Design and Implementation of Page Frame Allocation in 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

RAVISHANKAR PRASAD
College Roll No. 01/CTA/03
Delhi University Roll No. 3001

Under the guidance of
Prof. Asok De

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

July-2005

CERTIFICATE

This is to certify that the dissertation entitled "**Design and Implementation of Page Frame allocation in an Operating System**" submitted by **Ravishankar Prasad** in the partial fulfillment of the requirement for the award of degree of **Master of Engineering** in Computer Technology and Application, Delhi College of Engineering is an account of his work carried out under my guidance and supervision.

Professor D. Roy Choudhury

Head of Department
Department of Computer Engineering
Delhi College of Engineering
Delhi

Prof. Asok De

Head of Department
Department of Information Technology
Delhi College of Engineering
Delhi

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 **Professor Asok De**, Head of Department, Department of Information Technology 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. Goldie Gabrani, 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.

Ravishankar Prasad
M.E. (Computer Technology & Applications)
College Roll No. 01/CTA/03
Delhi University Roll No. 3001

Abstract

In a memory management system with paging, the entire physical memory is divided in form of page frames. A suitable mechanism is needed to dynamically allocate and de-allocate these page frames to various processes as and when needs arise. This dissertation proposes a mechanism for the page frame management and allocation to support virtual addressing in MINIX operating system for Intel 80x86 architecture, from 80386 onwards. The standard MINIX distribution does not support virtual addressing.

The proposed design for the memory management system is non-paging (i.e. paging without virtual memory support). This dissertation proposes all the necessary changes that should be made to the memory manager and kernel of MINIX operating system to achieve non-paging. This design uses two level paging system of Intel 80x86 architecture with single page directory for all the processes in the system.

As paging unit of base architecture is enabled to support page frame management for virtual addressing in MINIX, page fault may occur. So there is a need to handle the page fault. The dissertation also proposes an exception handler for the page fault. As it is a non-paging system, page fault handler is simple and it does not need any page replacement algorithm.

CONTENTS

1 Introduction……..…...…… 1

1.1 Basic memory management……..…… 2
1.2 Fixed partitioning ……..……... ... 3
1.2.1 single partition allocation ……..…… 3
1.2.2 multiple partition allocation ……. 4
1.3 Variable partitioning……. 5
1.3.1 Memory management using bit maps……... 7
1.3.2 Memory management using linked lists……... 8
1.3.3 Swapping……... 9
1.4 Paging……..……..……... 10
1.4.1 Hardware support for paging ……... 10
1.4.2 Non paging……..…… 11
1.4.3 Paging with virtual memory……. 12
1.4.4 Page size.…….……. 13
1.4.5 Page fault…….……. 14
1.5 Segmentation……. 15
1.6 Page frame management……...…… 17
1.7 Problem description……... 18
1.8 Dissertation organization.....……..... 20

2 Memory Management In MINIX...……...…… 21

2.1 Memory allocation to processes…… 21
2.2 Internal memory layout of a process …….…... 23
2.2.1 Non-separate I & D representation…… 25
2.2.2 Separate I & D representation……... 26
2.2.3 Shared text in separate I & D representation……. 27
2.3 Memory manager’s data structures……... 28
2.3.1 Data structures for internal memory representation of a process….. 28
2.3.2 The process table……... 29
2.3.3 Hole list……. 30
2.4 Memory management algorithm……... 32
2.4.1 Memory initialization 32
2.4.2 Memory allocation algorithm…… 33
2.4.3 Algorithm for freeing memory…….. 33

3 Paging In Intel 80x86…….…… 35

3.1 Virtual memory management in 32-bit Intel architecture……. 35
3.1.1 Page table…….. 37
3.1.2 Two level paging……... 38
3.1.3 Page directory table. …… 40
3.1.4 Page table in two level paging……... 41
3.1.5 Page table entries…… 41
3.2 Page Fault…… 43

3.2.1 Page Fault handling…… 43
3.2.2 Hardware support…… 45

4 System Design ……..……... 47

4.1 Data structures …… 47
4.1.1 Chunk table ……..…… 47
4.1.2 Page frame table ……. 49
4.2 Design ……..……... 51
4.2.1 Page frame management……... 52
4.2.2 Creation of page directory and page table ……... 53
4.2.3 Page fault handling ……. 54

5 Design Implementation... ……. 56

5.1 Routines related to operation in chunk table……... 56
5.1.1 chunk_init( ) routine ……..……... 56
5.1.2 chunk_add( ) routine ……... 56
5.1.3 chunk_del( ) routine…… 58
5.1.4 chunk_find( ) routine……... 60
5.2 Routines related to operation in page frame table…….. 60
5.2.1 rlmem_init( ) routine……... 60
5.2.2 rlmem_free( ) routine……... 61
5.2.3 rlmem_getpage( ) routine…… 62
5.3 Routines related to operation in page table and page directory……..……. 63
5.3.1 map_dir( ) routine……..……... 63
5.3.2 map_page( ) routine…….……..... 65
5.3.3 vm_init( ) routine…….……..... 66
5.4 Routine for page fault handling……... 67
5.4.1 vm_page_fault ( ) routine……..…….. 67

6 Conclusions and Future Work…… 71

6.1 Conclusion……... 71
6.2 Future work……. 72

References……..…….…..... 74

Source Code……..…… 78

CHAPTER 1 INTRODUCTION

Memory is one of the primary resources in a computer system. The Memory hierarchy of a computer system includes a small amount of very fast, expensive, volatile cache memory, a few hundreds of MB of a medium-sized, medium-priced, volatile main memory and several GB of low-priced, non-volatile secondary storage like Hard Disks.

From a programmer or user's point of view, typically main memory of the system should be an infinite size, very fast and non-volatile storage which could be relied upon even in case of power failures. If such would be the case then as many programs as a user want to use could be loaded into main memory without ever worrying about freeing memory space for other competing processes. Also, there wouldn't be any data loss due to power failures, etc. If such a case could have been possible then probably no complex memory management schemes would be required for operating systems in general.

But, unfortunately the current day situation is at extreme ends with this idealistic notion. Although in recent years, the memory sizes have gone up and the rates are down, and work on non-volatile and faster memories have progressed very well, but we are still very far from the ideal scenario stated above.

Also, with advent of larger and cheaper main memories program sizes are increasing at a very fast rate. In fact, even the memory requirement for running operating systems alone has gone very high. Microsoft Windows XP requires a least 128MB of Main Memory to perform at any acceptable level. The Next Version of Operating System from Microsoft Family code named 'Longhorn' would need still higher Main memory as a System Requirement. The recommended configuration for this Operating System according to the published report is going to 512 MB of Main Memory.

Similarly all Modern day games and applications are stretching system memory to its limits. It is not possible for any Operating System to work properly unless it manages the system memory in a highly efficient manner. In effect the need for better and more efficient memory management schemes are fast becoming not only a requirement, but also more of a norm for all modern day operating systems.

Memory Management deals with managing the memory hierarchy in such a manner that the limitations of each of the three types of storage could be kept to as limited as possible and best possible use of advantages of each one could be sought. The primary aim of all memory management schemes is to run different application simultaneously in such a manner that the best possible use of faster memories could be made.

1.1 Basic memory management

Memory management systems can be divided into two classes: those that allocate contiguous memory to the processes and those that allocate non contiguous memory. As the word *contiguous* refers to continuation, so the contiguous memory allocation means allocation of memory in continuous locations without any break in between. So, if a program needs 8 MB of memory then all this memory will be allocated to it in a single block of 8MB with no breaks[24].

In contrast, in non-contiguous memory allocation a program may be scattered in many smaller parts through out the memory. These parts need not be allocated in any sequence can be randomly spaced throughout the memory.

Both these schemes have merits and demerits and are sub-categorized as shown in the Figure 1.1.

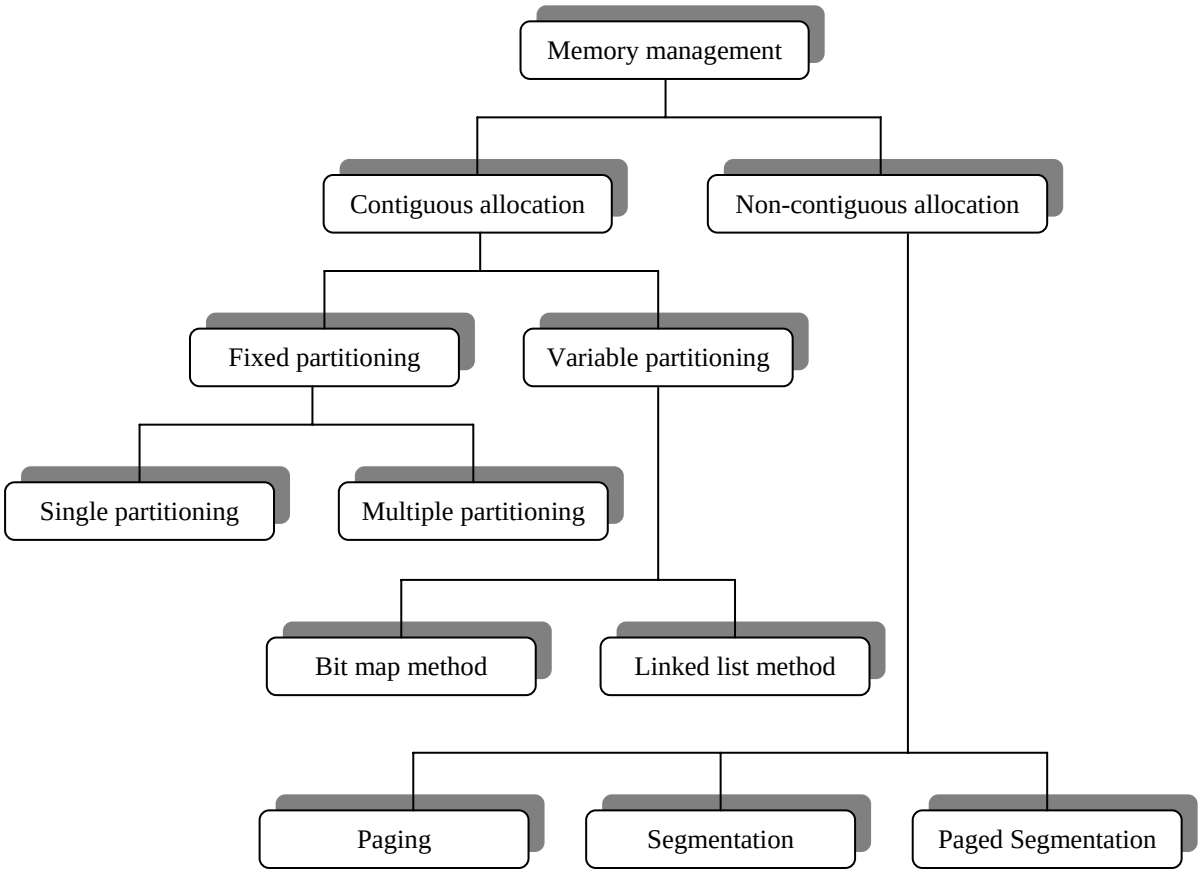


Figure 1.1: Classification of memory management schemes

1.2 Fixed partitioning

In fixed partitioning the total available memory for user processes is managed as a single entity and it is divided into partitions of predetermined sizes. As and when a request from a user process is received one of these partitions is allocated to that user process according to partition size and memory requirement of the process. In this type of memory management schemes only one programs could reside in a memory space, during the complete time of its execution and that memory block is freed only when the program terminates[9].

1.2.1 Single partition allocation

Single partition allocation is the simplest memory management scheme available. When the system is organized in this way, only one process can run at a time. As soon as the user

types a command, the operating system copies the requested program from disk to memory and executes it. When the process finishes, the operating system displays a prompt character and waits for a new command. When it receives the command, it loads a new program into memory, overwriting the first one.

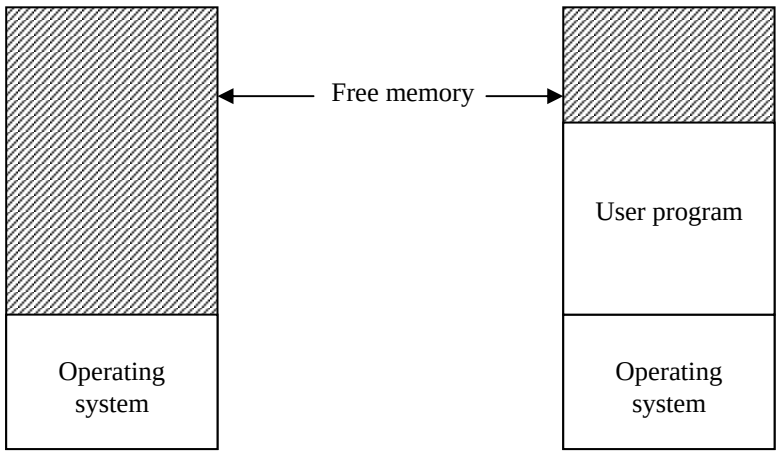


Figure 1.2: Single partition memory allocation

Figure 1.2 (a) shows the main memory of a computer system in case of single partition allocation scheme, while Figure 1.2(b) shows the memory after a program has been loaded. Note that although there is free memory available but no other program can be loaded in the free space available. Although this scheme is simple and easy to implement, but it can not support multiprogramming[9].

1.2.2 Multiple partition allocation

This is the simplest memory management scheme for multiprogramming operating systems. In this scheme the available user memory is divided into a number partitions of predetermined sizes. Only one user process can reside in a partition at a time. When a user wants to execute a process, the memory manager searches for a free available partition that is sufficient to hold this process. If such partition exists it is allocated to the user process. The process remains in this partition till completion. This memory allocation scheme is shown in the Figure 1.3.

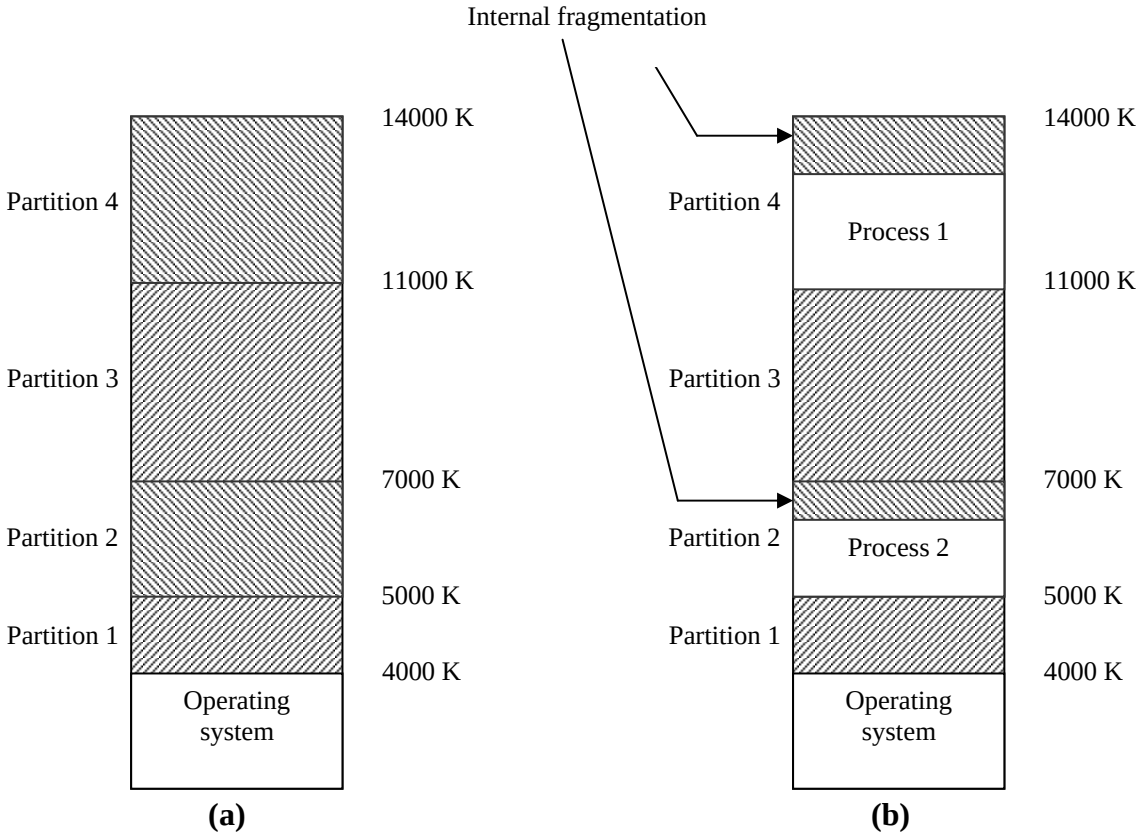


Figure 1.3 : (a) User memory divided into 4 different partitions. (b) Two processes (1 & 2) residing in main memory.

Internal fragmentation: when a process is loaded in a partition, in need be occupy all the partition and some space may remain unused. This scenario is known as internal fragmentation. Internal fragmentation occurs when memory is divided into fixed size blocks. Internal fragmentation is wastage of memory and is proportional to the size of the memory block.

1.3 Variable partitioning

In Variable partitioning scheme instead of managing the whole user space in terms of fixed size blocks, the whole memory is managed only in terms of free and used memory spaces. When a user process request for memory allocation, the required amount of memory from the free space is allocated to the user process and marked as used. Similarly when the process terminates the memory returns back to the free pool. The main

difference between the fixed partitions the variable partitions is that the number, location, and size of the partitions vary dynamically in the latter as processes come and go, whereas they are fixed in the former. The flexibility of not king tied to a fixed number of partitions that may be too large or too small improves memory utilization, but it also complicates allocating and deallocating memory, as well as keeping track of it[1].

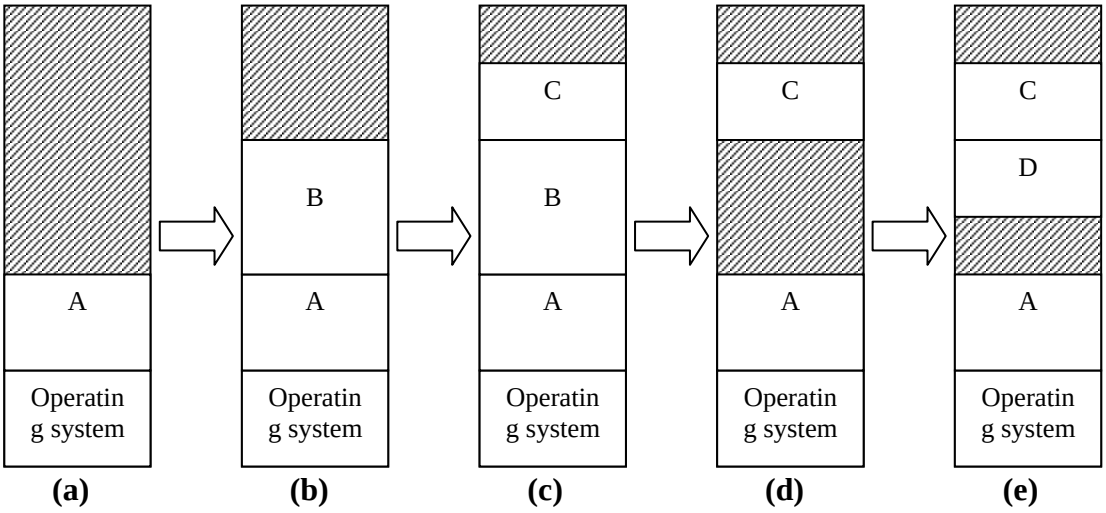


Figure 1.4: variable partition allocation

Operation of variable partitioning system is shown in the Figure 1.4. Initially only process A is in memory rest of the memory is a single free block. Then process B and C come into memory. After some time process B leaves the memory (completes its execution) free memory is divided into two parts. Then process D comes into memory.

External fragmentation: In Figure 1.4(d) there are two free memory blocks. Let their total size is 8k. Now a process E of size 8k wants to execute. We can not allocate space to this process although sufficient memory to hold this process is available. The reason is that the two free memory blocks are non-contiguous. This scenario is known as external fragmentation. External fragmentation occurs in case of variable partitioning. It can be avoided using *memory compaction.*

Memory compaction: When there are multiple holes in memory, it is possible to combine them all into one big one by moving all the processes downward as far as possible. This

technique is known as memory compaction. It is usually not done because it requires a lot of CPU time.

As process comes into memory and goes off, the number of holes in the memory may increase. These free memory blocks must be managed very carefully. There are different strategies available for this purpose. Now we will discuss two of them.

1.3.1 Memory management using bit maps

In this method, memory is divided up into allocation units, perhaps as small as a few words and perhaps as large as several kilobytes. Corresponding to each allocation unit is a bit in the bit map, which is 0 if the unit is free and 1 if it is occupied (or vice versa). Figure 1.5 shows part of memory and the corresponding bit map.

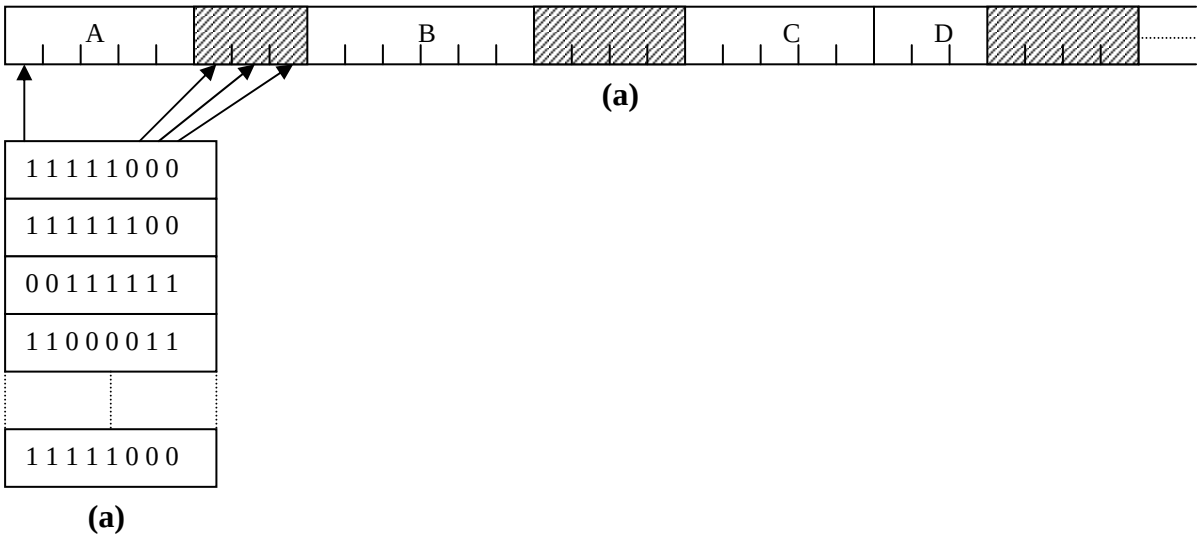


Figure 1.5: Memory management using bit maps

The size of the allocation unit is an important design issue. The smaller the allocation unit, the larger the bit map. If the allocation unit is chosen large, the bit map will be smaller, but appreciable memory may be wasted in the last unit if the process size is not an exact multiple of the allocation unit[1].

A bit map provides a simple way to keep track of memory words in a fixed amount of memory because the size of the bit map depends only on the size of memory and the size of the allocation unit. The main problem with it is that when it has been decided to bring a k unit process into memory, the memory manager must search the bit map to find a run of k consecutive 0 bits in the map. Searching a bit map for a run of a given length is a slow operation (because the run may straddle word boundaries in the map); this is an argument against bit maps.

1.3.2 Memory management using linked lists

Another way of keeping track of memory is to maintain 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 1.6(a) is represented in Figure 1.6(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.

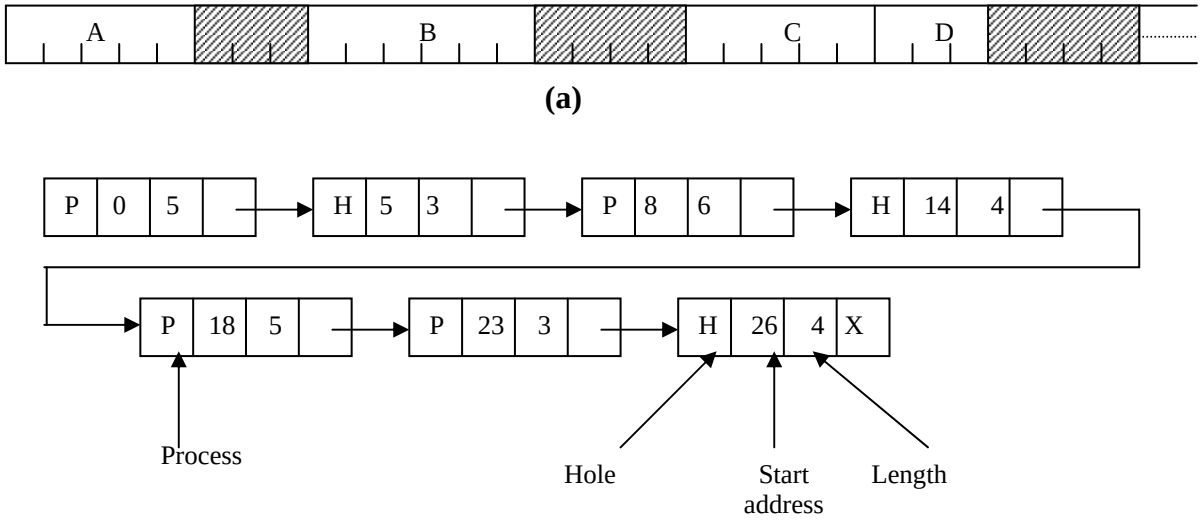


Figure 1.6: Linked list representation of memory

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[1].

- 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[1].

- 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.

1.3.3 Swapping

In timesharing systems or graphically oriented personal computers, sometimes there is not enough main memory to hold all the currently active processes. In this case the memory manager may select one or more processes to remove temporary from the main memory and bring back when memory is free. The part of memory manager performs this task is known as *swapper* and this process is known as *swapping*. Mechanism of swapping is shown in the Figure 1.7.

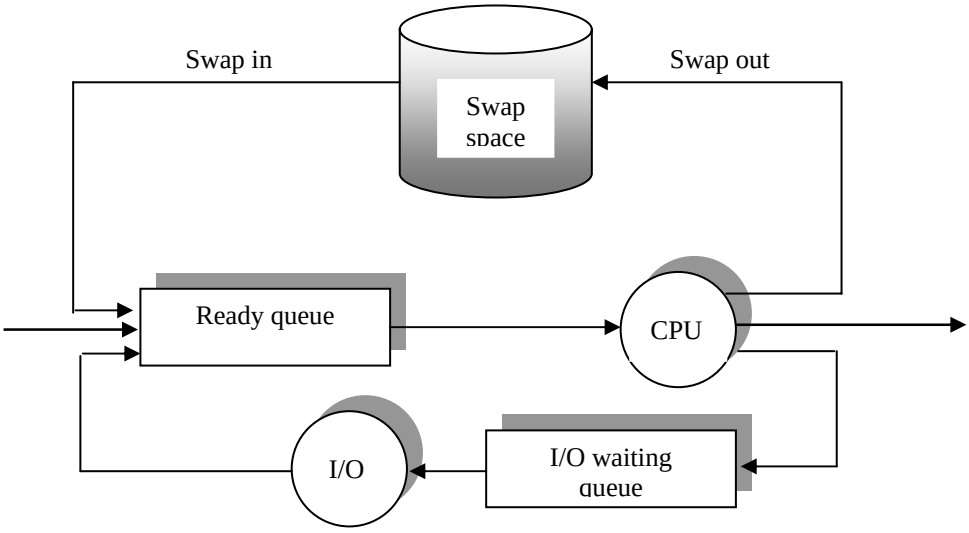


Figure 1.7: Swapping

Swapping requires a backing store. The backing store is commonly a fast disk. It must be large enough to accommodate the copies of all processes for all users[9].

1.4 Paging

In this memory management scheme, whole memory is divided into fixed-sized blocks called *page frames*. The program is also divided into fixed sized blocks called *pages.* Size of page is equal to the size of the page frame. 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[9].

1.4.1 Hardware support for paging

All modern computer systems consist of hardware for paging support. The basic hardware for this purpose is shown in the Figure 1.8. Every address generated by the CPU is divided into two parts: a *page number (p)* and a *page offset (d).* 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 *p* we get the *frame number (f)* from the page table. Now as a final step, page offset *d* is added to the frame number *f* to get the actual data. The page table is also stored in a page frame.

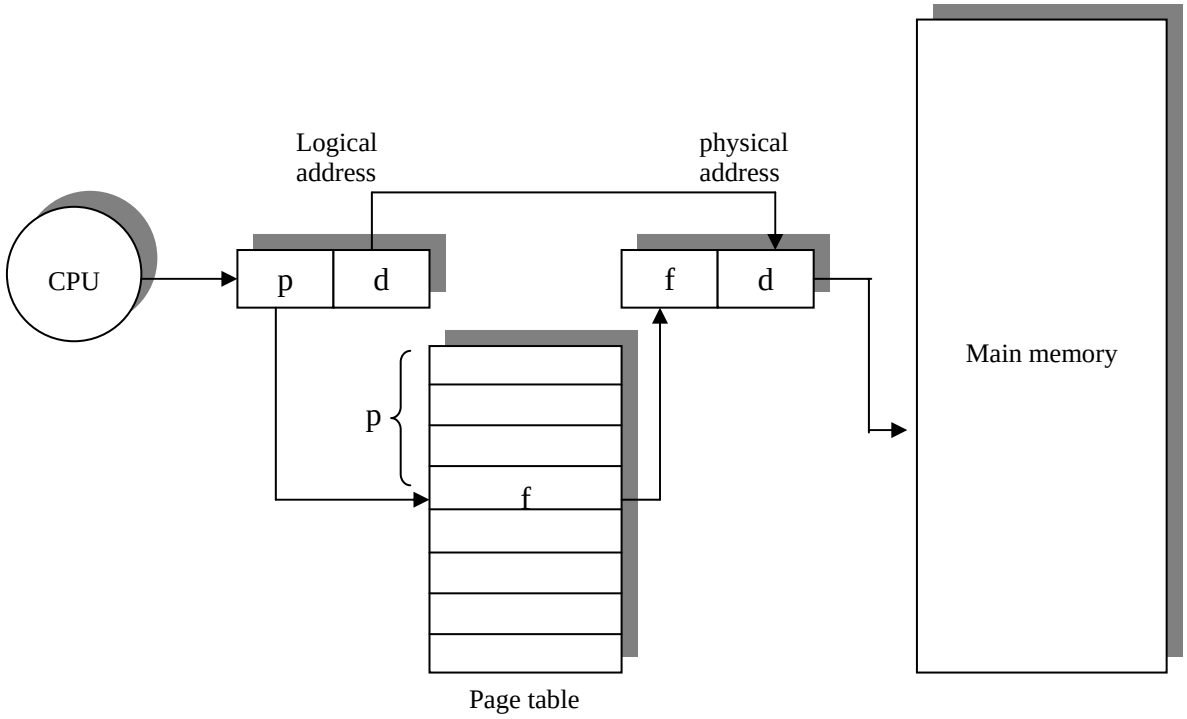


Figure 1.8: paging hardware

1.4.2 Non paging

It is very basic method of paging. In this paging scheme all the pages of a program resides in main memory. But these pages may be scattered anywhere in the memory. Once pages are loaded into page frame, they can not be removed until the process completes its execution. To load a process into main memory, the memory manager searches for the required number of free page frames sufficient to hold that process. These page frames need nod be contiguous. Once such page frames has been found, memory manager allocates these page frames to the process. Now these page frames are marked as used page frame. The next step is to create the page table for this process[15].

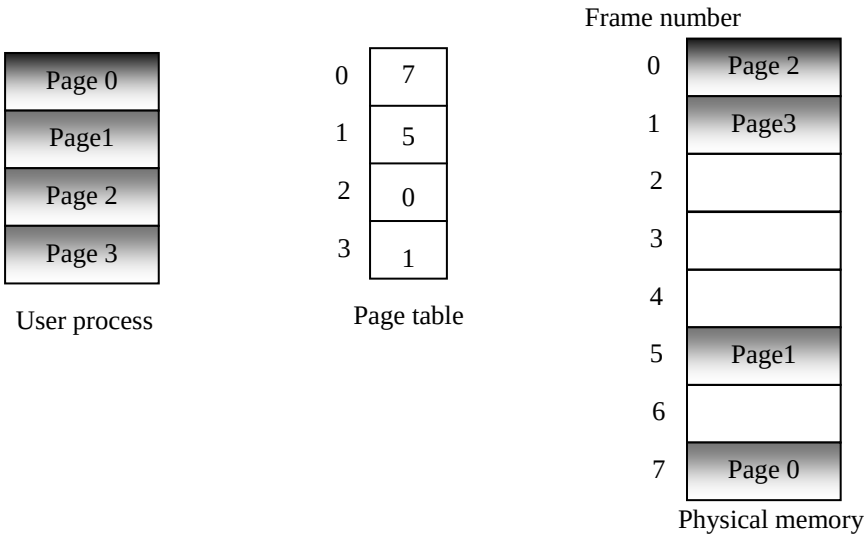


Figure 1.8: Example of non-paging

1.4.3 Paging with virtual memory

This memory management scheme is enhancement to the non-paging scheme. It provides the full benefit of the paging circuitry of a computer system. In this case it is not necessary to allocate page frames to all the pages of a process. Each process in the system is allocated some page frames (less ten the number of pages in the process) and process resides partially in the memory. The page table consists of a valid bit entry for each page. This is a one-bit entry and indicates whether this page is present in memory or not (1 – present, 0- not present). When the user process want to execute an instruction, the valid bit entry corresponding to the page containing this instruction is searched. If the bit is one, page is present in the main memory. If not, *page fault* occurs. The memory manager the desired page into an empty frame. This paging mechanism is shown in the Figure 1.9. As clear from the picture, the process resides partially in the main memory.

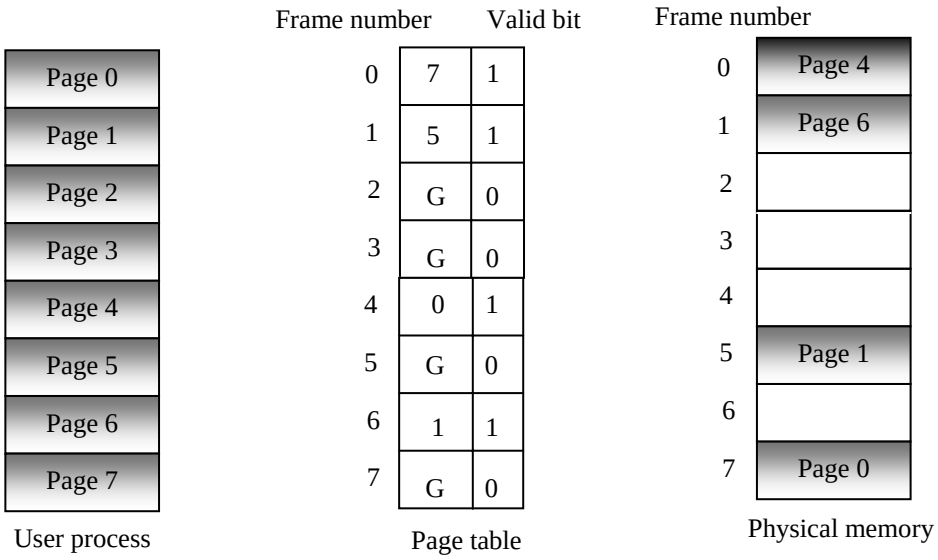


Figure 1.9: Example of paging with virtual memory

1.4.4 Page size

How big should a page be? This is really a hardware design question, but since it depends on OS considerations, we will discuss it here. If pages 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[14].

If space overhead is the *only* consideration, it can be shown that the optimal size of a page is square root of 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.

1.4.5 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 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.

Page fault in non-paging

As already explained earlier, in non-paging memory management scheme pages are never paged out of the physical memory throughout the period of execution of the process. But still page faults occurs. To understand the reason for page faults in this case, it is necessary to have a look at the concept of demand paging.

Demand Paging: Policy that allocates a page in physical memory only if an address on that page is actually referenced (demanded) in the executing program is known as demand paging.

Whenever page frames are allocated to a process for the first time, the pages are not brought into the physical memory de facto. As and when a request is generated to reference a page, a page fault occurs and the page is transferred to physical memory. So, in effect for a process at the most as many page faults could occur as the number of pages in

it. One page could generate only a single page fault during its execution and will never be paged out of physical memory[16].

Page Fault in Paging

In this memory management all pages are not necessarily present in main memory at all the times. Unlike, non-paging pages could be swapped back and forth into the physical memory. So, one page could generate any number of page faults during its execution.

Page fault handling mechanisms are dealt with greater details in Chapter 3 of this dissertation.

1.5 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.

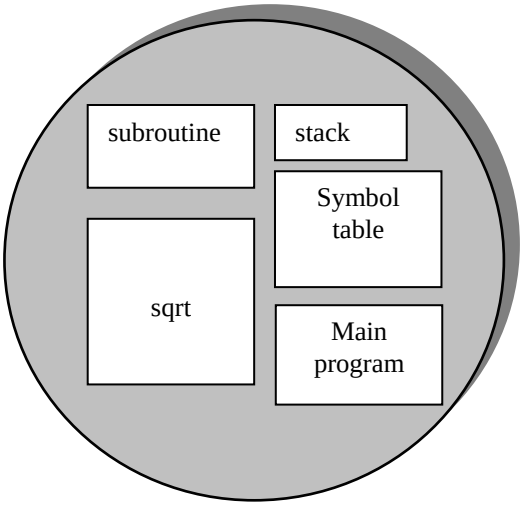


Figure 1.10: Different segments of a process

Segmentation is a memory management scheme, designed to handle the situations described above. In this memory management scheme whole logical address space is divided into different *segments*. Each segment 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 length. 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 1.10. Address translation mechanism of segmentation based memory management method is shown in Figure 1.11.

Segment table: Segment table stores information about all the segments of a process. A typical segment table is shown in Figure 1.11. First field *limit* represents the length of the segment and second field *base* stores the base address of segment.

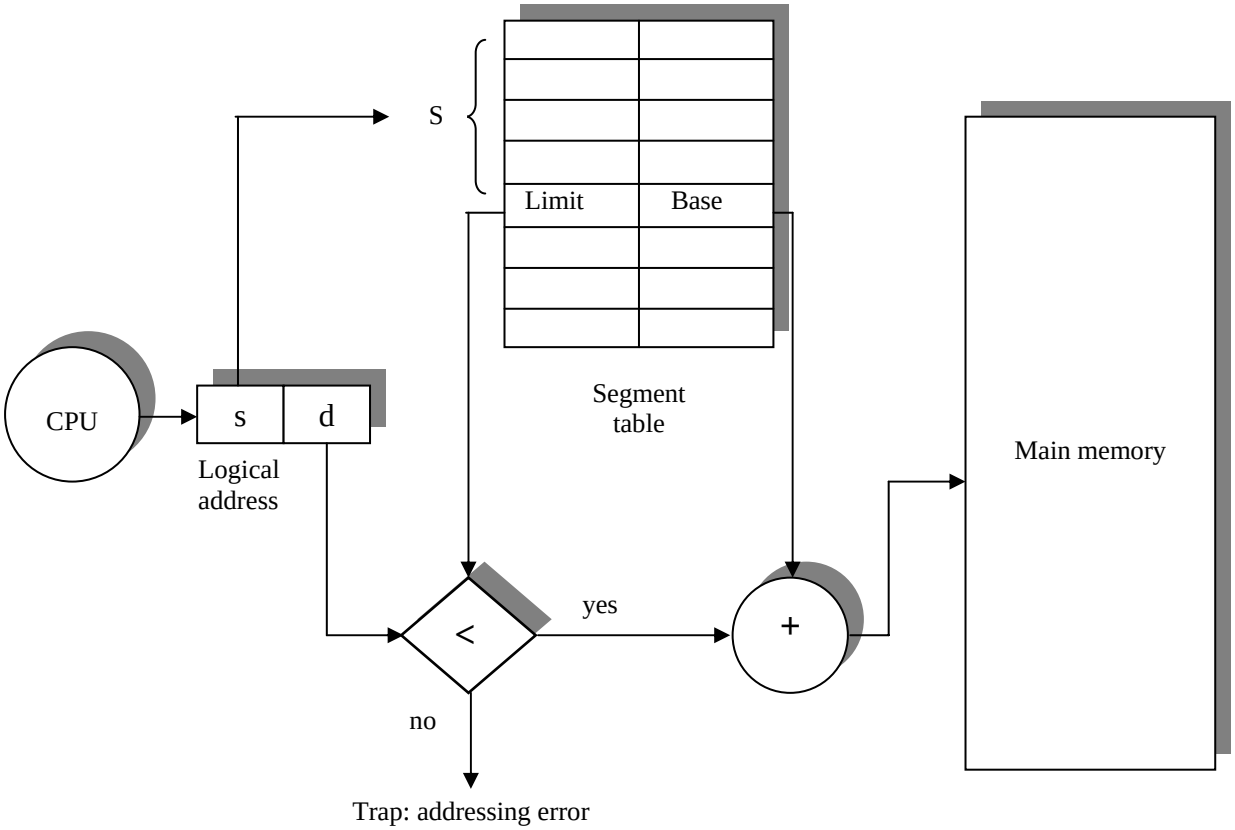


Figure 1.11: Segmentation hardware

1.6 Page frame management

In a paged environment when at the same time multiple processes are running in parallel, any particular page frame could be allocated to one of these processes or else wouldn't be allocated at all. Such pages are free and could be allocated whenever a future request for page frame allocation is received. With larger memories the number of these page frames could be really massive and it becomes tougher to track the status of each individual page frame as and when new requests are received from processes.

A suitable mechanism is needed to handle these requirements of dynamic page frame allocation. To manage all the information about the page frames OS typically creates a table of descriptors, one descriptor for each page frame. If memory size is 128 MB , using 4K page size, $2^{27}/2^{12} = 2^{15}$ = 32K descriptors are needed.

If each descriptor is 4 bytes long, the 128KB are needed for descriptors. These descriptors stores all the necessary information about the Page frames. These information includes physical address of the page frames. Whether page frame is free or it is in use. If it is in use, how many processes are using it. It should keep track of total number of page frames present in the memory[13].

Whenever a process demands a page, memory manager searches the descriptor table to find a free frame in the memory and allocates it. It also marks it as a used page frame. Similarly when a page becomes free, it is added to the list of free frames by the memory manager.

1.7 Problem description

This dissertation suggests a design to implement a page frame allocation scheme for non-paging in an existing operating system named MINIX. Standard model of MINIX does not support paging. It uses variable partitioning for memory management where free memory is represented by linked list. We are implementing non-paging memory management scheme for MINIX with main emphasis given on page frame allocation.

The dissertation also suggests a design for implementing page fault handler for this new memory management scheme. For non-paging, page fault handler is simple and does not need to use any page replacement algorithm.

Implementing this scheme for every available platform and operating system or writing a new operating system from scratch is beyond the constraints of this project. Thus the best alternative is to select a platform with well documented standards with an operating system which don’t provide any native support for page frame management as a starting point.

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

There are numerous open source operating system available, which could be modified freely without much restrictions. Example of such operating system are various flavours of Linux like Red Hat, Debian, Caldera, SUSE etc., MINIX and many others.

MINIX is an open source operating system by Andrew S. Tanenbaum, designed to teach students about the fundamentals of operating systems. MINIX is a stripped down version of UNIX without any paging architecture of its own. It is widely available and worldover popular between students and academecians.

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

1.8 Dissertation organization

The organization of this dissertation is as follows.

Chapter 1: Explains the basic memory management schemes. It also explains page frame allocation mechanism in detail.

Chapter 2: Provides the memory management mechanism used by MINIX operating system.

Chapter 3: Explains the paging scheme on Intel 80x86 architecture. This chapter also explains the page fault handling mechanism supported by Intel architecture.

Chapter 4: Provides complete system design for page frame management in MINIX operating system. It also includes the description of various data structures involved.

Chapter 5: Explains all the major routines involved in the page frame management in MINIX operating system.

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

References

Source code

CHAPTER 2 MEMORY MANAGEMENT IN MINIX

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[1].

2.1 Memory allocation to processes

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[1].

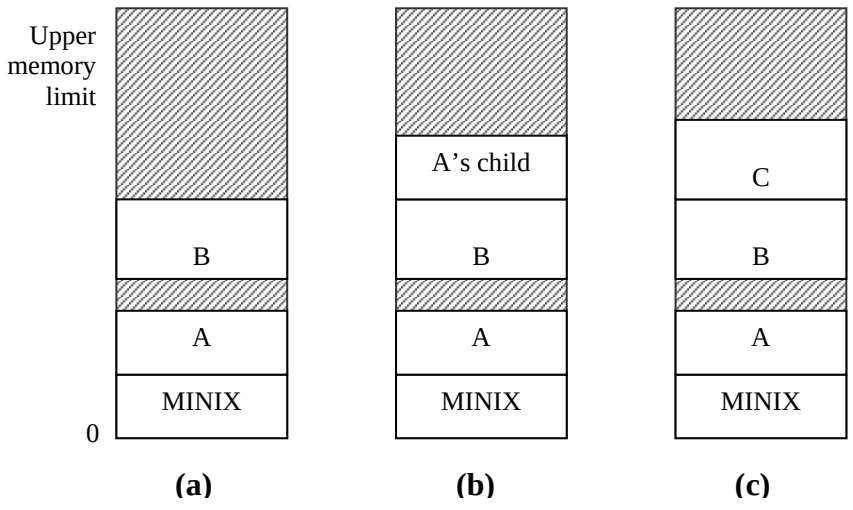


Figure 2.2: Memory allocation. (a) Originally. (b) After a SY FORK. (c) After the child does an EXEC . The shaded regions are unused memory. The process is a common I & D one.

Figure 2-1 shows both ways of allocating memory. In Figure 2.1 (a) we see two processes, A and B, in memory. If A forks, we get the situation of Figure 2.1 (b). The child is an exact copy of A. If the child now executes the file C, the memory looks like Figure 2.1 (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[1].

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.

2.2 Internal memory layout of a process

Simple MINIX processes use combined I and D space (I stands for instruction and D stands for data), 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[1].

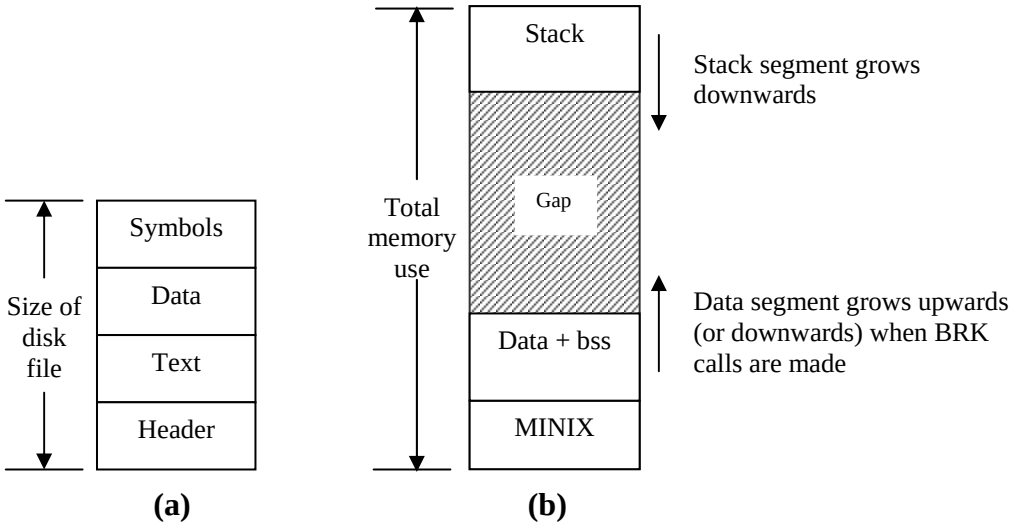


Figure 2.2: (a) A program as stored in a disk file. (b) Internal memory layout for a single process. In both parts of the figure the lowest disk or memory address is at the bottom and the highest address is at the top.

Figure 2.2 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. This is for use in debugging and is not copied into memory[1].

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.

To represent the internal memory layout of a process we use a special data structure shown in the Figure 2.3. It is a matrix of 3*3. Each row has three entries:

- Virtual address of the segment in terms of clicks.
- Physical address of the segment in terms of clicks.
- Length of segment in terms of clicks.

The first row stores information about the stack segment of the program. Second about text segment and third about data segment. Data structures used for this purpose will be discussed in section 2.3**.**

	Virtual	Physical	Length
Stack	0x20	0x340	0x8
Text	0	0x3c0	0x1c
Data	0	0x320	0

Figure 2.3: Representation of internal memory layout of a process in memory.

Now we will discuss the internal memory representation technique for both non-separate and separate I and D space.

2.2.1 Non-separate I & D representation

The method used for recording memory allocation for non-separate I & D representation is shown in Figure 2.4. 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 2.4 (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)[1].

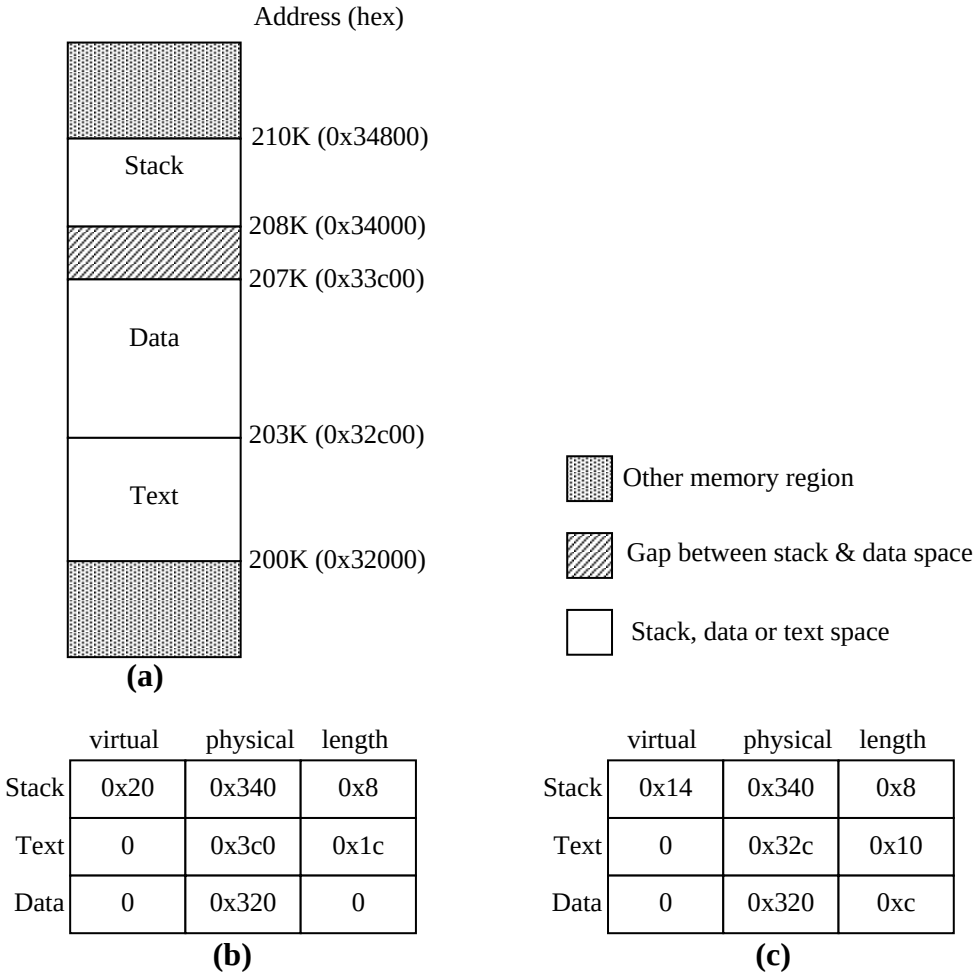


Figure 2.4: (a) A process in memory. (b) Its memory representation for Non-separate I and D space. (c) Its memory representation for separate I and D space.

2.2.2 Separate I & D representation

Figure 2.4 (c) shows the segment entries for the memory layout of Figure 2.4 (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 2.2 (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.

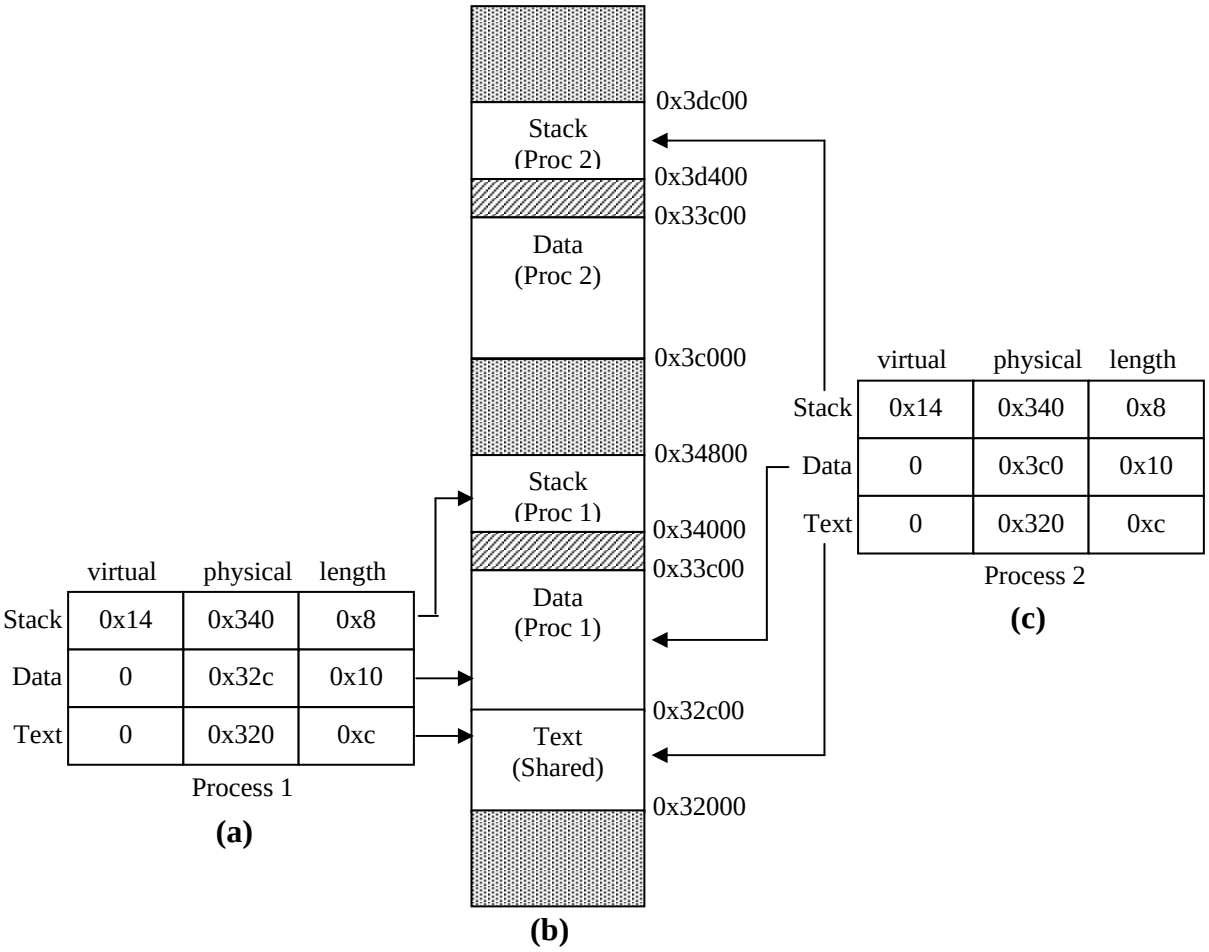


Figure 2.5: (a) A process in memory. (b) Its memory representation for Non-separate I and D space. (c) Its memory representation for separate I and D space.

2.2.3 Shared text in separate I & D representation

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.

If a process uses separate I and D space, wants to execute, a search is made for a process that is already using the text belongs to this process. If such a process is found, there is no need to load this text to the memory. What all we need now is to load only data and stack segment in memory, and copy information about the text (i.e. physical address, virtual address and length) in the memory map of the process. This is shown in Figure 2.5.

2.3 Memory manager's data structures

Till now we have discussed how memory is allocated to process and its internal memory layout. In this section we will discuss the data structures used for this purpose. The memory manager has two key data structures: the *process table* and the *hole table*. The process table also has a data structure for internal memory representation of a process. First we will discuss this data structure.

2.3.1 Data structures for internal memory representation of a process

As we explained earlier in section 2.2 that matrix of Figure 2.3 is used for internal representation of a process. Structure *mem_map* is used to represent one row of this matrix. Its declaration 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 */
};
```

It consists of three fields for each, virtual address, physical address and length. Now an array *mp_seg[ ]* of type *mem_mep* is declared and this array actually represents the matrix of Figure 2.3. The *mp_seg[ ]* array is actually defined in the *mproc* structure.

```
struct mem_map mp_seg[NR_SEGS];         /* points to text, data, stack */
```

Here *NR_SEGS* is a constant and its value is 3. It represents the number of segments per process. First row of *mp_seg[ ]* array stores information about stack segment. Similarly second and third rows store information about data and text segment respectively. All the

information stored is in terms of clicks. 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.

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 2^40 bytes, which is 1024 gigabytes.

Data and stack segments have given a common space in MINIX, growing towards each other. The 8088 hardware does not have a stack limit trap, and MINIX defines the stack in a way that will not trigger the trap on 32-bit processors until the stack has already overwritten the data segment. Thus, this change will not be made until the next BRK system call, at which point the operating system explicitly reads SP and recomputes the segment entries. On a machine with a stack trap, the stack segment's entry could be updated as soon as the stack outgrew its segment. This is not done by MINIX on 32-bit Intel processors, for reasons we will now discuss.

2.3.2 The process table

The memory manager's process table is called *mproc*. For each process in the system there is an entry in the *mproc[]* array. 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* and is already explained in section 2.3.1.

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. Definition of *mproc[ ]* 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];
```

mp_ ino, mp_dev, mp_ctime fields are used for shared text in case of separate I & D representation. Now we will see how these fields are used for shared text.

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_ino*, and *mp_ctime* fields in each slot of *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[T]* 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.

2.3.3 Hole list

MINIX uses linked list representation for maintaining free memory holes. The major data structure used for this purpose is *hole* table, which 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. The definition of the *hole* structure is given 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];
```

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.

An array of length 128 of type structure *hole* is declared. Each element of this array either points to a hole or it is empty. So this array has two parts: first represents the hole list and the second represents empty slots (also stored as a linked list).

So we need two data structures first *hole_head* which will point to hole list and second *free_slots* which will point to linked list of table entries that are not in use. Definition of these data structures is given below.

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

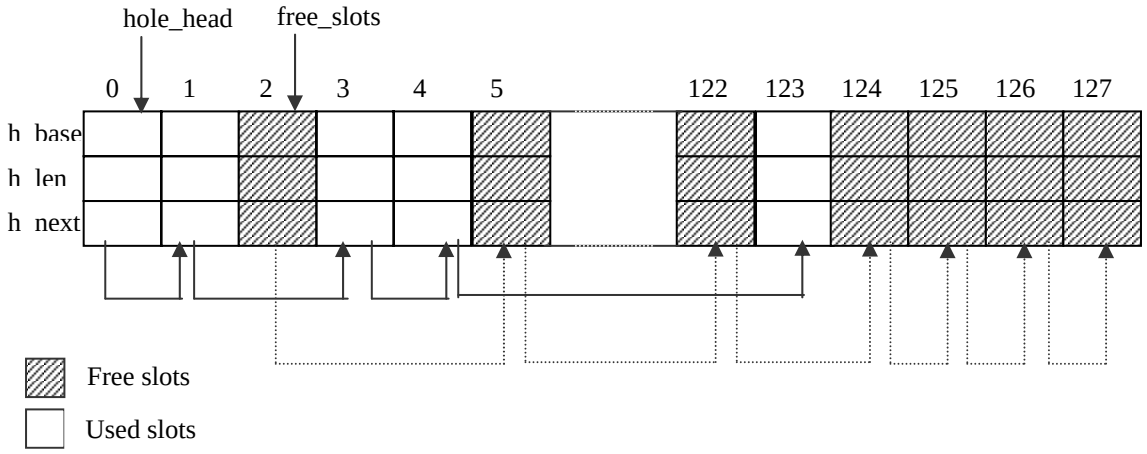


Figure 2.6 : Hole list and free slots list

Figure 2.6 shows the hole structure**.** As shown each slot contains three fields. Dotted arrows shows the linked list of free slots in the table while thick arrows shows the hole list.

Figure 2.7 shows a fragment of memory and its hole list representation. The hole list contains the information about the hole list in the increasing order of their base address.

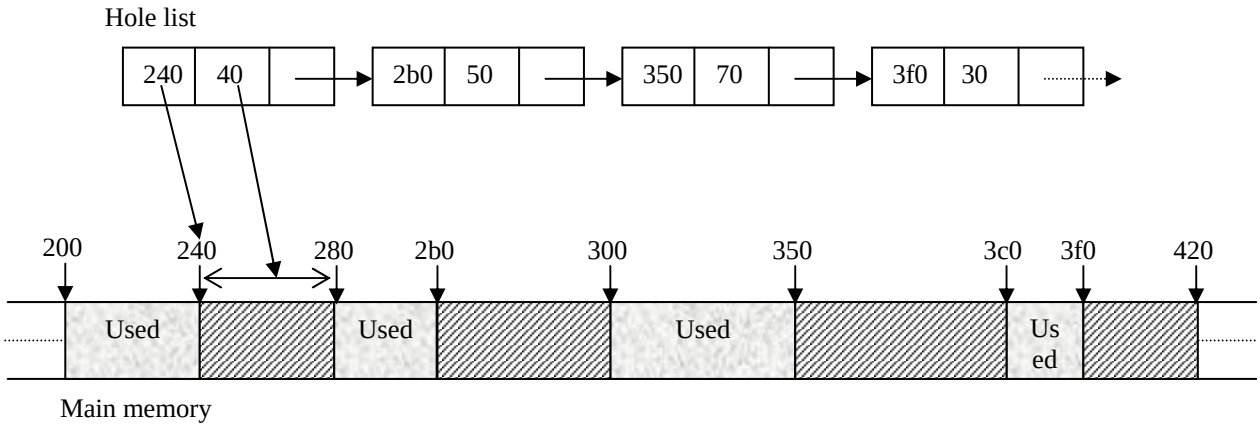


Figure 2.7: A hole list for a particular instance of memory

2.4 Memory management algorithm

This section describes in brief the memory management algorithms used by MINIX operating system.

2.4.1 Memory initialization

In MINIX memory is divided into chunks. Chunks are non-consecutive memory areas having unusable gap between them. Information about these memory chunks is stored in data structure named chunk table.

As explained in section 2.3.3 that 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*.

2.4.2 Memory allocation algorithm

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.

2.4.3 Algorithm for freeing memory

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 MB.

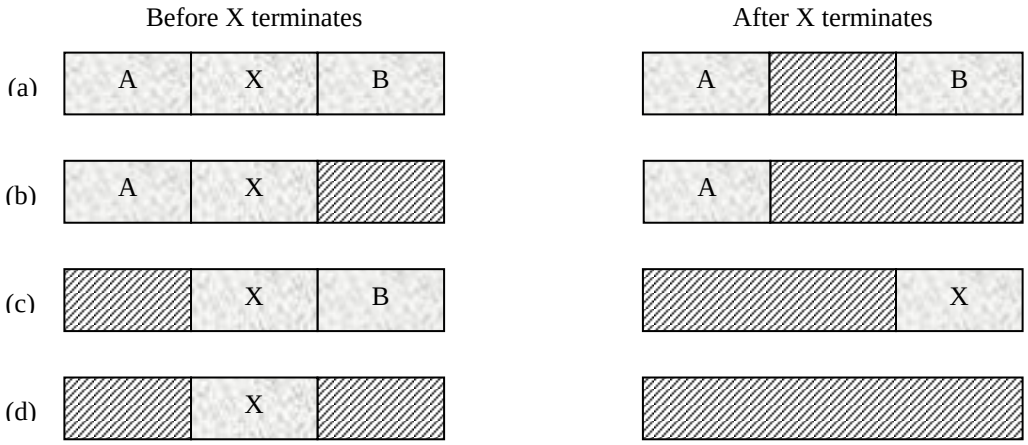


Figure 2.8: Four neighbor combinations for a terminating process X.

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 2.8. In Figure 2.8 (a) updating the list requires replacing a process by a hole. In Figure 2.8 (b) and Figure 2.8 (c), two entries are coalesced into one, and the list becomes one entry shorter. In Figure 2.8 (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 2.8 (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.

CHAPTER 3 PAGING IN INTEL 80x86

3.1 Virtual memory management in 32-bit Intel architecture

In this section we will see the basic paging mechanism with virtual memory management for Intel 80x86 architecture.

Programs running under modern operating system use linear 32-bit addressing to locate instructions and data. The use of 32-bit addresses means that applications (and operating system software) have a total address space of 4GB (2^{32}). If a computer has 4 GB of real (physical) memory, then locating programs and data in memory and addressing programs and data is unproblematic. However, such computers are rare and limited to high-end server platforms[2].

It is clear then that the address space occupied by programs and data will always exceed the limits of real, physical memory. Hence, memory management will always be a feature of modern operating systems. The basic challenge in memory management relates to the ability of operating systems to multitask processes. To do this, processes must be in memory. However, operating system designers have long abandoned the practice of loading entire programs and data into memory, so that they can be executed. This is due to the increase in the number and size of operating system and application programs, and to the enormous inefficiencies of transferring such programs to and from secondary storage. So, how do modern operating systems meet this challenge?

The first thing that an operating systems memory manager does is to create an address space that matches the address range of the operating system. In 32-bit system this is 4 GB. To do this, a file, called a paging file for reasons later outlined, is created on secondary storage. This file emulates what's known as virtual memory. The size of this file can vary, depending on operating system settings and hard disk space. It can never be more than 4 GB, however. When applications are run they are allocated a range of addresses within this space. Application run in Level 3 protected mode and are therefore normally limited to their allocated address range. This protects the address space of other applications and the operating system from mischievous or buggy applications. The

memory manager allocates program memory space from information in the programs header at load time. The challenge facing memory managers is to map this address space on disk to locations in real memory. Given that only the active regions of programs (i.e. the routines/modules containing instructions currently under execution) and associated data sets need to be resident in memory at any one time, the memory manager must be able to map specific address ranges in virtual memory to the real memory addresses where the instructions and data are actually stored. (remember, address ranges across a 4 GB virtual memory range are mapped unto a real memory address range that can be 256, 512 or 1024 MB.) Intel processors provide support for this by allowing operating systems to implement segmentation and/or paging strategies[2].

Dividing memory into pages facilitates the division of programs and data sets into discrete addressable regions or units that can be transferred from virtual memory to real memory and vice versa. The address of the first byte in a page is used to locate individual pages in both virtual and real memory. In 32-bit systems, the 4 GB virtual memory space is divided into of 1 million 4 KB pages (1,000,000 x 4,000 = 4GB). Each individual page is addressable using the upper (most significant) 20 bits of a 32-bit virtual address (bits 12-31: note that 2^{20} = 1 million possible combinations of 20 1's and 0's). The remaining 12 bits (0-11) of the 32-bit address are used to locate individual byte addresses in a page (i.e. 2^{12} = 4,048 or 4KB of addresses)[13]. Figure 3.1 illustrates this.

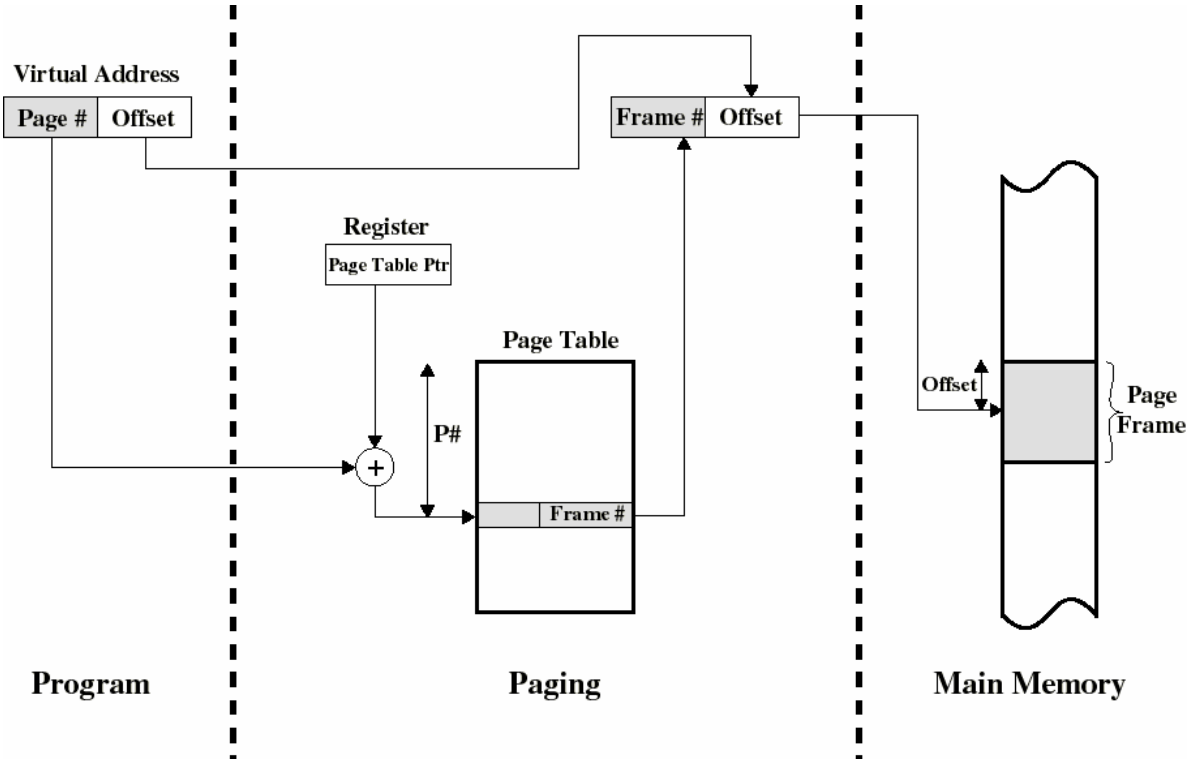


Figure 3.1: Basic paging mechanism

3.1.1 Page table

To keep track of pages in virtual memory, the memory manager would simply keep a single table in RAM each record of which would map the first address of each **page** in virtual memory to the address of the **page frame** in real memory that holds the program instructions/data. This table is called *page table*. However, the size of the page table would be enormous, given that each record would need to be 64 bits (8 bytes), giving a total size of 8MB. Why 64 bits? A 32-bit virtual memory address would need to be mapped onto a 32-bit real memory address. However, only 20 bits in each virtual and real address would be needed to map a virtual page onto a real memory page frame. Why? Simply because the first address in each 4096 byte page is used, and this byte address can be uniquely identified using the most significant 20 bits (the least significant 12 bits will always be all 0s in the first byte address of every 4 KB page). What of the other 4095 bytes in the page? The 12 least significant bits are used to locate these. For example, suppose the virtual memory manger wishes to map the 32-bit virtual address in the CPU's program counter to the address in real memory that contains the instruction. In order to find the address of the real memory page frame, the virtual memory manager would simply look up the memory

management table, use the most significant 20 bits of the virtual address to locate the record for the referenced page, and fetch to the CPU the 20 bit real memory page address which is stored in second part of the 64 bit record. The least significant 12 bits (the byte offset for the page) of the virtual address is then added or appended to the 20 bit real memory page address to arrive at the byte address in RAM of the referenced instruction[4].

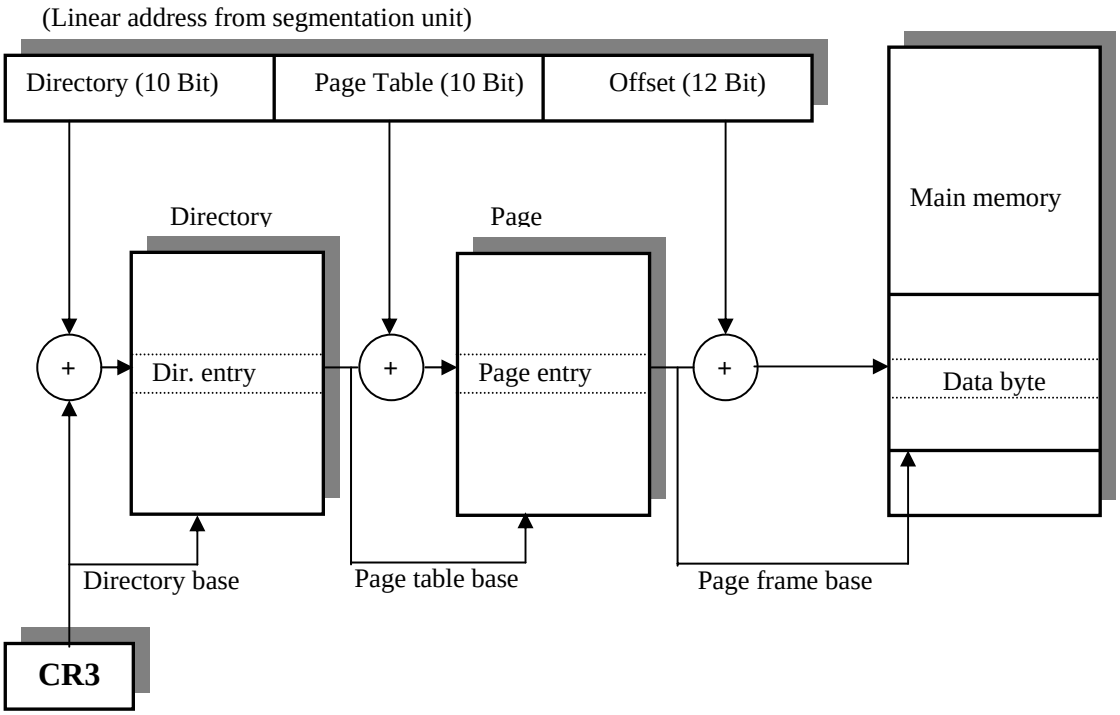


Figure 3.2: Paging Unit

3.1.2 Two level paging

The Intel Pentium IV supports (beginning with the Intel 386) a two level paging architecture that avoids the creation of an 8-MB paging table. In this scheme of things, the 32-bit linear address is divided into three: first, the 12 least significant linear address bits (0-11) of the virtual memory address are used as an offset value to generate the 32-bit physical address used to locate a byte in a particular page (refer to the Figure 3.2). The most significant 20 bits are used to locate the data entries in two memory management tables—the *Page Directory Table* and the *Page Table*(s). The outcome of this exercise is to locate the 20-bit page frame address of the page in RAM, which stores the virtual

memory page. Hence, bits 12-31 of the virtual memory address (which will be the same for all 4,096 byte addresses in a particular page) are divided into a 10-bit *page directory index* (bits 22-31) and a 10-bit *page table index* (bits 12-21). A page directory is a simple data structure created by the memory manager for each process in virtual memory—hence, there is just one Page Directory for each program. This data structure is maintained in RAM and is just 4 KB in size. The size of this data structure or table is not a coincidence, as will be seen[18].

In order to manage RAM and map pages in virtual memory to RAM so that the CPU can access them, the memory manager divides RAM into 4KB Page Frames. Each page frame will have an address in the range of 00000000 hex to FFFFF000 h. FFFFF000 is the address of the last page in virtual memory (the 32-bit linear address space): individual byte addresses in this page will range from FFFFF000 to FFFFFFFF. Examples of page addresses are 00AB2000 h, 00FFF000 h, 0001B000 h, and so on. The important thing to note in examining these addresses is that the 12 least significant bits are all 0s (i.e. 00FFF000 h equals 0000 0000 1111 1111 0000 0000 0000). Accordingly, as the first address in each page is give by the 20 most significant bits.

When a process is created and run, the operating system examines the header information in the binary image to obtain important data on the program. One critical piece of information is the size of the program in bytes; this influences the amount of virtual memory allocated to the process. Also important are the size of the data objects, files or related routines that the process will use to fulfil its role. The process's address space will be expanded to include these 'objects.' However, the in certain instances the virtual memory manager may simply 'reserve' virtual memory addresses and 'commit' them when the objects (files, data structures, or program code) are actually loaded. When the size of a process's address space is calculated the memory manager can then create the Page Directory and associated Page Tables. While each process will have just one Page Directory, it may have up to 1024 Page Tables.

3.1.3 Page directory table

The Page Directory is first created in RAM and the 32-bit address of the first byte in the page (in the form XXXXX000) is deposited in the CPU's CR3 (Page Directory) register. Format of the control registers is shown in the Figure 3.3. This will be used to locate the Page Directory in RAM during 32-bit virtual to real memory address translation. A Page Directory will have1024 entries of 4 bytes each (32-bits, 4 bytes x 1024 = 4096 bytes, 4KB). Remember, the maximum number of combinations of 10 bits is 1024 (2^{10} = 1024). The 10 most significant bits (22-31) of the virtual or linear address can take values from 0000000000 to 1111111111 (000-3FF h). These are added to the 32-bit value in the CR3 register (e.g. 000F2000 (CR3) + AF0 (bits 22-31) = 000F2AF0) to give the byte address of the Page Directory Entry (PDE). Each of PDE will hold the 20 bit address of a Page Table and 12 associated status and control bits. The number of Page Directory Entries (PDE) and the number of corresponding Page Tables depends on the size of the process and associated objects[35].

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

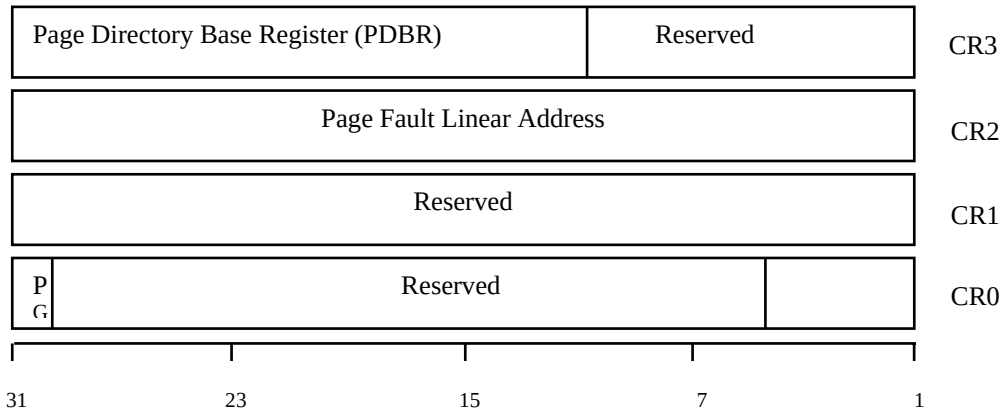


Figure 3.3: Control Registers

3.1.4 Page table in two level paging

In 80x86, a page table is simply an array of 32-bit page specifiers. A page table is itself a page, and therefore contains 4 Kilobytes of memory or at most 1K 32-bit entries.

As indicated, the 20-bit value in a PDE is used to locate the Page Frame Number of a Page Table in RAM. That is, the first byte address of the Page Table. Each Page Table is 4KB in size and contains 1024, 32-bit entries (i.e., records). 512 of these entries reference Page Frames in RAM that map onto the process's own address space, while the other 512 are used to map the system address space (i.e. the upper 2GB of virtual memory used by the operating system and its objects) to physical pages. As with PDEs, PTE entries are 32-bits in length. 20 bits of each PTE contains the Page Frame Number/address in RAM of the 4KB page which a process or object is physically located. The remaining 12 bits are used for indicating the status of pages. Hence, using a two-level data structure, the virtual memory manager is now able to locate individual 4 KB pages in RAM. To address an individual byte within the 4096 bytes in a page, the memory manager transfers the 20 bit Page Frame address to the CPU and appends the 12 bit offset (bits 0-11) of the virtual address in the program counter to that value to give the 32-bit real memory address of the referenced byte of memory.

3.1.5 Page table entries

Entries in either level of page tables have the same format. Figure 3.4 illustrates this format. The most important field is the *page frame number*. After all, the goal of the page mapping is to locate this value. Next to it we have the *present/absent* bit. If this bit is 1, the entry is valid and can be used. If it is 0, the virtual page to which the entry belongs is not currently in memory. Accessing a page table entry with this bit set to 0 causes a page fault.

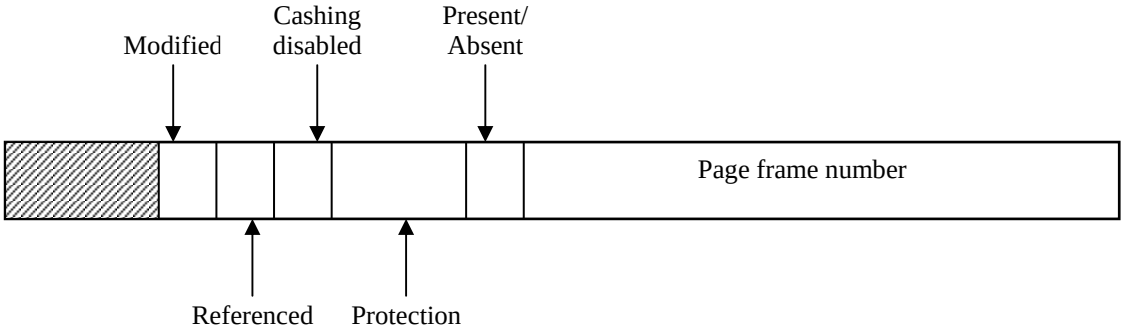


Figure 3.4 : A typical page table entry

The *protection* bits tell what kinds of access are permitted. In the simplest form, this field contains 1 bit, with 0 for read/write and 1 for read only. A more sophisticated arrangement is having 3 bits, one bit each for enabling reading, writing, and executing the page[1].

The *modified* and *referenced* bits keep track of page usage. When a page is written to, the hardware automatically sets the Modi3ed bit. This bit is of value when the operating system decides to reclaim a page frame. If the page in it has been modified (i.e., is "dirty"), it must be written back to the disk. If it has not been modified (i.e., is "clean"), it can just be abandoned, since the disk copy is still valid. The bit is sometimes called the *dirty bit*, since it reflects the page's state.

The *referenced* bit is set whenever a page is referenced, either for reading or writing. Its value is to help the operating system choose a page to evict when a page fault occurs. Pages that are not being used are better candidates than pages that are, and this bit plays an important role in several of the page replacement algorithms that we will study later in this chapter.

Finally, the last bit allows caching to be disabled for the page. This feature is important for pages that map onto device registers rather than memory. If the operating system is sitting in a tight loop waiting for some I/O device to respond to a command it was just given, it is essential that the hardware keep fetching the word from the device, and not use an old

cached copy. With this bit, caching can be turned off. Machines that have a separate I/O space and do not use memory mapped I/O do not need this bit.

A brief word about the Page Directory and Page Table and process size. Processes of 2 MB or less have one Page Directory and one Page Table. This means is that the Page Directory has only one entry. This 20 bit Page Frame Number in the entry references a single page in RAM that stores the process's singe Page Table. Since the Page Table has 512 entries to store the 20 bit Page Frame addresses of a process's 512 4KB pages in RAM, this means that the process's address space is less than or equal to 2.062 Mb. Remember that processes may access the system address space, hence the second 512 entries are allocated for that purpose. Note also that if many processes are running and there is tight competition for access to the CPU, only this process's 'working set', that is the set of instructions compromising the program module presently under execution, needs to be stored in RAM. Hence, the number of Page Table Entries will often be far less than the total number possible[30].

3.2 Page fault

When a process tries to access a page that is not present into physical memory, a page fault occurs. The general mechanism to handle page fault is described in the following section.

3.2.1 Page fault handling

The paging hardware, in translating the address through the page table, will notice that an invalid bit is set, causing a trap to the operating system. This trap is the result of the operating systems failure to bring the desired page into memory (in an attempt to minimise disk transfer overhead and memory requirements), rather then an invalid address error as a result of an attempt to use an illegal memory address (such as an incorrect array subscript)[9].

Figure 3.6 shows the mechanism needed to handle a page fault.

1. A check is performed in an internal table usually kept with the process control block to determine whether the reference was a valid or invalid memory access.
2. If the reference was invalid, process will be terminated. If it was valid further steps should be performed.
3. A Free page frame is found in memory.
4. The desired page is brought to this free page frame.
5. The internal table and the page table are modified to indicate that the page is now in memory.
6. The instruction that was interrupted by the illegal address trap should be restarted.

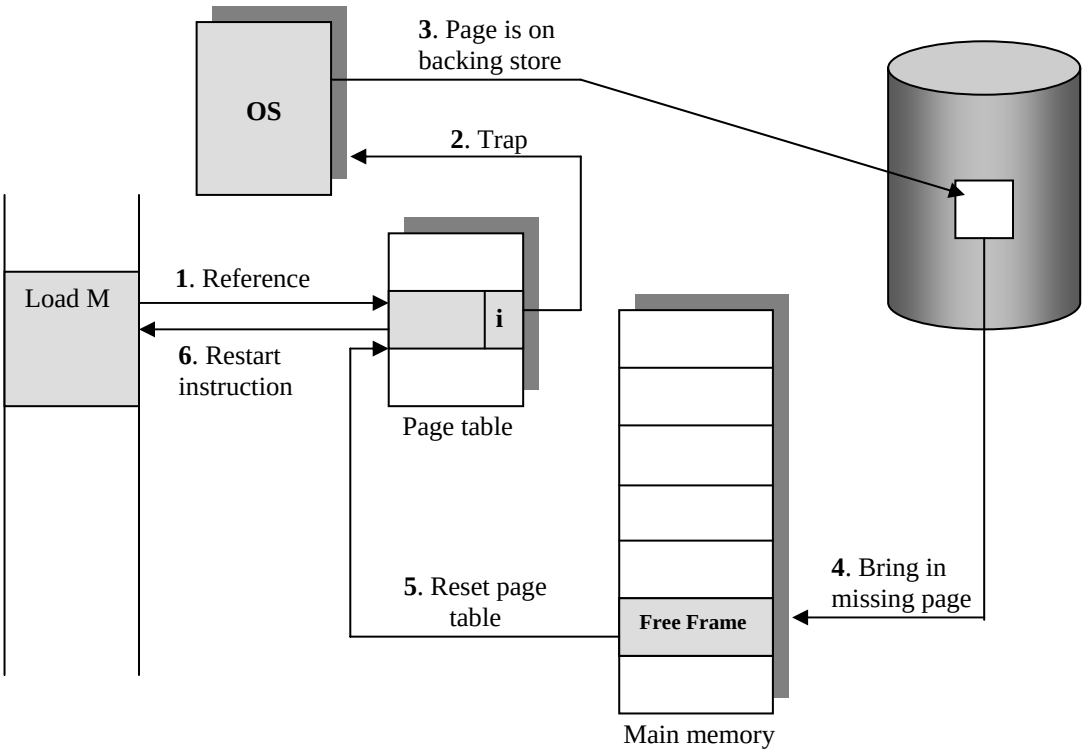


Figure 3.6: Steps in handling page fault

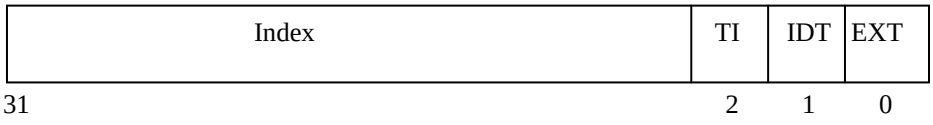
3.2.2 Hardware support

This exception occurs when paging is enabled (PG=1) and the processor detects one of the following conditions while translating a linear address to a physical address[10]:

- The page-directory or page-table entry needed for the address translation has zero in its present bit.
- The current procedure does not have sufficient privilege to access the indicated page.

The processor makes available to the page fault handler two items of information that aid in diagnosing the exception and recovering from it:

The first one is an error code on the stack. The error code for a page fault has a format different from that for other exceptions (see Figure 3.7). The error code tells the exception handler three things:



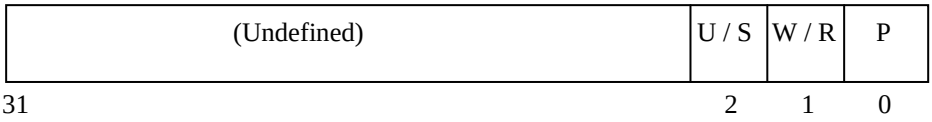
	TI	IDT	EXT
0	Use LDT	Use LDT and ignore bit 2.	An event external to the program caused the exception (i.e., external interrupt, single step, processor extension error)
1	Use GDT	Bit 2 indicates table usage.	An exception occurred while at CS:IP saved on the stack. processing the instruction

Figure 3.7: Page-Fault Error Code Format

1. Whether the exception was due to a not present page or to an access rights violation.

2. Whether the processor was executing at user or supervisor level at the time of the exception.
3. Whether the memory access that caused the exception was a read or write.

Second one is CR2 (control register two). The processor stores in CR2 the linear address used in the access that caused the exception (see Figure 3.8). The exception handler can use this address to locate the corresponding page directory and page table entries. If another page fault can occur during execution of the page fault handler, the handler should push CR2 onto the stack[10].



	U/S	W/R	P
0	The access causing the fault originated when the processor was executing in supervisor mode.	The access causing the fault was a read.	The fault was caused by a not-present page.
1	The access causing the fault originated when the processor was executing in user mode.	The access causing the fault was a write.	The fault was caused by a page-level protection violation.

Figure 3.8: CR2 format

CHAPTER 4 SYSTEM DESIGN

This chapter presents the design for page frame management and allocation in an operating system. To understand this design we need some basic acquaintance with the major data structures involved in the design. So, the first section describes the major data structures and in the next section design is explained.

4.1 Data structures

Data structures explained here are chunk table and page frame table. Page frame table keeps the information about the page frames while the chunk table represents all the memory chunks available in physical memory.

4.1.1 Chunk table

In a computer system complete main memory is not available for the operating system and user processes. Some of the memory is reserved and it can not be used by them. For example a specific memory area is reserved of mapping of BIOS routines as shown in the Figure 4.1.

Due to these reserved areas, usable memory is divided into different parts. These parts are called memory *chunks*. Number of memory chunks in main memory of a computer system is architecture dependent. In Intel architecture, usable memory is divided into three parts so there are three memory chunks (Figure 4.1)[1].

To store information about these memory chunks we need a data structure. A data structure named *memory* is used for this purpose. Its definition is given below.

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

And the data type *phys_clicks* is defined below.

```
typedef unsigned int phys_clicks;
```

First element *base* of the structure *memory* represents the base address of the chunk in terms of click. Second element *size* represents the size of the chunk in terms of click.

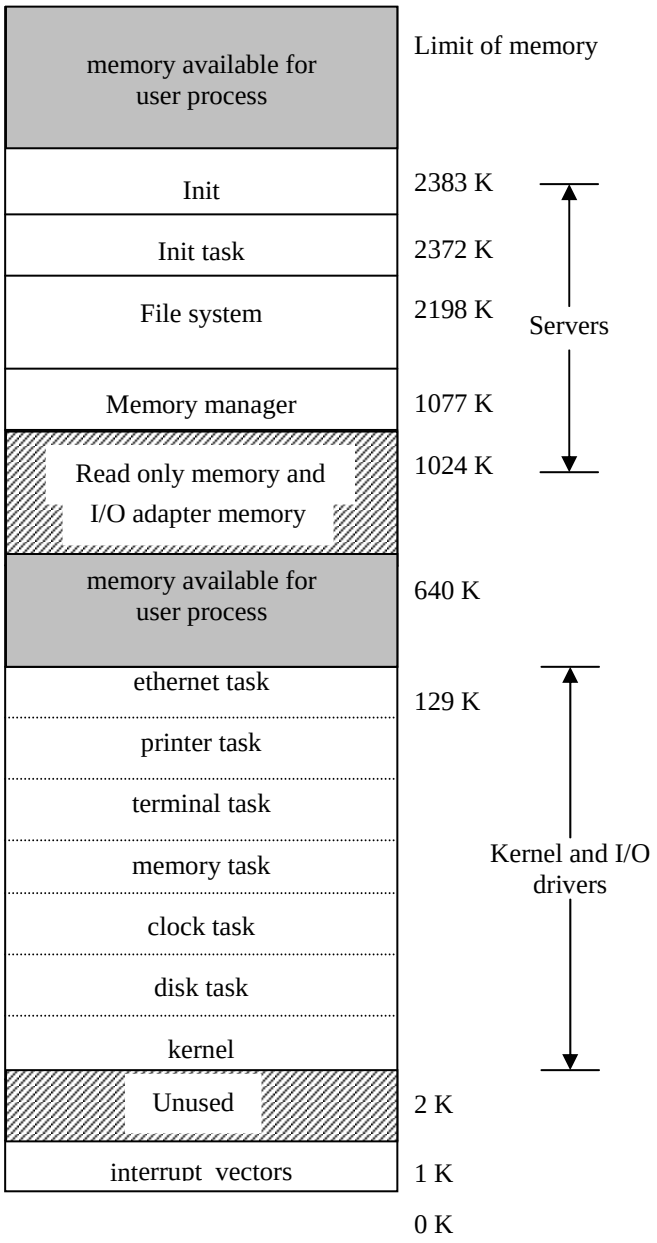


Figure 4.1: Memory layout after MINIX has been loaded from the disk into memory, and division of memory in to chunks.

An array named *chunk_table* is used to store all the memory chunks. Its definition is given below.

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

Here *NR_CHUNKS* is a constant and its value is 16. So this array can represent

maximum16 chunks of memory. Memory chunks stored in this array are random i.e. they are not stored in any specific order. A *chunk_table* entry with *size* equal to zero represents an empty chunk.

Figure 4.2 (a) shows the main memory divided into different chunks and Figure 4.2 (b) shows its representation by chunk table.

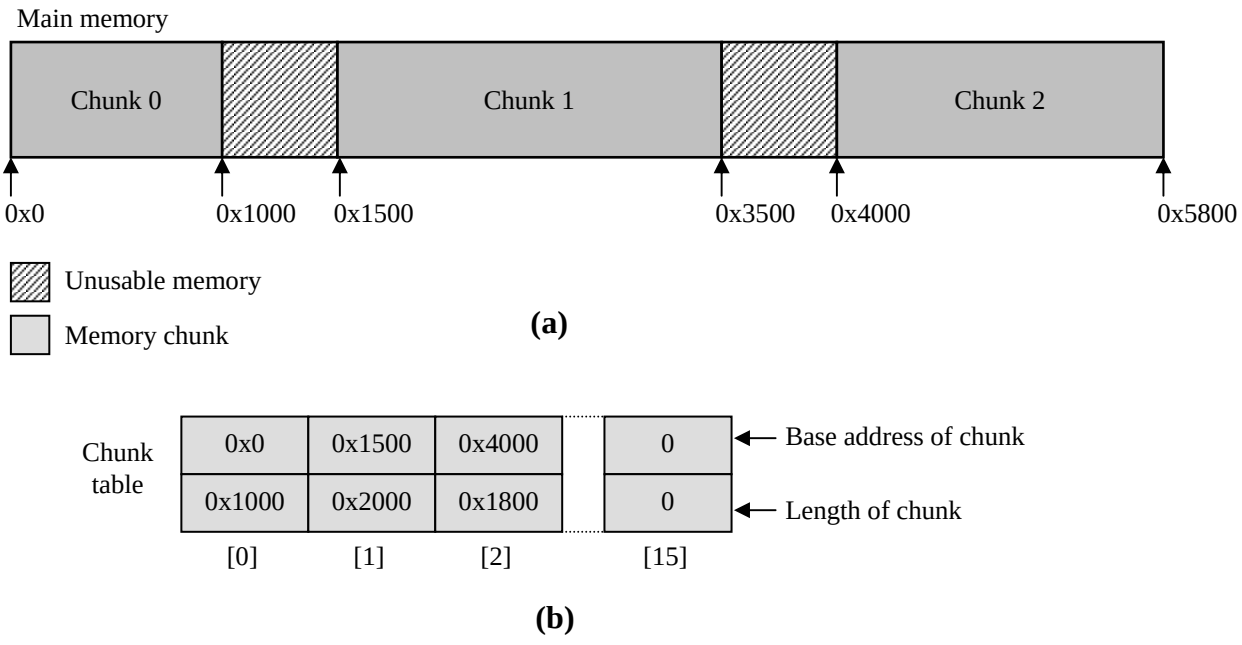


Figure 4.2: (a) Main memory it no chunks. (b) Its representation by chunk table

4.1.2 Page frame table

Page frame management for an operating system requires maintaining and managing a structure which could keep the status of all the available page frames updated through out the operation of the whole operating system. Kernel could use this structure to keep track

of free page frames and is also designated with the responsibility of updating this structure as and when a busy page frame is freed or a free page frame is allocated to a process. In this proposed design this structure is known as *rlmem_table*[35].

Each individual index in this table refers to an individual page frame in physical memory. For example, the index 3 of this table manages the page frame number 3 of the physical memory (see Figure 4.3). Each entry in this table is of single byte representing the reference count for the corresponding page frame. A reference count of 0 indicates that page frame is free. A reference count of 1 indicates that page frame is used by one process, a reference count of 2 indicates that page frame is used by two process and so on.

Size of this table depends on the number of page frames in physical memory. For example if the RAM size is 64 Mb, then the total number of page frames in the memory are 2^{14} (page frame size is 4 Kb) and hence the size of the *rlmem_table* is 16 Kb. 4 page frames are required to store this table in the memory. A variable *rlmem_table_base* stores base address of this table in terms of bytes. Definition of this variable is given below.

```
PRIVATE phys_bytes rlmem_table_base, rlmem_table_size;
```

Second variable stores the size of table in terms of bytes. An example of *rlmem_table* representing the information about page frames is shown in the Figure 4.3.

A variable named *free_mem* stores the count of free page frames available in the memory. Its definition is given below.

```
PRIVATE phys_clicks free_mem;
```

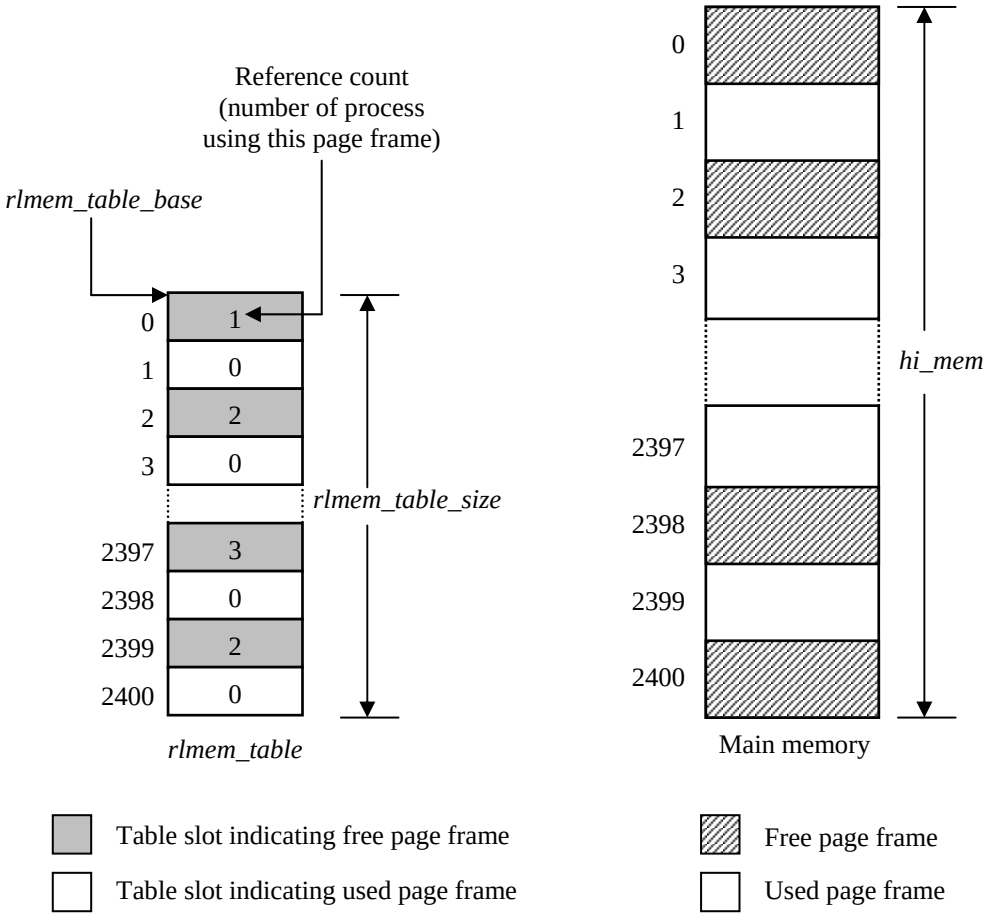


Figure 4.3: Representing information about the page frames

4.2 Design

In this section we will discuss the overall design of the page frame management phase to support virtual addressing in MINIX. The design of the system is divided into three parts. First part deals with the page frame management. In the second part page directory and page tables are created for mapping of all page frames. Finally, third part deals with the page fault handling.

As explained earlier, 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.

4.2.1 Page frame management

Initialization of rlmem_table

During the system initialization phase information about all the chunks is copied to the *chunk_table* by the kernel of the MINIX operating system. First step is to find the total amount of physical memory available. For this purpose, entire *chunk_table* is scanned and size of each chunk is added to a variable *hi_mem.* After this operation, *hi_mem* contains the size of memory in terms of clicks.

As explained earlier that the click size is equal to the page frame size, so *hi_mem* indicates the total number of page frames in the system. Now, next step is to allocate space for *rlmem_table*. The size of this table in terms of bytes is equal to the total number of page frames in the memory i.e. *hi_mem* bytes.

Entire *chunk_table* is again scanned to find a chunk that is sufficient to hold this table. Chunk of appropriate size is removed from the chunk table and this space is allocated to *rlmem_table* and variables *rlmem_table_base* and *rlmem_table_size* are set accordingly.

In the next step, all entries in the *rlmem_table* are set to 1. This indicates that all page frames are in use.

Then each chunk in the chunk table is scanned and for each page frame in this chunk following operation is performed: if this frame is in not in use set the corresponding entry in the *rlmem_table* to zero. When the entire chunk is scanned, it is removed from the chunk table. For each free page frame, variable *free_mem* is incremented.

Finally, the entire chunk table is empty and *rlmem_table* table contains the status of each page frame. Also *free_mem* represents the total number of free page frames available in the memory.

Whenever a request for a free page frame is received *rlmem_table* is scanned for a 0 entry and the corresponding page frame is allocated to the process with an increment in the

entry. Similarly when a process frees any particular page frame, the corresponding entry into *rlmem_table* is decrements.

Allocating and freeing page frames

Whenever a request for a free page frame is received, *free_mem* is read and if it is non zero the *rlmem_table* is scanned for a 0 entry. When such an entry is found the corresponding page frame is allocated to the process and the reference count of this page frame in the *rlmem_table* is incremented. Similarly when a process frees any particular page frame, the corresponding entry into *rlmem_table* is decrements. If the reference count becomes zero, it indicates that the page frame is now free and *free_mem* is incremented.

4.2.2 Creation of page directory and page table

Each entry in a page global directory or a page table takes 4 bytes. The first 20 bits contains the physical address of a page table or a page frame in terms of clicks and remaining bits are available for protection flags.

To create page directory, first step is to find the space for it. A page frame is allocated to it and all of the bytes of this frame is set to zero. A variable named *page_base* is set to point to this page frame. So this variable actually indicates the page directory of the system. Its definition is given below.

```
phys_bytes page_base;
```

Next step is to make entries in the page global directory for linear address range of 0 to *hi_mem.* For this purpose for each 4 Mb portion of memory, a page table is created and its physical address is put in the corresponding entry in the page directory.

Space is allocated for the page table in the similar way used for allocating space for page directory. Now in the next step, page tables are mapped on to page frames. These mappings are on the identical physical address space. This identical mapping helps the kernel to access RAM directly. After setting the tables for two-level paging to map RAM, the paging unit is enabled.

As we know in virtual addressing scheme, the user processes use linear address space instead of physical address space. To fulfill this requirement we have to create a mechanism. This could be achieved by removing all the chunks of physical memory in chunk table and adding a linear address space chunk.

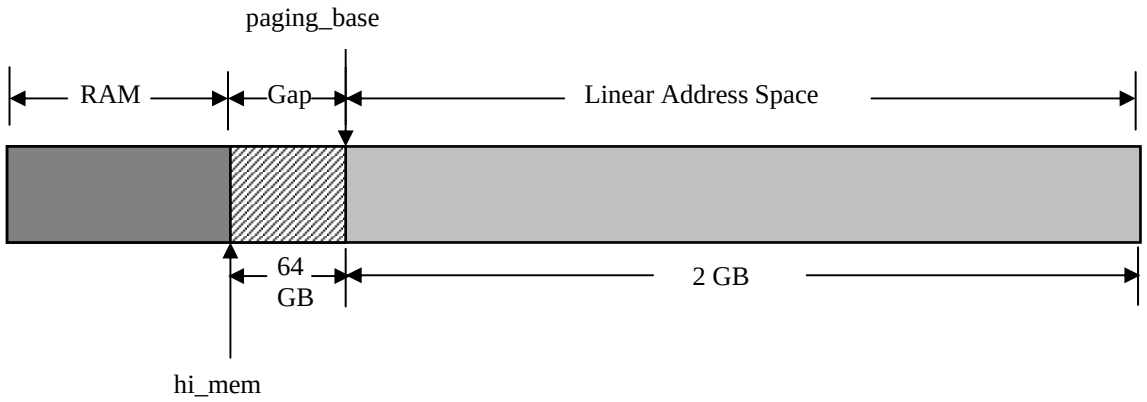


Figure 4.4 : Virtual Addressing

The base of linear address space to be used by servers and processes is calculated and copied in a variable named *paging_base*. Now this linear address space starting from *paging_base* and having range of *VM_SIZE_CLICKS* (2 GB), as shown in Figure 4.3, is added to *chunk_table* so that servers and user processes can treat it as normal memory. In this stage this is the only chunk present in *chunk_table* since all the physical chunks had been removed already as discussed above.

4.2.3 Page fault handling

In MINIX, memory manager allocates the memory to processes from the chunk table. As now, the chunk table consists of linear address space, so the memory allocated to the processes will not be in the form of physical memory, but instead in the terms of linear address space.

This linear address space should to be mapped back to the physical memory. The kernel carries out this function whenever a page fault occurs. This page fault is handled by a page fault handler, which maps the linear address space to physical memory and the process generating this page fault could be resumed.

The page fault handler first checks whether the faulting address is valid or not, and if it is not a valid address kernel halts. If the address is valid, page fault handler finds the corresponding page directory entry. With the help of this entry it finds the address of the page table. Fro here it gets the logical address of the page that cause the page fault. Finally this it brings this page into free frame.

CHAPTER 5 DESIGN IMPLEMENTATION

In this chapter we will discuss all the algorithms used for implementing the design explained in the previous chapter.

5.1 Routines related to operation in chunk table

This section describes all the routines related to operation in chunk table.

5.1.1 chunk_init( ) routine

This routine initializes the chunk table. It simply putts the chunk size equal to zero for each entry in the chunk table. This zero indicates that the chunk is empty.

5.1.2 chunk_add( ) routine

The *chunk_add( )* routine adds entry for a new chunk to the chunk table. Its prototype is given below.

```
PUBLIC void chunk_add(phys_clicks  base, phys_clicks size);
```

Here parameter *base* specifies the base address, and parameter *size* specifies the size of the chunk to be deleted. As shown in the Figure 5.1(a) through 5.1(c) there may be three kinds of situations, that may by develop while adding a new chunk to the chunk table. In each Figure, empty part of the page table is not shown and each address is in terms of clicks.

Let the chunk to be added to the chunk table is X and Y is a chunk already present in the chunk table. In case one, the base of the chunk X is equal to end address of a chunk Y as shown in the Figure 5.1(a). In this case the chunk Y is removed from the table and added to the chunk X. To remove the chunk *chunk_del( )* routine is called. This routine puts the last chunk entry in place of the chunk entry to be deleted. Now the chunk X will start from the base of chunk Y and its length is the sum of the lengths of X and Y. This new larger

chunk is again compared with all the chunks to find the similar situation described above. At the end chunk X is added to the end of the chunk table.

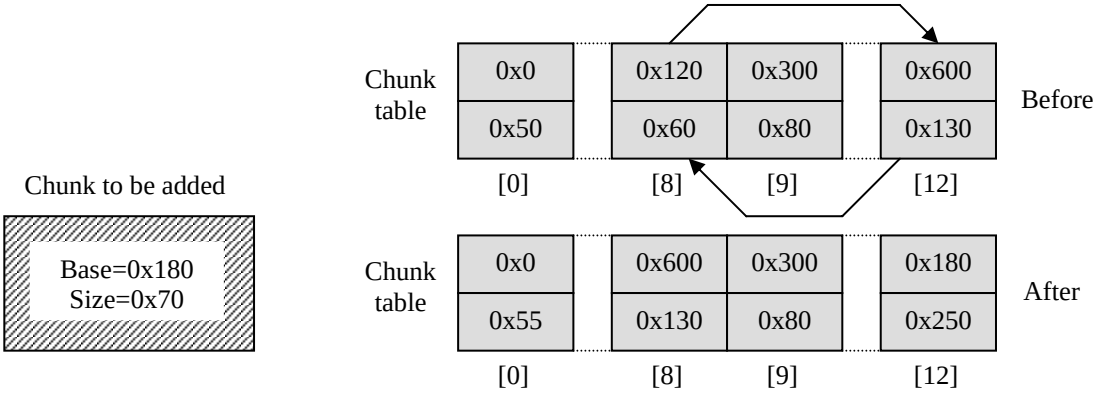


Figure : 5.1 (a)

In case two, the base of the chunk Y is equal to end address of a chunk X as shown in the Figure 5.1(b). In this case the chunk also Y is removed from the table and added to the chunk X. This new larger chunk is again compared with all the chunks and if it can be merged with some other chunk, it is merged with that. At the end chunk X is added to the end of the chunk table.

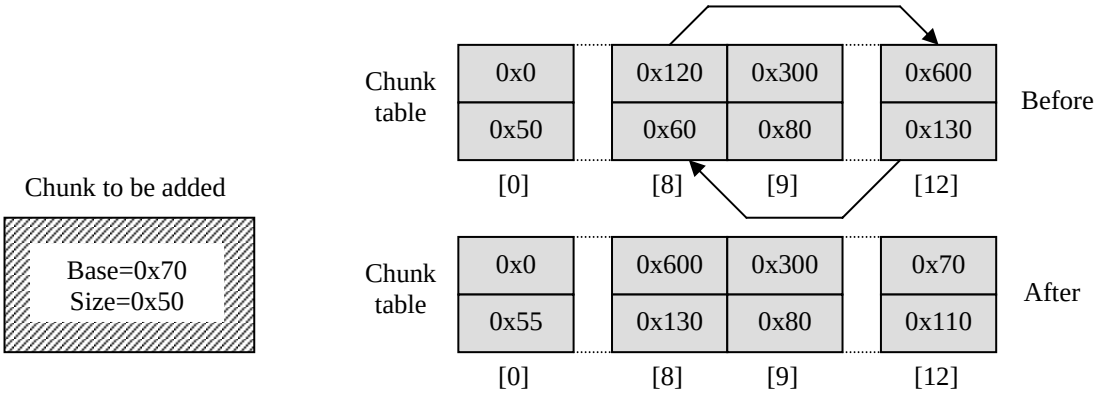


Figure : 5.1 (b)

In case three, the chunk X is not touching any other chunk boundary. In this case the chunk X is simply added to the end of the chunk table. Thus this situation increases the chunk table size while the above two situations may decrease the chunk table size. This case is shown in the Figure 5.1(c).

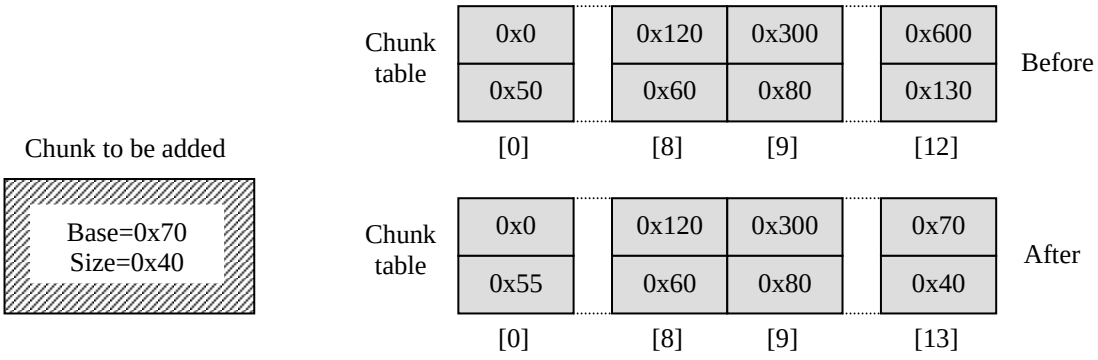


Figure : 5.1 (c)

5.1.3 chunk_del( ) routine

The chunk delete function deletes a specified length chunk from the chunk table. Its prototype is given below.

```
PUBLIC void chunk_del(phys_clicks  base, phys_clicks size);
```

Here parameter *base* specifies the base address, and parameter *size* specifies the size of the chunk to be deleted. In case of deletion, two kinds of situations may arrive. These situations are shown in the Figure 5.2. Let the chunk to be deleted is X and it is subpart of a chunk Y that is already present in the chunk table. Note that X can not overlap two chunks.

In first case, neither the base line of chunk X touching the base line of chunk Y, nor the ending line of X is touching the ending line of Y. That mans chunk X is completely inside the chunk Y. This situation is shown in the Figure 5.2 (a).

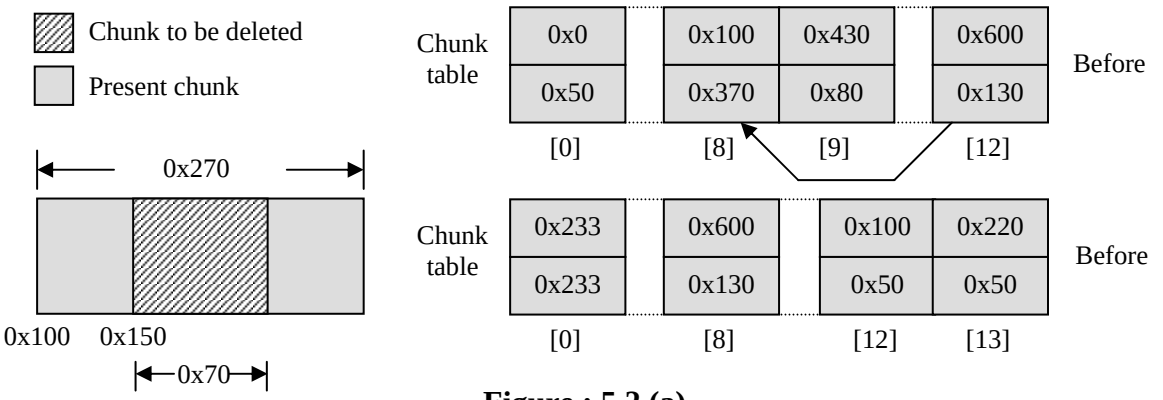


Figure : 5.2 (a)

So after removing chunk X form the chunk Y, the letter will break up in two chunks. What *chunk_del( )* function does is that it first removes the chunk Y from the chunk table. In the second step it calls the *chunk_add( )* function to add these newly created chunks to the chunk table.

In second case, the size of the chunk X is equal to the size of the chunk Y. In this cast chunk Y is removed from the chunk table and the last chunk entry is put at Y's position. This is shown in the Figure 5.2 (b).

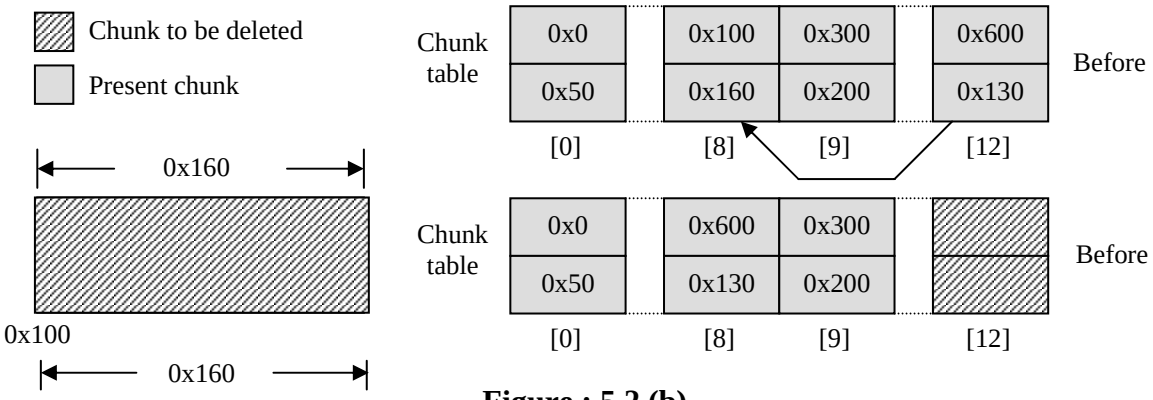


Figure : 5.2 (b)

5.1.4 chunk_find( )

This function finds a specified length chunk in the chunk table. its prototype is given below.

```
PUBLIC void chunk_find(phys_clicks  *base, phys_clicks *size);
```

The parameter *size* specifies the size of the chunk to be found. The routine simply scans through the chunk table to find a chunk having size greater than or equal to the *size*. Once it find such chunk, it sets the pointer *base* to point to the base address of the chunk found and sets the *size* pointer to point to the size of the chunk.

5.2 Routines related to operation in page frame table

This section describes all the routines related to operation in page frame table.

5.2.1 The rlmem_init( ) routine

This routine creates and initializes the table *rlmem_table*. The prototype of this routine is

void rlmem_init();

It calculates the highest address of physical memory by scanning through *chunk_table*, and store this value in a variable named *hi_mem*. Its value is equal to the sum of the base address and the size of the last chunk and is represented in terms of clicks. For example if the base address of the last chunk is 0x3f01 chunks and its size is 0xff chunks then the *hi_mem* is equal to 0x4000 chunks and is equal to 64 Mb. As *hi_mem* is represented in terms of clicks and one click is equal to one page frame so it represents the total number of page frames present in the memory.

Now, the *rlmem_table* has been initialized and its size is known the next step is to allocate memory for *rlmem_table*. To do this the routine *rlmem_init( ) scanes* the *chunk_table* until it finds a chunk that is large enough to hold the *rlmem_table*. The required numbers of bytes are extracted from the chunk and are allocated to *rlmem_table*. After space has been

allocated to the table the variables *rlmem_table_base* and *rlmem_table_size* are set to point to the starting address and the length of the table respectively.

Initially all entries in the *rlmem_table* are set to 1 through a assembly routine *put_phys_byte( )*. Its prototype is given below.

```
void put_phys_byte (phys_bytes phys_addr, int byte);
```

here *phys_addr* is the physical address of the byte where data is to be written and *byte* represents the value to be written. As explained earlier this one in each entry of the *rlmem_table* represent that page frame is in use and is used by one process.

Now in the next step is to mark all the free frames (i.e. set 0 to its reference count). To do this, *rlmem_init( )* scans the entire *chunk_table* and for each chunk, it performs the following steps: in first step, for each frame in this chunk it calls a function *rlmem_free().* The *rlmem_free( )* function checks to see whether this frame is free or in use, and if it is free it sets is reference count to zero. Detailed working of this routine is explained in the next section. In the second step, this chunk is removed from the chunk table with the help of *chunk_del( )* function. The *rlmem_table* finally denotes the status of each page frame and *chunk_table* contains no chunk. A variable *vm_not_alloc* represents the number of free frames available in the system.

5.2.2 rlmem_free( ) routine

As explained in the previous section, the job of this routine is to check the status of a page frame that whether it is free or in use and set accordingly its reference count. Prototype of this function is given below.

```
PRIVATE int rlmem_free(phys_bytes page_addr);
```

Parameter *page_addr* represents the base address of the page frame. To check the status of this page frame, its corresponding entry in the *rlmem_table* has to be found. This is done with the help of following code.

```
entry_addr= rlmem_table_base + (page_addr >> CLICK_SHIFT);
```

After the execution of above statement, variable *entry_addr* contains the physical address of the page frame entry corresponding to the desired page frame. Now the table entry is red with the help of a assembly language routine *get_phys_byte( ).* Prototype of this routine is given below.

```
int get_phys_byte (phys_bytes phys_addr);
```

Here parameter *phys_addr* represents the physical address of the byte to be read.

Now the *rlmem_free( )* reads the byte returned by the *get_phys_byte( )* and decrements it. If the reference count is zero after decrement, it increments the number of free page frames available.

5.2.3 rlmem_getpage( ) routine

This routine searches for a free page frame, and if available returns its base address. Prototype of this routine is given below.

```
PRIVATE phys_bytes rlmem_getpage();
```

The routine first reads the free memory counter to check if any page frame is free or not. If the counter is non zero, there are some free frames available in the memory and routines continues otherwise the system is halted. r*lmem_getpage( )* now calls a assembly language routine *phys_zero_scan( )* with parameters *rlmem_table_base*, *rlmem_table_size*. Prototype of this routine is given below.

```
phys_bytes phys_zero_scan (phys_bytes table_base, phys_bytes table_size);
```

Input to this routine is a table of single bytes whose base address is represented by *table_base* and size (in terms of bytes) is represented by *table_size*. This routine scans the entire table to find an entry having value zero and returns its base address.

Thus *phys_zero_scan( )* finds a free frame entry in the *rlmem_free* table and returns the base address of the free frame. Now *rlmem_getpage( )* routine marks this page frame as used with the help of assembly language routine *put_phys_byte( )* and decrements the free memory counter by one.

Finally routine returns the of the free page frame.

5.3 Routines related to operation in page table and page directory

This section describes all the routines related to creation of page table and page directory.

5.3.1 map_dir( ) routine

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

```
PRIVATE 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 5.3.

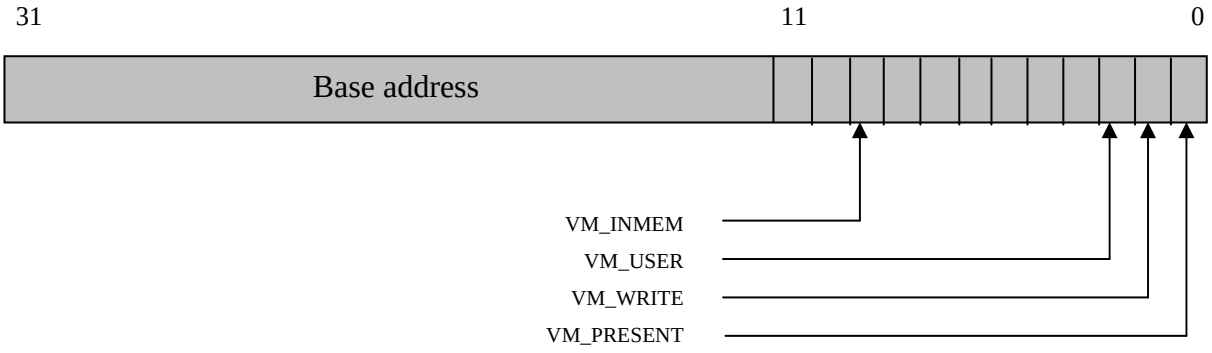


Figure 5.3: Format of page table/ page directory entry

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.

5.3.2 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.

```
PRIVATE 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 5.3.

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( ).*

5.3.3 vm_init( ) routine

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

void vm_init()

It invokes *rlmem_init( )* to cretae *rlmem_table* and to remove chunks from chunk table. *vm_init()* will now get a free page frame through the routine *rlmem_getpage( )* for creating page global directory. The routine now clears this page frame with the help of the routine *phys_clr_page( ).*

Now *vm_init( )* routine will make entries in the page global directory for linear address range of 0 to *hi_mem* through a loop structure given below.

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

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

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

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

In this loop, it updates the corresponding entry in the page table by invoking another routine *map_page().* At the end of each cycle, loop index *page_addr* is increased by a constant CLICK_SIZE, which is the size of linear address interval spanned by a single page frame mapped in page table entry.

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

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

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

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

5.4 Routine for page fault handling

This section describes the routines related to page fault handling.

5.4.1 vm_page_fault ( ) routine

This routine is page fault handler. Prototype of the routine is given below.

```
void vm_page_fault(err, addr)
u32_t err;
phys_bytes addr;
```

The parameter *err* contains the error code thrown by the page fault exception. First bit (LSB) of *err* is used to denote *protection violation* (bit value =1) or *page not present* (bit value =0). Second bit of *err* is used to denote *read* (bit value =0) or *write* (bit value =1) operation which the process responsible for page fault was performing. Third bit of *err* is used to denote *user*(bit value =1) or *supervisor* (bit value =0) mode of process responsible for page fault. The second parameter *addr* is the linear address that caused the page fault.

The routine first checks that address is actually a linear address. If not, kernel exits. The routine then reads and stores the page global directory entry corresponding to the linear address *addr* in a variable *dir_ent* as given below.

```
dir_ent_addr= page_base + vm_addr_to_dir(addr)*4;
        dir_ent= get_phys_dword(dir_ent_addr);
```

Then *dir_ent* is checked for *VM_PRESENT* bit. If the bit is enabled, it indicates the entry is filled to point to a page table. Then the routine reads and stores the page table entry corresponding to the linear address *addr* in a variable *page_ent* as given below.

```
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);
        }
```

After that it is checked whether the process is a system task or kernel. If it is kernel then kernel panics. If it is a user process then another routine named *check_user_fault* is invoked with *addr* as a parameter. Its prototype is:

```
int check_user_fault(addr).
```

The routine checks whether the page fault is in the range of text, code, stack or heap (growing stack) segment. If it is not in any of the segments i.e. out of boundaries, an error

is returned. Otherwise *OK* is returned. If *check_user_fault()* does not return *OK* the *vm_page_fault()* returns. These steps are shown below.

```
if (proc_number(proc_ptr) < 0)
        {
                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;
        }
```

In the above segment of routine *vm_cp_mess* is a kernel flag to indicate that message-copying routine is responsible for page fault.

Now if page directory entry is blank an entry is made and routine returns as given below.

```
if (!dir_ent)
        {
                dir_addr= rlmem_getpage();
                phys_clr_page(dir_addr);  /* No pages */
                map_dir(addr & ~VM_DIRMASK, dir_addr);

                vm_reload();
                return;
        }
```

If page directory entry was already their then it is checked for a proper entry as shown below. If strange entry is found, kernel panics.

```
        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);
        }
```

Then the other possibility of page fault related to page table entry is considered. If page table entry is empty, a page frame is allocated and mapped in the page table entry as given below.

```
if (!page_ent)
        {
                page_addr= rlmem_getpage();
                phys_clr_page(page_addr);           /* Empty page*/
                map_page(addr & VM_ADDRMASK, page_addr);
                vm_reload();
                return;
        }
```

If page entry is their then the copy on access possibility is checked and page is copied in a new page frame. Then this new page frame is mapped in the page table entry and link count is updated accordingly. *VM_PRESENT* bit is also enabled in the page table entry. These steps are given below.

```
/* Copy on access */


if (!(page_ent & VM_PRESENT))
{
        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);
}                                                                   /* End of routine*/
```

CHAPTER 6 CONCLUSIONS AND FUTURE WORK

6.1 Conclusions

In this dissertation a design is proposed for page frame management to support virtual addressing in MINIX operating system. During the course of this project, emphasis is also given on the page fault handling for achieving this enhancement in the MINIX operating system. Kernel and 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 project results in a MINIX operating system design in which servers and user processes use the linear address space instead of physical address space as before.

This design is proposed keeping Intel based 80x86 architecture, starting from 80386 onwards as target architecture. The target machines on which this design is implemented and tested are Pentium II and Pentium III based machines. The primary target machine is an Intel Pentium II based system with 64MB of physical memory available. The design is also tested for a multiple boot operation with a standard Windows 98 distribution and a Red Hat Linux 9.0 distribution on this target machine.

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

To keep the focus of this dissertation as much on the primary problem taken up in this project, some of the changes which are not directly related with the key goals in the design for page frame management to support virtual addressing for MINIX operating system are not included in this dissertation. Instead the focus of this dissertation is to present these primary changes in an elaborate manner.

6..2 Future work

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

Varying length instruction between different architectures:
Some CPU architectures have instructions with varying numbers of arguments. For example the Motorola 68000 has a move instruction with two arguments (source and target of the move). It can cause faults for three different reasons: the instruction itself or either of the two operands. The fault handler has to determine which reference faulted. On some computers, the OS has to figure that out by interpreting the instruction and in effect simulating the hardware. The 68000 made it easier for the OS by updating the PC as it goes, so the PC 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.

Side effects
Some computers have addressing modes that automatically increment or decrement index registers as a side effect, making it easy to simulate in one step the effect of the C statement:

```
*p++ = *q++;.
```

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.

Second enhancement that can be made to this design is to modify the page fault handler to use page replacement algorithm, so that the total design can be converted from non paging system to a paging system. To support these changes, page frame allocation mechanism should also be modified. This new page frame allocation mechanism will not allocate all the page frames needed by the demanding process. The page frame allocator can use working set model or page fault frequency count to decide the number of pages frames to be allocated to a process. After all these changes, this design could be converted to paging with virtual memory.

Third enhancement is to allocate separate page directory for each process. In this way the page frames can be managed more efficiently.

One more enhancement is to allocate only a portion of page frame to a process if the process is very small. The remaining free portion of the page frame can be used by some other small process(s). This will reduce the internal fragmentation. This strategy is also used in Linux.

REFERENCES

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

[2] 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.

[3] 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.

[4] 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.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[35] Elana D. Granston, Harry A. G. Wijshoff: "Managing pages in shared virtual memory systems: getting the compiler into the game", Proceedings of the 7th international conference on Supercomputing, Tokyo, Japan, 1993.

/***

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 */
```

/***

End of vm386.h

***/

/***

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);
}


/*===========================================================================*
 *                                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

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;
}

/*****************************************************************************************************************
```

End of vm386.c

```
*****************************************************************************************************************/
```

```
!**************************************************************************************************/
```

! vm386_s.x
!
! This file contains assembler routines for the 386 virual 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

!*****************************************************************************************************/
```

! end of vm386_s.x

```
!*****************************************************************************************************/
```

```
/*===========================================================================*
 *                                system.c                                   *
 *===========================================================================*/
#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;
/*          case SYS_UNMAP:             r = do_unmap(&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;
{

  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 */
{
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()
{
  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 (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 realy 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;
/* Allocate physical memory */
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 */
  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);
return(OK);
}
#endif /* (CHIP == INTEL) && VIRT_MEM */
```