Design and Development of a Linear Address Space Allocation for an Operating System

A Dissertation

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

MASTER OF ENGINEERING
(COMPUTER TECHNOLOGY & APPLICATIONS)

By

NAVNEET RAI
College Roll No. 02/CTA/03
Delhi University Roll No. 3002

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.

Navneet Rai
M.E. (Computer Technology & Applications)
College Roll No. 02/CTA/03
Delhi University Roll No. 3002

CERTIFICATE

This is to certify that the Dissertation entitled "**Design and Development of a Linear Address Space Allocation for an Operating System**", submitted by **Navneet Rai** 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	*Dr. Goldie Gabrani*
Head of Department,	**Assistant Professor,**
Department of Computer Engineering,	**Department of Computer Engineering,**
Delhi College of Engineering,	**Delhi College of Engineering,**
Delhi – 110042.	**Delhi – 110042.**

ABSTRACT

This dissertation proposes a design for linear address space management in MINIX operating system on Intel 80x86 architecture based systems. Standard MINIX distribution does not support virtual addressing i.e. linear address space.

In Intel based advanced microprocessors systems paging circuitry takes 32 bit linear address as input and map it into a specific physical address with the help of page tables. When paging circuitry is enabled, all user processes work on linear addresses instead of the actual physical addresses of memory. These linear addresses are mapped to physical addresses by paging circuitry with the help of page tables. To guarantee the protection of kernel address space from access by other user processes and between the user processes within themselves an operating system needs to have a mechanism for linear address space allocation.

Linear address space management is needed to tap the extents of benefits extended by the paging circuitry of modern Intel 80x86 architecture based systems. This dissertation provides a platform for a memory management scheme based on linear addresses in an operating system, which could support processes in a paged memory.

Contents

CHAPTER 1 INTRODUCTION

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

1.1 Virtual memory

Virtual memory is memory that appears to be allocated to application programs. The operating system uses a portion of the hard disk as virtual memory, and swaps data between the hard disk and physical memory. Virtual memory enables multitasking. If your computer needs to run several programs simultaneously, and the memory that all these programs require exceeds the amount of physical memory available, the operating system allocates virtual memory [1] to meet the total memory requirements of each program, and then manages the available physical memory to meet the actual memory requirements at each point in time. Therefore, the amount of virtual memory that is allocated can be much greater than the amount of physical memory that is installed in the computer.

In virtual memory, process is either divided into pages or segments. Virtual memory works in the following way. All addresses generated by a user process are virtual addresses. Address translation hardware checks every such address generated by a user process (read/write, instruction/data) as it tries to map it to the corresponding physical address. If the page/segment that contains the virtual address referenced by the user process is currently resident in physical memory, the translation proceeds as it used to without virtual memory. If, on the other hand, the page/segment is not in physical memory, a Page/segment Fault Exception is generated. This causes the processor to leave the user mode and switch to kernel mode. The kernel responds a Page/Segment Fault Exception by running a routine whose purpose is to bring the referenced page/segment into physical memory (for this, it may have to move some other page to the disk-based backing store).

First, we must introduce a new concept: virtual address space. Virtual address space is the maximum amount of address space available to an application. The virtual address space varies according to the system's architecture and operating system. Virtual address space depends on the architecture because it is the architecture that defines how many bits are available for addressing purposes. Virtual address space also depends on the operating system because the manner in which the operating system was implemented may introduce additional limits over and above those imposed by the architecture.

The word "virtual" in virtual address space means this is the total number of uniquely-addressable memory locations available to an application, but not the amount of physical memory either installed in the system, or dedicated to the application at any given time.

To implement virtual memory, it is necessary for the computer system to have special memory management hardware. This hardware is often known as an MMU (Memory Management Unit).

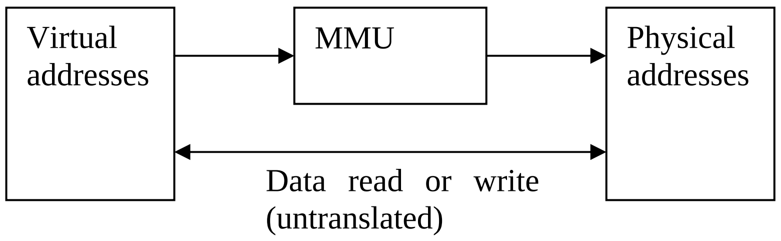


Figure 1.1: Address translation through memory management unit

Figure 1.1 shows translation of translation of virtual address space into physical address space through MMU. In case of virtual memory this MMU can be a paging hardware, segmentation hardware or combination of these two. We will describe paging and segmentation in section 1.2 and 1.3. Combination of these two, i.e. segmentation with paging will be described in chapter 2.

1.2 Paging

Before introducing the concept of paging we will describe some basic terms.

1.2.1 Some terms related to paging

Dirty Bit: Data structure (single-bit suffices) that tells whether the associated page was written after its last swap-in (or creation).

Global Page: A page that is used in more than one program; typically found in multi-programming environment with shared pages.

Logical Address: Address as defined by the architecture. Synonym on Intel architecture: Linear address.

Page: A portion of logical memory that is fixed in size. The start address of a page is an integer multiple of the page size. Antonym: Segment. A logical *page* is placed into a physical *page frame*.

Page Frame: A portion of physical memory that is fixed in size to one page. It starts at a boundary that is evenly divisible by the page size. Total physical memory should be an integral multiple of the page size.

Physical Memory: Main memory actually available *physically* on a processor.

Present Bit: Single-bit data structure that tells, whether the associated page is *resident* or swapped out onto disk.

1.2.2 Paging: Basic method

In paging whole memory is divided into *page frames,* and the program is also divided into *pages.* When the program wants to execute, its pages are loaded in the available free page frames. A data structure called *page table* is used to provide mapping between the pages and the page frame that stores that page.

All modern computer systems consist of hardware for paging support. The basic mechanism of paging [2] is shown in the Figure 1.2. Every address generated by the CPU is divided into two parts: a *virtual page number* and a *page offset.* The page number is used as an index into *page table*. As already explained page table is used for mapping between the pages and page frames that hold that page frame. It contains one entry per page for a process. The entry *x* consists of the frame number of the page frame that holds this page *x*. Through page number we get the *frame number* from the page table. Now as a final step, page offset is added to the frame number to get the actual data. The page table is also stored in a page frame.

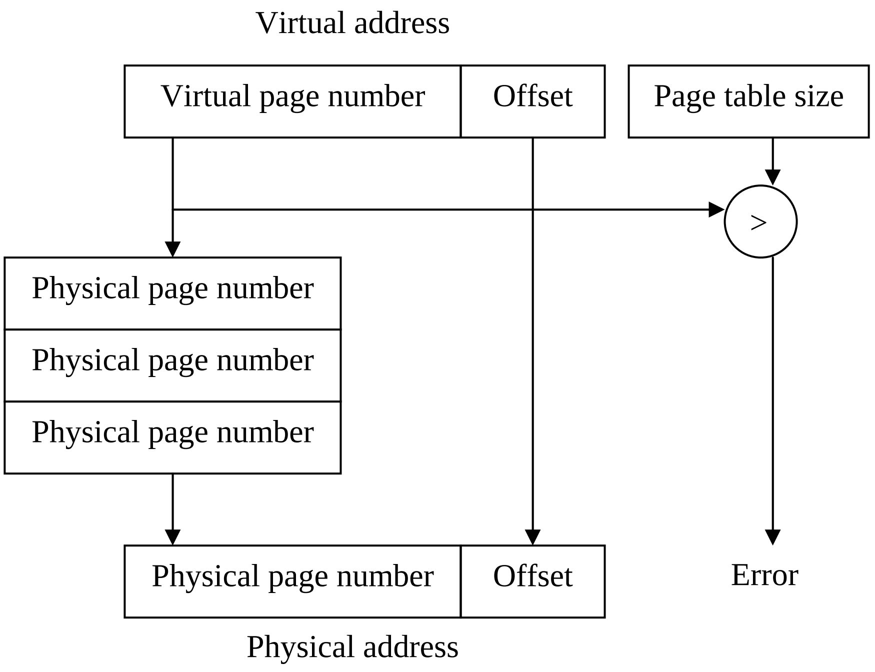


Figure 1.2: Paging mechanism

Figure 1.3(a) shows an example of paging. In this example a process is divided into 4 pages. Figure 1.3 (b) shows main memory holding these pages. Note that pages are scattered randomly in the page frames of main memory. Corresponding page table is shown in the Figure 1.4. In this figure present bit shows whether the page is present in memory or not. Present=1 indicates that page is present in the main memory.

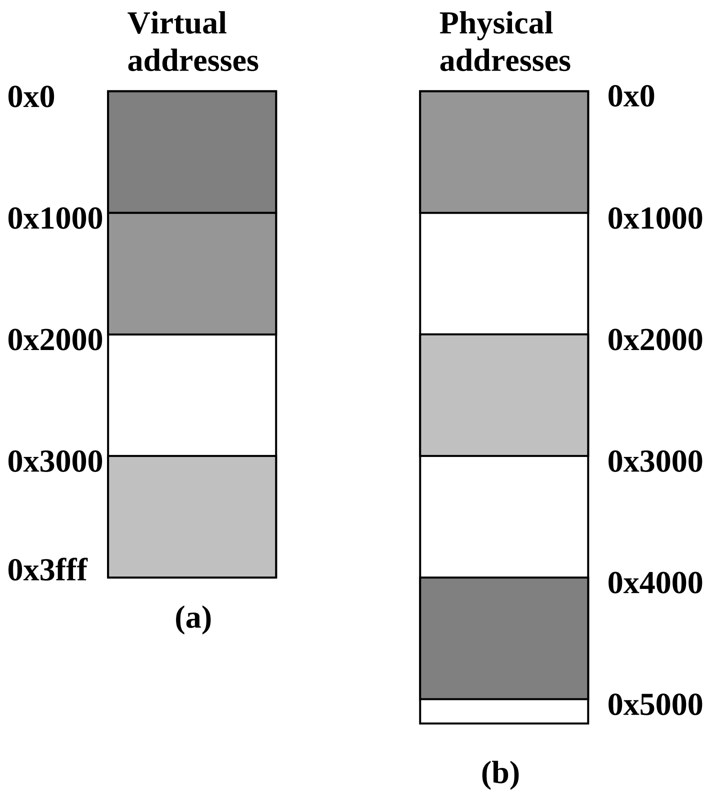


Figure 1.3: (a) A process divided into pages. (b) Main memory holding pages

Virtual page number	Physical page number	Present bit
0	4	1
1	0	1
2	Garbage	0
3	2	1

Figure 1.4: Page table for example of figure 1.3

Paging method described above is single level paging. Consider an example in which process size (logical address space) is 1 GB. Main memory is divided into page frames of size 4 KB. Then total number of pages present in the logical address of the process is equal to $2^{30}/2^{12}$ or 2^{18} (see Figure 1.5). Therefore, the total number entries in the page table are equal to 2^{18}. If an entry in the page table occupies 4 bytes then the page table size is 1 MB and it will occupy 256 page frames. This is an unrealistic approach.

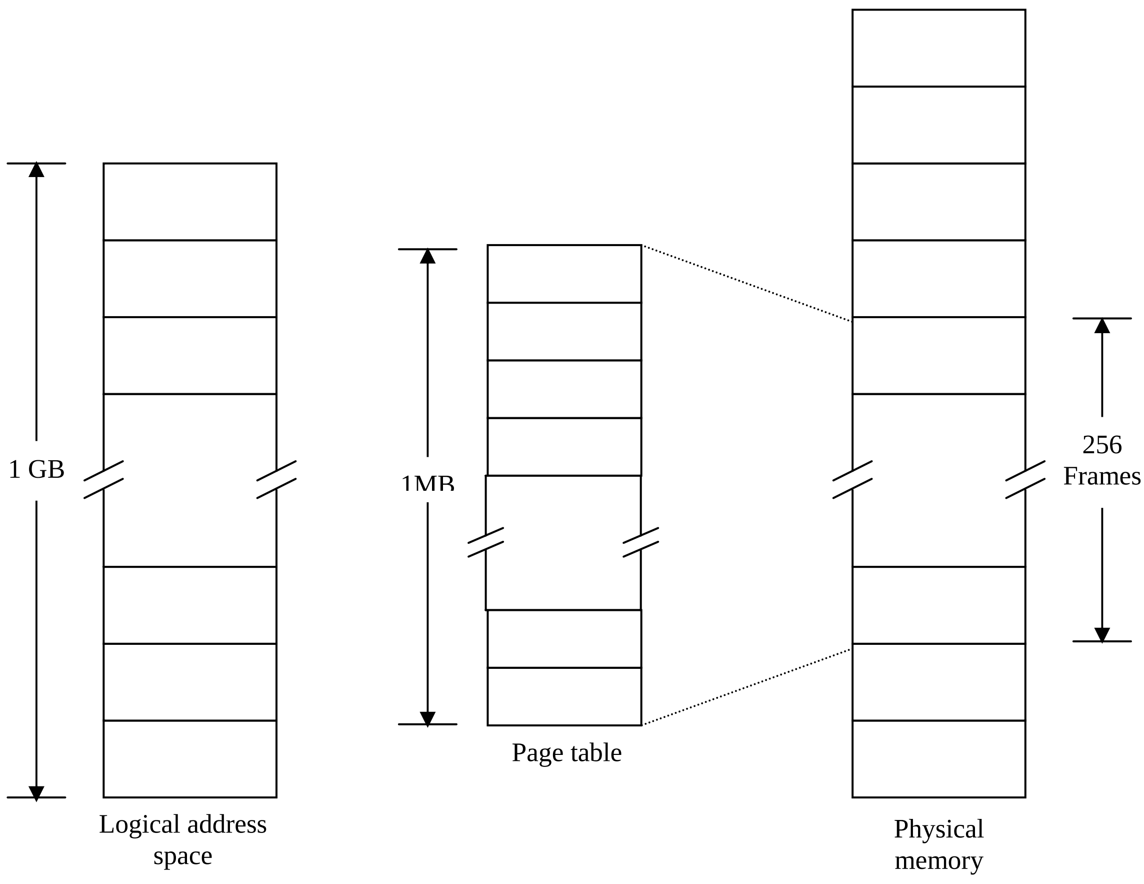


Figure 1.5: An example of large page table

1.2.3 Multilevel paging

To solve the problem of large page tables, page tables are also divided into pages and only some pages of this page table are kept into main memory. This is known as two level paging (see Figure 1.6). To keep track of pages of page table (which pages are present in memory), one more page table is used. This page table is called top-level page table.

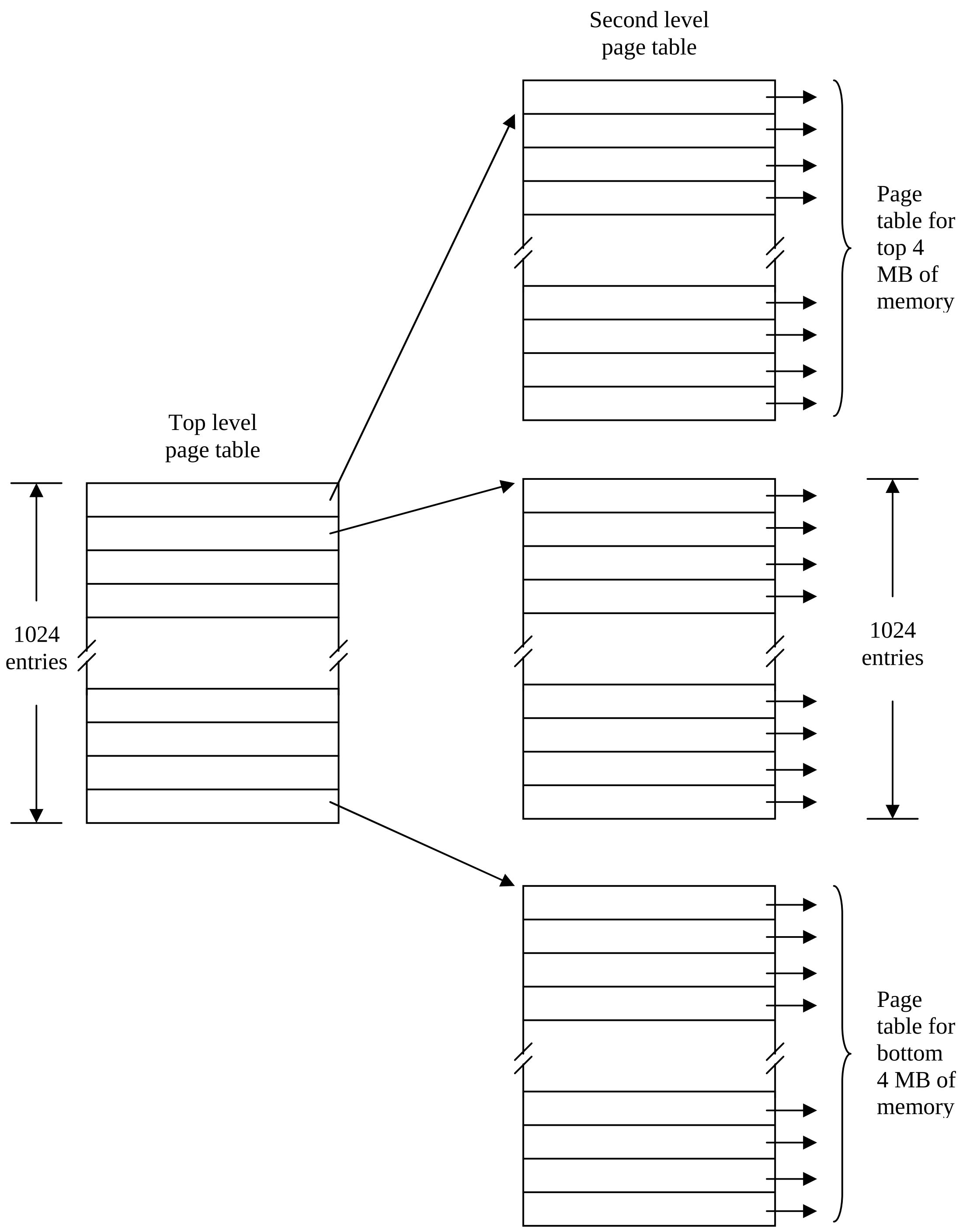

Second level page table
Page table for top 4 MB of memory
Top level page table
1024 entries
1024 entries
Page table for bottom 4 MB of memory

Figure 1.6: An example of two level paging

In case of two-level paging, address generated by the CPU is divided into three parts as shown in Figure 1.7. First part specifies the displacement in top-level page table, second part specifies the displacement in second level page table and the last part specifies the displacement in page frame.

Logical address

Displacement in top level page table	Displacement in second level page table	Displacement in page frame

Figure 1.7: Logical address divided into three parts two level paging

If the size of the top level page table is also very large, further it is divided into pages and logical address is divided into four parts. In this way we can increase the levels of paging. But as the levels of paging increases, the memory access speed decreases. This is due to the fact that accessing each level of page table entry requires some CPU clocks.

1.2.4 Demand Paging

Demand paging is a policy that allocates a page in physical memory only if an address on that page is actually referenced (demanded) in the executing program. Initially, when process wants to execute, required numbers of page frames for this process are reserved for it. But pages of the process are not copied to these page frames. When the process starts execution, CPU tries to fetch the very first instruction of the process. But it is not present in the memory and causes *page fault*. The page fault handler brings this page into a page frame allocated to process.

Thus initially each page in the processes virtual address space causes a page fault. The memory manager should have an efficient page fault handler to handle these page faults.

1.2.4.1 Page fault

A page fault is the sequence of events occurring when a program attempts to access data (or code) that is in its address space, but is not currently located in the system's RAM. The operating system [3] must handle page faults by somehow making the accessed data memory resident, allowing the program to continue operation as if the page fault had never occurred.

In the case of our hypothetical application, the CPU first presents the desired address to the MMU. However, the MMU has no translation for this address. So, it interrupts the

CPU and causes software, known as a page fault handler, to be executed. The page fault handler then determines what must be done to resolve this page fault. It can:
Find where the desired page resides on disk and read it in (this is normally the case if the page fault is for a page of code)
Determine that the desired page is already in RAM (but not allocated to the current process) and reconfigure the MMU to point to it
Point to a special page containing only zeros, and allocate a new page for the process only if the process ever attempts to write to the special page (this is called a copy on write page, and is often used for pages containing zero-initialized data)
Get the desired page from somewhere else (which is discussed in more detail later)
While the first three actions are relatively straightforward, the last one is not. For that, we need to cover some additional topics.

1.2.4.2 Page replacement strategy

What we do require for paged systems is a replacement strategy, and there are a number of these also:

least recently used – replace the page which was used least recently. The assumption is that future behaviour will closely follow recent behaviour. The overhead is that of recording the sequence of access to all pages.

least frequently used – replace the page which has been used least frequently during some immediately preceding time interval. The justification is similar to (i), and the overhead is that of maintaining a usage count for each page. One drawback is that a recently loaded page will generally possess a low usage count and may be replaced inadvisably. A way to avoid this is to inhibit the replacement of pages loaded within the last time interval.

first-in first-out – replace the page that has been resident longest. This is a simpler algorithm (therefore lower overhead). Of course, it completely ignores the possibility that the oldest resident page may be the most heavily referenced.

It is worth noting here that pages which have not been modified to do not need to be written back to secondary storage, and can be replaced very cheaply. Additionally, a record of whether or not the page has been written to can be recorded by the use of a single bit in the corresponding page table entry.

1.2.5 Page size

How big should a page be? This is really a hardware design question, but since it depends on operating system considerations, we will discuss it here. If pages [4] are too large, lots of space will be wasted by internal fragmentation: A process only needs a few bytes, but must take a full page. As a rough estimate, about half of the last page of a process will be wasted on the average. Actually, the average waste will be somewhat larger, if the typical process is small compared to the size of a page. For example, if a page is 8K bytes and the typical process is only 1K, 7/8 of the space will be wasted. Also, the relative amount of waste as a percentage of the space used depends on the size of a typical process. All these considerations imply that as typical processes get bigger and bigger, internal fragmentation becomes less and less of a problem.

On the other hand, with smaller pages it takes more page table entries to describe a given process, leading to space overhead for the page tables, but more importantly time overhead for any operation that manipulates them. In particular, it adds to the time needed to switch form one process to another. The details depend on how page tables are organized. For example, if the page tables are in registers, those registers have to be reloaded. A TLB will need more entries to cover the same size ``working set,'' making it more expensive and require more time to re-load the TLB when changing processes. In short, all current trends point to larger and larger pages in the future.

If space overhead is the *only* consideration, it can be shown that the optimal size of a page is sqrt(2*se*), where *s* is the size of an average process and *e* is the size of a page-table entry. This calculation is based on balancing the space wasted by internal fragmentation against the space used for page tables. This formula should be taken with a big grain of salt however, because it overlooks the time overhead incurred by smaller pages.

1.3 Segmentation

A process generally consists of various data structures like tables, arrays, stacks, variables and so on. These data may grow dynamically during the execution of the process. If the program process is allocated a single address space the dynamic nature of these data may cause some problems like memory overflow, memory overwrite etc. A process consists of various functions, which need not be present in the memory simultaneously. Similarly various files used by a process need not be present in the memory simultaneously. *Segmentation* is a memory management scheme, designed to handle the situations described above.

In this memory management scheme we divide the whole logical address space into different *segments*. Each segment [5] consists of a linear sequence of addresses, from 0 to some maximum. The length of each segment' may be anything from 0 to the maximum allowed. Different segments may, and usually do, have different lengths. Moreover, segment lengths may change during execution. The length of a stack segment may be increased whenever something is pushed onto the stack and decreased whenever something is popped off the stack. These segments can be loaded in different memory areas and all the segments need not be present in the memory at the same time.

Because each segment constitutes a separate address space, different segments can grow or shrink independently, without affecting each other. If a stack in a certain segment needs more address space to grow, it can have it, because there is nothing else in its address space to bump into. Of course, a segment can fill up but segments are usually very large, so this occurrence is rare. A process with five different segments is shown in the Figure 1.8.

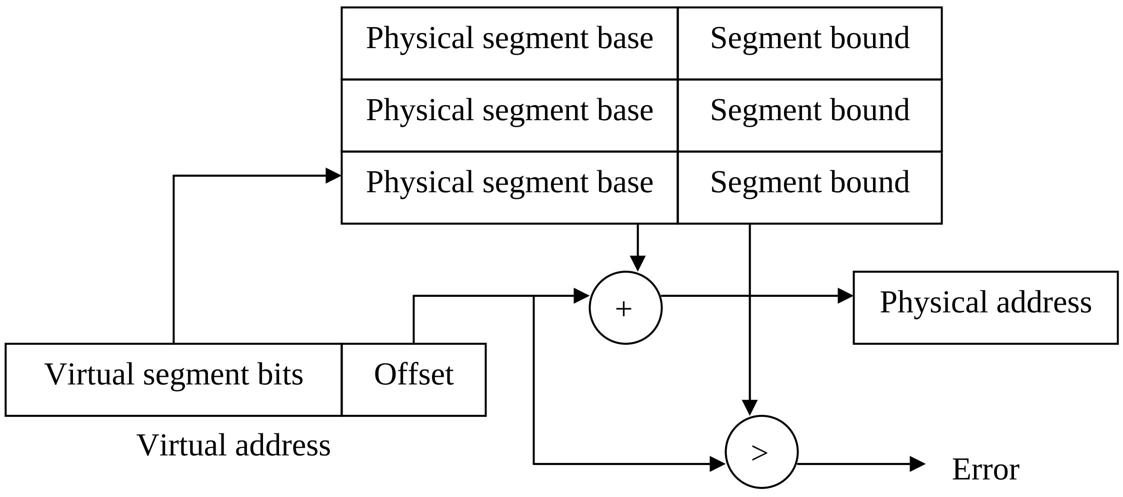


Figure 1.8: Segmentation mechanism

Figure 1.9 shows an example of segmentation. In this example virtual address space is divided into 4 segments. Out of these 4 segments, only three are present in main memory.

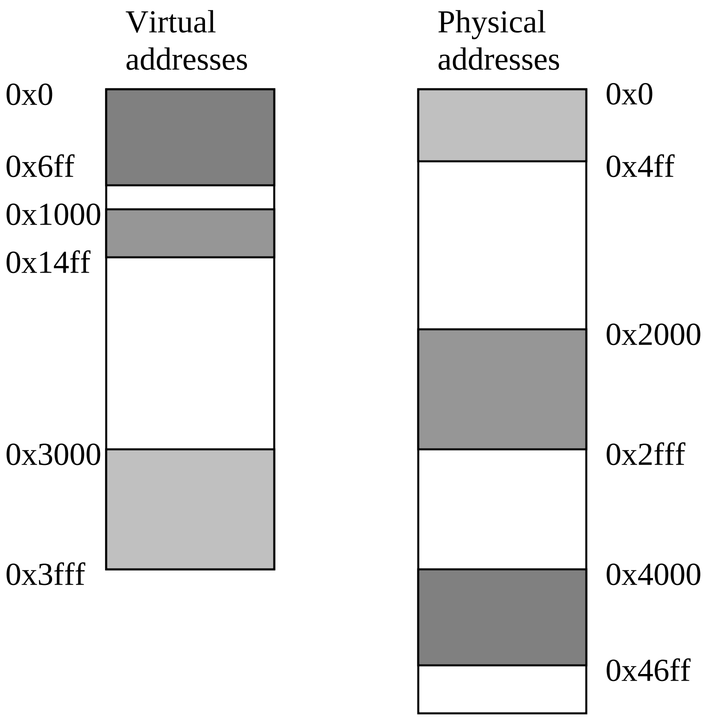


Figure 1.9: An example of segmentation

Virtual segment number	Physical segment base	Segment bound
Code	0x4000	0x700
Data	0	0x500
-	0	0
Stack	0x2000	0x1000

Figure 1.10: Segment table for example of figure 1.9

Each segment gets mapped to a contiguous location in physical memory, but there may be gaps between segments. These gaps allow heap and stack segments of a process to grow by changing the segment bound in the table entry. Also, by adding a protection mode to each segment, we can have a finer control of segment accesses. For example, the code segment should be set to read-only (only execution and loads are allowed). Data and stack segments are set to read-write (stores allowed).

1.3.1 Segment fault handling

In order to achieve replacement for segmented systems, the main objective is the same for paging. That is, to attempt to replace the segment this is least likely to be referenced in the immediate future. One might therefore expect the same policies to be applicable, and this is indeed the case, but with one major qualification. This qualification naturally arises from the fact that the segments to be placed are – unlike their paged counterparts – of varying size. Therefore, the segments that are to be candidates for relegation to secondary memory are influenced by the size of the incoming segment. If a small segment is to be brought in, then only a small segment need be replaced. However, if the incoming segment is large then a large segment must be replaced or alternatively, several smaller ones.

Possibly the simplest technique is to replace a single segment (if one exists) which, together with adjacent gaps, will free enough space for the incoming segment. If there are several such segments, then a policy such as *least recently used* – described above for paged systems – may be employed to discriminate between the candidates. If no single segment is large enough to create sufficient space, then several segments have to be replaced. A possible choice is the smallest set of contiguous segments will free the space required.

The danger with this type of approach is that the segment (or segments) being replaced on the basis of size only, may be referenced again very shortly after their replacement. Selecting segments purely on a *least recently used* basis (for example) can reduce this danger, but since the selected segments are not likely to be contiguous some *compaction* of memory will be required. This technique involves an algorithm, which regularly trawls through the entire memory and “pushes” all the segments up to become contiguous blocks at the head of memory. This is very similar to the idea of defragmentation on the disk system, but in main memory is often referred to a *garbage collection*.

1.4 Problem description

This dissertation aims to propose a design for linear address space allocation for an operating system. For this purpose, ideally we have to choose an operating system which doesn’t already have any support for virtual addressing, i.e., which works on physical address space, instead on linear address space.

Andrew S. Tanenbaum’s Minix is used for this purpose in this project. Minix is a very popular operating system among students, as it is a open source operating system without much licensing restrictions. It is a Unix look-alike operating system, and its source code is universally available for study and modification. Many successful projects including Linux have there origin in Minix, due to these reasons only.

Standard Minix does not have any native virtual memory. It allocates physical address space to the user processes directly. This physical space allocation is handled by a memory manager server in Minix. To support virtual memory in Minix we have to design and develop a suitable mechanism for handling the linear address space allocation to the user processes.

The hardware platform used in this project is Intel 80x86 architecture. It is the most widely used and most easily available computer architecture present today. This architecture includes 80386, 80486 and Pentium family of processors which are in wide spread use throughout the world.

The details of this architecture are widely published and could easily be referenced for the purpose of this project. This makes Intel 80x86 a suitable architecture for proposing and testing this design.

Intel 80x86 architecture supports two level paging. The linear address space is needed to be assigned to the processes for using virtual memory which is mapped to the physical address space by paging unit using page global directory and page tables.

To allocate the linear address space instead of physical address space to processes, we need to make some changes to the memory manager of Minix. This modification includes the changes in the system calls of memory manager used to allocate the memory to the user processes.

To implement these changes some new functions at kernel level are also required. These functions are used to manage the page tables to provide the mapping of linear address space on the physical memory at kernel level. For this purpose, kernel of standard Minix should be modified and recompiled.

Selecting modular kernel based Minix; instead of monolithic kernel based operating system like Linux will make it easier to perform these changes. To make any change to a monolithic kernel would have needed 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.

Selecting a widely used operating system like Linux would have been relatively tougher as 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.

The primary aim of the project described in this dissertation is to make the modifications described above in the memory manager and to add the necessary functions at the kernel level to support the linear address space in Minix.

1.5 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. Minix memory management is also discussed in details.

Chapter 4: Provides the basic mechanism to support the linear address space management in Minix. The design issues and the modification in memory manager are discussed in detail. It includes the various routines to update the page tables.

Chapter 5: All the routines involved in the mechanism discussed in chapter 4 are provided 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 MEMORY MANAGEMENT IN INTEL

2.1 Address translation in Intel

The 80x86 transforms [6] 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 2.1 illustrates the two translations at a high level of abstraction.

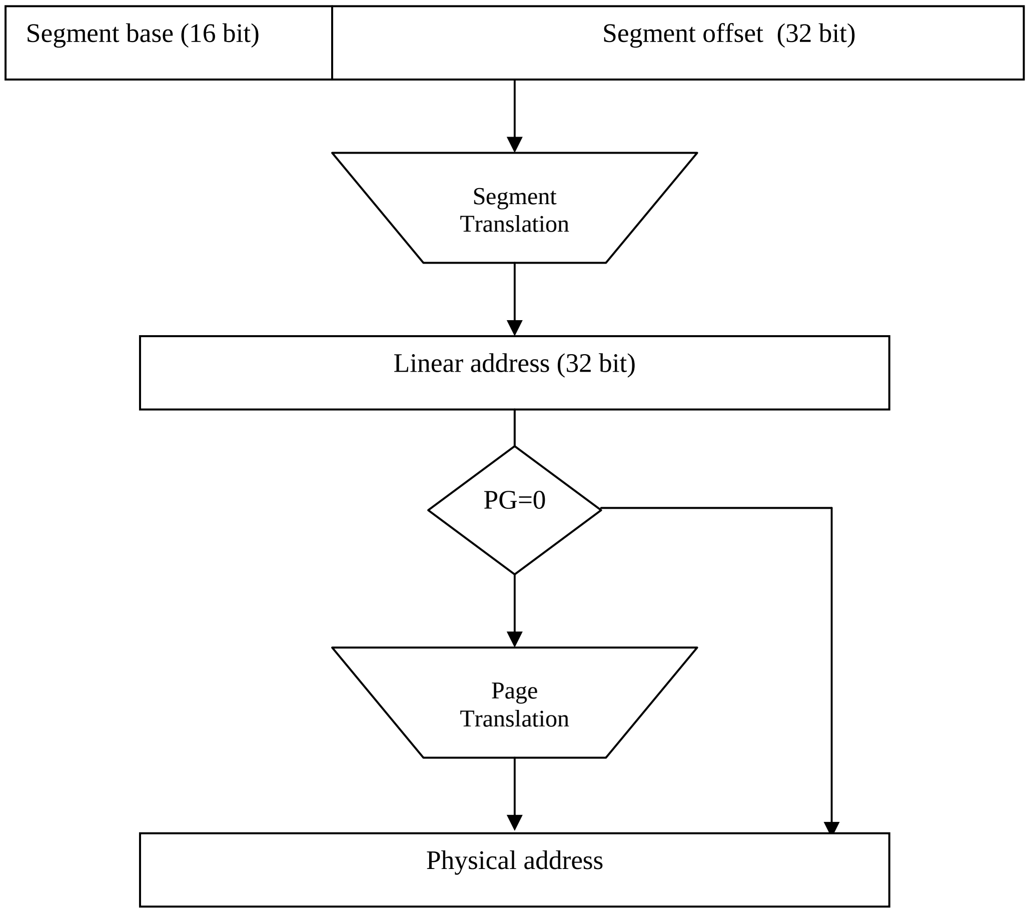


Figure 2.1: Address translation in Intel

2.2 Segmentation with Paging

2.2.1 Segment translation

Pentium has 16K independent segments, each holding up to 1 billion 32-bit words. Although there are fewer segments, the larger segment size is far more important, as few programs need more than 1000 segments, but many programs need segments holding megabytes.

The heart of the Pentium [7] virtual memory consists of two tables, the LDT (Local Descriptor Table) and the GDT (Global Descriptor Table). Each program has its own LDT, but there is a single GDT, shared by all the programs on the computer. The LDT describes segments local to each program, including its code, data, stack, and so on, whereas the GDT describes system segments, including the operating system itself.

To access a segment, a Pentium program first loads a selector for that segment into one of the machine's six segment registers. During execution, the CS register holds the selector for the code segment and the DS register holds the selector for the data segment. The other segment registers are less important. Each selector is a 16-bit number, as shown in Figure 2.2.

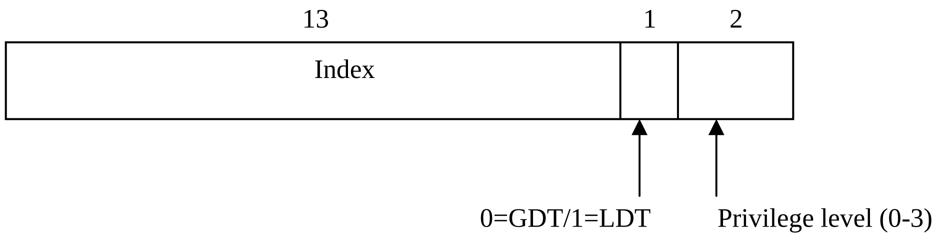
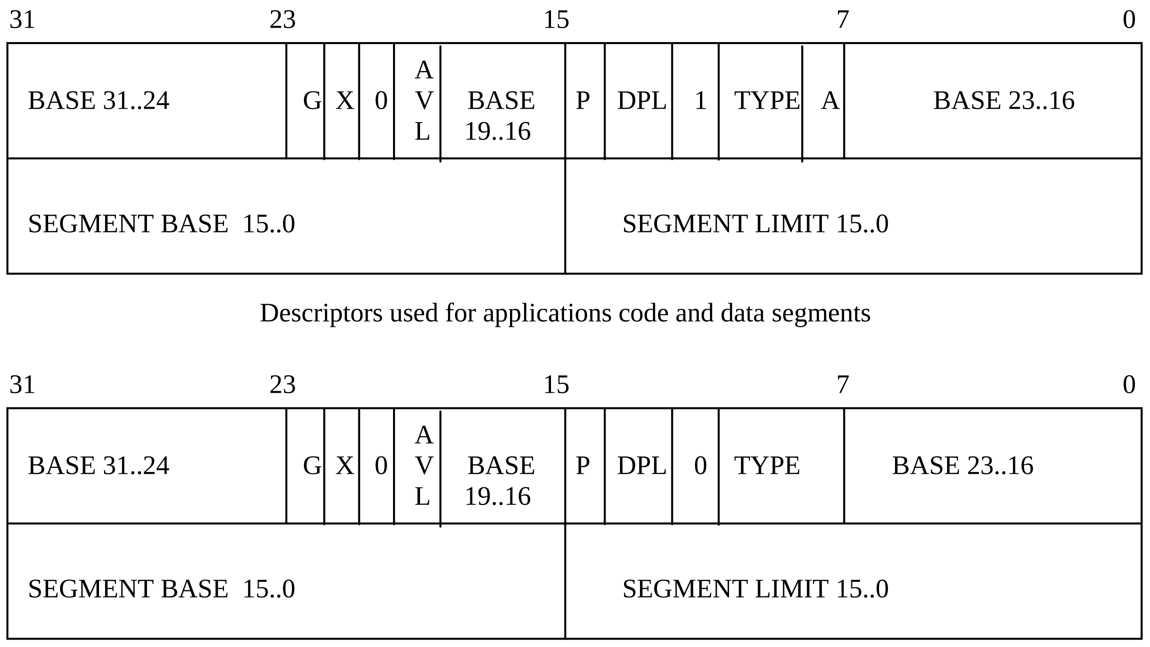


Figure 2.2: A Pentium selector

One of the selector bits tells whether the segment is local or global (i.e., whether it is in the LDT or GDT). Thirteen other bits specify the LDT or GDT entry number, so these tables are each restricted to holding 8K segment descriptors. The other 2 bits relate to protection, and will be described later. Descriptor 0 is forbidden. It may be safely loaded into a segment register to indicate that the segment register is not currently available. It causes a trap if used.

At the time a selector is loaded into a segment register, the corresponding descriptor is fetched from the LDT or GDT and stored in microprogram registers, so it can be accessed quickly. A descriptor [8] consists of 8 bytes, including the segment's base address, size, and other information, as depicted in Figure 2.3.



Descriptors used for special system segments

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

Figure 2.3: Different descriptor formats

The format of the selector has been cleverly chosen to make locating the descriptor easy. First either the LDT or GDT is selected, based on selector bit 2. Then the selector is copied to an internal scratch register, and the 3 low-order bits set to 0. Finally, the address of either the LDT or GDT table is added to it, to give a direct pointer to the descriptor. For example, selector 72 refers to entry 9 in the GDT, which is located at address GDT + 72.

Let us trace the steps by which a (selector, offset) pair is converted to a physical address. As soon as the microprogram knows which segment register is being used, it can find the complete descriptor corresponding to that selector in its internal registers. If the segment does not exist (selector 0), or is currently paged out, a trap occurs.

It then checks to see if the offset is beyond the end of the segment, in which case a trap also occurs. Logically, there should simply be a 32-bit field in the descriptor giving the size of the segment, but there are only 20 bits available, so a different scheme is used. If the G bit (Granularity) field is 0, the Limit field is the exact segment size, up to 1 MB. If it is 1, the Limit field gives the segment size in pages instead of bytes. The Pentium page size is fixed at 4K bytes, so 20 bits are enough for segments up to 232 bytes.

Assuming that the segment is in memory and the offset is in range, the Pentium then adds the 32-bit Base field in the descriptor to the offset to form what is called a linear address, as shown in Figure 2.4. The Base field is broken up into three pieces and spread all over the descriptor for compatibility with the 286, in which the Base is only 24 bits. In effect, the Base field allows each segment to start at an arbitrary place within the 32-bit linear address space.

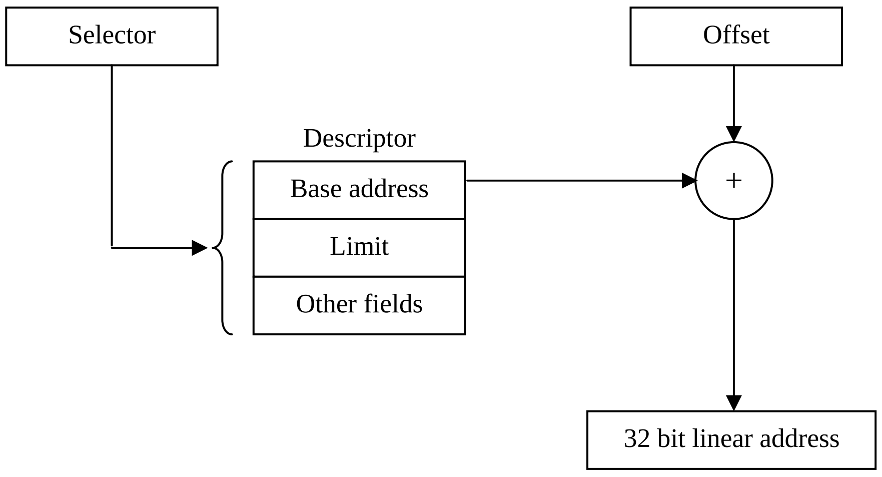


Figure 2.4: Conversion of a (selector, offset) pair to a linear address

If paging is disabled (by a bit in a global control register), the linear address is interpreted as the physical address and sent to the memory for the read or write. Thus with paging disabled, we have a pure segmentation scheme, with each segment's base address given in its descriptor. Segments are permitted to overlap, incidentally, probably because it would be too much trouble and take too much time to verify that they were all disjoint.

On the other hand, if paging is enabled, the linear address is interpreted as a virtual address and mapped onto the physical address using page tables, pretty much as in our earlier examples. The only real complication is that with a 32-bit virtual address and a 4 K page, a segment might contain I million pages, so a two-level mapping is used to reduce the page table size for small segments.

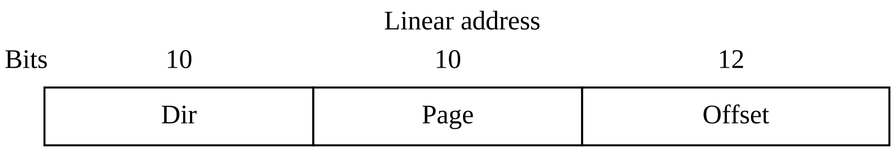
2.2.2 Two level paging

Intel uses two level paging scheme. Each running program has a page directory consisting of 1024 32-bit entries. It is located at an address pointed to by a global register. Each entry in this directory points to a page table also containing 1024 32-bit entries. The page table entries point to page frames. The scheme is shown in Figure 2.5.

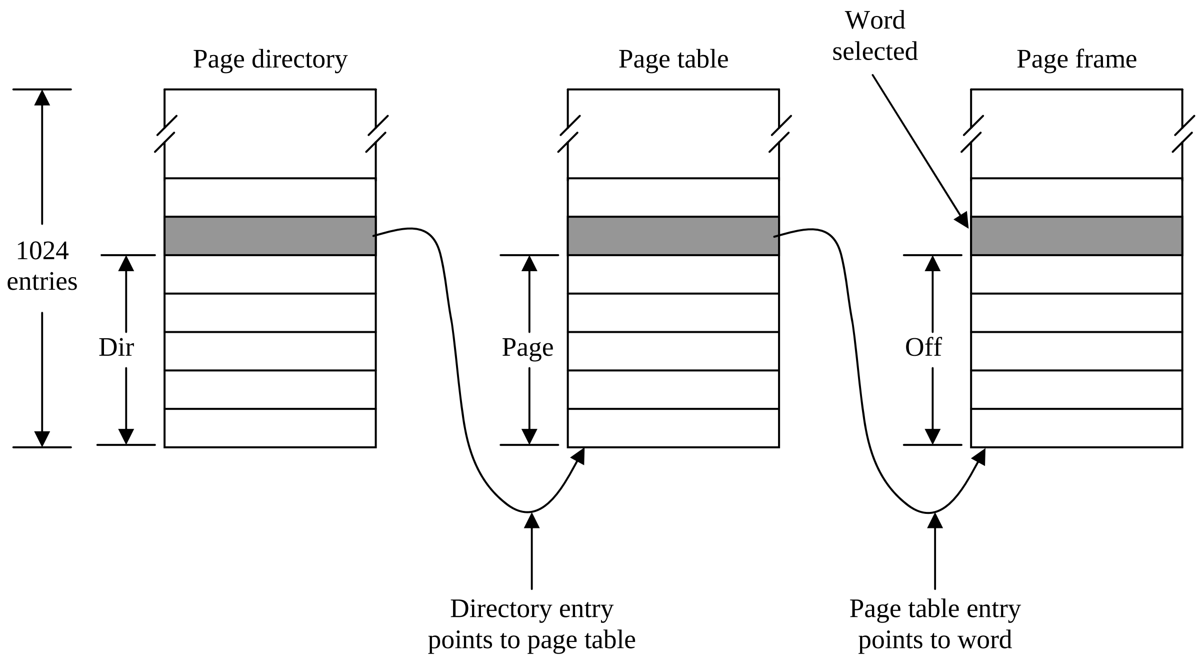
In Figure 2.5(a) we see a linear address divided into three fields, *Dir, Page,* and *Off*. The *Dir* field is used to index into the page directory to locate a pointer to the proper page table. Then the *Page* [9] field is used as an index into the page table to find the physical address of the page frame. Finally, *Off* is added to the address of the page frame to get the physical address of the byte or word needed.

The page table entries are 32 bits each, 20 of which contain a page frame number. The remaining bits contain access and dirty bits, set by the hardware for the benefit of the operating system, protection bits, and other utility bits. Each page table has entries for 1024 4K page frames, so a single page table handles 4 megabytes of memory. A segment shorter than 4M will have a page directory with a single entry, a pointer to its one and only page table. In this way, the overhead for short segments is only two pages, instead of the million pages that would be needed in a one-level page table.

To avoid making repeated references to memory, the Pentium, has a small TLB that directly maps the most recently used *Dir-Page* combinations onto the physical address of the page frame. Only when the current combination is not present in the TLB is the mechanism of Figure 2.5 actually carried out and the TLB updated.



(a)



(b)

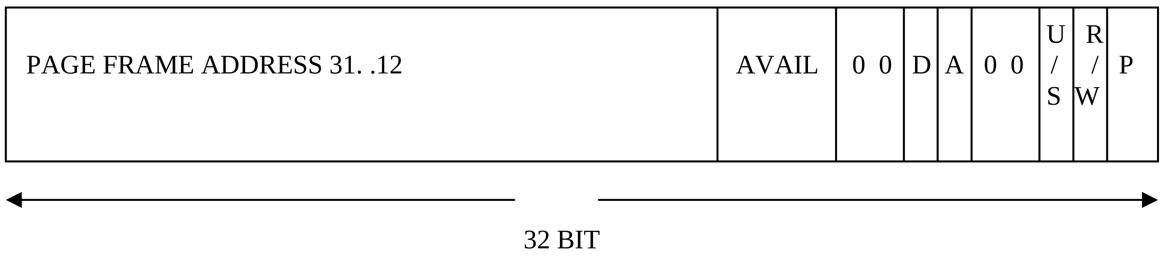
Figure 2.5: Mapping of a linear address onto a physical address.

A little thought will reveal the fact that when paging is used, there is really no point in having the *Base* field in the descriptor be nonzero. All that *Base* does is cause a small offset to use an entry in the middle of the page directory, instead of at the beginning. The real reason for including Base at all is to allow pure (now paged) segmentation, and for compatibility with the 286, which always has paging disabled (i-e., the 286 has only pure segmentation, but not paging).

It is also worth noting that if some application does not need segmentation but is content with a single, paged, 32-bit address space, that model is possible. All the segment registers can be set up with the same selector, whose descriptor has *Base* = 0 and *Limit* set to the maximum. The instruction offset will then be the linear address, with only a single address space used-in effect, normal paging.

2.2.2.1 Page Table entries

Entries in either level of page tables [9] have the same format. Figure 2.6 illustrates this format.



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

Figure 2.6: 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.

2.3 Protection

Although the complete architecture of the Pentium virtual memory has been covered briefly, it is worth saying a few words about protection, since this subject is intimately related to the virtual memory.

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 even 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 criteria. These mechanisms may be used or ignored, according to system design objectives.

2.3.1 Privilege levels

The Pentium supports four protection levels with level 0 being the most privileged and level 3 the least. These are shown in Figure 2.7. At each instant, a running program is at a certain level, indicated by a Z bit field in its PSW. Each segment in the system also has a level.

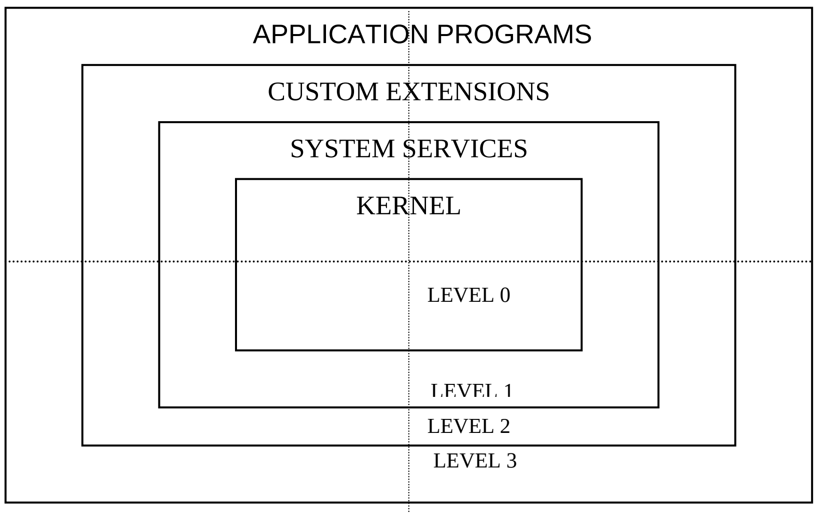


Figure 2.7: Protection on the Pentium

As long as a program restricts itself to using segments at its own level, everything works fine. Attempts to access data at a higher level are permitted. Attempts to access data at a lower level are illegal and cause traps. Attempts to call procedures [10] at a different level (higher or lower) are allowed, but in a carefully controlled way. To make an inter-level call, the CALL instruction must contain a selector instead of an address. This selector designates a descriptor called a *call gate*, which gives the address of the procedure to be called. Thus it is not possible to jump into the middle of an arbitrary code segment at a different level. Only official entry points may be used. The concepts of protection levels and call gates were pioneered in MULTICS, where they were viewed as protection rings.

A typical use for this mechanism is suggested in Figure 2.7. At level O, we find the kernel of the operating system, which handles UO, memory management, and other critical matters. At level I , the system call handler is present. User programs may call procedures here to have system calls carried out, but only a specific, and protected list of procedures may be called. Level 2 contains library procedures, possibly shared among many running programs. User programs may call these procedures and read their data, but they may not modify them. Finally, user programs run at level 3, which has the least protection.

Traps and interrupts use a mechanism similar to the call gates. They, too, reference descriptors, rather than absolute addresses, and these descriptors point to specific procedures to be executed. The Type field in Figure 2.3 distinguishes between code segments, data segments, and the various kinds of gates.

2.3.2 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.

CHAPTER 3 THE MINIX OPERATING SYSTEM

3.1 Minix Architecture

Let us begin our study of Minix by taking a bird's-eye view of the system. Minix is structured in four layers [11], with each layer performing a well-defined function. The four layers are illustrated in Figure 3.1.

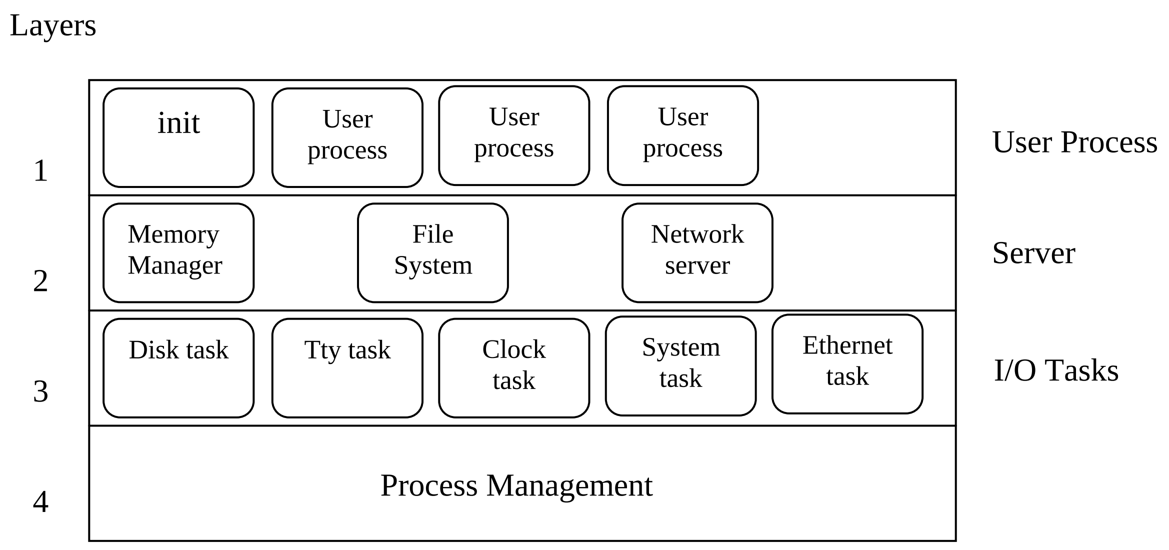


Figure 3.1: Minix architecture

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 interrupts 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 110 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 l/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 VO 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 CHDZR.

3.2 Minix memory management

Memory management in Minix is simple: neither paging nor swapping is used. The memory manager maintains a list of holes sorted in memory address order. When memory is needed, either due to a FORK or an EXEC system call, the hole list is searched using first fit for a hole that is big enough. Once a process has been placed in memory, it remains in exactly the same place until it terminates. It is never swapped out and also never moved to another place in memory. Nor does the allocated area ever grow or shrink.

3.2.1 Memory management using linked lists

This section describes the concept of memory management used by Minix. This memory management scheme maintains a linked list of allocated and free memory segments, where a segment is either a process or a hole between two processes. The memory of Figure 3.2 (a) is represented in Figure 3.2 (b) as a linked list of segments. Each entry in the list specifies a hole (H) or process (P), the address at which it starts, the length, and a pointer to the next entry.

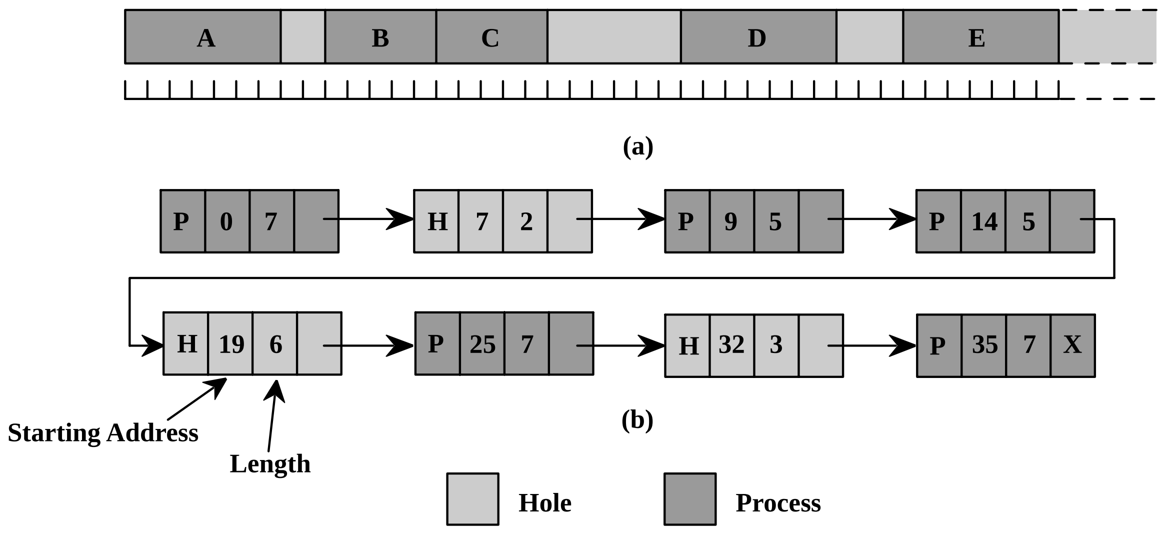


Figure 3.2: Linked list representation of memory

3.2.1.1 Memory allocation algorithms

When the processes and holes are kept on a list sorted by address, several algorithms can be used to allocate memory for a newly created process. These algorithms are described below.
First fit: The memory manager scans along the list of segments until it finds a hole that is big enough. The hole is then broken up into two pieces, one for the process and one for the unused memory, except in the unlikely case of an exact fit. First fit is a fast algorithm because it searches as little as possible.

Next fit: A minor variation of first fit is next fit. It works the same way as first fit, except that it keeps track of where it is whenever it finds a suitable hole. The next time it is called to find a hole, it starts searching the list from the place where it left off last time instead of always at the beginning, as first fit does.

Best fit: It searches the entire list and takes the smallest hole that is adequate. Rather than breaking up a big hole that might be needed later, best fit tries to find a hole that is close to the actual size needed.

Worst fit: It always takes the largest available hole, so that the hole broken off will be big enough to be useful. Simulation has shown that worst fit is not a very good idea either.

Among these algorithms, Minix uses first fit algorithm to allocate memory to processes.

3.2.1.2 Memory freeing algorithm

When a process terminates and is cleaned up, its data and stack memory are returned to the free list. If it uses common I and D, this releases all its memory, since such programs never have a separate allocation of memory for text. If the program uses separate I and D and a search of the process table reveals no other process is sharing the text, the text allocation will also be returned. Since with shared text the text and data regions are not necessarily contiguous, two regions of memory may be returned. For each region returned, if either or both of the region's neighbors are holes, they are merged, so adjacent holes never occur. In this way, the number, location, and sizes of the holes vary continuously during system operation, whenever all user processes have terminated, all of available memory is once again ready for allocation. This isn't necessarily a single hole, however, since physical memory may be interrupted by regions unusable by the operating system, as in IBM compatible systems where read-only memory (ROM) and memory reserved for I/0 transfers separate usable memory below address 640K from memory above 1 M.

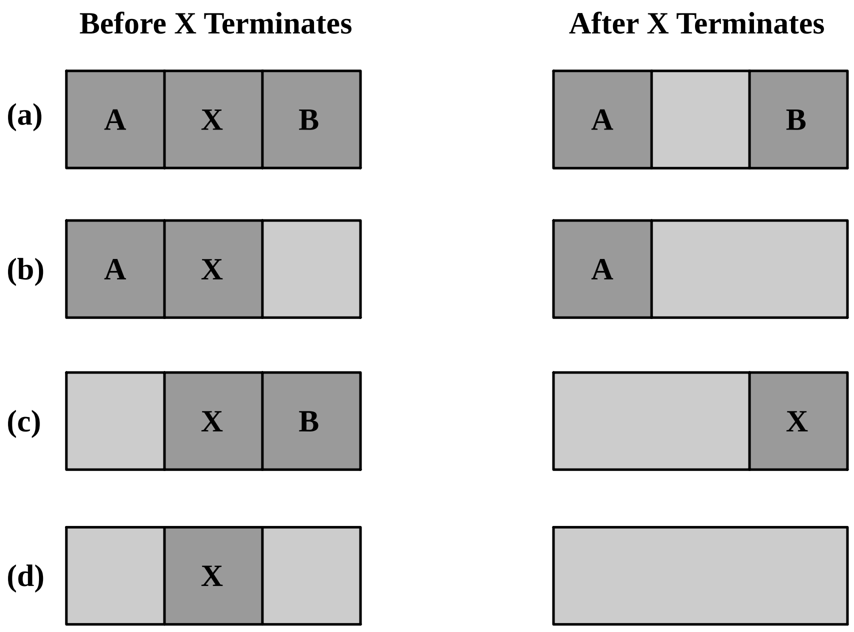


Figure 3.3: Four neighbor combination for terminating process X

A terminating process normally has two neighbors (except when it is at the very top or bottom of memory). These may be either processes or holes leading to the four combinations of Figure 3.3. In Figure 3.3 (a) updating the list requires replacing a process by a hole. In Figure 3.3 (b) and Figure 3.3 (c), two entries are coalesced into one, and the list becomes one entry shorter. In Figure 3.3 (d), three entries are merged and two items are removed from the list. Since the process table slot for the terminating process will normally point to the list entry for the process itself, it may be more convenient to have the list as a double-linked list, rather than the single-linked list of Figure 3.3 (c). This structure makes it easier to entry and to see if a merge is possible. Find the previous entry and to see if a merge is possible.

3.2.2 Memory layout of process

Simple Minix processes use combined I and D space, in which all parts of the process (text, data, and stack) share a block of memory which is allocated and released as one block. Processes can also be compiled to use separate I and D space. For clarity, allocation of memory for the simpler model will be discussed first. Processes using separate I and D space can use memory more efficiently, but taking advantage of this feature complicates things. We will discuss the complications after the simple case has been outlined.

Memory is allocated in Minix on two occasions. First, when a process forks, the amount of memory needed by the child is allocated. Second, when a process changes its memory image via the EXEC system call, the old image is returned to the free list as a hole, and memory is allocated for the new image. The new image may be in a part of memory different from the released memory. Its location will depend upon where an adequate hole is found. Memory is also released whenever a process terminates, either by exiting or by being killed by a signal.

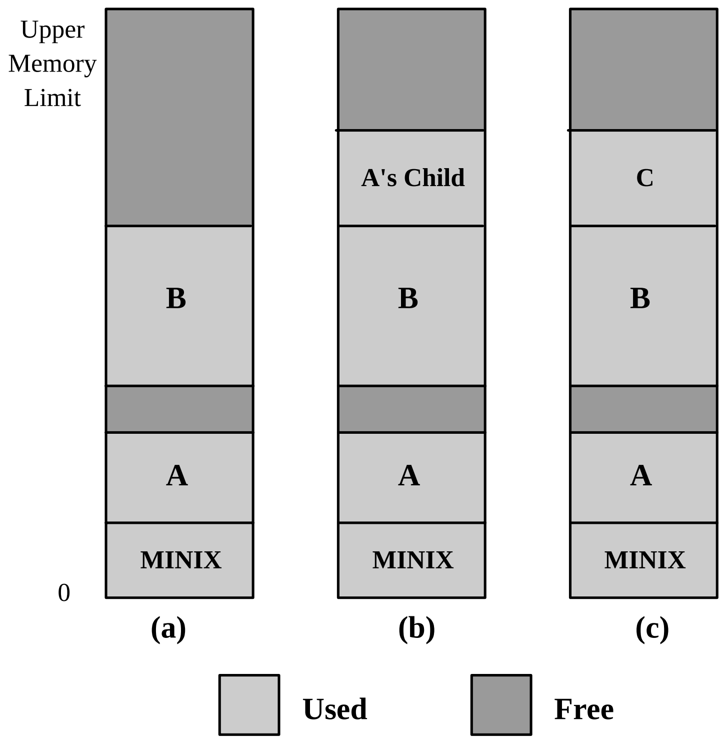


Figure 3.4: Memory allocation. (a) Originally. (b) After a SYS FORK. (c) After the child does an EXEC . The shaded regions are unused memory. The process is a common I & D one.

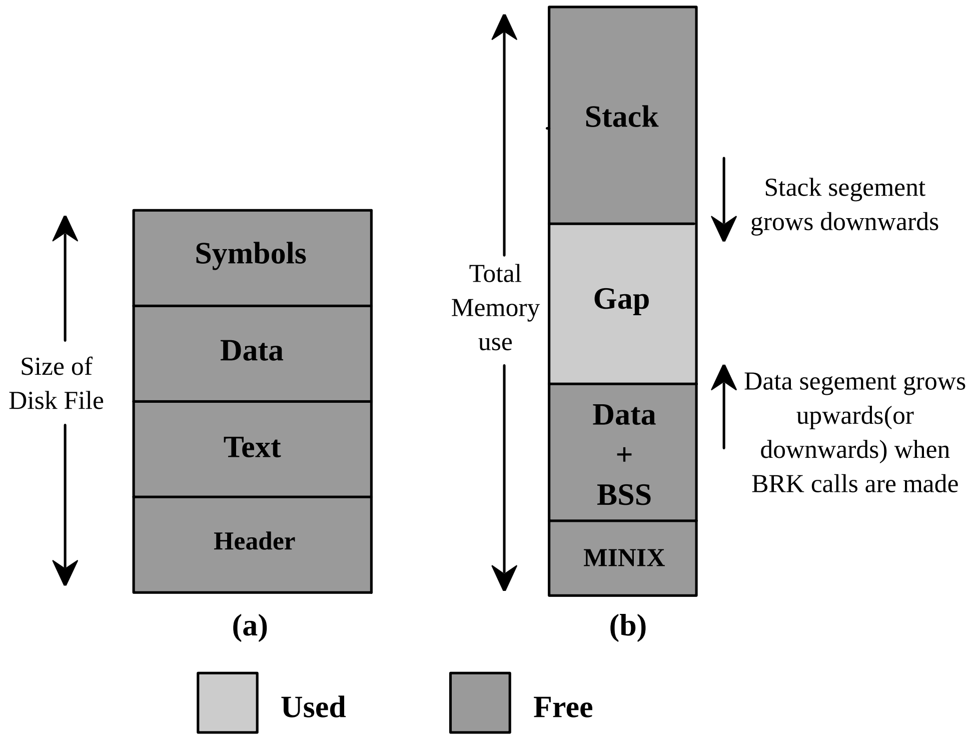
Figure 3.4 shows both ways of allocating memory. In Figure 3.4 (a) we see two processes, A and B, in memory. If A forks, we get the situation of Figure 3.4 (b). The child is an exact copy of A. If the child now executes the file C, the memory looks like Figure 3.4 (c). The child's image is replaced by C.

Note that the old memory for the child is released before the new memory for C is allocated, so that C can use the child's memory. In this way, a series of FORK and EXEC pairs (such as the shell setting up a pipeline) results in all the processes being adjacent, with no holes between them, as would have been the case had the new memory been allocated before the old memory had been released.

When memory is allocated, either by the FORK or EXEC system calls, a certain amount of it is taken for the new process. In the former case, the amount taken is identical to what the parent process has. In the latter case, the memory manager takes the amount specified in the header of the file executed. Once this allocation has been made, under no conditions is the process ever allocated any more total memory.

What has been said so far applies to programs that have been compiled with combined I and D space. Programs with separate 1 and D space take advantage of an enhanced mode of memory management called shared text. When such a process does a FORK, only the amount of memory needed for a copy of the new process' data and stack is allocated. Both the parent and the child share the executable code already in use by the parent. When such a process does an EXEC, a search is made of the process table to see if another process already is using the executable code needed. If one is found, new memory is allocated only for the data and stack, and the text already in memory is shared. Shared text complicates termination of a process. When a process terminates it always releases the memory occupied by its data and stack. But it only releases the memory occupied by its text segment after a search of the process table reveals that no other current process is sharing that memory. Thus a process may be allocated more memory when it starts than it releases when it terminates, if it loaded its own text when it started but that text is being shared by one or more other processes when the first process terminates.

Figure 3.5 shows how a program is stored as a disk file and how this is transferred to the internal memory layout of a Minix process. The header on the disk file contains information about the sizes of the different parts of the image, as well as the total size. In the header of a program with common I and D space, a field specifies the total size of the text and data parts; these parts are copied directly to the memory image. The data part in the image is enlarged by the amount specified in the *bss* field in the header. This area is cleared to contain all zeroes and is used for uninitialized static data. The total amount of memory to be allocated is specified by the total field in the header. If, for example, a program has 4K of text, 2K of data plus *bss*, and 1K of stack, and the header says to allocate 40K total, the gap of unused memory between the data segment and the stack segment will be 33K. A program file on the disk may also contain a symbol table.



Size of Disk File
Symbols
Data
Text
Header
(a)
Total Memory use
Stack
Gap
Data + BSS
MINIX
(b)
Stack segement grows downwards
Data segement grows upwards(or downwards) when BRK calls are made
Used
Free

Figure 3.5. (a) A program as stored in a disk file. (b) Internal memory layout for a single process. In both parts of the figure the lowest disk or memory address is at the bottom and the highest address is at the top.

This is for use in debugging and is not copied into memory. If the programmer knows that the total memory needed for the combined growth of the data and stack segments for the file *a.out* is at most 10K, he can give the command

```
chmem =10240 a.out
```

which changes the header field so that upon EXEC the memory manager allocates a space 10240 bytes more than the sum of the initial text and data segments. For the above example, a total of 16K will be allocated on all subsequent EXECS of the file. Of this amount, the topmost 1K will be used for the stack, and 9K will be in the gap, where it can be used by growth of the stack, the data area, or both.

For a program using separate I and D space (indicated by a bit in the header that is set by the linker), the total field in the header applies to the combined data and stack space only. A program with 4K of text, 2K of data, 1 K of stack, and a total size of 44K will be allocated 68K (4K instruction space, 64K data space), leaving 61K for the data segment and stack to consume during execution. The boundary of the data segment can be moved only by the BRK system call. All BRK does is check to see if the new data segment bumps into the current stack pointer, and if not, notes the change in some internal tables. This is entirely internal to the memory originally allocated to the process; no additional memory is allocated by the operating system. If the new data segment bumps into the stack, the call fails.

This strategy was chosen to make it possible to run Minix on an TBM PC with an 8088 processor, which does not check for stack overflow in hardware. A user program can push as many words as it wants onto the stack without the operating system being aware of it. On computers with more sophisticated memory management hardware, the stack is allocated a certain amount of memory initially. If it attempts to grow beyond this amount, a trap to the operating system occurs, and the system allocates another piece of memory to the stack, if possible. This trap does not exist on the 8088, making it dangerous to have the stack adjacent to anything except a large chunk of unused memory, since the stack can grow quickly and without warning. Minix has been designed so that when it is implemented on a computer with better memory management, it is straightforward to change the Minix memory manager.

3.3 Memory management data structures

The memory manager has two key data structures: the *process table* and the *hole table*. We will now look at each of these in turn.

3.3.1 The mproc table

Different parts of Minix (file system, memory manager and kernel) needs to keep information about all the process present in the memory. These informations are kept in a data structure called *process table*. In Minix, each of these three pieces of the operating system has its own *process table*, containing just those fields that it needs. The entries correspond exactly, to keep things simple. Thus, slot k of the memory manager's table refers to the same process as slot k of the file system's table. When a process is created or destroyed, all three parts update their tables to reflect the new situation, in order to keep them synchronized.

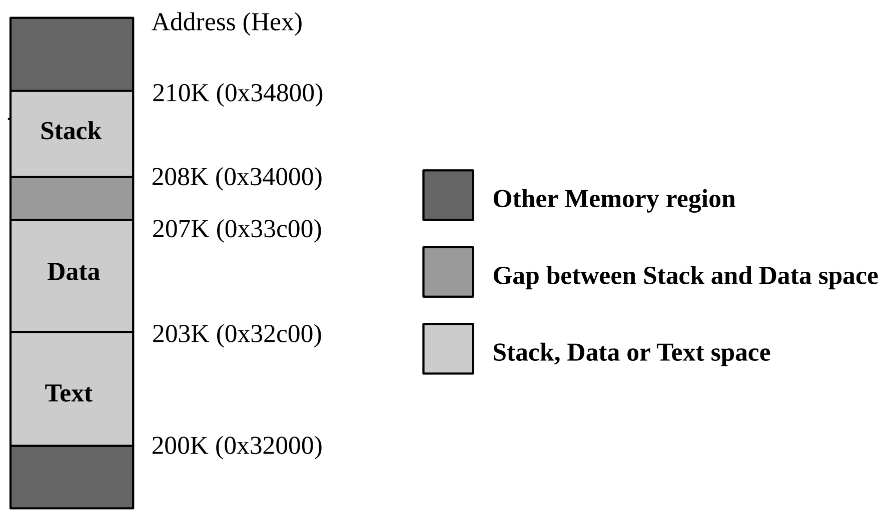
The memory manager's process table is called *mproc*; its definition is given below:

```
EXTERN struct mproc {
  struct mem_map mp_seg[NR_SEGS];        /* points to text, data, stack */
  char mp_exitstatus;                    /* storage for status when process exits */
  char mp_sigstatus;                     /* storage for signal # for killed procs */
  pid_t mp_pid;                          /* process id */
  pid_t mp_procgrp;                      /* pid of process group (used for signals) */
  pid_t mp_wpid;                         /* pid this process is waiting for */
  int mp_parent;                         /* index of parent process */

  /* Real and effective uids and gids. */
  uid_t mp_realuid;                      /* process' real uid */
  uid_t mp_effuid;                       /* process' effective uid */
  gid_t mp_realgid;                      /* process' real gid */
  gid_t mp_effgid;                       /* process' effective gid */

  /* File identification for sharing. */
  ino_t mp_ino;                          /* inode number of file */
  dev_t mp_dev;                          /* device number of file system */
  time_t mp_ctime;                       /* inode changed time */

  /* Signal handling information. */
  sigset_t mp_ignore;                    /* 1 means ignore the signal, 0 means don't */
  sigset_t mp_catch;                     /* 1 means catch the signal, 0 means don't */
  sigset_t mp_sigmask;                   /* signals to be blocked */
  sigset_t mp_sigmask2;                  /* saved copy of mp_sigmask */
  sigset_t mp_sigpending;                /* signals being blocked */
  struct sigaction mp_sigact[_NSIG + 1]; /* as in sigaction(2) */
  vir_bytes mp_sigreturn;                /* address of C library __sigreturn function */

  unsigned mp_flags;                     /* flag bits */
  vir_bytes mp_procargs;                 /* ptr to proc's initial stack arguments */
} mproc[NR_PROCS];
```

It contains all the fields related to a process' memory allocation, as well as some additional items. The most important field is the array mp_seg. Its difinition is given below:

```
typedef unsigned int vir_clicks;          /* virtual  addresses and lengths in clicks */
typedef unsigned int phys_clicks;         /* physical addresses and lengths in clicks */

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

Structure mp_seg has three entries, for the text, data, and stack segments, respectively. Each entry is a structure containing the virtual address, physical address, and length of the segment, all measured in clicks rather than in bytes. The size of a click is implementation dependent; for standard Minix it is 256 bytes. All segments must start on a click boundary and occupy an integral number of clicks.



(a)

	virtual	physical	length
Stack	0x20	0x340	0x8
Data	0	0x320	0x1c
Text	0	0x320	0

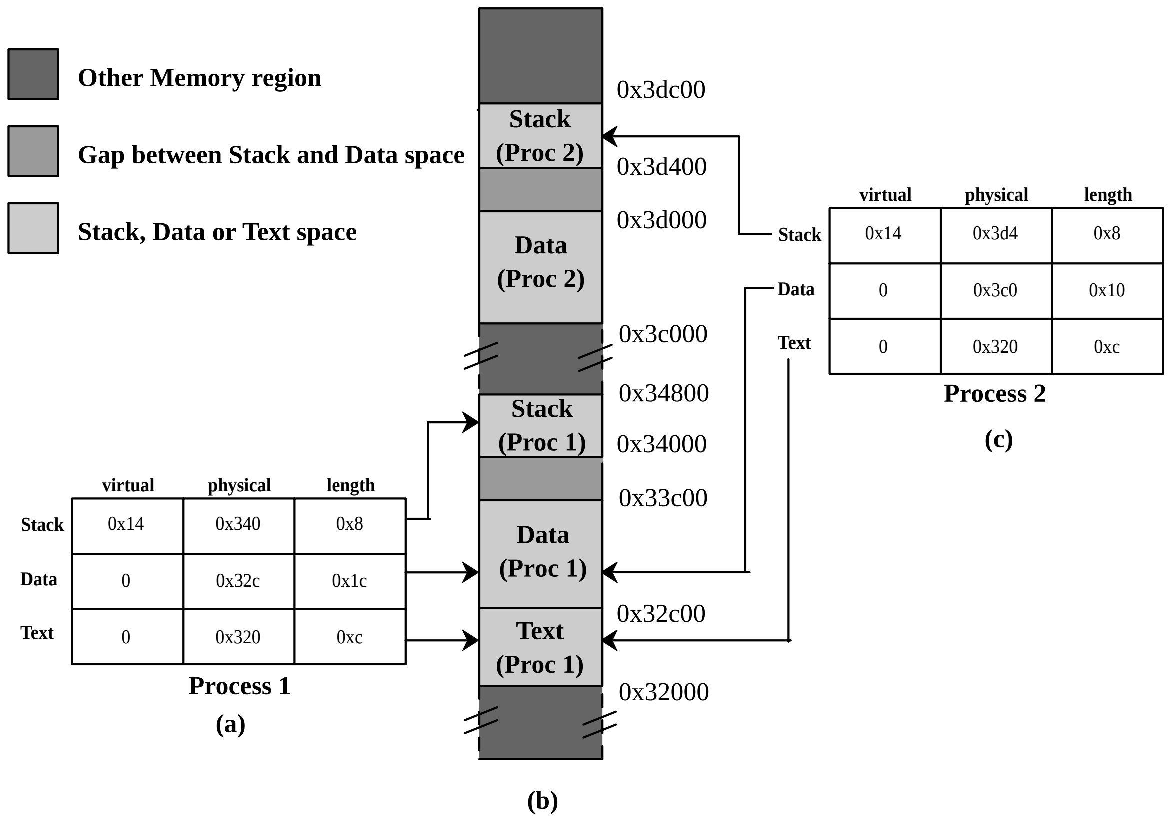
(b)

	virtual	physical	length
Stack	0x14	0x340	0x8
Data	0	0x32c	0x10
Text	0	0x320	0xc

(c)

Figure 3.6: (a) A process in memory. (b) Its memory representation for Non-separate I and D space. (c) Its memory representation for separate I and D space

The method used for recording memory allocation is shown in Figure 3.6. In this figure we have a process with 3K of text, 4K of data, a gap of 1K, and then a 2K stack, for a total memory allocation of IOK. In Figure 3.6(b) we see what the virtual, physical, and length fields for each of the three segments are, assuming that the process does not have separate I and D space. In this model, the text segment is always empty, and the data segment contains both text and data. When a process references virtual address 0, either to jump to it or to read it (i.e., as instruction space or as data space), physical address 0x32000 (in decimal, 200K) will be used. This address is at click 0x320. Note that the virtual address at which the stack begins depends initially on the total amount of memory allocated to the process. If the *chmem* command were used to modify the file header to provide a larger dynamic allocation area (bigger gap between data and stack segments), the next time the file was executed, the stack would start at a higher virtual address. If the stack grows longer by one click, the stack entry should change from the triple (0x20,0x340,0x8) to the triple (0xlF, 0x33F, 0x9).



	virtual	physical	length
Stack	0x14	0x340	0x8
Data	0	0x32c	0x1c
Text	0	0x320	0xc

	virtual	physical	length
Stack	0x14	0x3d4	0x8
Data	0	0x3c0	0x10
Text	0	0x320	0xc

Figure 3.7: (a) A process in memory. (b) Its memory representation for Non-separate I and D space. (c) Its memory representation for separate I and D space.

Figure 3.7(c) shows the segment entries for the memory layout of Figure 3.7 (a) for separate I and D space. Here both the text and data segments are nonzero in length. The *mp_seg* array shown in Figure 3.7 (b) or (c) is primarily used to map virtual addresses onto physical memory addresses. Given a virtual address and the space to which it belongs, it is a simple matter to see whether the virtual address is legal or not (i.e., falls inside a segment), and if legal, what the corresponding physical address is. The kernel procedure *umap* performs this mapping for the I/O tasks and for copying to and from user space, for example.

The contents of the data and stack areas belonging to a process may change as the process executes, but the text does not change. It is common for several processes to be executing copies of the same program, for instance several users may be executing the same shell. Memory efficiency is improved by using shared text. When EXEC is about to load a process, it opens the file. When EXEC is about to load a process, it opens the file holding the disk image of the program to be loaded and reads the file header. If the process uses separate I and D space, a search of the *mp_dev*, *mp_no*, and *mp_ctime* fields in each slot of When EXEC is about to load a process, it opens the file holding the disk image of the program to be loaded and reads the file header. *mproc* is made. These hold the device and i-node numbers and changed-status times of the images being executed by other processes. If a process already loaded is found to be executing the same program that is about to be loaded, there is no need to allocate memory for another copy of the text. Instead the *mp_seg[q]* portion of the new process' memory map is initialized to point to the same place where the text segment is already loaded, and only the data and stack portions are set up in a new memory allocation. This is shown in Figure 3.7. If the program uses combined I and D space or no match is found, memory is allocated as shown in Figure 3.6 and the text and data for the new process are copied in from the disk.

In addition to the segment information, *mproc* also holds the process *ID* (*pid*) of the process itself and of its parent, the *uids* and *gids* (both real and effective), information about signals, and the exit status, if the process has already terminated but its parent has not yet done a WAIT for it.

3.3.2 The hole table

The other major memory manager table is the hole table, *hole*, defined below:

```
#define NR_HOLES        128        /* max # entries in hole table */

PRIVATE struct hole {
  phys_clicks h_base;                /* where does the hole begin? */
  phys_clicks h_len;                 /* how big is the hole? */
  struct hole *h_next;               /* pointer to next entry on the list */
} hole[NR_HOLES];
```

The *hole* lists every hole in memory in order of increasing memory address. The gaps between the data and stack segments are not considered holes; they have already been allocated to processes. Consequently, they are not contained in the free hole list. Each hole list entry has three fields: the base address of the hole, in clicks; the length of the hole, in clicks; and a pointer to the next entry on the list. The list is singly linked, so it is easy to find the next hole starting from any given hole, but to find the previous hole, you have to search the entire list from the beginning until you come to the given hole.

The reason for recording everything about segments and holes in clicks rather than bytes is simple: it is much more efficient, In 16-bit mode, 16-bit integers are used for recording memory addresses, so with 256-bit clicks, up to 16 MB of memory can be supported. In 32-bit mode, address fields can refer to up to 2w bytes, which is 1024 gigabytes.

The principal operations on the hole list are allocating a piece of memory of a given size and returning an existing allocation. To allocate memory, the hole list is searched, starting at the hole with the lowest address, until a hole that is large enough is found (first fit). The segment is then allocated by reducing the hole by the amount needed for the segment, or in the rare case of an exact fit, removing the hole from the list. This scheme is fast and simple but suffers from both a small amount of internal fragmentation (up to 255 bytes may be wasted in the final click, since an integral number of clicks is always taken) and external fragmentation.

When a process terminates and is cleaned up, its data and stack memory are returned to the free list. If it uses common I and D, this releases all its memory, since such programs never have a separate allocation of memory for text. If the program uses separate I and D and a search of the process table reveals no other process is sharing the text, the text allocation will also be returned. Since with shared text the text and data regions are not necessarily contiguous, two regions of memory may be returned. For each region returned, if either or both of the region's neighbors are holes, they are merged, so adjacent holes never occur. In this way, the number, location, and sizes of the holes vary continuously during system operation, whenever all user processes have terminated, all of available memory is once again ready for allocation. This isn't necessarily a single hole, however, since physical memory may be interrupted by regions unusable by the operating system, as in IBM compatible systems where read-only memory (ROM) and memory reserved for I/O transfers separate usable memory below address 640K from memory above 1 M.

3.4 System calls related to memory management

In this section we will describe the system calls related to memory management.

3.4.1 The FORK system call

When processes are created or destroyed, memory must be allocated or deallocated. Also, the process table must be updated, including the parts held by the kernel and FS. The memory manager coordinates all this activity. Process creation is done by FORK, and carried out in the series of steps shown below.

1. Check to see if process table b full.
2. Try to allocate memory for the child's data and stack.
3. Copy the parent's data and stack to the child's memory.
4. Find a free process slot and copy parent's slot to it.
5. Enter child's memory map in process table.
6. Choose a *pid* for the child.
7. Tell kernel and file system about child.
8. Report child's memory map to kernel.
9. Send reply messages to parent and child.

It is difficult and inconvenient to stop a FORK call part way through, so the memory manager maintains a count at all times of the number of processes currently in existence in order to see easily if a process table slot is available. If the table is not full, an attempt is made to allocate memory for the child. If the program is one with separate I and D space, only enough memory for new data and stack allocations is requested. If this step also succeeds, the FORK is guaranteed to work. The newly allocated memory is then filled in, a process slot is located and filled in, a *pid* is chosen, and the other parts of the system are informed that a new process has been created.

A process fully terminates when two events have both happened: (1) the process itself has exited (or has been killed by a signal), and (2) its parent has executed a WAIT system call to find out what happened. A process that has exited or has been killed, but whose parent has not (yet) done a WAIT for it, enters a kind of suspended animation, sometimes known as *zombie state*. It is prevented from being scheduled and has its alarm timer turned off (if it was on), but it is not removed from the process table. Its memory is freed. *Zombie state* is temporary and rarely lasts long. When the parent finally does the WAIT, the process table slot is freed, and the file system and kernel are informed.

A problem arises if the parent of an exiting process is itself already dead. If no special action were taken, the exiting process would remain a zombie forever. Instead, the tables are changed to make it a child of the init process. When the system comes up, init reads the */etc/ttytab* file to get a list of all terminals, and then forks off a login process to handle each one. It then blocks, waiting for processes to terminate. In this way, orphan zombies are cleaned up quickly.

3.4.2 The EXEC system call

When a command is typed at the terminal, the shell forks off a new process, which then executes the command requested. It would have been possible to have a single system call to do both FORK and EXEC at once, but they were provided as two distinct calls for a very good reason: to make it easy to implement I/0 redirection, When the shell forks, if standard input is redirected, the child closes standard input and then opens .the new standard input before executing the command. In this way the newly started process inherits the redirected standard input. Standard output is handled the same way.

EXEC is the most complex system call in Minix. It must replace the current memory image with a new one, including 'setting up a new stack. It carries out its job in a series of steps, as shown below.

1. Check permissions-is the file executable?
2. Read the header to get the segment and total sizes.
3. Fetch the arguments and environment from the caller.
4. Allocate new memory end release unneeded old memory.
5. Copy stack to new memory image.
6. Copy data (and possibly text) segment to new memory image.
7. Check for and handle *setuid, setgid* bits.
8. Fix up process table entry.
9. Tell kernel that process is now runnable.

Each step consists, in turn, of yet smaller steps, some of which can fail. For example, there might be insufficient memory available. The order in which the tests are made has been carefully chosen to make sure the old memory image is not released until it is certain that the EXEC will succeed, to avoid the embarrassing situation of not being able to set up a new memory image, but not having the old one to go back to, either. Normally EXEC does not return, but if it fails, the calling process must get control again, with an error indication.

There are a few steps in described above that deserve some more comment. First is the question of whether or not there is enough room or not. After determining how much memory is needed, which requires determining if the text memory of another process can be shared, the hole list is searched to check whether there is sufficient physical memory before freeing the old memory-if the old memory were freed first and there were insufficient memory, it would be hard to get the old image back again.

However, this test is overly strict. It sometimes rejects EXEC calls that, in fact, could succeed. Suppose, for example, the process doing the EXEC call occupies 20K and any other process does not share its text. Further suppose that there is a 30K hole available and that the new image requires 50K. By testing before releasing, we will discover that only 30K are available and reject the call. If we had released first, we might have succeeded, depending on whether or not the new 20K hole were adjacent to, and thus now merged with, the 30K hole. A more sophisticated implementation could handle this situation a little better.

Another possible improvement would be to search for two holes, one for the text segment and one for the data segment, if the process to be *EXECed* uses separate I and D space. There is no need for the segments to be contiguous.

A more subtle issue is whether the executable file fits in the virtual address space. The problem is that memory is allocated not in bytes, but in 256-byte clicks. Each click must belong to a single segment, and may not be, for example, half data, half stack, because the entire memory administration is in clicks. To see how this restriction can give trouble, note that the address space on 16-bit systems (8088 and 80286) is limited to 64K, which can be divided into 256 clicks. Suppose that a separate I and D space program has 40,000 bytes of text, 32,770 bytes of data, and 32,760 bytes of stack. The data segment occupies 129 clicks, of which the last one is only partially used; still, the whole click is part of the data segment. The stack segment is 128 clicks. Together they exceed 256 clicks, and thus cannot co-exist, even though the number of bytes needed fits in the virtual address space (barely). In theory this problem exists on all machines whose click size is larger than 1 byte, but in practice it rarely occurs on Pentium class processors, since they permit large (4-GB) segments.

Another important issue is how the initial stack is set up. The library call normally used to invoke EXEC with arguments and an environment is

execve(name, argv, envp);

where name is a pointer to the name of the file to be executed, *argv* is a pointer to an array of pointers, each one pointing to an argument, and *envp* is a pointer to an array of pointers, each one pointing to an environment string.

It would be easy enough to implement EXEC by just putting the three pointers in the message to the memory manager and letting it fetch the file name and two arrays by itself. Then it would have to fetch each argument and each string one at a time. Doing it this way requires at least one message to the system task per argument or string and probably more, since the memory manager has no way of knowing how big each one is in advance.

To avoid the overhead of multiple messages to read all these pieces, a completely different strategy has been chosen. The *execve* library procedure builds the entire initial stack inside itself and passes its base address and size to the memory manager. Building the new stack within the user space is highly efficient, because references to the arguments and strings are just local memory references, not references to a different address space.

3.4.3 The BRK system call

The library procedures *brk* and *sbrk* are used to adjust the upper bound of the data segment. The former takes an absolute size (in bytes) and calls BRK. The latter takes a positive or negative increment to the current size, computes the new data segment size, and then calls BRK. There is no actual SBRK system call.

An interesting question is: "How does *sbrk* keep track of the current size, so it can compute the new size?" The answer is that a variable, *brksite*, always holds the current size so *sbrk* can find it. This variable is initialized to a compiler generated symbol giving the initial size of text plus data (non-separate 1 and D) or just data (separate I and D). The name, and, in fact, very existence of such a symbol is compiler dependent, and thus it will not be found defined in any header file in the source file directories. It is defined in the library, in the file brksi2e.s. Exactly where it will be found depends on the system, but it will be in the same directory as crts0.s.

Carrying out BRK is easy for the memory manager. All that must be done is to check to see that everything still fits in the address space, adjust the tables, and tell the kernel.

CHAPTER 4 DESIGN

Standard Minix uses physical address space for processes and this address space is managed by memory manage (MM) server. To introduce linear address space in Minix its memory manager has to be modified. The major issue here is that modifications should not affect the basic framework of Minix.

4.1 The basic mechanism: design issues and concepts

Memory manager in Minix gets the physical address space from Kernel during system initialization phase. To support linear address space, changes could be made in the kernel so that instead of passing physical address space it passes the linear address space during system initialization.

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 [12].

Once all this is done, memory manager is using linear address space for memory allocation and de-allocation to processes. 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 numbers 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. This linear address space starts from *paging_base* and has a range of *VM_SIZE_CLICKS* (2 GB), as shown in Figure 4.1.

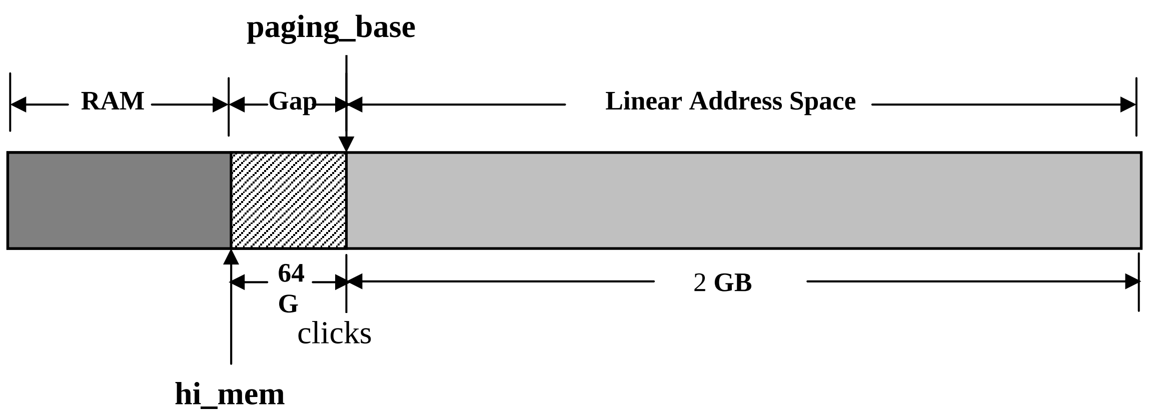


Figure 4.1 : Virtual Addressing

Now memory manager will fulfill the requests of user processes for memory from this linear address space. A hole list is used in the same way as standard Minix uses for physical address space. This linear address space will be mapped on the page frames by the Kernel.

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 it is 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.

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 [13] 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 greater 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. Also, a Gap of 4M [14] is taken between contiguous processes in the linear address space. This gap is needed to avoid the overlap between processes' address space as shown in figure 4.2 and hence provide protection.

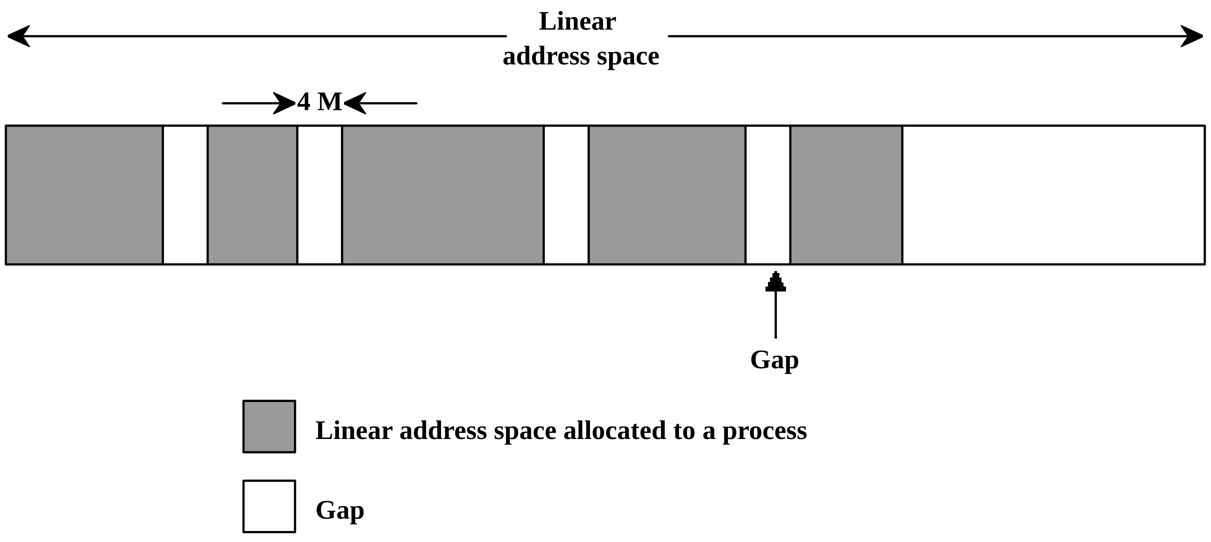


Figure 4.2: Linear address space allocation

The memory to user processes is allocated by system calls Fork and Exec. These system calls are implemented in the memory manager. Now these system calls need to be slightly modified for handling linear address space. These system calls also uses kernel functions at the lowest level defined in s*ystem.c* file for their implementation. We also have to change and add some functions for the linear address space management in s*ystem.c*. These system calls are discussed below. Before explaining the modifications in the system calls of memory manager, a few ideas and routines related to the Page Global Directory and Page Tables are described.

4.2 Handling page global directory and page tables

To use linear address space in any operating system it is must to enable the paging unit of the underlying architecture. As we know that Intel architecture supports a two level paging model. At the fist level, a page global directory is used to point to the page tables of second level. A page table at second level actually contains the physical addresses of page frames.

In this design, Minix kernel 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 during system initialization phase. 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()*.

Then it calculates the base of linear address space to be used by servers (MM and FS) 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 passed to memory manager for usage.

For adding an entry in the Page global Directory, a routine named *map_dir()* is defined and similarly, for making entries in page tables another routine named *map_page()* is defined. These two routines are explained below.

4.2.1 map_dir( ) routine

This routine puts an entry in the page directory. Prototype of the routine is given below.

```
void map_dir(phys_bytes vm_addr, phys_bytes real_addr);
```

The first parameter *vm_addr* is the 32 bit linear address and the second parameter *real_addr* denotes the physical address of the page frame containing a page table. The routine enters the information about this page table in the corresponding entry of the page directory. The information includes the physical address of the page frame containing page table, whether this page is present in main memory or not, the read/write bit indicating whether the page is write protected or not. Format of a page directory entry is given in Figure 4.3.

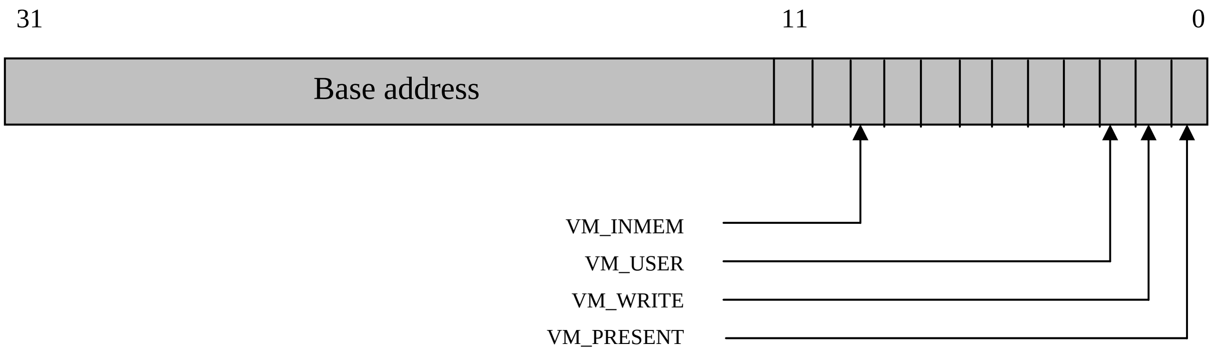


Figure 4.3: Format of page table/ page directory entry

map_dir( ) routine first prepares this page directory entry in a variable *dir_ent* with the help of following statement.

```
dir_ent= real_addr | VM_INMEM_N_PRESENT | VM_WRITE | VM_USER;
```

The constants used in this statement are declared as follows:

```
#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)
```

Now the next step is to find the address of page directory entry and then put dir*_ent* at this place. Statement given below copies the address of the directory entry in a variable *ent_addr*.

```
ent_addr= page_base+ vm_addr_to_dir(vm_addr)*4;
```

Here *vm_addr_to_dir( )* is a macro and its definition along with the definition a constant used by it is given below.

```
#define VM_DIRSHIFT    22           /* 2log VM_DIRSIZE */
#define vm_addr_to_dir(a)           ((a >> VM_DIRSHIFT) & 0x3ff)
```

Finally *map_dir( )* copies this entry in the page directory with the help of an assembly language routine *put_phys_dword( )*. Prototype of this routine is given below.

```
void put_phys_dword (phys_bytes phys_addr, u32_t dword);
```

This routine writes a double word in the specified location in main memory. The first argument *phys_addr* is the physical address of the double word where data is to be written and *dword* is a double word to be written.

4.2.2 map_page( ) routine

The job of this routine puts an entry in a page table. Working of this routine is almost similar to *map_dir( )* routine. Prototype of the routine is given below.

```
void map_page(phys_bytes vm_addr, phys_bytes real_addr);
```

The first parameter *vm_addr* is the 32 bit linear address and the second parameter *real_addr* denotes the physical address of the page frame. The routine enters the information about this page frame in the corresponding entry of the page table. The format of the page table entry is similar to page directory entry and is shown in the Figure 4.3.

map_page( ) routine first prepares this page table entry a variable *page_ent* with the help of following statement.

```
page_ent= real_addr | VM_INMEM_N_PRESENT | VM_WRITE | VM_USER;
```

The constants used in this statement are already defined in the previous section.

Now the next step is to find the address of page table entry and then put page*_ent* at this place. The following statement does this.

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

Here *vm_addr_to_page( )* is a macro and its definition along with the definition a constant used by it is given below.

```
#define VM_PAGESHIFT12          /* 2log VM_PAGESIZE */
#define vm_addr_to_page(a)      ((a >> VM_PAGESHIFT) & 0x3ff)
```

Finally *map_page( )* routine copies this entry in the page table with the help of the assembly language routine *put_phys_dword( ).*

4.3 Modifications in system calls of memory manager

In this section we will discuss the required modifications in the system calls of the memory manager to support linear address space in Minix.

4.3.1 The FORK System Call

In standard Minix, FORK routine takes a suitable sized hole, if available, from hole list and then copy the image of parent process in this area. But now memory manager is using linear address space so FORK is getting linear address space for the process from memory manager and not the physical memory. So, it is not feasible to copy the image of parent process here.

First this linear address space has to be mapped on physical address space (page frames) and then the image is copied in this address space. The physical address space is not exclusively allocated for mapping instead parent process' page table is copied in the child’s page table and flags in the child’s page table are used to denote page sharing or copy on access. A page fault handler is also needed to finally load the pages in the physical memory when page fault occurs.

This copying of tables is done by kernel through a routine *vm_fork()* defined in *vm386.c*. Actually, FORK routine in memory manager send a SYS_FORK message to kernel which is received by kernel through a routine *sys_task()* which then invoke *do_fork()* routine to process this message. Both these routines are defined in *system.c* file in kernel. The *do_fork()* routine then invokes vm_fork() routine.

4.3.2 The EXIT system call

This system call now has to unmapped the linear address space of the process also. This requires the clearing of corresponding entries from the page global directory and the page tables. This is done by the kernel through the modified routine *do_xit()* defined in System.c. The do_xit() does this by invoking a new routine *vm_unmap()* defined in vm386.c.

4.3.3 The EXEC System Call

EXEC system call is the most complex system call in Minix so it needs a careful handling. This system call now gets a linear address space from memory manager for the new process to be loaded. It needs the help of the kernel for mapping this linear address space on the physical memory.

For this purpose it sends a message SYS_EXECMAP to the kernel. This message is received by the *sys_task()* routine defined in *system.c* file in kernel. The *sys_task()* routine invokes a routine *do_execmap()* to process this message. The *do_exexmap()* routine check for the physical memory availability and if possible allocates the physical memory for the process. After this, control returns back to EXEC routine in memory manager which then works as in Standard minix.

4.3.4 The BRK system call

This system call has to ask the kernel for adjusting the size of stack and data segment because now only kernel has the actual physical memory map of processes and not the memory manager. So, kernel adjust the size of stack and text segments through a new routine *do_adjmap()*. Previously in standard Minix, memory manager itself used to adjust the segments' sizes as it was having the physical map of processes and then it informed the kernel to update its copy of memory map.

CHAPTER 5 IMPLEMENTATION DETAILS

In the previous chapter we discussed the basic design and modifications in system calls to support linear address space in Minix. Now we will discuss in detail the major routines involved in that design and also the modified system calls of memory manager.

5.1 Routines related to linear address space allocation in memory manager

This section is concerned with allocating and freeing arbitrary-size blocks of physical memory on behalf of the FORK and EXEC system calls. The key data structure used is the hole table, which maintains a list of holes in memory. It is kept sorted in order of increasing memory address. The addresses it contains refer to linear address space starting at absolute address *paging_base* (i.e., they are not relative to the start of memory manager). During system initialization, that part of memory containing the interrupt vectors, kernel, and memory manager are "allocated" to mark them as not available and to remove them from the hole list.

The data structure hole table is defined as follows.

```
PRIVATE struct hole {
  phys_clicks h_base;            /* where does the hole begin? */
  phys_clicks h_len;             /* how big is the hole? */
  struct hole *h_next;           /* pointer to next entry on the list */
} hole[NR_HOLES];
```

Where NR_HOLES is defined as:

```
#define NR_HOLES      128     /* max # entries in hole table */
```

The routines involved to manage this linear address space are given below.

5.1.1 The alloc_mem( ) routine

This routine allocates a block of memory from the free list using first fit. Its prototype is given below.

```
phys_clicks alloc_mem(clicks)
phys_clicks clicks;
```

The block consists of a sequence of contiguous bytes, whose length in clicks is given by 'clicks'. A pointer to the block is returned. The block is always on a click boundary. This procedure is called when memory is needed for FORK or EXEC.

On Intel architecture with virtual memory enabled, memory is allocated in chunks of 4M as already explained in the previous chapter. So, click is padded with extra bytes to make it a multiple of 4M as follows.

```
clicks= (clicks + MEM_PAD_CLICKS - 1) & ~(MEM_PAD_CLICKS - 1);
```

MEM_PAD_CLICKS is defined as follows.

```
#define MEM_PAD_CLICKS                    (0x400000 >> CLICK_SHIFT)
```

5.1.2 The free_mem( ) routine

This routine returns a block of free memory to the hole list. The prototype is as follows.

```
void free_mem(base, clicks)
phys_clicks base;                /* base address of block to free */
phys_clicks clicks;              /* number of clicks to free */
```

The parameters tell where the block starts in physical memory and how big it is. The block is added to the hole list. If it is contiguous with an existing hole on either end, it is merged with the hole or holes.

On Intel architecture with virtual memory enabled, memory should be allocated and freed at directory boundaries and this has been considered here through the following code.

```
clicks= (clicks + MEM_PAD_CLICKS - 1) & ~(MEM_PAD_CLICKS - 1);
  if (base & (MEM_PAD_CLICKS-1))
  {
            panic("Got non aligned free: ", base);
  }
```

5.1.3 The max_hole( ) routine

This routine scans the hole list and return the largest hole. The prototype is given below.

```
phys_clicks max_hole();
```

5.1.4 The mem_init( ) routine

This routine initializes hole lists. The prototype is given below.

```
void mem_init();
```

There are two lists: 'hole_head' points to a linked list of all the holes (unused memory) in the system; 'free_slots' points to a linked list of table entries that are not in use. Initially, the former list has one entry for each chunk of physical memory, and the second list links together the remaining table slots. As memory becomes more fragmented in the course of time (i.e., the initial big holes break up into smaller holes), new table slots are needed to represent them. These slots are taken from the list headed by 'free_slots'.

5.1.5 The mem_left( ) routine

This routine determines how much memory is left. This is called just after initialization to find the original amount. The prototype is given below.

```
phys_clicks mem_left();
```

5.2 Kernel routines to support linear address space in system calls of memory manager

To support linear address space in system calls of memory manage some new kernel routines are required as explained in the previous chapter. In this section those routines are elaborated.

5.2.1 The do_adjmap( ) routine

This routine changes the memory map for memory manage. This procedure is called when memory map is needed to be adjusted for BRK. The Prototype of this routine is as follows.

```
int do_adjmap(m_ptr)
message *m_ptr;
```

The parameter *m_ptr* is a pointer to request message. Message parameters are given below.

1. Where the map has to be stored.
2. Process whose map is to be loaded.
3. New size of the data segment.
4. Location of the stack pointer.
5. Virtual address of map inside caller (memory manage).

These message parameters are extracted using the following code.

```
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;
```

Then the change in data segment is calculated as follows.

```
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;
```

Now the change in stack segment is calculated as follows.

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

After that availability of physical memory is checked as segments are defined in terms of linear address space and that space has to be mapped on physical memory.

```
  if (vm_not_alloc < data_change + stack_change)
            return ENOMEM;
```

In the above lines, *vm_mot_alloc* represents the size of physical memory available at that particular instant. This variable is constantly updated by the kernel to reflect the changes in the size of available physical memory. This variable stores the size in terms of clicks.

After that it checks the gaps as shown below.

```
  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;
```

After checking the gaps, memory map is changed and available physical memory is updated accordingly as shown below.

```
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;
```

Finally updated memory map is reported back to memory manage by invoking a standard minix routine *do_getmap()* and *OK* is returned.

5.2.2 The do_execmap( ) routine

This routine removes old map and fetch new memory map from memory manage. This procedure is called when physical address space is needed for loading new process by EXEC. The Prototype of this routine is as follows.

```
int do_execmap(m_ptr)
message *m_ptr;
```

The parameter *m_ptr* is a pointer to request message. Message parameters are given below.

1. Whose space has the new map (usually memory manage).
2. Process whose map is to be loaded.
3. Value of flags before modification.
4. Virtual address of map inside caller (memory manage).

These message parameters are extracted using the following code.

```
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;
```

Then it copies new memory map from memory manage using the following code.

```
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);
```

In the above lines of code, *proc_ptr* points to kernel’s proc entry in the process table of kernel. After execution of the *phys_copy()*, *new_map* contains the memory map from memory manage.

After that availability of physical memory is checked as segments are defined in terms of linear address space and that space has to be mapped on physical memory.

```
if (vm_not_alloc < new_map[T].mem_len + new_map[D].mem_len +
        new_map[S].mem_len)
        return ENOMEM;
```

Then old memory is released by invoking a routine do_unmap(). This routine is explained in the next section.

The new map is copied in the entry of user getting the map in kernel’s process table as follows.

```
for (i= 0; i<NR_SEGS; i++)
        rp->p_map[i]= new_map[i];
```

After that base address and the top address of map is calculated as follow.

```
 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;
```

This linear address space starting from *base_address* upto *top_addr* is checked for un-mapping by invoking a routine *vm_check_umapped( ).* The *vm_check_unmapped* routine is explained in the coming sections. If this address space is un-mapped then it can be safely used for the user process invoking the EXEC system call.

Available Physical Memory size is decremented to allocate the physical address space to the user process.

```
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);
```

Finally ldt entries of the user process’ slot in the kernel’s process table are updated for the new linear address space by invoking the *alloc_segments()* routine defined in standard Minix, process is marked ready and *OK* is returned as shown below.

```
 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);
```

5.2.3 The do_unmap( ) routine

This routine removes memory map of a process. This is invoked by do_execmap() as explained above. The Prototype of this routine is as follows.

```
PRIVATE int do_unmap(m_ptr)
message *m_ptr;
```

The message parameter is extracted as shown below. k is the process whose map is to be freed i.e. the linear address space corresponding to the memory map of this process is to be unmapped.

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

After that base address and the size of memory map i.e. linear address space is calculated as follow.

```
 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;
```

To un-map this linear address space a routine named *vm_unmap()* is invoked as shown below. This routine is explained in the next section.

```
 vm_unmap(base, size, rp->p_map[T].mem_len + rp->p_map[D].mem_len +
        rp->p_map[S].mem_len);
```

Finally *OK* is returned.

5.2.4 The vm_unmap( ) routine

This routine removes physical address map of a process corresponding to linear address map passed to it as parameter. To do this it clears the entries from the page global directory and page tables corresponding to the linear address space passed to it. This is invoked by do_unmap() as explained above. The Prototype of this routine is as follows.

```
void vm_unmap(addr, vm_size, alloc_size)
phys_bytes addr;
phys_bytes vm_size;
phys_clicks alloc_size;
```

addr is the base address of the linear address space and *vm_size* is the size of linear address space. *alloc_size* is the size of linear address space actually mapped on physical address space.

Each time an entry from page global directory is taken starting from *addr* up to *top* = *addr* + *vm_size*. If it is unmapped next entry is taken as shown below.

```
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;
                }
```

The entry in the page global directory is cleared using the following code. *put_phys_dword( )* is an assembly routine which put the double word (second parameter) on the given physical address (first parameter).

```
put_phys_dword(dir_ent_addr, (u32_t)0);
```

After that, page table address is extracted from the directory entry as shown below.

```
page_ent_addr= dir_ent & VM_ADDRMASK;
```

Then each entry of this page table is read and the page frame on which it is mapped is freed by a routine rlmem_free(). This is shown below.

```
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--;
                }
```

After clearing the page table, the page frame containing the above page table is also freed using *rlmem_free( )* and *addr* is changed to point to the next entry in the page global directory as given below.

```
rlmem_free(dir_ent & VM_ADDRMASK);
                        addr += VM_DIRSIZE;
            }                                   /* end of while loop */
```

Finally after un-mapping the linear address space, size of available physical memory is updated as shown below.

vm_not_alloc += alloc_size + ((vm_size+VM_DIRSIZE-1) >> VM_DIRSHIFT);

5.2.5 The vm_check_unmapped( ) routine

This routine verifies whether the linear address space deduced by the provided arguments is unmapped or not. If not, the system gets terminated. The Prototype of this routine is as follows.

```
void vm_check_unmapped(base, top)
phys_bytes base;
phys_bytes top;
```

The parameter *base* is the base address of the linear address space and the parameter *top* is last address in the linear address space.

First of all it is checked that base is aligned on 4M boundary if not system terminates.

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

Then all entries corresponding to the linear address space in the page global directory is checked for un-mapping as shown below.

```
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 (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);
 }
assert (!dir_ent);  /* not mapped */
            }
```

5.2.6 The vm_fork() routine

This routine actually forks off a process at kernel level by mapping the linear address space of child process on the physical address space of parent. To do this, it copies the page global directory entries and page table entries of parent process in the respective slots of page global directory and page tables corresponding to the linear address space of the child process. The Prototype of this routine is as follows.

```
void vm_fork(parent, c_base_clicks)
struct proc *parent;
phys_clicks c_base_clicks;
```

The parameter *parent* is a pointer to the entry of parent process in the kernel's process table and the parameter *c_base_clicks* is the base address of linear address space in terms of clicks.

The linear address space of parent process is deduced from the process table entry as shown below.

```
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;
```

Parent's base is checked for alignment on 4M boundary. Then it is verified that parent's address space lies in the linear address space for processes.

```
assert (!(p_base & VM_DIRMASK));
assert (p_base >= paging_base);
```

Then the number of page global directory entries for parent is calculated as shown below.

```
dirs= ((p_top_clicks-p_base_clicks-1) >> (VM_DIRSHIFT-CLICK_SHIFT))+1;
```

Child’s base is checked for alignment on 4M boundary. Then it is verified that child’s address space lies in the linear address space for processes.

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

After that, the address of the first entry in the page global directory for the parent and chilld is calculated as follows.

```
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;
```

Entries from page global directory for parent are taken starting from *p_dir_ent_addr* one by one in a loop. Then the entry is checked whether it is blank. If it blank the next entry is taken. If an entry exists its flags are checked for validity. Then the   Page table address for this entry is calculated and stored in *p_page_ent_addr*.

```
for (i= 0; i<dirs; i++, p_dir_ent_addr += 4, c_dir_ent_addr += 4)        /* start of page global directory loop */
        {

                p_dir_ent= get_phys_dword(p_dir_ent_addr);
                if (!p_dir_ent)
                        {
                                continue;
                        }
                assert ((p_dir_ent & VM_INMEM_N_PRESENT) == VM_INMEM_N_PRESENT);
                p_page_ent_addr= p_dir_ent & VM_ADDRMASK;
```

Now entries from this page table for parent are read starting from *p_page_ent_addr* one by one in a loop. Then the entry is checked whether it is blank. If it blank the next entry is taken. If an entry exists its flags are checked for validity.

```
for (j= 0; j<1024; j++, p_page_ent_addr += 4)           /* Start of page table loop */
                {
                        p_page_ent= get_phys_dword(p_page_ent_addr);
                        if (!p_page_ent)
                        {
                                continue;
                        }
                        assert(p_page_ent & VM_INMEM);
```

Now the flags of the page table entry are checked and changed based on the type of page i.e. whether it is a text or data page.

```
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 */
          {
                    p_page_ent &= ~VM_WRITE;
                    vm_not_alloc++;
          }
          else      /* Data page */
          {
                    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;
}
```

In the above lines, page frames status is also updated by using a pointer *rlmem_table_base* to the page frame status table.

Then it is checked if page is copy on access or read only. It can't be both and INMEM has already been checked.

```
assert(p_page_ent & (VM_WRITE | VM_PRESENT));

                              if (p_page_ent & VM_PRESENT) /* Read only page */
                              {
                                        vm_not_alloc++;
                              }
```

As we are sharing the page frames of parent with child, so link count of page frames is increased as shown below.

```
          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);
}         /* End of page table loop */
```

After that a page frame is allocated to store the page table of the child and thenthe page table of the parent is copied in the newly created page table for child as shown below.

```
c_page_ent_addr= rlmem_getpage();
phys_copy(p_dir_ent & VM_ADDRMASK, c_page_ent_addr,
                                    VM_PAGESIZE);
```

Finally, the entry in the page global directory is done to map the page table created above. This entry is for the child as shown below.

```
map_dir(c_base+ (i<<VM_DIRSHIFT), c_page_ent_addr);
            }           /* End of page global directory loop */
```

CHAPTER 6 CONCLUSIONS AND FUTURE WORK

6.1 Conclusions

In this dissertation a design is proposed for linear address space allocation for Minix operating system. During the course of this project, emphasis is given on the memory manager for achieving this enhancement in the Minix operating system. Memory manager 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 proposed design aims to present a Minix operating system 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.

This dissertation presents a design using Intel based 80x86 family of processors in mind. This includes earlier 80386 (popularly known as 386), 80486 (popularly known as 486) and Pentium ^{TM} series of processors, which are almost omnipresent in today's computing world. The target machines chosen for this project are Pentium II and Pentium III based machines with 64 or 128 MB of physical memory (RAM) available.

This design is also tested for multi boot operation with Microsoft Windows 98^{TM} and Red Hat Linux 9.0^{TM} distributions on a target machine with hardware configuration as described above.

Due to intertwined nature of operating system, a few changes which are not directly related with linear space management are also needed in this project. For the purpose of not obfuscating the primary design proposed in this dissertation, these changes are not included here. The focus of this dissertation is kept on presenting the proposed design for linear address space management for Minix only. The design proposed in this dissertation is also successfully implemented on the target machines described above.

6.2 Future work

As already elaborated earlier the design presented in this dissertation focuses on Intel 80x86 architecture only. With a few minor changes, this design could be further extended or implemented as standalone on other architectures too. Some of the key changes should be kept in mind.

Contrary to Intel 80x86, some other architectures like Motorola 68000^{TM} have instructions with varying numbers of arguments. For example, the move instruction in this architecture has two arguments, source and target of the move. So, it can cause faults for three different reasons, i.e., the instruction itself and for either of the two operands (source or target).

The fault handler has to determine which reference faulted. On some computers, the operating system has to figure that out by interpreting the instruction and in effect simulating the hardware. Motorola 68000 made it easier for the operating system by updating the program counter as it goes. So, the program counter will be pointing at the word immediate following the part of the instruction that caused the fault. On the other hand, this makes it harder to restart the instruction.

Another important thing to remember is that some computers have addressing modes that automatically increment or decrement index registers as a side effect, making it easy to simulate the effect of C statement `*p++ = *q++;` in a single step.

Unfortunately, if an instruction faults part-way through, it may be difficult to figure out which registers have been modified so that they can be restored to their original state. Some computers also have instructions such as *move characters*,' which work on variable-length data fields, updating a pointer or count register. If an operand crosses a page boundary, the instruction may fault part-way through, leaving a pointer or counter register modified.

Another possible extension of this design is to propose a design to support virtual addressing with paging. This design has already created the framework necessary for this.

To implement paging, the design for a pager should be proposed. This pager could swap pages between physical memory and swap area.

Also, the design issues like working set model, thrashing and global versus local allocation policies are needed to be considered for a paging system. To support swap partition, some changes in the Minix file system would also be required. A page replacement policy will need to be required for the implementation of pager. These changes will extend Minix to support complete virtual memory management scheme like virtual addressing with paging.

REFERENCES

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

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

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

[4] 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.

[5] 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.

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

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

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

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

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

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

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

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

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

[15] U. Vahalia: UNIX Internals-The New Frontiers, Prentice Hall, 1996.

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

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

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

[19] 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.

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

[21] 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.

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

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

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

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

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

[27] 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.

[28] 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).

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

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

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

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

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

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

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

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

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

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 */
```

++

alloc.c

++

```
/* This file is concerned with allocating and freeing arbitrary-size blocks of
 * physical memory on behalf of the FORK and EXEC system calls.  The key data
 * structure used is the hole table, which maintains a list of holes in memory.
 * It is kept sorted in order of increasing memory address. The addresses
 * it contains refer to physical memory, starting at absolute address 0
 * (i.e., they are not relative to the start of MM).  During system
 * initialization, that part of memory containing the interrupt vectors,
 * kernel, and MM are "allocated" to mark them as not available and to
 * remove them from the hole list.
 *
 * The entry points into this file are:
 *   alloc_mem:    allocate a given sized chunk of memory
 *   free_mem:     release a previously allocated chunk of memory
 *   mem_init:     initialize the tables when MM start up
 *   max_hole:     returns the largest hole currently available
 *   mem_left:     returns the sum of the sizes of all current holes
 */

#if _VMD_EXT
/* Enable misc. extensions to alloc.c */
#ifndef VMDEXT_MM_ALLOC
#define VMDEXT_MM_ALLOC 1
#endif
#endif /* _VMD_EXT */

#include "mm.h"
#include "assert.h"
INIT_ASSERT

#define NR_HOLES         128     /* max # entries in hole table */
#define NIL_HOLE (struct hole *) 0

PRIVATE struct hole {
  phys_clicks h_base;                 /* where does the hole begin? */
  phys_clicks h_len;                  /* how big is the hole? */
  struct hole *h_next;                /* pointer to next entry on the list */
} hole[NR_HOLES];


PRIVATE struct hole *hole_head;               /* pointer to first hole */
PRIVATE struct hole *free_slots;              /* ptr to list of unused table slots */

FORWARD _PROTOTYPE( void del_slot, (struct hole *prev_ptr, struct hole *hp) );
FORWARD _PROTOTYPE( void merge, (struct hole *hp)                              );

/*===========================================================================*
 *                              alloc_mem                                    *
 *===========================================================================*/
PUBLIC phys_clicks alloc_mem(clicks)
phys_clicks clicks;                   /* amount of memory requested */
{
/* Allocate a block of memory from the free list using first fit. The block
 * consists of a sequence of contiguous bytes, whose length in clicks is
 * given by 'clicks'.  A pointer to the block is returned.  The block is
 * always on a click boundary.  This procedure is called when memory is
 * needed for FORK or EXEC.
 */

  register struct hole *hp, *prev_ptr;
  phys_clicks old_base;

#if (CHIP == INTEL) && VIRT_MEM
  /* On a 386 with virtual memory enabled, memory is allocated in chunks of
   * 4M.
   */
  clicks= (clicks + MEM_PAD_CLICKS - 1) & ~(MEM_PAD_CLICKS - 1);
#endif /* CHIP == INTEL && VIRT_MEM */

  hp = hole_head;
#if VMDEXT_MM_ALLOC
  /* Passing uninitialized variables to a function (del_slot) is bad style. */
  prev_ptr= NULL;
#endif /* VMDEXT_MM_ALLOC */
  while (hp != NIL_HOLE) {
	if (hp->h_len >= clicks) {
		/* We found a hole that is big enough.  Use it. */
		old_base = hp->h_base;		/* remember where it started */
		hp->h_base += clicks;	/* bite a piece off */
		hp->h_len -= clicks;	/* ditto */

		/* If hole is only partly used, reduce size and return. */
#if VMDEXT_MM_ALLOC
		if (hp->h_len == 0)
		{
			/* The entire hole has been used up.  Manipulate free
			 * list. */
			del_slot(prev_ptr, hp);
		}
#if (CHIP == INTEL) && VIRT_MEM
		if (old_base & (MEM_PAD_CLICKS-1))
		{
			panic("Got non aligned memory: ", old_base);
		/* on a 386 vm system memeory should be allocated at directory
		 * bounderies */
		 }
#endif /* CHIP == INTEL && VIRT_MEM */
#else /* !VMDEXT_MM_ALLOC */
		if (hp->h_len != 0) return(old_base);

		/* The entire hole has been used up.  Manipulate free list. */
		del_slot(prev_ptr, hp);
#endif /* VMDEXT_MM_ALLOC */
		return(old_base);
	}

	prev_ptr = hp;
	hp = hp->h_next;
  }
  return(NO_MEM);
}


/*===========================================================================*
 *				free_mem				     *
 *===========================================================================*/
PUBLIC void free_mem(base, clicks)
phys_clicks base;		/* base address of block to free */
phys_clicks clicks;		/* number of clicks to free */
{
/* Return a block of free memory to the hole list.  The parameters tell where
 * the block starts in physical memory and how big it is.  The block is added
 * to the hole list.  If it is contiguous with an existing hole on either end,
 * it is merged with the hole or holes.
 */

  register struct hole *hp, *new_ptr, *prev_ptr;

#if DEBUG & 256
 { where(); printf("free_mem(0x%x, 0x%x)\n", base, clicks); }
#endif

  assert(base != 0);

#if VMDEXT_MM_ALLOC
#if (CHIP == INTEL) && VIRT_MEM
  /* on a 386 vm system memeory should be allocated and freed at directory
   * bounderies */
  clicks= (clicks + MEM_PAD_CLICKS - 1) & ~(MEM_PAD_CLICKS - 1);
  if (base & (MEM_PAD_CLICKS-1))
  {
        panic("Got non aligned free: ", base);
  }
#endif /* CHIP == INTEL && VIRT_MEM */

  if ( (new_ptr = free_slots) == NIL_HOLE)
  {
        printf("dumping hole list:\n");
        for(hp= hole_head; hp; hp= hp->h_next)
        {
                printf("base= %d, len= %d\n", hp->h_base, hp->h_len);
        }
        panic("Hole table full", NO_NUM);
  }
#else /* !VMDEXT_MM_ALLOC */
  if ( (new_ptr = free_slots) == NIL_HOLE) panic("Hole table full", NO_NUM);
#endif /* VMDEXT_MM_ALLOC */
  new_ptr->h_base = base;
  new_ptr->h_len = clicks;
  free_slots = new_ptr->h_next;
  hp = hole_head;

  /* If this block's address is numerically less than the lowest hole currently
   * available, or if no holes are currently available, put this hole on the
   * front of the hole list.
   */
  if (hp == NIL_HOLE || base <= hp->h_base) {
        /* Block to be freed goes on front of the hole list. */
        new_ptr->h_next = hp;
        hole_head = new_ptr;
        merge(new_ptr);
        return;
  }

  /* Block to be returned does not go on front of hole list. */
  while (hp != NIL_HOLE && base > hp->h_base) {
        prev_ptr = hp;
        hp = hp->h_next;
  }

  /* We found where it goes.  Insert block after 'prev_ptr'. */
  new_ptr->h_next = prev_ptr->h_next;
  prev_ptr->h_next = new_ptr;
  merge(prev_ptr);              /* sequence is 'prev_ptr', 'new_ptr', 'hp' */
}


/*===========================================================================*
 *                              del_slot                                     *
 *===========================================================================*/
PRIVATE void del_slot(prev_ptr, hp)
register struct hole *prev_ptr; /* pointer to hole entry just ahead of 'hp' */
register struct hole *hp;       /* pointer to hole entry to be removed */
{
/* Remove an entry from the hole list.  This procedure is called when a
 * request to allocate memory removes a hole in its entirety, thus reducing
 * the numbers of holes in memory, and requiring the elimination of one
 * entry in the hole list.
 */

  if (hp == hole_head)
        hole_head = hp->h_next;
  else
        prev_ptr->h_next = hp->h_next;

  hp->h_next = free_slots;
  free_slots = hp;
}


/*===========================================================================*
 *                              merge                                        *
 *===========================================================================*/
PRIVATE void merge(hp)
register struct hole *hp;       /* ptr to hole to merge with its successors */
```

```
{
/* Check for contiguous holes and merge any found.  Contiguous holes can occur
 * when a block of memory is freed, and it happens to abut another hole on
 * either or both ends.  The pointer 'hp' points to the first of a series of
 * three holes that can potentially all be merged together.
 */

  register struct hole *next_ptr;

  /* If 'hp' points to the last hole, no merging is possible.  If it does not,
   * try to absorb its successor into it and free the successor's table entry.
   */
  if ( (next_ptr = hp->h_next) == NIL_HOLE) return;
#if PCH_DEBUG
 if (hp->h_base + hp->h_len > next_ptr->h_base)
 {
		printf("hp->h_base= 0x%x, hp->h_len= 0x%x, next_ptr->h_base= 0x%x\n",
			hp->h_base, hp->h_len, next_ptr->h_base);
		panic("merge: overlapping holes", NO_NUM);
 }
#endif
  if (hp->h_base + hp->h_len == next_ptr->h_base) {
		hp->h_len += next_ptr->h_len;	/* first one gets second one's mem */
		del_slot(hp, next_ptr);
  } else {
		hp = next_ptr;
  }

  /* If 'hp' now points to the last hole, return; otherwise, try to absorb its
   * successor into it.
   */
  if ( (next_ptr = hp->h_next) == NIL_HOLE) return;
#if PCH_DEBUG
 if (hp->h_base + hp->h_len > next_ptr->h_base)
 {
		printf("hp->h_base= 0x%x, hp->h_len= 0x%x, next_ptr->h_base= 0x%x\n",
			hp->h_base, hp->h_len, next_ptr->h_base);
		panic("merge: overlapping holes", NO_NUM);
 }
#endif
  if (hp->h_base + hp->h_len == next_ptr->h_base) {
		hp->h_len += next_ptr->h_len;
		del_slot(hp, next_ptr);
  }
}


/*===========================================================================*
 *				max_hole				     *
 *===========================================================================*/
PUBLIC phys_clicks max_hole()
{
/* Scan the hole list and return the largest hole. */

  register struct hole *hp;
  register phys_clicks max;

  hp = hole_head;
  max = 0;
  while (hp != NIL_HOLE) {
		if (hp->h_len > max) max = hp->h_len;
		hp = hp->h_next;
  }
  return(max);
}


/*===========================================================================*
 *				mem_init				     *
 *===========================================================================*/
PUBLIC void mem_init()
{
/* Initialize hole lists.  There are two lists: 'hole_head' points to a linked
 * list of all the holes (unused memory) in the system; 'free_slots' points to
 * a linked list of table entries that are not in use.  Initially, the former
 * list has one entry for each chunk of physical memory, and the second
 * list links together the remaining table slots.  As memory becomes more
 * fragmented in the course of time (i.e., the initial big holes break up into
 * smaller holes), new table slots are needed to represent them.  These slots
 * are taken from the list headed by 'free_slots'.
 */

  register struct hole *hp;
  phys_clicks base;                   /* base address of chunk */
  phys_clicks size;                   /* size of chunk */

  /* Put all holes on the free list. */
  for (hp = &hole[0]; hp < &hole[NR_HOLES]; hp++) hp->h_next = hp + 1;
  hole[NR_HOLES-1].h_next = NIL_HOLE;
  hole_head = NIL_HOLE;
  free_slots = &hole[0];

  /* Allocate a hole for each chunk of physical memory. */
  while (get_mem(&base, &size))
        free_mem(base, size);
}


/*===========================================================================*
 *                              mem_left                                     *
 *===========================================================================*/
PUBLIC phys_clicks mem_left()
{
/* Determine how much memory is left.  This procedure is called just after
 * initialization to find the original amount.
 */

  register struct hole *hp;
  phys_clicks tot;

  for (hp = hole_head, tot = 0; hp != NIL_HOLE; hp = hp->h_next)
        tot += hp->h_len;
  return(tot);
}

#if PCH_DEBUG
/*===========================================================================*
 *                              print_mem                                    *
 *===========================================================================*/
PUBLIC void print_mem()
{
/* Print the current hole map.
 */

  struct hole *hp;
  for (hp= hole_head; hp; hp= hp->h_next)
  {
        printf("0x%xK - 0x%xK (%ldK)\n", hp->h_base*CLICK_SIZE/1024,
                (hp->h_base + hp->h_len-1)*CLICK_SIZE/1024,
                hp->h_len*CLICK_SIZE/1024);
  }
}
#endif /* PCH_DEBUG */

/*
 * $PchId: alloc.c,v 1.3 1995/11/28 07:21:13 philip Exp $
 */
```

++

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 */
```

++

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