDING, child does not inherit trace status*/
  sigemptyset(&rpc->p_pending);
  rpc->p_pendcount = 0;
  rpc->p_pid = m_ptr->PID;          /* install child's pid */
  rpc->p_reg.retreg = 0;                /* child sees pid = 0 to know it is child */

  rpc->user_time = 0;                   /* set all the accounting times to 0 */
  rpc->sys_time = 0;
  rpc->child_utime = 0;
  rpc->child_stime = 0;

#if (SHADOWING == 1)
  rpc->p_nflips = 0;
  mkshadow(rpp, (phys_clicks)m_ptr->m1_p1);             /* run child first */
#endif

  return(OK);
}

/*===========================================================================*
 *                                do_getmap                                  *
 *===========================================================================*/
PRIVATE int do_getmap(m_ptr)
message *m_ptr;                         /* pointer to request message */
{
/* Handle sys_getmap().  Report the memory map to MM. */

  register struct proc *rp, *rdst;
  phys_bytes src_phys, dst_phys, pn;
  vir_bytes vmm, vsys, vn;
  int caller;                           /* where the map has to be stored */
  int k;                                /* process whose map is to be loaded */
```

```
  struct mem_map *map_ptr;      /* virtual address of map inside caller (MM) */

#if DEBUG & 256
 { printW(); printf("doing do_getmap\n"); }
#endif
  /* Extract message parameters and copy new memory map to MM. */
  caller = m_ptr->m_source;
  k = m_ptr->PROC1;
  map_ptr = (struct mem_map *) m_ptr->MEM_PTR;

  if (!isokprocn(k))
                panic("do_getmap got bad proc: ", m_ptr->PROC1);

  rp = proc_addr(k);                    /* ptr to entry of the map */
  rdst = proc_addr(caller);             /* ptr to MM's proc entry */

  vn = NR_SEGS * sizeof(struct mem_map);
  pn = vn;
  vmm = (vir_bytes) map_ptr;    /* careful about sign extension */
  vsys = (vir_bytes) rp->p_map; /* again, careful about sign extension */
  if ( (src_phys = umap(proc_ptr, D, vsys, vn)) == 0)
                panic("bad call to sys_getmap (src)", NO_NUM);
  if ( (dst_phys = umap(rdst, D, vmm, vn)) == 0)
                panic("bad call to sys_getmap (dst)", NO_NUM);
  phys_copy(src_phys, dst_phys, pn);

  return(OK);
}

#if (CHIP == INTEL) && VIRT_MEM
/* This function is replaced by do_adjmap and do_execmap */
#else

/*===========================================================================*
 *                                do_newmap                                  *
 *===========================================================================*/
PRIVATE int do_newmap(m_ptr)
message *m_ptr;                         /* pointer to request message */
{
/* Handle sys_newmap().  Fetch the memory map from MM. */

  register struct proc *rp;
  phys_bytes src_phys;
  int caller;                           /* whose space has the new map (usually MM) */
  int k;                                /* process whose map is to be loaded */
  int old_flags;                        /* value of flags before modification */
  struct mem_map *map_ptr;      /* virtual address of map inside caller (MM) */
  #if CHIP == INTEL && VIRT_MEM
   panic("do_newmap should not been called", NO_NUM);
  #endif
  /* Extract message parameters and copy new memory map from MM. */
  caller = m_ptr->m_source;
  k = m_ptr->PROC1;
  map_ptr = (struct mem_map *) m_ptr->MEM_PTR;
  if (!isokprocn(k)) return(E_BAD_PROC);
  rp = proc_addr(k);                    /* ptr to entry of user getting new map */

  /* Copy the map from MM. */
  src_phys = umap(proc_addr(caller), D, (vir_bytes) map_ptr, sizeof(rp->p_map));
  if (src_phys == 0) panic("bad call to sys_newmap", NO_NUM);
  phys_copy(src_phys, vir2phys(rp->p_map), (phys_bytes) sizeof(rp->p_map));

#if (SHADOWING == 0)
#if (CHIP != M68000)
  alloc_segments(rp);
#else
  pmmu_init_proc(rp);
#endif
  old_flags = rp->p_flags;              /* save the previous value of the flags */
  rp->p_flags &= ~NO_MAP;
```

```
  if (old_flags != 0 && rp->p_flags == 0) lock_ready(rp);
#endif
#if (CHIP == INTEL)
  if (rp->p_map[D].mem_phys != rp->p_map[S].mem_phys ||
                rp->p_map[D].mem_vir + rp->p_map[D].mem_len > rp->p_map[S].mem_vir)
                panic("newmap: invalid map for process ", proc_number(rp));
#endif


  return(OK);
}
#endif /* (CHIP == INTEL) && VIRT_MEM */

/*===========================================================================*
 *                                do_exec                                    *
 *===========================================================================*/
PRIVATE int do_exec(m_ptr)
register message *m_ptr;        /* pointer to request message */
{
/* Handle sys_exec().  A process has done a successful EXEC. Patch it up. */

  register struct proc *rp;
  reg_t sp;                     /* new sp */
  phys_bytes phys_name;
  char *np;
#define NLEN (sizeof(rp->p_name)-1)

  if (!isoksusern(m_ptr->PROC1)) return E_BAD_PROC;
  /* PROC2 field is used as flag to indicate process is being traced */
  if (m_ptr->PROC2) cause_sig(m_ptr->PROC1, SIGTRAP);
  sp = (reg_t) m_ptr->STACK_PTR;
  rp = proc_addr(m_ptr->PROC1);
  rp->p_reg.sp = sp;            /* set the stack pointer */
#if (CHIP == M68000)
  rp->p_splow = sp;             /* set the stack pointer low water */
#ifdef FPP
  /* Initialize fpp for this process */
  fpp_new_state(rp);
#endif
#endif
  rp->p_reg.pc = (reg_t) m_ptr->IP_PTR;     /* set pc */
  rp->p_alarm = 0;              /* reset alarm timer */
  rp->p_flags &= ~RECEIVING; /* MM does not reply to EXEC call */
  if (rp->p_flags == 0) lock_ready(rp);

  /* Save command name for debugging, ps(1) output, etc. */
  phys_name = numap(m_ptr->m_source, (vir_bytes) m_ptr->NAME_PTR,
                                                        (vir_bytes) NLEN);
  if (phys_name != 0) {
                phys_copy(phys_name, vir2phys(rp->p_name), (phys_bytes) NLEN);
                for (np = rp->p_name; (*np & BYTE) >= ' '; np++) {}
                *np = 0;
  }
  return(OK);
}


/*===========================================================================*
 *                                do_xit                                     *
 *===========================================================================*/
PRIVATE int do_xit(m_ptr)
message *m_ptr;                 /* pointer to request message */
{
/* Handle sys_xit().  A process has exited. */

  register struct proc *rp, *rc;
  struct proc *np, *xp;
  int parent;                   /* number of exiting proc's parent */
  int proc_nr;                  /* number of process doing the exit */
 #if CHIP == INTEL && VIRT_MEM
```

```
  phys_bytes base, size, top;
 #endif


  parent = m_ptr->PROC1;          /* slot number of parent process */
  proc_nr = m_ptr->PROC2;         /* slot number of exiting process */
  if (!isoksusern(parent) || !isoksusern(proc_nr)) return(E_BAD_PROC);
  rp = proc_addr(parent);
  rc = proc_addr(proc_nr);
  lock();
  rp->child_utime += rc->user_time + rc->child_utime;  /* accum child times */
  rp->child_stime += rc->sys_time + rc->child_stime;
  unlock();
  rc->p_alarm = 0;                        /* turn off alarm timer */
  if (rc->p_flags == 0) lock_unready(rc);

 #if (SHADOWING == 1)
  rmshadow(rc, &base, &size);
  m_ptr->m1_i1 = (int)base;
  m_ptr->m1_i2 = (int)size;
 #endif
 #if VIRT_MEM
  base= (rc->p_map[T].mem_phys) << CLICK_SHIFT;
  top= (rc->p_map[S].mem_phys + rc->p_map[S].mem_vir +
                rc->p_map[S].mem_len) << CLICK_SHIFT;
  if (top < base)
                panic("Stack not above text", NO_NUM);
  size= top-base;
  vm_unmap(base, size, rc->p_map[T].mem_len + rc->p_map[D].mem_len +
                rc->p_map[S].mem_len);
 #if DEBUG || 1
  vm_check_unmapped(base, top);
 #endif
 #endif
  strcpy(rc->p_name, "<noname>");             /* process no longer has a name */

  /* If the process being terminated happens to be queued trying to send a
   * message (i.e., the process was killed by a signal, rather than it doing an
   * EXIT), then it must be removed from the message queues.
   */
  if (rc->p_flags & SENDING) {
                /* Check all proc slots to see if the exiting process is queued. */
                for (rp = BEG_PROC_ADDR; rp < END_PROC_ADDR; rp++) {
                                if (rp->p_callerq == NIL_PROC) continue;
                                if (rp->p_callerq == rc) {
                                                /* Exiting process is on front of this queue. */
                                                rp->p_callerq = rc->p_sendlink;
                                                break;
                                } else {
                                                /* See if exiting process is in middle of queue. */
                                                np = rp->p_callerq;
                                                while ( ( xp = np->p_sendlink) != NIL_PROC)
                                                                if (xp == rc) {
                                                                                np->p_sendlink = xp->p_sendlink;
                                                                                break;
                                                                } else {
                                                                                np = xp;
                                                                }
                                }
                }
  }
 #if (CHIP == M68000) && (SHADOWING == 0)
   pmmu_delete(rc);     /* we're done remove tables */
 #endif

  if (rc->p_flags & PENDING) --sig_procs;
  sigemptyset(&rc->p_pending);
  rc->p_pendcount = 0;
  rc->p_flags = P_SLOT_FREE;
  return(OK);
```

```
}


/*===========================================================================*
 *                                do_getsp                                   *
 *===========================================================================*/
PRIVATE int do_getsp(m_ptr)
register message *m_ptr;        /* pointer to request message */
{
/* Handle sys_getsp().  MM wants to know what sp is. */

  register struct proc *rp;

  if (!isoksusern(m_ptr->PROC1)) return(E_BAD_PROC);
  rp = proc_addr(m_ptr->PROC1);
  m_ptr->STACK_PTR = (char *) rp->p_reg.sp;        /* return sp here (bad type) */
  return(OK);
}


/*===========================================================================*
 *                                do_times                                   *
 *===========================================================================*/
PRIVATE int do_times(m_ptr)
register message *m_ptr;        /* pointer to request message */
{
/* Handle sys_times().  Retrieve the accounting information. */

  register struct proc *rp;

  if (!isoksusern(m_ptr->PROC1)) return E_BAD_PROC;
  rp = proc_addr(m_ptr->PROC1);

  /* Insert the times needed by the TIMES system call in the message. */
  lock();                       /* halt the volatile time counters in rp */
  m_ptr->USER_TIME   = rp->user_time;
  m_ptr->SYSTEM_TIME = rp->sys_time;
  unlock();
  m_ptr->CHILD_UTIME = rp->child_utime;
  m_ptr->CHILD_STIME = rp->child_stime;
  m_ptr->BOOT_TICKS  = get_uptime();
  return(OK);
}


/*===========================================================================*
 *                                do_abort                                   *
 *===========================================================================*/
PRIVATE int do_abort(m_ptr)
message *m_ptr;                 /* pointer to request message */
{
/* Handle sys_abort.  MINIX is unable to continue.  Terminate operation. */
  char monitor_code[64];
  phys_bytes src_phys;

  if (m_ptr->m1_i1 == RBT_MONITOR) {
        /* The monitor is to run user specified instructions. */
        src_phys = numap(m_ptr->m_source, (vir_bytes) m_ptr->m1_p1,
                                        (vir_bytes) sizeof(monitor_code));
        if (src_phys == 0) panic("bad monitor code from", m_ptr->m_source);
        phys_copy(src_phys, vir2phys(monitor_code),
                                        (phys_bytes) sizeof(monitor_code));
        reboot_code = vir2phys(monitor_code);
  }
  wreboot(m_ptr->m1_i1);
  return(OK);                   /* pro-forma (really EDISASTER) */
}


#if (SHADOWING == 1)
```

```
/*===========================================================================*
 *                                do_fresh                                   *
 *===========================================================================*/
PRIVATE int do_fresh(m_ptr)     /* for 68000 only */
message *m_ptr;                 /* pointer to request message */
{
/* Handle sys_fresh.  Start with fresh process image during EXEC. */

  register struct proc *p;
  int proc_nr;                  /* number of process doing the exec */
  phys_clicks base, size;
  phys_clicks c1, nc;

  proc_nr = m_ptr->PROC1;       /* slot number of exec-ing process */
  if (!isokprocn(proc_nr)) return(E_BAD_PROC);
  p = proc_addr(proc_nr);
  rmshadow(p, &base, &size);
  do_newmap(m_ptr);
  c1 = p->p_map[D].mem_phys;
  nc = p->p_map[S].mem_phys - p->p_map[D].mem_phys + p->p_map[S].mem_len;
  c1 += m_ptr->m1_i2;
  nc -= m_ptr->m1_i2;
  zeroclicks(c1, nc);
  m_ptr->m1_i1 = (int)base;
  m_ptr->m1_i2 = (int)size;
  return(OK);
}
#endif /* (SHADOWING == 1) */


/*===========================================================================*
 *                                do_sendsig                                 *
 *===========================================================================*/
PRIVATE int do_sendsig(m_ptr)
message *m_ptr;                 /* pointer to request message */
{
/* Handle sys_sendsig, POSIX-style signal */

  struct sigmsg smsg;
  register struct proc *rp;
  phys_bytes src_phys, dst_phys;
  struct sigcontext sc, *scp;
  struct sigframe fr, *frp;

  if (!isokusern(m_ptr->PROC1)) return(E_BAD_PROC);
  rp = proc_addr(m_ptr->PROC1);

  /* Get the sigmsg structure into our address space.  */
  src_phys = umap(proc_addr(MM_PROC_NR), D, (vir_bytes) m_ptr->SIG_CTXT_PTR,
                        (vir_bytes) sizeof(struct sigmsg));
  if (src_phys == 0)
              panic("do_sendsig can't signal: bad sigmsg address from MM", NO_NUM);
  phys_copy(src_phys, vir2phys(&smsg), (phys_bytes) sizeof(struct sigmsg));

  /* Compute the usr stack pointer value where sigcontext will be stored. */
  scp = (struct sigcontext *) smsg.sm_stkptr - 1;

  /* Copy the registers to the sigcontext structure. */
  memcpy(&sc.sc_regs, &rp->p_reg, sizeof(struct sigregs));

  /* Finish the sigcontext initialization. */
  sc.sc_flags = SC_SIGCONTEXT;

  sc.sc_mask = smsg.sm_mask;

  /* Copy the sigcontext structure to the user's stack. */
  dst_phys = umap(rp, D, (vir_bytes) scp,
                        (vir_bytes) sizeof(struct sigcontext));
  if (dst_phys == 0) return(EFAULT);
  phys_copy(vir2phys(&sc), dst_phys, (phys_bytes) sizeof(struct sigcontext));
```

```
  /* Initialize the sigframe structure. */
  frp = (struct sigframe *) scp - 1;
  fr.sf_scpcopy = scp;
  fr.sf_retadr2= (void (*)()) rp->p_reg.pc;
  fr.sf_fp = rp->p_reg.fp;
  rp->p_reg.fp = (reg_t) &frp->sf_fp;
  fr.sf_scp = scp;
  fr.sf_code = 0;           /* XXX - should be used for type of FP exception */
  fr.sf_signo = smsg.sm_signo;
  fr.sf_retadr = (void (*)()) smsg.sm_sigreturn;

  /* Copy the sigframe structure to the user's stack. */
  dst_phys = umap(rp, D, (vir_bytes) frp, (vir_bytes) sizeof(struct sigframe));
  if (dst_phys == 0) return(EFAULT);
  phys_copy(vir2phys(&fr), dst_phys, (phys_bytes) sizeof(struct sigframe));

  /* Reset user registers to execute the signal handler. */
  rp->p_reg.sp = (reg_t) frp;
  rp->p_reg.pc = (reg_t) smsg.sm_sighandler;

  return(OK);
}

/*===========================================================================*
 *                                do_sigreturn                               *
 *===========================================================================*/
PRIVATE int do_sigreturn(m_ptr)
register message *m_ptr;
{
/* POSIX style signals require sys_sigreturn to put things in order before the
 * signalled process can resume execution
 */

  struct sigcontext sc;
  register struct proc *rp;
  phys_bytes src_phys;

  if (!isokusern(m_ptr->PROC1)) return(E_BAD_PROC);
  rp = proc_addr(m_ptr->PROC1);

  /* Copy in the sigcontext structure. */
  src_phys = umap(rp, D, (vir_bytes) m_ptr->SIG_CTXT_PTR,
                        (vir_bytes) sizeof(struct sigcontext));
  if (src_phys == 0) return(EFAULT);
  phys_copy(src_phys, vir2phys(&sc), (phys_bytes) sizeof(struct sigcontext));

  /* Make sure that this is not just a jmp_buf. */
  if ((sc.sc_flags & SC_SIGCONTEXT) == 0) return(EINVAL);

  /* Fix up only certain key registers if the compiler doesn't use
   * register variables within functions containing setjmp.
   */
  if (sc.sc_flags & SC_NOREGLOCALS) {
                rp->p_reg.retreg = sc.sc_retreg;
                rp->p_reg.fp = sc.sc_fp;
                rp->p_reg.pc = sc.sc_pc;
                rp->p_reg.sp = sc.sc_sp;
                return (OK);
  }
  sc.sc_psw  = rp->p_reg.psw;

#if (CHIP == INTEL)
  /* Don't panic kernel if user gave bad selectors. */
  sc.sc_cs = rp->p_reg.cs;
  sc.sc_ds = rp->p_reg.ds;
  sc.sc_es = rp->p_reg.es;
#if _WORD_SIZE == 4
  sc.sc_fs = rp->p_reg.fs;
  sc.sc_gs = rp->p_reg.gs;
```

```
#endif
#endif

  /* Restore the registers. */
  memcpy(&rp->p_reg, (char *)&sc.sc_regs, sizeof(struct sigregs));

  return(OK);
}

/*===========================================================================*
 *                                do_kill                                    *
 *===========================================================================*/
PRIVATE int do_kill(m_ptr)
register message *m_ptr;        /* pointer to request message */
{
/* Handle sys_kill(). Cause a signal to be sent to a process via MM.
 * Note that this has nothing to do with the kill (2) system call, this
 * is how the FS (and possibly other servers) get access to cause_sig to
 * send a KSIG message to MM
 */

  if (!isokusern(m_ptr->PR)) return(E_BAD_PROC);
  cause_sig(m_ptr->PR, m_ptr->SIGNUM);
  return(OK);
}


/*===========================================================================*
 *                                do_endsig                                  *
 *===========================================================================*/
PRIVATE int do_endsig(m_ptr)
register message *m_ptr;        /* pointer to request message */
{
/* Finish up after a KSIG-type signal, caused by a SYS_KILL message or a call
 * to cause_sig by a task
 */

  register struct proc *rp;

  if (!isokusern(m_ptr->PROC1)) return(E_BAD_PROC);
  rp = proc_addr(m_ptr->PROC1);

  /* MM has finished one KSIG. */
  if (rp->p_pendcount != 0 && --rp->p_pendcount == 0
      && (rp->p_flags &= ~SIG_PENDING) == 0)
          lock_ready(rp);
  return(OK);
}

/*===========================================================================*
 *                                do_copy                                    *
 *===========================================================================*/
PRIVATE int do_copy(m_ptr)
register message *m_ptr;        /* pointer to request message */
{
/* Handle sys_copy().  Copy data for MM or FS. */

  int src_proc, dst_proc, src_space, dst_space;
  vir_bytes src_vir, dst_vir;
  phys_bytes src_phys, dst_phys, bytes;

  /* Dismember the command message. */
  src_proc = m_ptr->SRC_PROC_NR;
  dst_proc = m_ptr->DST_PROC_NR;
  src_space = m_ptr->SRC_SPACE;
  dst_space = m_ptr->DST_SPACE;
  src_vir = (vir_bytes) m_ptr->SRC_BUFFER;
  dst_vir = (vir_bytes) m_ptr->DST_BUFFER;
  bytes = (phys_bytes) m_ptr->COPY_BYTES;
```

```
  /* Compute the source and destination addresses and do the copy. */
#if (SHADOWING == 0)
  if (src_proc == ABS)
        src_phys = (phys_bytes) m_ptr->SRC_BUFFER;
  else {
        if (bytes != (vir_bytes) bytes)
                /* This would happen for 64K segments and 16-bit vir_bytes.
                 * It would happen a lot for do_fork except MM uses ABS
                 * copies for that case.
                 */
                panic("overflow in count in do_copy", NO_NUM);
#endif

        src_phys = umap(proc_addr(src_proc), src_space, src_vir,
                                (vir_bytes) bytes);
#if (SHADOWING == 0)
        }
#endif

#if (SHADOWING == 0)
  if (dst_proc == ABS)
        dst_phys = (phys_bytes) m_ptr->DST_BUFFER;
  else
#endif
        dst_phys = umap(proc_addr(dst_proc), dst_space, dst_vir,
                                (vir_bytes) bytes);

  if (src_phys == 0 || dst_phys == 0) return(EFAULT);
  phys_copy(src_phys, dst_phys, bytes);
  return(OK);
}


/*==========================================================================*
 *                                    do_vcopy                              *
 *==========================================================================*/
PRIVATE int do_vcopy(m_ptr)
register message *m_ptr;        /* pointer to request message */
{
/* Handle sys_vcopy(). Copy multiple blocks of memory */

  int src_proc, dst_proc, vect_s, i;
  vir_bytes src_vir, dst_vir, vect_addr;
  phys_bytes src_phys, dst_phys, bytes;
  cpvec_t cpvec_table[CPVEC_NR];

  /* Dismember the command message. */
  src_proc = m_ptr->m1_i1;
  dst_proc = m_ptr->m1_i2;
  vect_s = m_ptr->m1_i3;
  vect_addr = (vir_bytes)m_ptr->m1_p1;

  if (vect_s > CPVEC_NR) return EDOM;

  src_phys= numap (m_ptr->m_source, vect_addr, vect_s * sizeof(cpvec_t));
  if (!src_phys) return EFAULT;
  phys_copy(src_phys, vir2phys(cpvec_table),
                                        (phys_bytes) (vect_s * sizeof(cpvec_t)));

  for (i = 0; i < vect_s; i++) {
        src_vir= cpvec_table[i].cpv_src;
        dst_vir= cpvec_table[i].cpv_dst;
        bytes= cpvec_table[i].cpv_size;
        src_phys = numap(src_proc,src_vir,(vir_bytes)bytes);
        dst_phys = numap(dst_proc,dst_vir,(vir_bytes)bytes);
        if (src_phys == 0 || dst_phys == 0) return(EFAULT);
        phys_copy(src_phys, dst_phys, bytes);
  }
  return(OK);
}
```

```
/*===========================================================================*
 *                                do_gboot                                   *
 *===========================================================================*/
PUBLIC struct bparam_s boot_parameters;

PRIVATE int do_gboot(m_ptr)
message *m_ptr;                         /* pointer to request message */
{
/* Copy the boot parameters.  Normally only called during fs init. */

  phys_bytes dst_phys;

  dst_phys = umap(proc_addr(m_ptr->PROC1), D, (vir_bytes) m_ptr->MEM_PTR,
                                        (vir_bytes) sizeof(boot_parameters));
  if (dst_phys == 0) panic("bad call to SYS_GBOOT", NO_NUM);
  phys_copy(vir2phys(&boot_parameters), dst_phys,
                                        (phys_bytes) sizeof(boot_parameters));
  return(OK);
}


/*===========================================================================*
 *                                do_mem                                     *
 *===========================================================================*/
PRIVATE int do_mem(m_ptr)
register message *m_ptr;        /* pointer to request message */
{
/* Return the base and size of the next chunk of memory. */

  phys_clicks mem_base, mem_size;

  mem_base= 0;
  mem_size= 0;
  if (chunk_find(&mem_base, &mem_size))
  {
        chunk_del(mem_base, mem_size);
  }
  m_ptr->m1_i1= mem_base;
  m_ptr->m1_i2= mem_size;
  return OK;
}




/*===========================================================================*
 *                                do_umap                                    *
 *===========================================================================*/
PRIVATE int do_umap(m_ptr)
register message *m_ptr;        /* pointer to request message */
{
/* Same as umap(), for non-kernel processes. */

  m_ptr->SRC_BUFFER = umap(proc_addr((int) m_ptr->SRC_PROC_NR),
                (int) m_ptr->SRC_SPACE,
                (vir_bytes) m_ptr->SRC_BUFFER,
                (vir_bytes) m_ptr->COPY_BYTES);
  return(OK);
}


/*===========================================================================*
 *                                do_trace                                   *
 *===========================================================================*/
#define TR_PROCNR               (m_ptr->m2_i1)
#define TR_REQUEST              (m_ptr->m2_i2)
#define TR_ADDR                 ((vir_bytes) m_ptr->m2_l1)
#define TR_DATA                 (m_ptr->m2_l2)
```

```
#define TR_VLSIZE  ((vir_bytes) sizeof(long))

PRIVATE int do_trace(m_ptr)
register message *m_ptr;
{
/* Handle the debugging commands supported by the ptrace system call
 * The commands are:
 * T_STOP               stop the process
 * T_OK                 enable tracing by parent for this process
 * T_GETINS             return value from instruction space
 * T_GETDATA            return value from data space
 * T_GETUSER            return value from user process table
 * T_SETINS             set value from instruction space
 * T_SETDATA            set value from data space
 * T_SETUSER            set value in user process table
 * T_RESUME             resume execution
 * T_EXIT               exit
 * T_STEP               set trace bit
 *
 * The T_OK and T_EXIT commands are handled completely by the memory manager,
 * all others come here.
 */

  register struct proc *rp;
  phys_bytes src, dst;
  int i;

  rp = proc_addr(TR_PROCNR);
  if (rp->p_flags & P_SLOT_FREE) return(EIO);
  switch (TR_REQUEST) {
  case T_STOP:                                  /* stop process */
                if (rp->p_flags == 0) lock_unready(rp);
                rp->p_flags |= P_STOP;
                rp->p_reg.psw &= ~TRACEBIT;/* clear trace bit */
                return(OK);

  case T_GETINS:                        /* return value from instruction space */
                if (rp->p_map[T].mem_len != 0) {
                        if ((src = umap(rp, T, TR_ADDR, TR_VLSIZE)) == 0) return(EIO);
                        phys_copy(src, vir2phys(&TR_DATA), (phys_bytes) sizeof(long));
                        break;
                }
                /* Text space is actually data space - fall through. */

  case T_GETDATA:                       /* return value from data space */
                if ((src = umap(rp, D, TR_ADDR, TR_VLSIZE)) == 0) return(EIO);
                phys_copy(src, vir2phys(&TR_DATA), (phys_bytes) sizeof(long));
                break;

  case T_GETUSER:                       /* return value from process table */
                if ((TR_ADDR & (sizeof(long) - 1)) != 0 ||
                   TR_ADDR > sizeof(struct proc) - sizeof(long))
                        return(EIO);
                TR_DATA = *(long *) ((char *) rp + (int) TR_ADDR);
                break;

  case T_SETINS:                        /* set value in instruction space */
                if (rp->p_map[T].mem_len != 0) {
                        if ((dst = umap(rp, T, TR_ADDR, TR_VLSIZE)) == 0) return(EIO);
                        phys_copy(vir2phys(&TR_DATA), dst, (phys_bytes) sizeof(long));
                        TR_DATA = 0;
                        break;
                }
                /* Text space is actually data space - fall through. */

  case T_SETDATA:                               /* set value in data space */
                if ((dst = umap(rp, D, TR_ADDR, TR_VLSIZE)) == 0) return(EIO);
                phys_copy(vir2phys(&TR_DATA), dst, (phys_bytes) sizeof(long));
                TR_DATA = 0;
                break;
```

```
  case T_SETUSER:                       /* set value in process table */
                if ((TR_ADDR & (sizeof(reg_t) - 1)) != 0 ||
                    TR_ADDR > sizeof(struct stackframe_s) - sizeof(reg_t))
                        return(EIO);
                i = (int) TR_ADDR;
#if (CHIP == INTEL)
                /* Altering segment registers might crash the kernel when it
                 * tries to load them prior to restarting a process, so do
                 * not allow it.
                 */
                if (i == (int) &((struct proc *) 0)->p_reg.cs ||
                    i == (int) &((struct proc *) 0)->p_reg.ds ||
                    i == (int) &((struct proc *) 0)->p_reg.es ||
#if _WORD_SIZE == 4
                    i == (int) &((struct proc *) 0)->p_reg.gs ||
                    i == (int) &((struct proc *) 0)->p_reg.fs ||
#endif
                    i == (int) &((struct proc *) 0)->p_reg.ss)
                        return(EIO);
#endif
                if (i == (int) &((struct proc *) 0)->p_reg.psw)
                        /* only selected bits are changeable */
                        SETPSW(rp, TR_DATA);
                else
                        *(reg_t *) ((char *) &rp->p_reg + i) = (reg_t) TR_DATA;
                TR_DATA = 0;
                break;

  case T_RESUME:                    /* resume execution */
                rp->p_flags &= ~P_STOP;
                if (rp->p_flags == 0) lock_ready(rp);
                TR_DATA = 0;
                break;

  case T_STEP:                          /* set trace bit */
                rp->p_reg.psw |= TRACEBIT;
                rp->p_flags &= ~P_STOP;
                if (rp->p_flags == 0) lock_ready(rp);
                TR_DATA = 0;
                break;

  default:
                return(EIO);
  }
  return(OK);
}

/*===========================================================================*
 *                                cause_sig                                  *
 *===========================================================================*/
PUBLIC void cause_sig(proc_nr, sig_nr)
int proc_nr;                            /* process to be signalled */
int sig_nr;                     /* signal to be sent, 1 to _NSIG */
{
/* A task wants to send a signal to a process.  Examples of such tasks are:
 *   TTY wanting to cause SIGINT upon getting a DEL
 *   CLOCK wanting to cause SIGALRM when timer expires
 * FS also uses this to send a signal, via the SYS_KILL message.
 * Signals are handled by sending a message to MM.  The tasks don't dare do
 * that directly, for fear of what would happen if MM were busy.  Instead they
 * call cause_sig, which sets bits in p_pending, and then carefully checks to
 * see if MM is free.  If so, a message is sent to it.  If not, when it becomes
 * free, a message is sent.  The process being signaled is blocked while MM
 * has not seen or finished with all signals for it.  These signals are
 * counted in p_pendcount, and the SIG_PENDING flag is kept nonzero while
 * there are some.  It is not sufficient to ready the process when MM is
 * informed, because MM can block waiting for FS to do a core dump.
 */
```

```
  register struct proc *rp, *mmp;

  rp = proc_addr(proc_nr);
  if (sigismember(&rp->p_pending, sig_nr))
                return;                         /* this signal already pending */
  sigaddset(&rp->p_pending, sig_nr);
  ++rp->p_pendcount;                /* count new signal pending */
  if (rp->p_flags & PENDING)
                return;                         /* another signal already pending */
  if (rp->p_flags == 0) lock_unready(rp);
  rp->p_flags |= PENDING | SIG_PENDING;
  ++sig_procs;                                  /* count new process pending */

  mmp = proc_addr(MM_PROC_NR);
  if ( ((mmp->p_flags & RECEIVING) == 0) || mmp->p_getfrom != ANY) return;
  inform();
}


/*===========================================================================*
 *                                              inform                       *
 *===========================================================================*/
PUBLIC void inform()
{
/* When a signal is detected by the kernel (e.g., DEL), or generated by a task
 * (e.g. clock task for SIGALRM), cause_sig() is called to set a bit in the
 * p_pending field of the process to signal.  Then inform() is called to see
 * if MM is idle and can be told about it.  Whenever MM blocks, a check is
 * made to see if 'sig_procs' is nonzero; if so, inform() is called.
 */

  register struct proc *rp;

  /* MM is waiting for new input.  Find a process with pending signals. */
  for (rp = BEG_SERV_ADDR; rp < END_PROC_ADDR; rp++)
                if (rp->p_flags & PENDING) {
                        m.m_type = KSIG;
                        m.SIG_PROC = proc_number(rp);
                        m.SIG_MAP = rp->p_pending;
                        sig_procs--;
                        if (lock_mini_send(proc_addr(HARDWARE), MM_PROC_NR, &m) != OK)
                                panic("can't inform MM", NO_NUM);
                        sigemptyset(&rp->p_pending); /* the ball is now in MM's court */
                        rp->p_flags &= ~PENDING;/* remains inhibited by SIG_PENDING */
                        lock_pick_proc();       /* avoid delay in scheduling MM */
                        return;
                }
}


/*===========================================================================*
 *                                              umap                         *
 *===========================================================================*/
PUBLIC phys_bytes umap(rp, seg, vir_addr, bytes)
register struct proc *rp;               /* pointer to proc table entry for process */
int seg;                                /* T, D, or S segment */
vir_bytes vir_addr;                     /* virtual address in bytes within the seg */
vir_bytes bytes;                        /* # of bytes to be copied */
{
/* Calculate the physical memory address for a given virtual address. */

  vir_clicks vc;                        /* the virtual address in clicks */
  phys_bytes pa;                        /* intermediate variables as phys_bytes */
  vir_clicks sp_click;                  /* click where the stack pointer is. */
  vir_clicks adjust;                    /* amount mem_vir has to be lowered to reach sp.*/

#if (CHIP == INTEL)
  phys_bytes seg_base;
#endif
```

```
  /* If 'seg' is D it could really be S and vice versa.  T really means T.
   * If the virtual address falls in the gap,  it causes a problem. On the
   * 8088 it is probably a legal stack reference, since "stackfaults" are
   * not detected by the hardware.  On 8088s, the gap is called S and
   * accepted, but on other machines it is called D and rejected.
   * The Atari ST behaves like the 8088 in this respect.
   */

  #if (CHIP == INTEL)
  if (rp->p_map[D].mem_phys != rp->p_map[S].mem_phys ||
                rp->p_map[D].mem_vir + rp->p_map[D].mem_len > rp->p_map[S].mem_vir)
  {
#if DEBUG
 { printW(); }
#endif
                panic("umap: invalid map for process ", proc_number(rp));
  }
#endif

  if (bytes <= 0 || (long)vir_addr + bytes < vir_addr)
  {
 { printW(); printf("umap(%d, %d, 0x%x, 0x%x) failed\n", proc_number(rp),
                seg, vir_addr, bytes); }
                return( (phys_bytes) 0);
  }

  vc = (vir_addr + bytes - 1) >> CLICK_SHIFT;             /* last click of data */

#if ((CHIP == INTEL) && !VIRT_MEM) || (CHIP == M68000)
  if (seg != T)
                seg = (vc < rp->p_map[D].mem_vir + rp->p_map[D].mem_len ? D : S);
#else
  if (seg != T)
                seg = (vc < rp->p_map[S].mem_vir ? D : S);
#endif
  if (seg == S)             /* Let's adjust the stack segment to (at most)
                             * the stack pointer. */
  {
                if (vir_addr >= rp->p_reg.sp)
                                sp_click= vir_addr >> CLICK_SHIFT;
                else            /* This causes umap to fail ... */
                                sp_click= rp->p_reg.sp >> CLICK_SHIFT;
                if (sp_click < rp->p_map[S].mem_vir)
                {
                                adjust= rp->p_map[S].mem_vir-sp_click;
                                rp->p_map[S].mem_vir -= adjust;
                                rp->p_map[S].mem_len += adjust;
#if !SEGMENTED_MEMORY
                                rp->p_map[S].mem_phys -= adjust;
#endif
#if (CHIP == INTEL)
  if (rp->p_map[D].mem_phys != rp->p_map[S].mem_phys ||
                rp->p_map[D].mem_vir + rp->p_map[D].mem_len > rp->p_map[S].mem_vir)
  {
#if DEBUG
 { printW(); }
#endif
                panic("umap: invalid map for process ", proc_number(rp));
  }
#endif
                }
  }

  if((vir_addr>>CLICK_SHIFT) < rp->p_map[seg].mem_vir)
  {
 { printW(); printf("umap(%d, %d, 0x%x, 0x%x) failed\n", proc_number(rp),
                seg, vir_addr, bytes); }
                return( (phys_bytes) 0 );
  }
```

```
if((vir_addr>>CLICK_SHIFT) >= rp->p_map[seg].mem_vir+ rp->p_map[seg].mem_len)
                {
                {printW(); printf("umap(%d, %d, 0x%x, 0x%x) failed\n", proc_number(rp),
                seg, vir_addr, bytes); }
                return( (phys_bytes) 0 );
  }
  if (((vir_addr + bytes + CLICK_SHIFT-1) >> CLICK_SHIFT) >
                                                rp->p_map[seg].mem_vir+ rp->p_map[seg].mem_len)
  {
  { printW(); printf("umap(%d, %d, 0x%x, 0x%x) failed\n", proc_number(rp),
                seg, vir_addr, bytes); }
                return( (phys_bytes) 0 );
  }

#if (CHIP == INTEL)
  seg_base = (phys_bytes) rp->p_map[seg].mem_phys;
  seg_base = seg_base << CLICK_SHIFT;    /* segment origin in bytes */
#endif
  pa = (phys_bytes) vir_addr;
#if (CHIP == INTEL)
  return(seg_base + pa);
#endif

#if (CHIP != M68000)
  pa -= rp->p_map[seg].mem_vir << CLICK_SHIFT;
  return(seg_base + pa);
#endif
#if (CHIP == M68000)
#if (SHADOWING == 0)
  pa -= (phys_bytes)rp->p_map[seg].mem_vir << CLICK_SHIFT;
  pa += (phys_bytes)rp->p_map[seg].mem_phys << CLICK_SHIFT;
#else
  if (rp->p_shadow && seg != T) {
                pa -= (phys_bytes)rp->p_map[D].mem_phys << CLICK_SHIFT;
                pa += (phys_bytes)rp->p_shadow << CLICK_SHIFT;
  }
#endif
  return(pa);
#endif
}


/*===========================================================================*
 *                                numap                                      *
 *===========================================================================*/
PUBLIC phys_bytes numap(proc_nr, vir_addr, bytes)
int proc_nr;                            /* process number to be mapped */
vir_bytes vir_addr;             /* virtual address in bytes within D seg */
vir_bytes bytes;                /* # of bytes required in segment  */
{
/* Do umap() starting from a process number instead of a pointer.  This
 * function is used by device drivers, so they need not know about the
 * process table.  To save time, there is no 'seg' parameter. The segment
 * is always D.
 */

  return(umap(proc_addr(proc_nr), D, vir_addr, bytes));
}


#if (CHIP == INTEL)
/*===========================================================================*
 *                                alloc_segments                             *
 *===========================================================================*/
PUBLIC void alloc_segments(rp)
register struct proc *rp;
{
/* This is called only by do_newmap, but is broken out as a separate function
 * because so much is hardware-dependent.
```

```
 */

  phys_bytes code_bytes;
  phys_bytes data_bytes;
  int privilege;

  if (protected_mode) {
                data_bytes = (phys_bytes) (rp->p_map[S].mem_vir + rp->p_map[S].mem_len)
                        << CLICK_SHIFT;
                if (rp->p_map[T].mem_len == 0)
                        code_bytes = data_bytes;            /* common I&D, poor protect */
                else
                                        code_bytes = ((phys_bytes) rp->p_map[T].mem_len+
                                        rp->p_map[T].mem_vir) << CLICK_SHIFT;

                privilege = istaskp(rp) ? TASK_PRIVILEGE : USER_PRIVILEGE;
                init_codeseg(&rp->p_ldt[CS_LDT_INDEX],
                                ((phys_bytes) rp->p_map[T].mem_phys) << CLICK_SHIFT,
                                        code_bytes, privilege);
                init_dataseg(&rp->p_ldt[DS_LDT_INDEX],
                                ((phys_bytes) rp->p_map[D].mem_phys) << CLICK_SHIFT,
                                        data_bytes, privilege);
                rp->p_reg.cs = (CS_LDT_INDEX * DESC_SIZE) | TI | privilege;
#if _WORD_SIZE == 4
                rp->p_reg.gs =
                rp->p_reg.fs =
#endif
                rp->p_reg.ss =
                rp->p_reg.es =
                rp->p_reg.ds = (DS_LDT_INDEX*DESC_SIZE) | TI | privilege;
  } else {
                rp->p_reg.cs = click_to_hclick(rp->p_map[T].mem_phys);
                rp->p_reg.ss =
                rp->p_reg.es =
                rp->p_reg.ds = click_to_hclick(rp->p_map[D].mem_phys);
  }
}
#endif /* (CHIP == INTEL) */



#if (CHIP == INTEL) && VIRT_MEM

/*===========================================================================*
 *                                    do_adjmap                              *
 *===========================================================================*/
PRIVATE int do_adjmap(m_ptr)
message *m_ptr;                         /* pointer to request message */
{
/* Handle sys_adjmap().  Change the memory map for MM. */

  int caller;                   /* where the map has to be stored */
  int k;                        /* process whose map is to be loaded */
  vir_clicks data_size;         /* New size of the data segment */
  vir_bytes new_sp;             /* Location of the stack pointer */
  struct mem_map *map_ptr;      /* virtual address of map inside caller (MM) */

  register struct proc *rp;
  vir_clicks data_change, stack_change;
  vir_clicks stack_click;

#if DEBUG & 256
  { printW(); printf("doing do_adjmap\n"); }
#endif
  /* Extract message parameters. */
  caller = m_ptr->m_source;
  k = m_ptr->PROC1;
  data_size= m_ptr->m1_i2;
  new_sp= m_ptr->m1_i3;
  map_ptr = (struct mem_map *) m_ptr->MEM_PTR;
```

```
  if (!isokprocn(k))
        panic("bad proc in do_adjmap: ", m_ptr->PROC1);

  rp = proc_addr(k);            /* ptr to entry of the map */
  if (data_size>rp->p_map[D].mem_len)
        data_change= data_size - rp->p_map[D].mem_len;
  else
        data_change= 0;

  stack_click= (new_sp >> CLICK_SHIFT);
  if (stack_click >= rp->p_map[S].mem_vir+rp->p_map[S].mem_len)
        return EFAULT;                          /* Strange stack pointer */

  /* One click extra to avoid problems on click boundaries */
  stack_click--;

  if (stack_click<rp->p_map[S].mem_vir)
        stack_change= rp->p_map[S].mem_vir - stack_click;
  else
        stack_change= 0;

  /* Let's check if the request memory is really there. */
  if (vm_not_alloc < data_change + stack_change)
        return ENOMEM;

  /* Let's check gaps etc */
  if (rp->p_map[D].mem_vir + rp->p_map[D].mem_len + data_change +
        STACK_SAFETY_CLICKS + stack_change > rp->p_map[S].mem_vir)
        return ENOMEM;

  rp->p_map[D].mem_len += data_change;
  rp->p_map[S].mem_vir -= stack_change;
  rp->p_map[S].mem_len += stack_change;

  vm_not_alloc -= data_change + stack_change;

  do_getmap(m_ptr);             /* Use getmap code to report map to MM */

  return OK;
}


/*===========================================================================*
 *                                do_execmap                                 *
 *===========================================================================*/
PRIVATE int do_execmap(m_ptr)
message *m_ptr;                         /* pointer to request message */
{
/* Handle sys_execmap().  Remove old map and fetch new memory map from MM. */

  register struct proc *rp, *rsrc;
  phys_bytes src_phys, dst_phys, pn;
  vir_bytes vmm, vsys, vn;
  int caller;                   /* whose space has the new map (usually MM) */
  int k;                        /* process whose map is to be loaded */
  int old_flags, i;             /* value of flags before modification */
  phys_bytes base_addr, top_addr;
  struct mem_map *map_ptr;      /* virtual address of map inside caller (MM) */
  int result;
  struct mem_map new_map[NR_SEGS];

#if DEBUG & 256
 { printW(); printf("doing do_execmap\n"); }
#endif
  /* Extract message parameters and copy new memory map from MM. */
  caller = m_ptr->m_source;
  k = m_ptr->PROC1;
  map_ptr = (struct mem_map *) m_ptr->MEM_PTR;
  if (!isokprocn(k)) return(E_BAD_PROC);
```

```
  rp = proc_addr(k);                    /* ptr to entry of user getting new map */
  rsrc = proc_addr(caller);             /* ptr to MM's proc entry */

  vn = NR_SEGS * sizeof(struct mem_map);
  pn = vn;
  vmm = (vir_bytes) map_ptr;        /* careful about sign extension */
  vsys = (vir_bytes) new_map;       /* again, careful about sign extension */
  if ( (src_phys = umap(rsrc, D, vmm, vn)) == 0)
                panic("bad call to sys_newmap (src)", NO_NUM);
  if ( (dst_phys = umap(proc_ptr, D, vsys, vn)) == 0)
                panic("bad call to sys_newmap (dst)", NO_NUM);
  phys_copy(src_phys, dst_phys, pn);

  /* Is there enough physical memory ? */
  if (vm_not_alloc < new_map[T].mem_len + new_map[D].mem_len +
                new_map[S].mem_len)
                return ENOMEM;

  /* Release old memory with do_unmap */
  result= do_unmap(m_ptr);
  if (result != OK)
                return result;

  /* Copy new map */
  for (i= 0; i<NR_SEGS; i++)
                rp->p_map[i]= new_map[i];

#if (CHIP == INTEL)
  if (rp->p_map[D].mem_phys != rp->p_map[S].mem_phys)
                panic("do_execmap: invalid map for process ", proc_number(rp));
#endif

  base_addr= rp->p_map[T].mem_phys << CLICK_SHIFT;
  top_addr= (rp->p_map[S].mem_phys+rp->p_map[S].mem_vir+
                                                        rp->p_map[S].mem_len) << CLICK_SHIFT;
#if DEBUG || 1
  vm_check_unmapped(base_addr, top_addr);
#endif
   /* Allocate physical memory */
#if DEBUG & 256
 { printW(); printf("vm_not_alloc -= %d + %d + %d + %d\n", rp->p_map[T].mem_len,
                rp->p_map[D].mem_len, rp->p_map[S].mem_len, ((top_addr-base_addr +
                        VM_DIRSIZE-1) >> VM_DIRSHIFT)); }
#endif
  vm_not_alloc -= rp->p_map[T].mem_len + rp->p_map[D].mem_len +
                rp->p_map[S].mem_len + ((top_addr-base_addr + VM_DIRSIZE-1) >>
                VM_DIRSHIFT);
  alloc_segments(rp);
  old_flags = rp->p_flags;              /* save the previous value of the flags */
  rp->p_flags &= ~NO_MAP;
  if (old_flags != 0 && rp->p_flags == 0) lock_ready(rp);

  return(OK);
}

/*===========================================================================*
 *                                      do_unmap                                 *
 *===========================================================================*/
PRIVATE int do_unmap(m_ptr)
message *m_ptr;                         /* pointer to request message */
{
/* Handle sys_unmap(). */

  register struct proc *rp;
  phys_bytes base, top, size;
  int k;                                /* process whose map is to be loaded */

#if DEBUG & 256
 { printW(); printf("doing do_unmap\n"); }
#endif
```

```
  k = m_ptr->PROC1;
  if (!isokprocn(k)) return(E_BAD_PROC);
  rp = proc_addr(k);                    /* ptr to entry of user getting new map */

  base= rp->p_map[T].mem_phys << CLICK_SHIFT;
  top= (rp->p_map[S].mem_phys + rp->p_map[T].mem_vir +
                                                        rp->p_map[S].mem_len) << CLICK_SHIFT;
  if (top < base)
                panic("Stack not above text", NO_NUM);
  size= top-base;
  vm_unmap(base, size, rp->p_map[T].mem_len + rp->p_map[D].mem_len +
                rp->p_map[S].mem_len);
#if DEBUG || 1
  vm_check_unmapped(base, top);
#endif

  return(OK);
}
#endif /* (CHIP == INTEL) && VIRT_MEM */
```