Chapter 6

MEMORY MANAGEMENT

6.1 Background

Memory is a large array of words or bytes, each with its own address. The CPU fetches instructions from memory according to the value of the program counter. These instructions may cause additional loading from and storing to specific memory addresses.

A typical instruction execution cycle, for example, will first fetch an instruction from memory. The instruction is then decoded and may cause operands to be fetched from memory. After the instruction has been executed on the operands, results may be stored back in memory. Notice that the memoryunit sees only a stream of memory addresses.

It does not know how they are generated (the instruction counter, indexing, indirection, literal addresses, and so on) or what they are for (instructions or data). Accordingly, we can ignore ***how*** a memory address is generated by a program. We are interested in only the sequence of memory addresses generated by the running program.

6.1.1 Address Binding

A program resides on a disk as a binary executable file. The program must be brought into memory and placed within a process for it to be executed. Depending on the memory management in use, the process may be moved between disk and memory during its execution. The collection of processes on the disk that are waiting to be brought into memory for execution forms the ***input queue.***

The normal procedure is to select one of the processes in the input queue and to load that process into memory. As the process is executed, it accesses instructions and data from memory. Eventually, the process terminates, and its memory space is declared available.

Most systems allow a user process to reside in any part of the physical memory. Thus, although the address space of the computer starts at 00000, the

first address of the user process does not need to be 00000. This arrangement affects the addresses that the user program can use.

Addresses in the source program are generally symbolic (such as COUNT). A compiler will typically ***bind*** these symbolic addresses to relocatable addresses (such as "14 bytes from the beginning of this module"). The linkage editor or loader will in turn bind these relocatable addresses to absolute addresses (such as 74014). Each binding is a mapping from one address space to another.

Classically, the ***binding*** of instructions and data to memory addresses can be done at any step along the way:

1. **Compile time:**

If it is known at compile time where the process will reside in memory, then ***absolute*** code can be generated.

For example, if it is known a priori that a user process resides starting at location R, then the generated compiler code will start at that location and extend up from there. If, at
some later time, the starting location changes, then it will be necessary to recompile this code. T he MS-DOS.COM-format programs are absolute code bound at compile time.

2. **Load time:**

If it is not known at compile time where the process will reside in memory, then the compiler must generate ***relocatable*** code. In this case, final binding is delayed until load time. If the starting address changes, we need only to reload the user code to incorporate this changed value.

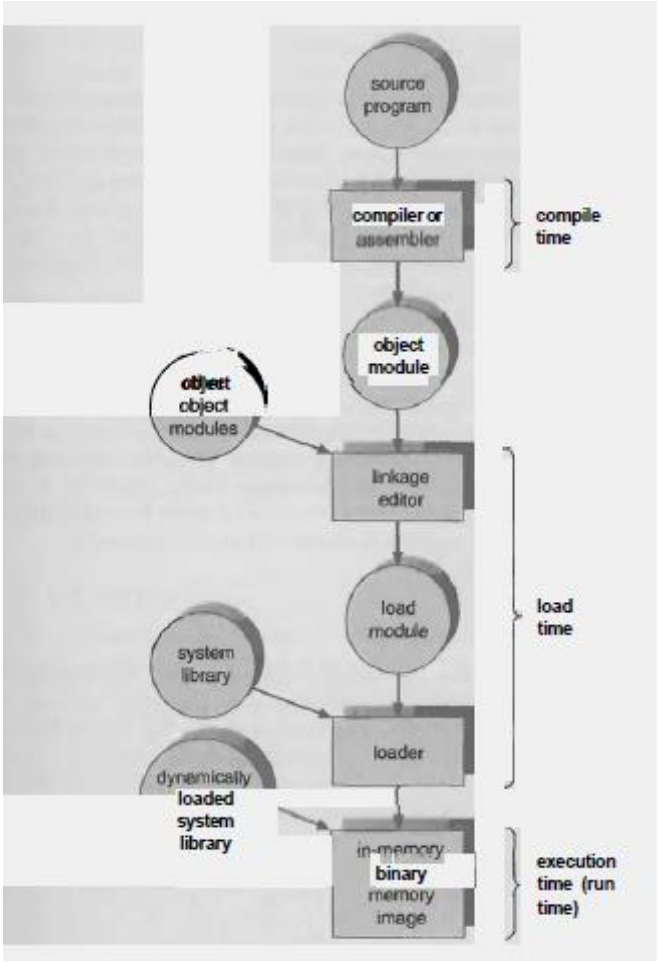


Figure 6.1 Multistep processing of a user program.

Execution time:

If the process can be moved during its execution from One memory segment to another, then binding must be delayed until run time. Special hardware must be available for this scheme to work.

A major portion of this chapter is devoted to showing how these various bindings can be implemented effectively in a computer system, and discussing appropriate hardware support.

6.1.2 Dynamic Loading

To obtain better memory-space utilization, we can use ***dynamic loading.*** With dynamic loading, a routine is not loaded until it is called. All routines are kept on disk in a relocatable load format.

The main program is loaded into memory and is executed. When a routine needs to call another routine, the calling routine first checks to see whether the other routine has been loaded. If it has not been, the relocatable linking loader is called to load the desired routine into memory and to update the program's address tables to reflect this change. Then, control is passed to the newly loaded routine.

The advantage of dynamic loading is that an unused routine is never loaded. This scheme is particularly useful when large amounts of code are needed to handle infrequently occurring cases, such as error routines.

In this case, although the total program size may be large, the portion that is actually used (and hence actually loaded) may be much smaller. Dynamic loading does not require special support from the operating system.

It is the responsibility of the users to design their programs to take advantage of such a scheme. Operating systems may help the programmer, however, by providing library routines to implement dynamic loading.

6.1.3 Dynamic Linking

Notice that Figure 6.1 also shows ***dynamically linked*** libraries. Most operating systems support only ***static linking,*** in which system language libraries are treated like any other object module and are combined by the loader into the binary program image.

The concept of dynamic linking is similar to that of dynamic loading. Rather than loading being postponed until execution time, linking is postponed. This feature is usually used with system libraries, such as language subroutine libraries. Without this facility, all programs on a system need to have a copy of their language library (or at least the routines referenced by the program) included in the executable image.

This requirement wastes both disk space and main memory. With dynamic linking, a ***stub*** is included in the image for each library-routine reference. This stub is a small piece of code that indicates how to locate the

appropriate memory-resident library routine, or how to load the library if the routine is not already present.

When this stub is executed, it checks to see whether the needed routine is already in memory. If the routine is not in memory, the program loads it into memory.

The stub replaces itself with the address of the routine, and executes the routine. Thus, the next time that that code segment is reached, the library routine is executed directly, incurring no cost for dynamic linking. Under this schee , all processes that use a language library execute only one copy of the library code.

A library may be replaced by a new version, and all programs that reference the library will automatically use the new version. Without dynamic linking, all such programs would need to be relinked to gain access to the new library. So that programs will not accidentally execute new, incompatible versions of libraries, version information is included in both the program and the library.

More than one version of a library may be loaded into memory, and each program uses its version information to decide which copy of the library to use. Minor changes retain the same version number, whereas major changes increment the version number.

Thus, only programs that are compiled with the new library version are affected by the incompatible changes incorporated in it. Other programs linked before the new library was installed will continue using the older library. This system is also known as ***shared libraries.***

Dynamic linking generally requires some help from the operating system. If the processes in memory are protected from one another then the operating system is the only entity that can check to see whether the needed routine is in another processes' memory space, and can allow multiple processes to access the same memory addresses.

6.1.4 Overlays

The entire program and data of a process must be in physical memory for the process to execute. The size of a process is limited to the size of physical memory. So that a process can be larger than the amount of memory allocated to it, a technique called ***overlays*** is sometimes used.

The idea of overlays is to keep in memory only those instructions and data that are needed at any given time. When other instructions are needed, they are loaded into space that was occupied previously by instructions that are no longer needed.

Example:

Consider a two-pass assembler. During pass 1, it constructs a symbol table; then, during pass 2, it generates machine-language code. We may be able to partition such an assembler into pass 1 code, pass 2 code, the symbol table, and common support routines used by both pass 1 and pass 2.

Assume that the sizes of these components are as follows **(K** stands for "kilobyte," which is 1024 bytes):

Pass 1	70K
Pass 2	80K
Symbol table	20K
Common routines	30K

To load everything at once, we would require 200K of memory. If only 150K is available, we cannot run our process. However, notice that pass 1 and pass 2 do not need to be in memory at the same time. We thus define two overlays:

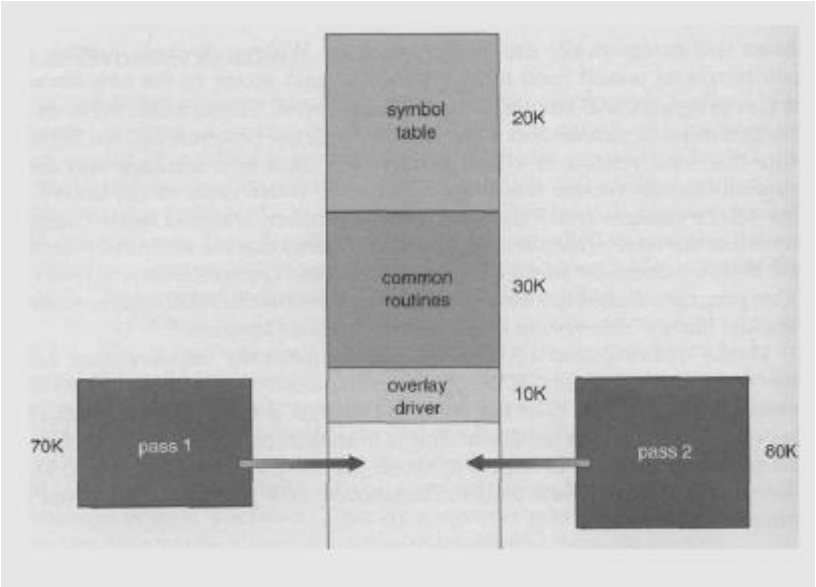


Figure **6.2** Overlays for a two-pass assembler.

Overlay A is the symbol table, common routines, and pass 1, and overlay B is the symbol table, common routines, and pass 2.

We add an overlay driver (10K) and start with overlay A in memory. When we finish pass 1, we jump to the overlay driver, which reads overlay B into memory, overwriting overlay A, and then transfers control to pass 2.

Overlay A needs only 120K, whereas overlay B needs 130K (Figure 8.2). We can now run our assembler in the 150K of memory. It will load somewhat faster because fewer data need to be transferred before execution starts.

However, it will run somewhat slower, due to the extra 1/0 to read the code for overlay B over the code for overlay A. The code for overlay A and the code for overlay B are kept on disk as absolute memory images, and are read by the overlay driver as needed.

Special relocation and linking algorithms are needed to construct the overlays. As in dynamic loading, overlays do not require any special support from the operating system. They can be implemented completely by the user with simple file structures, reading from the files into memory and then jumping to that memory and executing the newly read instructions. The operating system notices only that there is more 1/0 than usual.

The programmer, on the other hand, must design and program the overlay structure properly. This task can be a major undertaking, requiring complete knowledge of the structure of the program, its code, and its data structures.

Because the program is, by definition, large (small programs do not need to be overlayed), obtaining a sufficient understanding of the program may be difficult. For these reasons, the use of overlays is currently limited to microcomputer and other systems that have limited amounts of physical memory and that lack hardware support for more advanced techniques.

Some microcomputer compilers provide to the programmer support of overlays to make the task easier. Automatic techniques to run large programs in limited amounts of physical memory are certainly preferable.

6.2 Logical versus Physical Address Space

An address generated by the CPU is commonly referred to as a ***logical address,*** whereas an address seen by the memory unit (that is, the one loaded into the ***memory address register*** of the memory) is commonly referred to as a ***physical address.***

[cle]ar->compiletime , loadtime,execution time

The compile-time and load-time address-binding schemes result in an environment where the logical and physical addresses are the same. the execution-time address-binding scheme results in an environment where the logical and physical addresses differ. In this case, we usually refer to the logical address as a ***virtual address.***

We use ***logical address*** and ***virtual address*** interchangeably in this text. The set of all logical addresses generated by a program is referred to as a ***logical address space;*** the set of all physical addresses corresponding to these logical addresses is referred to as a ***physical address space.***

In the execution-time address-binding scheme, the logical and physical address spaces differ. The run-time mapping from virtual to physical addresses is done by the ***memory-management unit*** *(MMU),* which is a hardware device.

The base register is now called a ***relocation*** register. The value in the relocation register is ***added*** to every address generated by a user process at the time it is sent to memory.

Example:

If the base is at 14,000, then an attempt by the user to address location 0 is dynamically relocated to location 14,000; an access to location 346 is mapped to location 14,346.

The MS-DOS operating system running on the Intel 80x86 family of processors uses four relocation registers when loading and running processes.

Notice that the user program never sees the ***real*** physical addresses. The program can create a pointer to location 346, store it in memory, manipulate it, compare it to other addresses - all as the number 346.

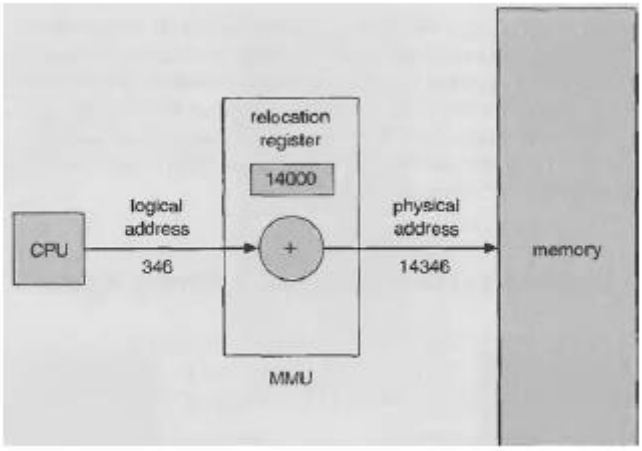


Figure 6.3 Dynamic relocation using a relocation register.

Only when it is used as a memory address (in an indirect load or store, perhaps) is it relocated relative to the base register. The user program deals with ***logical*** addresses. The memory-mapping hardware converts logical addresses into physical addresses.

The final location of a referenced memory address is not determined until the reference is made. Notice also that we now have two different types of addresses: logical addresses (in the range 0 to ***max)*** and physical addresses (in the range R + 0 to R + ***max*** for a base value R).

The user generates only logical addresses and thinks that the process runs in locations 0 to ***max.*** The user program supplies logical addresses; these logical addresses must be mapped to physical addresses before they are used.

The concept of a ***logical address space*** that is bound to a separate ***physical address space*** is central to proper memory management.

6.3 Swapping

A process needs to be in memory to be executed. A process, however, can be ***swapped*** temporarily out of memory to a ***backing store,*** and then brought back into memory for continued execution. For example, assume a multipmgramming environment with a round-robin CPU-scheduling algorithm. When a quantum expires, the memory manager will start to swap out the process that just finished, and to swap in another process to the memory space that has been freed (Figure 6***.4).***

In the meantime, the CPU scheduler will allocate a time slice to some other process in memory. When each process finishes its quantum, it will be swapped with another process. Ideally, the memory manager can swap processes fast enough that there are always processes in memory, ready to execute, when the CPU scheduler wants to reschedule the CPU. The quantum must also be sufficiently large that reasonable amounts of computing are done between swaps.

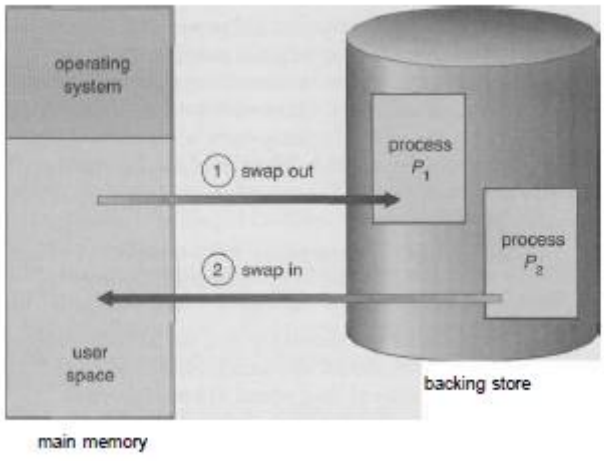


Figure 6.4 Swapping of two processes using a disk as a backing store.

A variant of this swapping policy is used for priority-based scheduling algorithms. If a higher-priority process arrives and wants service, the memory manager can swap out the lower-priority process so that it can load and execute the higher-priority process. When the higher-priority process finishes, the lower-priority process can be swapped back in and continued. This variant of swapping is sometimes called *roll out, roll in.*

Normally a process that is swapped out will be swapped back into the same memory space that it occupied previously. This restriction is dictated by the method of address binding. If binding is done at assembly or load time, then the process cannot be moved to different locations. If execution-time binding is being used, then it is possible to swap a process into a different memory space, because the physical addresses are computed during execution time.

Swapping requires a *backing store.* The backing store is commonly a fast disk. It must be large enough to accommodate copies of all memory images for all users, and must provide direct access to these memory images. The system maintains a *ready queue* consisting of all processes whose memory images are on the backing store or in memory and are ready to run. Whenever the CPU scheduler decides to execute a process, it calls the dispatcher.

The dispatcher checks to see whether the next process in the queue is in memory. If the process is not, and there is no free memory region, the dispatcher swaps out a process currently in memory and swaps in the desired process. It then reloads registers as normal and transfers control to the selected process.

It should be clear that the context-switch time in such a swapping system is fairly high. To get an idea of the context-switch time, let us assume that the user process is of size lOOK and the backing store is a standard hard disk with a transfer rate of 1 megabyte per second. The actual transfer of the lOOK process to or from memory takes

lOOK / lOOOK per second = 1/10 second
= 100 milliseconds

Assuming that no head seeks are necessary and an average latency of 8 milliseconds, the swap time takes 108 milliseconds. Since we must both swap out and swap in, the total swap time is then about 216 milliseconds.

For efficient CPU utilization, we want our execution time for each process to be long relative to the swap time. Thus, in a round-robin CPU-scheduling algorithm, for example, the time quantum should be substantially larger than 0.216 seconds.

Notice that the major part of the swap time is transfer time. The total transfer time is directly proportional to the amount of memory swapped. If we have a computer system with 1 megabyte of main memory and a resident operating system taking 100K, the maximum size of the user process is 900K. However, many user processes may be much smaller than this size - say, 100K.

A lOOK process could be swapped out in 108 milliseconds, compared to the 908 milliseconds for swapping 900K. Therefore, it would be useful to know exactly how much memory a user process is using, not simply how much it ***might*** be using. Then, we would need to swap only what is actually used, reducing swap time.

For this scheme to be effective, the user must keep the system informed of any changes in memory requirements. Thus, a process with dynamic memory requirements will need to issue system calls **(request memory** and **release memory)** to inform the operating system of its changing memory needs.

There are other constraints on swapping. If we want to swap a process, we must be sure that it is completely idle. Of particular concern is any pending I/O. If a process is waiting for an I/O operation, we may want to swap that process to free up its memory. However, if the I/O is asynchronously accessing the user memory for [/O buffers, then the process cannot be swapped.

Assume that the I/O operation was queued because the device was busy. Then, if we were to swap out process **P1** and swap in process ***P2,*** the I/O operation might then attempt to use memory that now belongs to process ***P2.***

The two main solutions to this problem are (1) never to swap a process with pending I/O, or (2) to execute 1/0 operations only into operating-system

buffers. Transfers between operating-system and process memory then occur only when the process is swapped in.

The assumption that swapping requires few if any head seeks needs further explanation. where secondary-storage structure is covered. Generally, swap space is allocated as a separate chunk of disk, separate from the file system, so that its use is as fast as possible.

Currently, standard swapping is used in few systems. It requires too much swapping time and provides too little execution time to be a reasonable memory-management solution. Modified versions of swapping, however, are found on many systems.

A modification of swapping is used in many versions of UNIX. Swapping was normally disabled, but would start if many processes were running and were using a threshold amount of memory. Swapping would again be halted if the load on the system was reduced.

Early PCs lacked sophisticated hardware (or operating systems that take advantage of the hardware) to implement more advanced memory management methods, but they were used to run multiple, large processes by a modified version of swapping. A prime example is the Microsoft Windows 3.1 operating system, which supports concurrent execution of processes in memory. If a new process is loaded and there is insufficient main memory, an old process is swapped to disk.

This operating system, however, does not provide full swapping, because the user, rather than the scheduler, decides when it is time to preempt one process for another. Any swapped-out process remains swapped out (and not executing) until the user selects that process to run.

Follow-on Microsoft operating systems, such as Windows/NT, take advantage of advanced MMU features now found even on PCs. In Section 6.7.2, we describe the memory-management hardware found on the Intel 386 family of processors used in many PCs. In that section, we also describe the memory management used on this CPU by another advanced operating system for PCs: IBM OS/2.

6.4 Contiguous Allocation

The main memory must accommodate both the operating system and the various user processes. The memory is usually divided into two partitions, one for the resident operating system, and one for the user processes. It is possible to place the operating system in either low memory or high memory.

The major factor affecting this decision is the location of the interrupt vector. Since the interrupt vector is often in low memory, it is more common to place the operating system in low memory. Thus, we shall discuss only the situation where the operating system resides in low memory (Figure 8.5). The development of the other situation is similar.

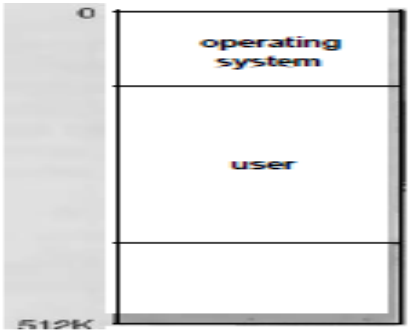


Figure 6.5 Memory partition.

6.4.1 Single-Partition Allocation

If the operating system is residing in low memory and the user processes are executing in high memory, we need to protect the operating-system code and data from changes (accidental or malicious) by the user processes. We also need to protect the user processes from one another.

We can provide this protection by using a relocation register, as discussed in Section 6.2, with a limit register, as discussed in Section 2.5.3. The relocation register contains the value of the smallest physical address; the limit register contains the range of logical addresses (for example, relocation = 100,040 and limit = 74,600).

With relocation and limit registers, each logical address must be less than the limit register; the MMU maps the logical address ***dynamically*** by adding

the value in the relocation register. This mapped address is sent to memory (Figure 6.6).

When the CPU scheduler selects a process for execution, the dispatcher loads the relocation and limit registers with the correct values as part of the context switch. Because every address generated by the CPU is checked against these registers, we can protect both the operating system and the other users' programs and data from being modified by this running process.

Note that the relocation-register scheme provides an effective way to allow the operating-system size to change dynamically. This flexibility is desirable in many situations.

For example, the operating system contains code and buffer space for device drivers. If a device driver (or other operating-system service) is not commonly used, it is undesirable to keep the code and data in memory, as we might be able to use that space for other purposes.

Such code is sometimes called ***transient*** operating-system code;it comes and goes as needed.Thus, using this code changes the size of the operating system during program execution.

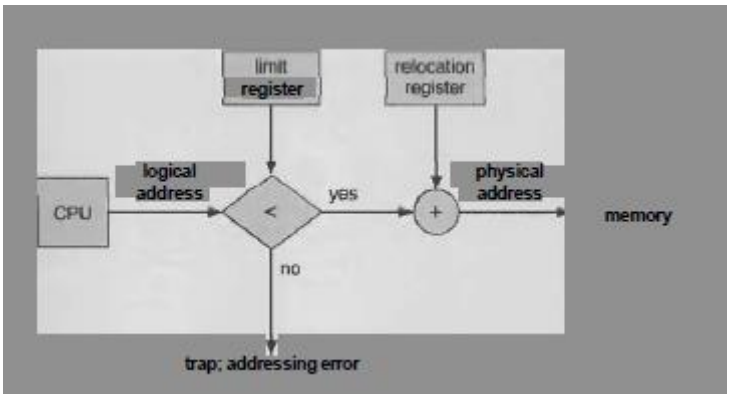


Figure 6.6 Hardware support for relocation and limit registers.

6.4.2 Multiple Partition Allocation

Because it is desirable, in general, that there be several user processes residing in memory at the same time, we need to consider the problem of how

to allocate available memory to the various processes that are in the input queue waiting to be brought into memory.

One of the simplest schemes for memory allocation is to divide memory into a number of fixed-sized ***partitions.*** Each partition may contain exactly one process. Thus, the degree of multiprogramming is bound by the number of partitions. When a partition is free, a process is selected from the input queue and is loaded into the free partition. When the process terminates, the partition becomes available for another process.

This scheme was originally used by the IBM **OS/360** operating system (called MET); it is no longer in use. The scheme described next is a generalization of the fixedpartition scheme (called MVT) and is used primarily in a batch environment.

We note, however, that many of the ideas presented here are also applicable to a time-sharing environment where pure segmentation is used for memory management (Section 6.6).

The operating system keeps a table indicating which parts of memory are available and which are occupied. Initially, all memory is available for user processes, and is considered as one large block of available memory, a ***hole.***

When a process arrives and needs memory, we search for a hole large enough for this process. If we find one, we allocate only as much memory as is needed, keeping the rest available to satisfy future requests.

For example, assume that we have 2560K of memory available and a resident operating system of 400K.

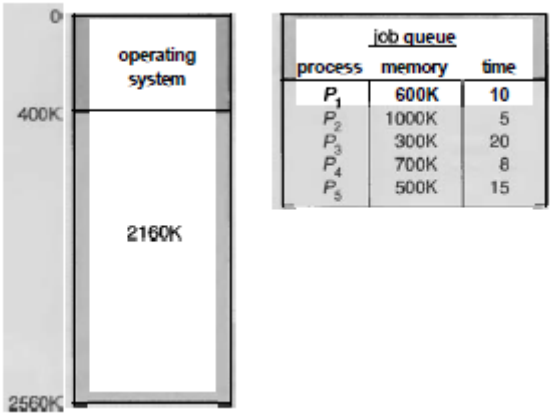


Figure 6.7 Scheduling example.

This situation leaves 2160K for user processes, as shown in Figure 8.7. Given the input queue in the figure, and FCFS job scheduling, we can immediately allocate memory to processes **PI, P2,** and **P3,** creating the memory map of Figure 8.8(a). We have a hole of size 260K that cannot be used by any of the remaining processes in the input queue.

Using a round-robin CPU-scheduling with a quantum of 1 time unit, process **P2** will terminate at time 14, releasing its memory. This situation is illustrated in Figure 6.8(b). We then return to our job queue and schedule the next process, process **P4,** to produce the memory map of Figure 6.8(c). Process **P1** will terminate at time 28 to produce Figure 6.8(d); process **P5** is then scheduled, producing Figure 6.8(e).

This example illustrates the general structure of the allocation scheme. As processes enter the system, they are put into an input queue. The operating system takes into account the memory requirements of each process and the amount of available memory space in determining which processes are allocated memory.

When a process is allocated space, it is loaded into memory and it can then compete for the CPU. When a process terminates, it releases its memory, which the operating system may then fill with another process from the input queue.

At any given time, we have a list of available block sizes and the input queue. The operating system can order the input queue according to a scheduling algorithm.

Memory is allocated to processes until, finally, the memory requirements of the next process cannot be satisfied; no available block of memory (hole) is large enough to hold that process.

The operating system can then wait until a large enough block is available, or it can skip down the input queue to see whether the smaller memory requirements of some other process can be met.

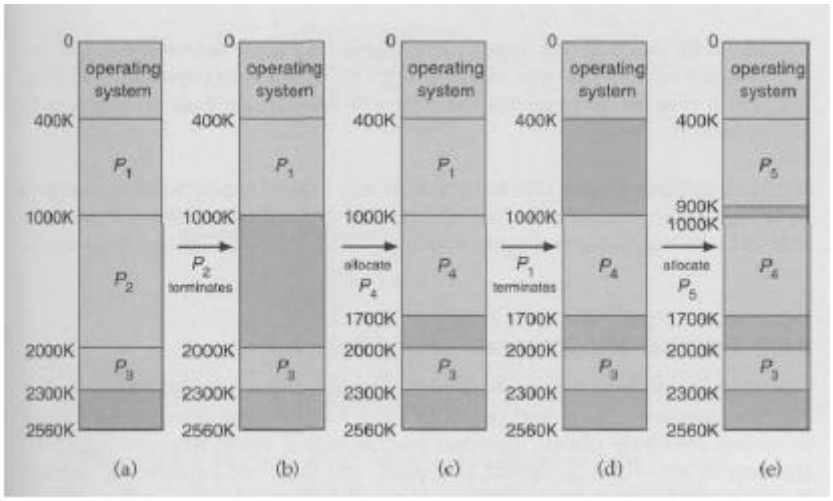


Figure 6.8 Memory allocation and long-term scheduling.

In general, there is at any time a *set* of holes, of various sizes, scattered throughout memory. When a process arrives and needs memory, we search this set for a hole that is large enough for this process. If the hole is too large, it is split into two: One part is allocated to the arriving process; the other is returned to the set of holes.

When a process terminates, it releases its block of memory, which is then placed back in the set of holes. If the new hole is adjacent to other holes, we merge these adjacent holes to form one larger hole.

At this point, we may need to check whether there are processes waiting for memory and whether this newly freed and recombined memory could satisfy the demands of any of these waiting processes. This procedure is a particular instance of the general *dynamic storageallocation* problem, which is how to satisfy a request of size *n* from a list of free holes.

There are many solutions to this problem. The set of holes is searched to determine which hole is best to allocate. *First-fit, best-fit,* and *worst-fit* are the most common strategies used to select a free hole from the set of available holes.

• **First-fit:** Allocate the *first* hole that is big enough. Searching can start either at the beginning of the set of holes or where the previous first-fit search ended. We can stop searching as soon as we find a free hole that is large enough.

- **Best-fit:** Allocate the *smallest* hole that is big enough. We must search the entire list, unless the list is kept ordered by size. This strategy produces the smallest leftover hole.

- **Worst-fit:** Allocate the largest hole. Again, we must search the entire list, unless it is sorted by size. This strategy produces the largest leftover hole, which may be more useful than the smaller leftover hole from a best-fit approach.

Simulations have shown that both first-fit and best-fit are better than worst-fit in terms of decreasing both time and storage utilization. Neither first-fit nor best-fit is clearly better in terms of storage utilization, but first-fit is generally faster.

6.4.3 External and Internal Fragmentation

The algorithms described in Section 6.4.2 suffer from external fragmentation. As processes are loaded and removed from memory, the free memory space is broken into little pieces.

External fragmentation exists when enough total memory space exists to satisfy a request, but it is not contiguous; storage is fragmented into a large number of small holes. Looking back at Figure 6.8, we can see two such situations. In Figure 6.8(a), there is a total external fragmentation of 260K, a space that is too small to satisfy the requests of either of the two remaining processes, ***P4*** and

P5. In Figure 6.8(c), however, we have a total external fragmentation of 560K (= 300K + 260K).

This space would be large enough to run process ***P5*** (which needs 500K), except that this free memory is not contiguous. The free memory space is fragmented into two pieces, neither one of which is large enough, by itself, to satisfy the memory request of process ***P5.***

This fragmentation problem can be severe. In the worst case, we could have a block of free (wasted) memory between every two processes. If all this memory were in one big free block, we might be able to run several more processes.

The selection of first-fit versus best-fit can affect the amount of fragmentation. (First-fit is better for some systems, and best-fit is better for others.) Another factor is which end of a free block is allocated. (Which is the leftover piece - the one on the top, or the one on the bottom?) No matter which algorithms are used, however, external fragmentation will be a problem.

Depending on the total amount of memory storage and the average process size, external fragmentation may be either a minor or a major problem. Statistical analysis of first-fit, for instance, reveals that, even with some optimization, given N allocated blocks, another 0.5N blocks will be lost due to fragmentation.
That is, one-third of memory may be unusable! This property is known as the 50-percent rule.

Another problem that arises with the multiple partition allocation scheme is illustrated by Figure 8.9. Consider the hole of 18,464 bytes. Suppose that the next process requests 18,462 bytes. If we allocate exactly the requested block, we are left with a hole of 2 bytes. The overhead to keep track of this hole will be substantially larger than the hole itself. The general approach is to allocate very small holes as part of the larger request. Thus, the allocated memory may

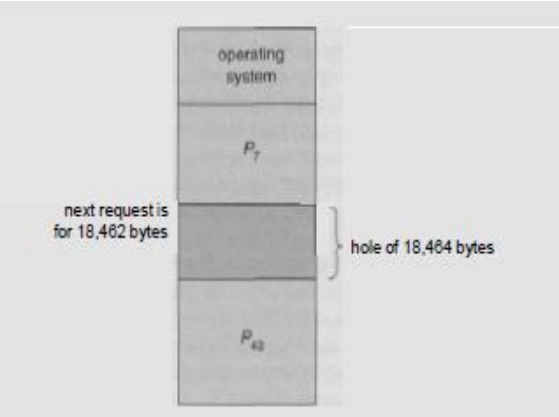


Figure 6.9 Memory allocation made in some multiple of bytes.

be slightly larger than the requested memory. The difference between these two numbers is *internal fragmentation* - memory that is internal to a partition, but is not being used.

One solution to the problem of external fragmentation is *compaction.* The goal is to shuffle the memory contents to place all free memory together in one large block. For example, the memory map of Figure 8.8(e) can be compacted, as shown in Figure 8.10. The three holes of sizes loOK, 300K, and 260K can be compacted into one hole of size 660K.

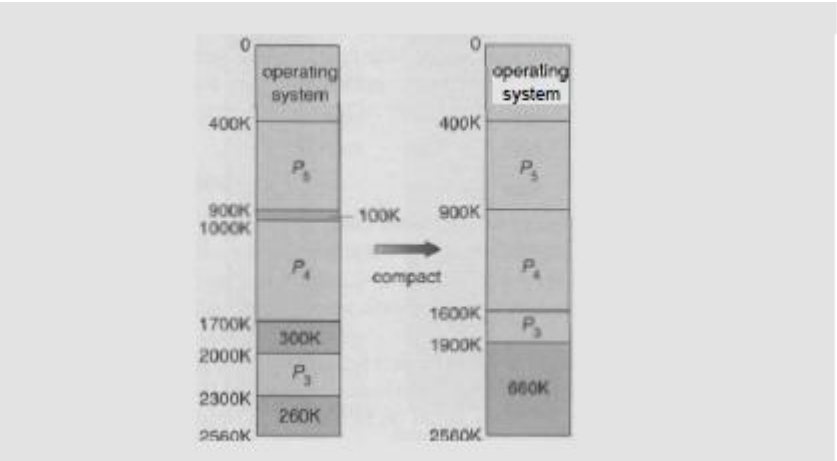


Figure 6.10 Compaction.

Compaction is not always possible. Notice that, in Figure 6.10, we moved processes P4 and P3. For these processes to be able to execute in their new

locations, all internal addresses must be relocated. If relocation is static and is done at assembly or load time, compaction cannot be done; compaction is possible only if relocation is dynamic, and is done at execution time.

If addresses are relocated dynamically, relocation requires only moving the program and data, and then changing the base register to reflect the new base address.

When compaction is possible, we must determine its cost. The simplest compaction algorithm is simply to move all processes toward one end of memory; all holes move in the other direction, producing one large hole of available memory. This scheme can be quite expensive.

Consider the memory allocation shown in Figure 6.11. If we use this simple algorithm, we must move processes P3 and P4, for a total of 600K moved. In this situation, we could simply move process Pq above process P3, moving only 400K, or move process P3 below process P4, moving only 200K. Note that, in this last instance, our one large hole of available memory is not at the end of memory, but rather is in the middle. Also notice that, if the queue contained only one process that wanted 450K, we could satisfy that ***particular*** request by moving process P2 somewhere else (such as below process P4).

Although ths solution does not create a single large hole, it does create a hole big enough to satisfy the immediate request. Selecting an optimal compaction strategy is quite difficult.

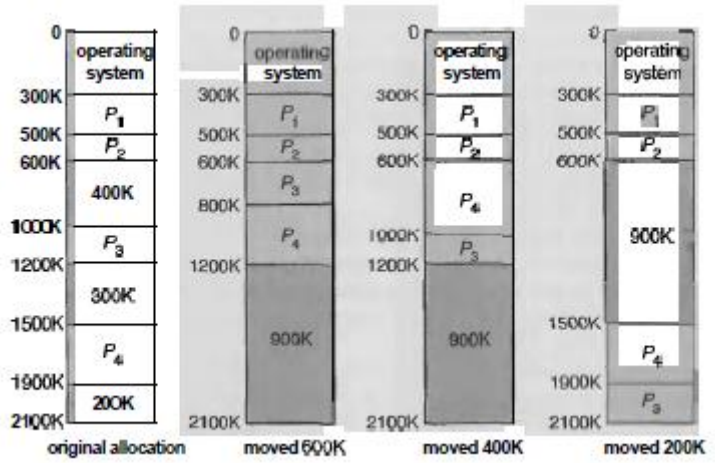


Figure 6.11 Comparison of some different ways to compact memory.

Swapping can also be combined with compaction. A process can be rolled out of main memory to a backing store and rolled in again later. When the process is rolled out, its memory is released, and perhaps is reused for another process. When the process is to be rolled back in, several problems may arise. If static relocation is used, the process must be rolled into the exact same memory locations that it occupied previously. This restriction may require that other processes be rolled out to free that memory.

If dynamic relocation (such as with base and limit registers) is used, then a process can be rolled into a different location. In this case, we find a free block, compacting if necessary, and roll in the process.
One approach to compaction is to roll out those processes to be moved, and to roll them into different memory locations. If swapping or roll-in, rollout is already a part of the system, the additional code for compaction may be minimal.

6.5 Paging

Another possible solution to the external fragmentation problem is to permit the logical address space of a process to be noncontiguous, thus allowing a process to be allocated physical memory wherever the latter is available. One way of implementing this solution is through the use of a *paging* scheme.
Paging avoids the considerable problem of fitting the varying-sized memory chunks onto the backing store, from which most of the previous memory-management schemes suffered. When some code fragments or data residing in main memory need to be swapped out, space must be found on the backing store.

The fragmentation problems discussed in connection with main memory are also prevalent with backing store, except that access is much slower, so compaction is impossible. Because of its advantages over the previous methods, paging in its various forms is commonly used in many operating systems.

6.5.1 Basic Method

Physical memory is broken into fixed-sized blocks called *frames.* Logical memory is also broken into blocks of the same size called *pages.* When a process is to be executed, its pages are loaded into any available memory frames from the backing store. The backing store is divided into fixed-sized blocks that are of the same size as the memory frames.

The hardware support for paging is illustrated in Figure 6.12. 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 a *page table.* The page table contains the base address of each page in physical memory. This base address is combined with the page offset to define the physical memory address that is sent to the memory unit. The paging model of memory is shown in Figure 6.13.

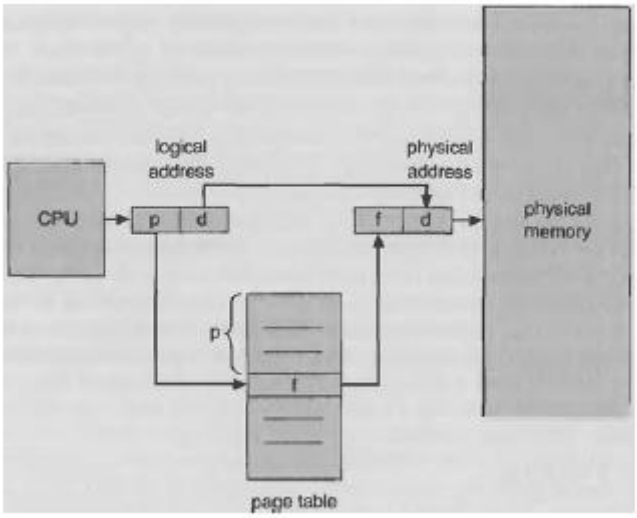
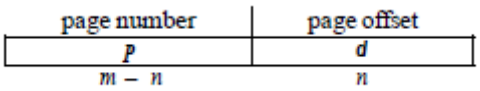


Figure 6.12 Paging hardware.

The page size (like the frame size) is defined by the hardware. The size of a page is typically a power of 2 varying between 512 bytes and 8192 bytes per page, depending on the computer architecture. The selection of a power of 2 as a page size makes the translation of a logical address into a page number and page offset particularly easy. If the size of logical address space is 2^m, and a page size is 2" addressing units (bytes or words), then the high-order $m - n$ bits of a logical address designate the page number, and the n low-order bits designate the page offset. Thus, the logical address is as follows:



where p is an index into the page table and d is the displacement within the page. For a concrete, although minuscule, example, consider the memory of Figure 6.14 (page 260). Using a page size of 4 bytes and a physical memory of 32 bytes (8 pages), we show an example of how the user's view of memory can be mapped into physical memory. Logical address 0 is page 0, offset 0.

Indexing into the page table, we find that page 0 is in frame 5. Thus, logical address 0 maps to physical address 20 (= (5 x 4) + 0). Logical address 3 (page 0, offset **3)** maps to physical address 23 (= (5 x 4) + 3). Logical address 4 is page 1, offset 0; according to the page table, page 1 is mapped to frame 6.

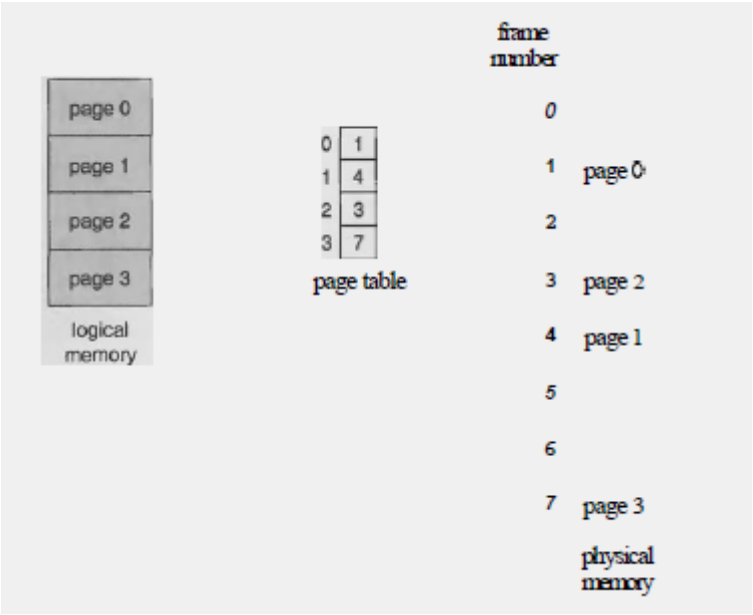


Figure 6.13 Paging model of logical and physical memory.

Thus, address 4 maps to physical address 24 (= (6 x 4) + 0). Logical address 13 maps to physical address 9.

Notice that paging itself is a form of dynamic relocation. Every logical address is bound by the paging hardware to some physical address. The observant reader will have realized that paging is similar to using a table of base (relocation) registers, one for each frame of memory.

When we use a paging scheme, we have no external fragmentation: Any free frame can be allocated to a process that needs it. However, we may have some internal fragmentation. Notice that frames are allocated as units. If the memory requirements of a process do not happen to fall on page boundaries, the ***last*** frame allocated may not be completely full. For example, if pages are 2048 bytes, a process of 72,766 bytes would need 35 pages plus 1086 bytes.

It would be allocated 36 frames, resulting in an internal fragmentation of 2048 -1086 = 962 bytes. In the worst case, a process would need *n* pages plus one byte. It would be allocated *n* + 1 frames, resulting in an internal fragmentation of almost an entire frame.

If process size is independent of page size, we expect internal fragmentation to average one-half page per process. This consideration suggests that small page sizes are desirable. However, there is quite a bit of overhead involved in each page-table entry, and this overhead is reduced as the size of the pages increases.

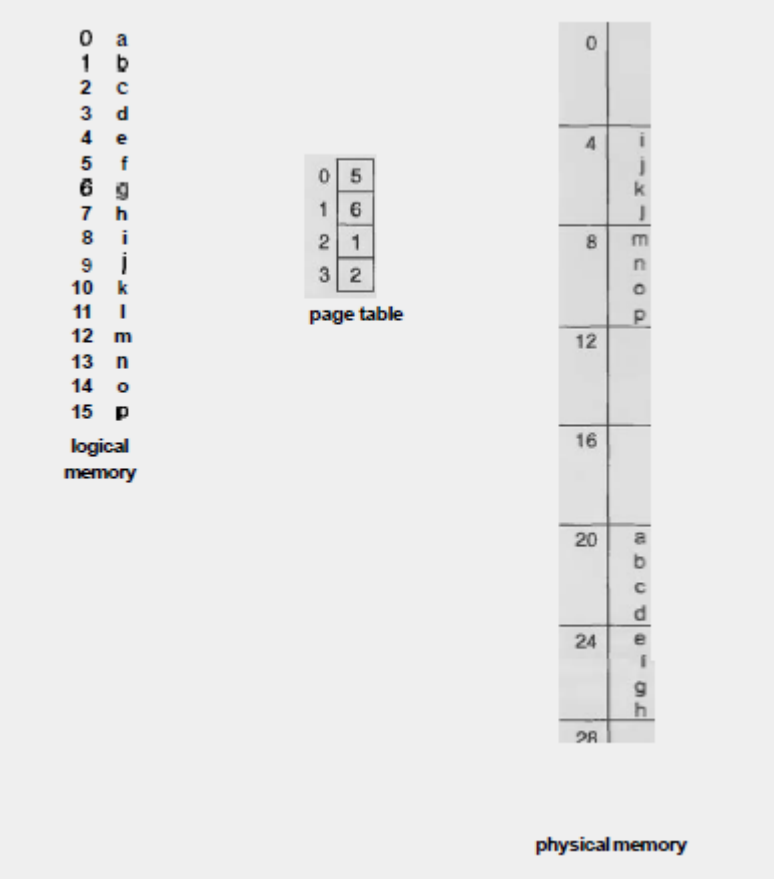


Figure 6.14 Paging example for a 32-byte memory with 4-byte pages.

Also, disk 1/0 is more efficient when the number of data being transferred is larger (Chapter 13). Generally, page sizes have grown over time as processes, data sets, and main memory have become larger. Today, pages typically are either 2 or 4 kilobytes.

When a process arrives in the system to be executed, its size, expressed in pages, is examined. Each page of the process needs one frame. Thus, if the process requires *n* pages, there must be at least *n* frames available in memory. If there are *n* frames available, they are allocated to this arriving process.

The first page of the process is loaded into one of the allocated frames, and the frame number is put in the page table for this process. The next page is loaded into another frame, and its frame number is put into the page table, and so on (Figure 6.15).

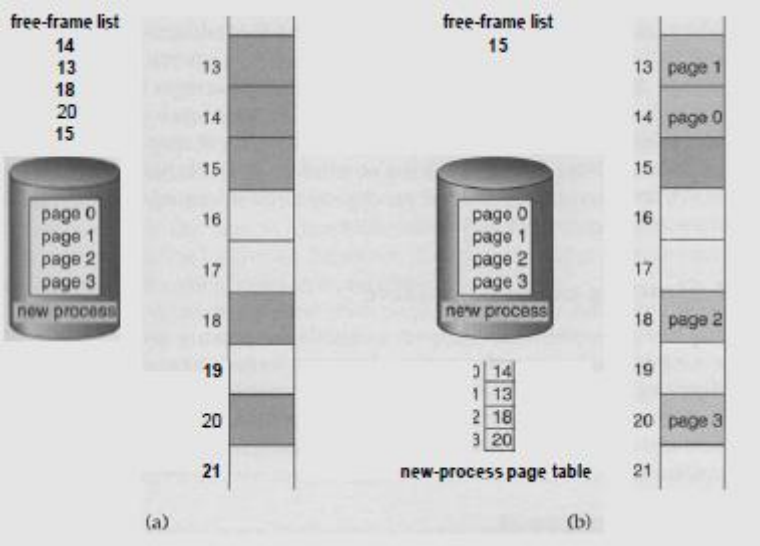


Figure 6.15 Free frames. (a) Before allocation, and (b) after allocation.

An important aspect of paging is the clear separation between the user's view of memory and the actual physical memory. The user program views that memory as one single contiguous space, containing only this one program.

In fact, the user program is scattered throughout physical memory, which also holds other programs. The difference between the user's view of memory and the actual physical memory is reconciled by the address-translation hardware.

The logical addresses are translated into physical addresses. This mapping is hidden from the user and is controlled by the operating system. Notice that the user process by definition is unable to access memory it does not own. It has no way of addressing memory outside of its page table, and the table includes only those pages that the process owns.

Since the operating system is managing physical memory, it must be aware of the allocation details of physical memory: which frames are allocated, which frames are available, how many total frames there are, and so on. This information is generally kept in a data structure called a *frame table.* The frame table has one entry for each physical page frame, indicating whether the latter is free or allocated and, if it is allocated, to which page of which process or processes.

In addition, the operating system must be aware that user processes operate in user space, and all logical addresses must be mapped to produce physical addresses. If a user makes a system call (to do I/O, for example) and provides an address as a parameter (a buffer, for instance), that address must be mapped to produce the correct physical address.

The operating system maintains a copy of the page table for each process, just as it maintains a copy of the instruction counter and register contents. This copy is used to translate logical addresses to physical addresses whenever the operating system must map a logical address to a physical address manually. It is also used by the CPU dispatcher to define the hardware page table when a process is to be allocated the CPU. Paging therefore increases the context-switch time.

6.5.2 Structure of the Page Table

Each operating system has its own methods for storing page tables. Most allocate a page table for each process. A pointer to the page table is stored with the other register values (like the instruction counter) in the process control block. When the dispatcher is told to start a process, it must reload the user registers and define the correct hardware page-table values from the stored user page table.

6.5.2.1 Hardware Support

The hardware implementation of the page table can be done in a number of different ways. In the simplest case, the page table is implemented as a set of dedicated registers. These registers should be built with very high-speed logic to make the paging address translation efficient.

Every access to memory must go through the paging map, so efficiency is a major consideration. The CPU dispatcher reloads these registers, just as it reloads the other registers.

Instructions to load or modify the page-table registers are, of course, privileged, so that only the operating system can change the memory map. The DEC PDP-I1 is an example of such an architecture. The address consists of 16 bits, and the page size is **8K.** The page table, thus, consists of eight entries that are kept in fast registers.

The use of registers for the page table is satisfactory if the page table is reasonably small (for example, 256 entries). Most contemporary computers, however, allow the page table to be very large (for example, 1 million entries). For these machines, the use of fast registers to implement the page table is not feasible. Rather, the page table is kept in main memory, and a page-table base register ***(PTBR)*** points to the page table. Changing page tables requires changing only this one register, substantially reducing context-switch time.

The problem with this approach is the time required to access a user memory location. If we want to access location i, we must first index into the page table, using the value in the PTBR offset by the page number for ***i.*** This task requires a memory access. It provides us with the frame number, which is combined with the page offset to produce the actual address.

We can then access the desired place in memory. With this scheme, ***two*** memory accesses are needed to access a byte (one for the page-table entry, one for the byte). Thus, memory access is slowed by a factor of 2. This delay would be intolerable under most circumstances.

We might as well resort to swapping! The standard solution to this problem is to use a special, small, fast-lookup hardware cache, variously called *associative registers* or *translation look-aside buffers* (TLBs). A set of associative registers is built of especially high-speed memory.

Each register consists of two parts: a key and a value. When the associative registers are presented with an item, it is compared with all keys simultaneously. If the item is found, the corresponding value field is output.

The search is fast; the hardware, however, is expensive. Typically, the number of entries in a TLB varies between 8 and 2048. Associative registers are used with page tables in the following way. The associative registers contain only a few of the page-table entries.

When a logical address is generated by the CPU, its page number is presented to a set of associative registers that contain page numbers and their corresponding frame numbers. If the page number is found in the associative registers, its frame number is immediately available and is used to access memory.

The whole task may take less than 10 percent longer than it would were an unmapped memory reference used. If the page number is not in the associative registers, a memory reference to the page table must be made. When the frame number is obtained, we can use it to access memory (Figure 6.16). In addition, we add the page number and frame number to the associative registers, so that they will be found quickly on the next reference If the TLB is already full of entries, the operating system must select one for replacement. Unfortunately, every time a new page table is selected (for instance, each context switch), the TLB must be *flushed* (erased) to ensure that the next executing process does not use the wrong translation information.

Otherwise, there could be old entries in the TLB that contain valid virtual addresses but have incorrect or invalid physical addresses left over from the previous process.

The percentage of times that a page number is found in the associative registers is called the *hit ratio.* An 80-percent hit ratio means that we find the desired page number in the associative registers 80 percent of the time. If it takes 20 nanoseconds to search the associative registers, and 100 nanoseconds to access memory, then a mapped memory access takes 120 nanoseconds when the page number is in the associative registers. If we fail to find the page number in the associative registers (20 nanoseconds), then we must first access memory for the page table and frame number (100 nanoseconds), and then access the desired byte in memory (100 nanoseconds), for a total of 220

nanoseconds. To find the *effective memory-access time*, we must weigh each case by its probability:

effective access time = 0.80 x 120 + 0.20 x 220

= 140 nanoseconds

In this example,

We suffer a 40-percent slowdown in memory access time (from 100 to 140 nanoseconds).

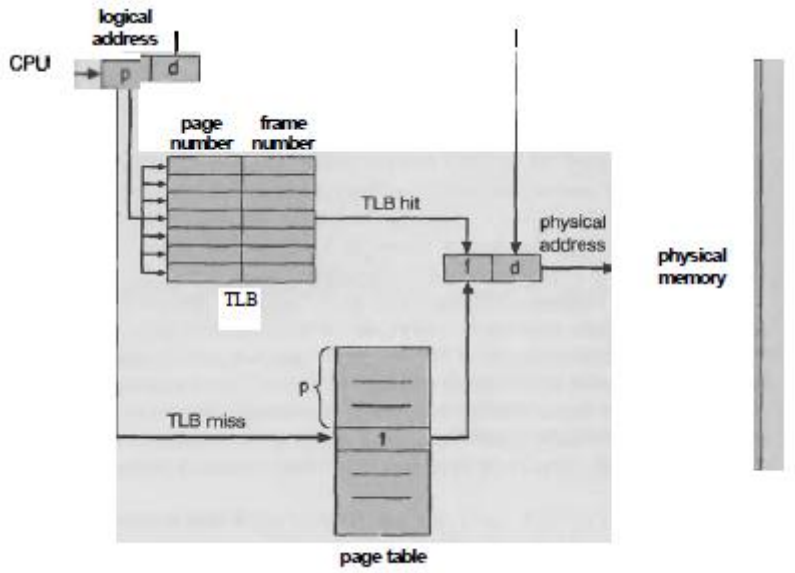


Figure 6.16 Paging hardware with TLB.

For a 98-percent hit ratio, we have effective access time

= 0.98 x 120 + 0.02 x 220

= 122 nanoseconds.

This increased hit rate produces only a 22-percent slowdown in memory access time. The hit ratio is clearly related to the number of associative registers. With the number of associative registers ranging between 26 and 512, a hit ratio of 80 to 98 percent can be obtained.

The Motorola 68030 processor (used in Apple Macintosh systems) has a 22-entry TLB. The Intel 80486 CPU (found in some PCs) has 32 registers, and claims a 98-percent hit ratio**. 6.5.2.2 Protection** Memory protection in a paged environment is accomplished by protection bits that are associated with each frame.

Normally, these bits are kept in the page table. One bit can define a page to be read and write or read-only. Every reference to memory goes through the page table to find the correct frame number. At the same time that the physical address is being computed, the protection bits can be checked to verify that no writes are being made to a read only page.

An attempt to write to a read-only page causes a hardware trap to the operating system (memory-protection violation). This approach to protection can be expanded easily to provide a finer level of protection.
We can create hardware to provide read-only, read-write, or execute-only protection. Or, by providing separate protection bits for each kind of access, any combination of these accesses can be allowed, and illegal attempts will be trapped to the operating system.

One more bit is generally attached to each entry in the page table: a validinvalid bit. When this bit is set to "valid," this value indicates that the associated page is in the process's logical address space, and is thus a legal (valid) page. If the bit is set to "invalid," this value indicates that the page is not in the process's logical address space. Illegal addresses are trapped by using the valid-invalid bit.

The operating system sets this bit for each page to allow or disallow accesses to that page. For example, in a system with a 14-bit address space (0 to 16,3831, we may have a program that should use only addresses 0 to 10,468. Given a page size of 2K, we get the situation shown in Figure 6.17. Addresses in pages 0,1,2,3,4 and 5 are mapped normally through the page table. Any attempt to generate an address in pages 6 or 7, however, finds that the valid-invalid bit is

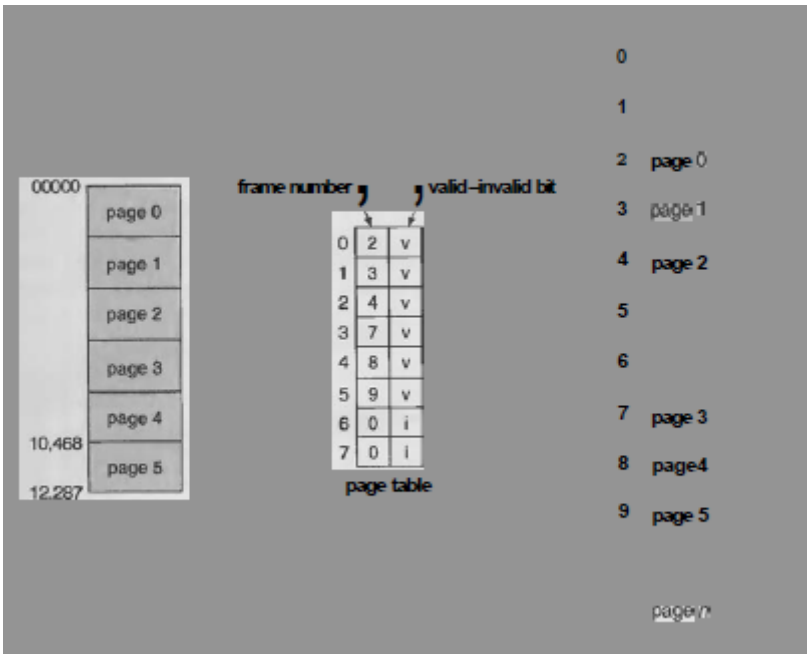


Figure 6.17 Valid (v) or invalid (i) bit in a page table.

set to invalid, and the computer will trap to the operating system (invalid page reference). Notice that as the program extends only to address 10,468; any reference beyond that address is illegal. However, references to page 5 are classified as valid, so accesses to addresses up to 12,287 are valid. Only addresses from 12,288 to 16,383 are invalid. This problem is a result of the 2K page size and reflects the internal fragmentation of paging.

Rarely does a process use all of its address range. In fact, many processes use only a small fraction of the address space available to them. It would be wasteful in these cases to create a page table with entries for every page in the address range. Most of this table would be unused, but would take up valuable memory space.

Some systems provide hardware, in the form of a ***page-table length register*** *(PTLR),* to indicate the size of the page table. This value is checked against every logical address to verify that the address is in the valid range for the process. Failure of this test causes an error trap to the operating system.

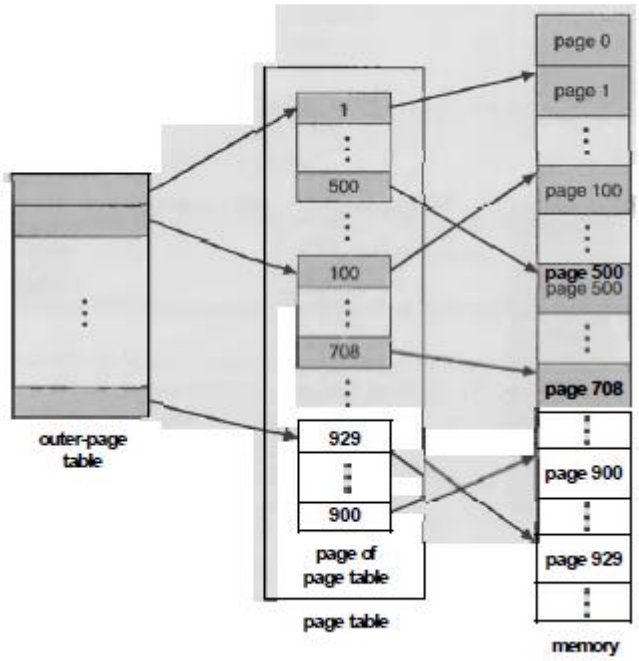


Figure 6.18 A two-level page-table scheme.

6.5.3 Multilevel Paging

Most modern computer systems support a very large logical address space (232 to 264). In such an environment the page table itself becomes excessively large. For example, consider a system with a 32-bit logical address space. If the page size in such a system is 4K bytes (212), then a page table may consist of up to 1 million entries (232/212). Because each entry consists of 4 bytes, each process may need up to 4 megabytes of physical address space for the page table alone.

Clearly, we would not want to allocate the page table contiguously in main memory. One simple solution to this is to divide the page table into smaller pieces. There are several different ways to accomplish this. One way is to use a two-level paging scheme, in which the page table itself is also paged (Figure 6.18). To illustrate this, let us return to our 32-bit machine example, with a page size of 4K bytes. A logical address is divided into a page number consisting of 20 bits, and a page offset consisting of 12 bits. Because we page

the page table, the page number is further divided into a 10-bit page number, and a 10-bit page offset. Thus, a logical address is as follows:

page number	page	offset
10	10	12

where *p1* is an index into the outer page table, and p2 is the displacement within the page of the outer page table. The address-translation scheme for this architecture is shown in Figure 6.19. The VAX architecture supports twolevel paging. The VAX is a 32-bit machine with page size of 512 bytes.

The logical address space of a process is divided into four equal sections, each of which consists of 230 bytes. Each section represents a different part of the logical address space of a process. The first 2 high-order bits of the logical address designate the appropriate section.

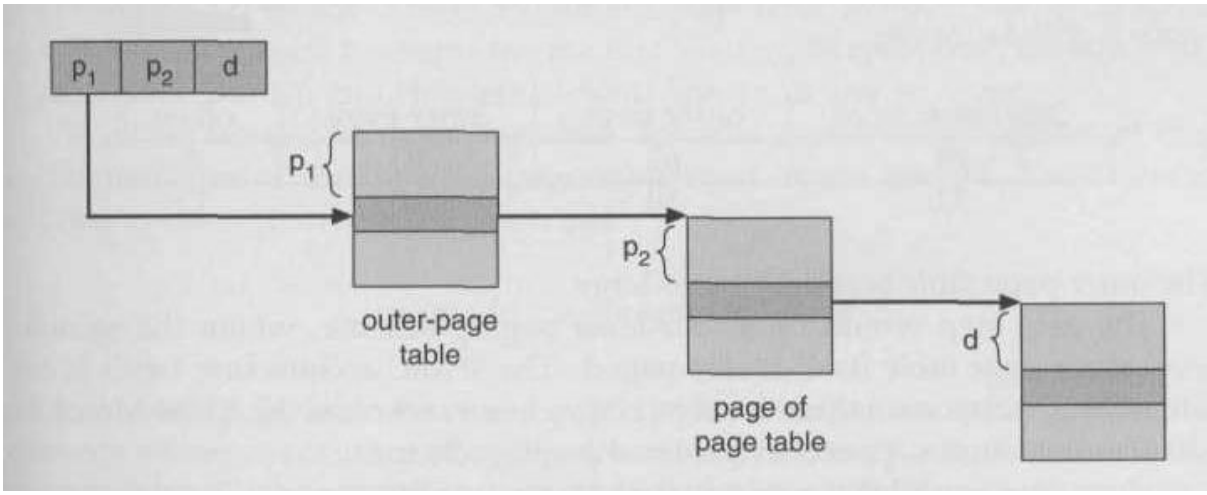


Figure 6.19 Address translation for a two-level 32-bit paging architecture.

The next 21 bits represent the logical page number of that section, and the last 9 bits represent an offset in the desired page. By partitioning the page table in this manner, the operating system can leave partitions unused until a process needs them. An address on the **VAX** architecture is as follows:

section	vaee	offset
s	*p*	*d*
2	21	9

where s designates the section number, p is an index into the page table, and d is the displacement within the page. The size of a one-level page table for a **VAX** process using one segment still is 2^{21} bits * 4 bytes per entry = 8 megabytes. To further reduce main memory use, the **VAX** pages the user-process page tables.

For a system with a 64-bit logical address space, a two-level paging scheme is no longer appropriate. To illustrate this point, let us suppose that the page size in such a system is 4K bytes (2^{12}). In this case, the page table will consist of up to 2^{52} entries. If we use a two-level paging scheme, then the inner page tables could conveniently be 1 page long, or contain 21° four-byte entries.

The addresses would look like:

outer page	inner page	offset
*p*1	*p*2	d
42	10	12

The outer page table will consist of 2^{42} entries, or 2^{44} bytes. The obvious method to avoid such a large table is to divide the outer page table into smaller pieces. This approach is also used on some 32-bit processors for added flexibility and efficiency.

This can be accomplished in various ways. We can page the outer page table, giving us a three-level paging scheme. Suppose that the outer page table is made up of standard-size pages (2^{10} entries, or 2^{12} bytes); a 64-bit address space is still daunting:

The outer page table is still 2^{34} bytes large.

2nd outer page	outer page	inner page	offset
*p*1	*p*2	*p*3	d
32	10	10	12

The next step would be a four-level paging scheme, where the secondlevel outer page table itself is also paged. The **SPARC** architecture (with

32-bit addressing) supports a three-level paging scheme, whereas the 32-bit Motorola 68030 architecture supports a four-level paging scheme.

How does multilevel paging affect system performance? Given that each level is stored as a separate table in memory, converting a logical address to a physical one may take four memory accesses. We have now quintupled the amount of time needed for one memory access! Caching again pays dividends, however, and performance remains reasonable. Given a cache hit rate of 98 percent, we have effective access time = 0.98 x 120 + 0.02 x 520 = 128 nanoseconds.Thus, even with the extra levels of table lookup, we have only a 28-percent slowdown in memory access time.

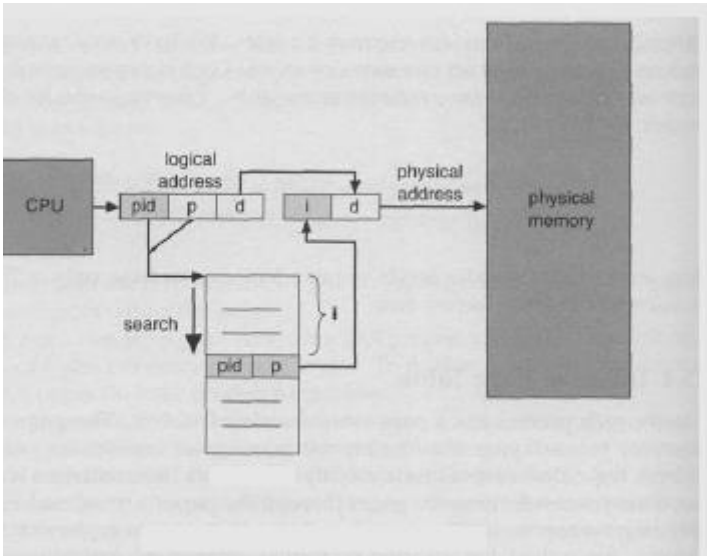
6.5.4 Inverted Page Table

Usually, each process has a page table associated with it. The page table has one entry for each page that the process is using (or one slot for each virtual address, regardless of the latter's validity). This table representation is a natural one, since processes reference pages through the pages' virtual addresses.

The operating system must then translate this reference into a physical memory address. Since the table is sorted by virtual address, the operating system is able to calculate where in the table the associated physical-address entry is, and to use that value directly. One of the drawbacks of this scheme is that each page table may consist of millions of entries. These tables may consume large amounts of physical memory, which is required just to keep track of how the other physical memory is being used.

To solve this problem, we can use an ***inverted page table.*** An inverted page table has one entry for each real page (frame) of memory. Each entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page. Thus, there is only one page table in the system, and it has only one entry for each page of physical memory. Figure 6.20 shows the operation of an inverted page table. Compare it to Figure 6.12, which depicts a standard page table in operation. Examples of systems using such a scheme are the IBM System/38 computer, the IBM RISC System 6000, IBM RT, and Hewlett-Packard Spectrum workstations.

To illustrate this scheme, we shall describe a simplified version of the implementation of the inverted page table used in the IBM RT. Each virtual address in the system consists of a triple <process-id, page-number, offset>.
Each inverted page-table entry is a pair <process-id, page-number>. When a memory reference occurs, part of the virtual address, consisting of <processid, pagenumber>, is presented to the memory subsystem. The inverted page table is then searched for a match. If a match is found - say, at entry ***i*** - then the



physical address ***<i,*** offset> is generated. If no match is found, then an illegal address access has been attempted. Although this scheme decreases the amount of memory needed to store each page table, it increases the amount of time needed to search the table when a page reference occurs. Because the table is sorted by a physical address, but lookups occur on virtual addresses, the whole table might need to be searched for a match.

This search would take far too long. To alleviate this problem, we use a hash table to limit the search to one - or at most a few - page-table entries. E procedc Y Of course, each access to the hash table adds a memory reference to thire, so one virtual-memory reference requires at least two real-memory reads:One for the hash-table entry and one for the page table performance, we use associative memory registers to hold recently located entries. These registers are searched first, before the hash table is consulted.

6.5.5 Shared Pages

Another advantage of paging is the possibility of *sharing* common code. This consideration is particularly important in a time-sharing environment. Consider a system that supports 40 users, each of whom executes a text editor.

If the text editor consists of 150K of code and 50K of data space, we would need 8000K to support the 40 users. If the code is *reentrant,* however, it can be shared, as shown in Figure 8.21. Here we see a three-page editor (each page of size 50K;

The large page size is used to simplify the figure) being shared among three processes. Each process has its own data page.

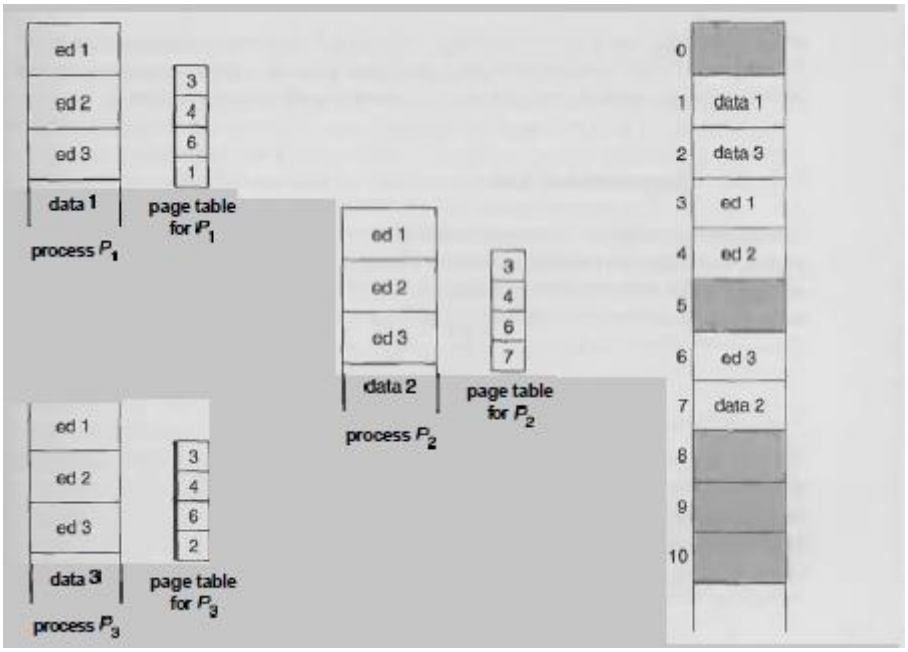


Figure 6.21 Sharing of code in a paging environment.

Reentrant code (also called pure code) is non-self-modifying code. If the code is reentrant, then it never changes during execution. Thus, two or more processes can execute the same code at the same time. Each process has its own copy of registers and data storage to hold the data for the process' execution.

The data for two different processes will, of course, vary for each process. Only one copy of the editor needs to be kept in physical memory. Each user's page table maps onto the same physical copy of the editor, but data pages

are mapped onto different frames. Thus, to support 40 users, we need only one copy of the editor (150K1, plus 40 copies of the 50K of data space per user.

The total space required is now 2150K, instead of 8000K - a significant savings. Other heavily used programs also can be shared: compilers, window systems, database systems, and so on.

To be sharable, the code must be reentrant. The read-only nature of shared code should not be left to the correctness of the code; the operating system should enforce this property. This sharing of memory between processes on a system is similar to the way threads share the address space of a task.

Systems that use inverted page tables have difficulty implementing shared memory. Shared memory is usually implemented as two virtual addresses that are mapped to one physical address. This standard method cannot be used, however, as there is only one virtual page entry for every physical page, so one physical page cannot have the two (or more) shared virtual addresses.

6.6 Segmentation

An important aspect of memory management that became unavoidable with paging is the separation of the user's view of memory and the actual physical memory. The user's view of memory is not the same as the actual physical memory.

The user's view is mapped onto physical memory. The mapping allows differentiation between logical memory and physical memory. 6**.6.1 Basic Method** What is the user's view of memory? Does the user think of memory as a linear array of bytes, some containing instructions and others containing data, or is there some other preferred memory view? There is general agreement that the user or programmer of a system does not think of memory as a linear array of bytes.

Rather, the user prefers to view memory as a collection of variable-sized segments, with no necessary ordering among segments (Figure 6.22).

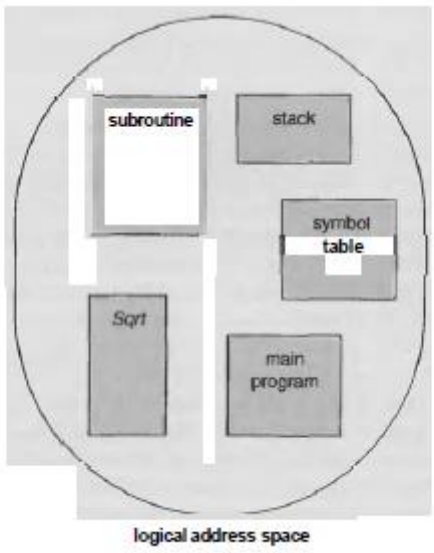


Figure 6.22 User's view of a program.

Consider how you think of a program when you are writing it. You think of it as a main program with a set of subroutines, procedures, functions, or modules. There may also be various data structures: tables, arrays, stacks, variables, and so on. Each of these modules or data elements is referred to by name. You talk about "the symbol table," "function Sqrt," "the main program," without caring what addresses in memory these elements occupy. You are not concerned with whether the symbol table is stored before or after the Sqrt function.

Each of these segments is of variable length; the length is intrinsically defined by the purpose of the segment in the program. Elements within a segment are identified by their offset from the beginning of the segment: the first statement of the program, the seventeenth entry in the symbol table, the fifth instruction of the Sqrt function, and so on.

Segmentation is a memory-management scheme that supports this user view of memory. A logical address space is a collection of segments. Each segment has a name and a length. The addresses specify both the segment name and the offset within the segment. The user therefore specifies each address by two quantities: a segment name and an offset. (Contrast this scheme with the paging scheme, where the user specified only a single address, which was partitioned by the hardware into a page number and an offset, all invisible to the programmer.)

For simplicity of implementation, segments are numbered and are referred to by a segment number, rather than by a segment name. Thus, a logical address consists of a two tuple:

<segment-number, offset>.

Normally the user program is compiled, and the compiler automatically constructs segments reflecting the input program. A Pascal compiler might create separate segments for **(1)** the global variables; (2) the procedure call stack, to store parameters and return addresses; **(3)** the code portion of each procedure or function; and (4) the local variables of each procedure and function.

A FORTRAN compiler might create a separate segment for each common block. Arrays might be assigned separate segments. The loader would take all these segments and assign them segment numbers.

6.6.2 Hardware

Although the user can now refer to objects in the program by a two-dimensional address, the actual physical memory is still, of course, a one-dimensional sequence of bytes. Thus, we must define an implementation to map two dimensional user-defined addresses into one-dimensional physical addresses.

This mapping is affected by a segment table. Each entry of the segment table as a segment base and a segment ***limit.*** The segment base contains the starting

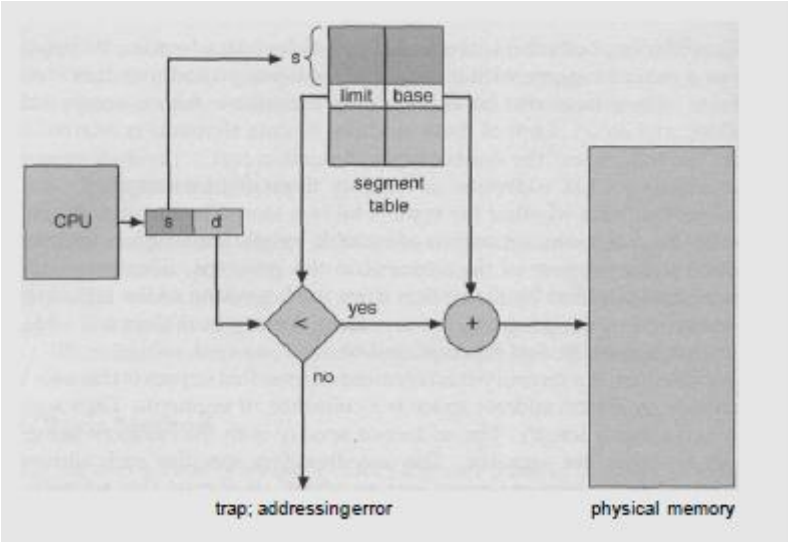


Figure 6.**23** Segmentation hardware.

physical address where the segment resides in memory, whereas the segment limit specifies the length of the segment. The use of a segment table is illustrated in Figure 8.23. A logical address consists of two parts: a segment number, s, and an offset into that segment, d.

The segment number is used as an index into the segment table. The offset d of the logical address must be between 0 and the segment limit. If it is not, we trap to the operating system (logical addressing attempt beyond end of segment). If this offset is legal, it is added to the segment base to produce the address in physical memory of the desired byte. The segment table is thus essentially an array of base-limit register pairs.

As an example, consider the situation shown in Figure 6.24. We have five segments numbered from 0 through 4. The segments are stored in physical memory as shown. The segment table has a separate entry for each segment, giving the beginning address of the segment in physical memory (the base) and the length of that segment (the limit).

For example, segment 2 is 400 bytes long, and begins at location 4300. Thus, a reference to byte 53 of segment 2 is mapped onto location 4300 + 53 = 4353. A reference to segment 3, byte 852, is mapped to 3200 (the base of segment 3) + 852 = 4052. A reference to byte 1222 of segment 0 would result in a trap to the operating system, as this segment is only 1000 bytes long.

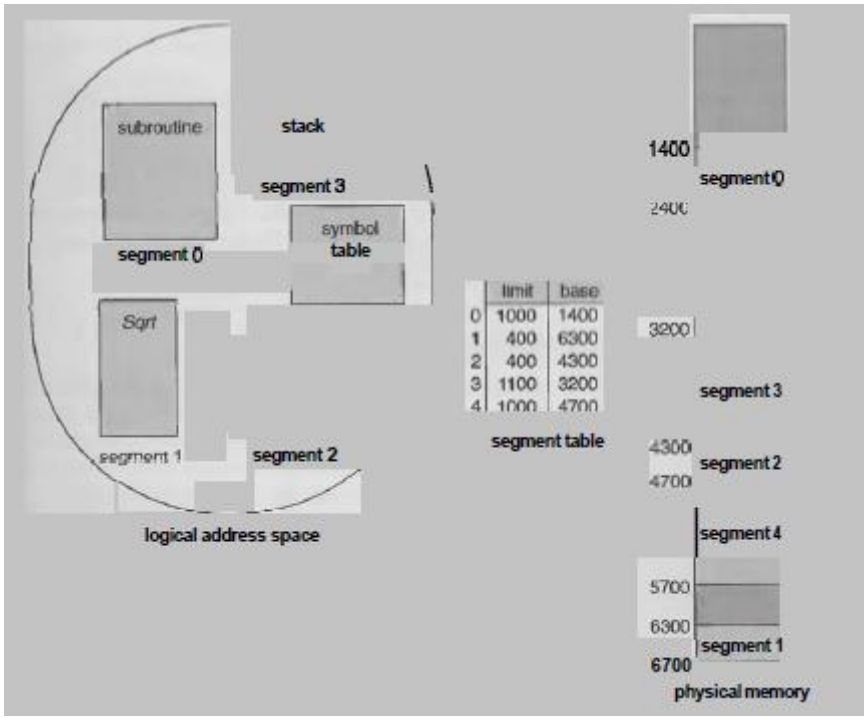


Figure 6.24 Example of segmentation.

6.6.3 Implementation of Segment Tables

Segmentation is closely related to the partition models of memory management presented earlier, the main difference being that one program may consist of several segments. Segmentation is a more complex concept, however, which is why we are describing it after discussing paging. Like the page table, the segment table can be put either in fast registers or in memory.

A segment table kept in registers can be referenced quickly; the addition to the base and comparison with the limit can be done simultaneously to save time.

In the case where a program may consist of a large number of segments, it is not feasible to keep the segment table in registers, so we must keep it in memory. A segment-table base register (STBR) points to the segment table.

Also, because the number of segments used by a program may vary widely, a segment-table length register (STLR) is used.

For a logical address ***(s,d),*** we first check that the segment number s is legal (that is, s < STLR). Then, we add the segment number to the STBR, resulting in the address (STBR + s) in memory of the segment-table entry.

This entry is read from memory and we proceed as before: Check the offset against the segment length and compute the physical address of the desired byte as the sum of the segment base and offset.

As occurs with paging, this mapping requires two memory references per logical address, effectively slowing the computer system by a factor of 2, unless something is done. The normal solution is to use a set of associative registers to hold the most recently used segment-table entries.

Again, a small set of associative registers can generally reduce the time required for memory accesses to no more than 10 or 15 percent slower than unmapped memory accesses.

6.6.4 Protection and Sharing

A particular advantage of segmentation is the association of protection with the segments. Because the segments represent a semantically defined portion of the program, it is likely that all entries in the segment will be used the same way.

Hence, we have some segments that are instructions, whereas other segments are data. In a modern architecture, instructions are non-self-modifying, so instruction segments can be defined as read-only or execute-only.

The memorymapping hardware will check the protection bits associated with each segmenttable entry to prevent illegal accesses to memory, such as attempts to write into a read-only segment, or to use an execute-only segment as data. By placing an array in its own segment, the memory-management hardware will automatically check that array indexes are legal and do not stray outside the array boundaries. Thus, many common program errors will be detected by the hardware before they can cause serious damage.

Another advantage of segmentation involves the ***sharing*** of code or data. Each process has a segment table associated with it, which the dispatcher uses to define the hardware segment table when this process is given the **CPU.**

Segments are shared when entries in the segment tables of two different processes point to the same physical locations. (Figure 6.25). The sharing occurs at the segment level. Thus, any information can be shared if it is defined to be a segment. Several segments can be shared, so a program composed of several segments can be shared.

For example, consider the use of a text editor in a time-sharing system. A complete editor might be quite large, composed of many segments. These segments can be shared among all users, limiting the physical memory needed to support editing tasks. Rather than ***n*** copies of the editor, we need only one copy.

For each user, we still need separate, unique segments to store local variables. These segments, of course, would not be shared. It is also possible to share only parts of programs. For example, common subroutine packages can be shared among many users if they are defined as sharable, read-only segments.

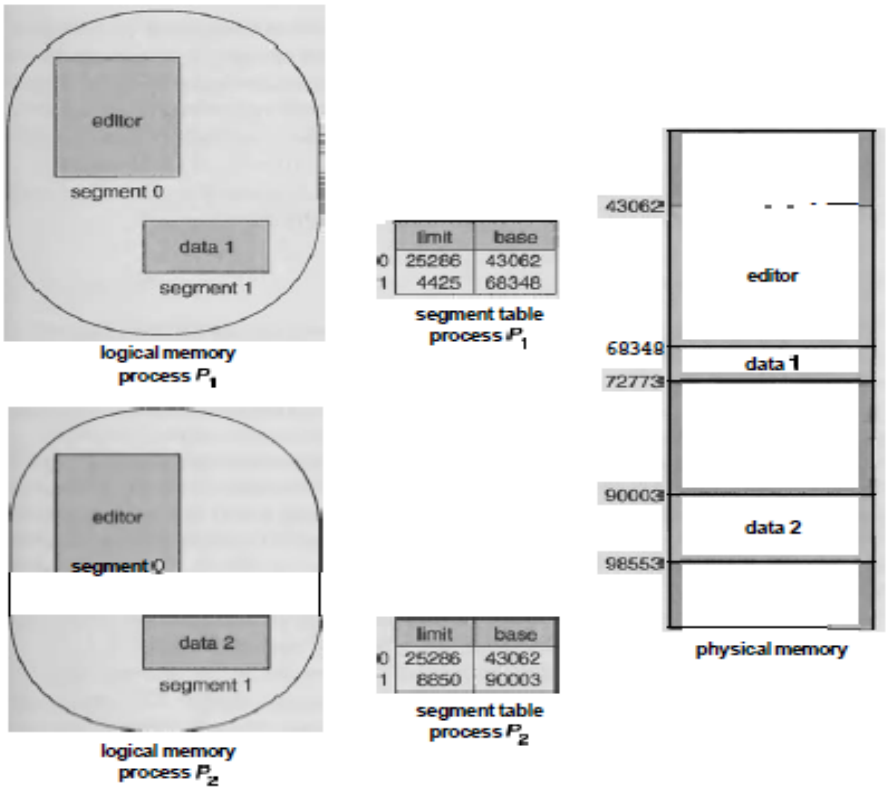


Figure 6.25 Sharing of segments in a segmented memory system.

Two FORTRAN programs, for instance, may use the same ***Sqrt*** subroutine, but only one physical copy of the ***Sqrt*** routine would be needed.

Although this sharing appears simple, there are subtle considerations. Code segments typically contain references to themselves. For example, a conditional jump normally has a transfer address.

The transfer address is a segment number and offset. The segment number of the transfer address will be the segment number of the code segment. If we try to share this segment, all sharing processes must define the shared code segment to have the same segment number.

For instance, if we want to share the ***Sqrt*** routine, and one process wants to make it segment 4 and another wants to make it segment 17, how should the ***Sqrt*** routine refer to itself? Because there is only one physical copy of ***Sqrt,*** it must refer to itself in the same way for both users - it must have a unique segment number. As the number of users sharing the segment increases, so does the difficulty of finding an acceptable segment number.

Read-only data segments that contain no physical pointers may be shared as different segment numbers, as may code segments that refer to themselves not directly, but rather only indirectly.

For example, conditional branches that specify the branch address as an offset from the current program counter or relative to a register containing the current segment number would allow code to avoid direct reference to the current segment number.

6.6.5 Fragmentation

The long-term scheduler must find and allocate memory for all the segments of a user program. This situation is similar to paging except that the segments are of ***variable*** length; pages are all the same size. Thus, as with the variable- ized partition scheme, memory allocation is a dynamic storage-allocation problem, usually solved with a best-fit or first-fit algorithm.

Segmentation may then cause external fragmentation, when all blocks of free memory are too small to accommodate a segment. In this case, the process may simply have to wait until more memory (or at least a larger hole) becomes available, or compaction may be used to create a larger hole.

Because segmentation is by its nature a dynamic relocation algorithm, we can compact memory whenever we want. [f the CPU scheduler must wait for one process, due to a memory-allocation problem, it may (or may not) skip through the CPU queue looking for a smaller, lower-priority process to run.

How serious a problem is external fragmentation for a segmentation scheme? Would long-term scheduling with compaction help? The answers to these questions depend mainly on the average segment size. At one extreme, we could define each process to be one segment. This approach reduces to the variable-sized partition scheme. At the other extreme, every byte could be put in its own segment and relocated separately.

This arrangement eliminates external fragmentation altogether; however, every byte would need a base register for its relocation, doubling memory use! Of course, the next logical step - fixed-sized, small segments - is paging. Generally, if the average segment size is small, external fragmentation will also be small. (By analogy, consider putting suitcases in the trunk of a car; they never quite seem to fit. However, if you open the suitcases and put the individual items in the trunk, everything fits.) Because the individual segments are smaller than the overall process, they are more likely to fit in the available memory blocks.

6.7 Segmentation with Paging

Both paging and segmentation have their advantages and disadvantages. In fact, of the two most popular microprocessors now being used, the Motorola 68000 line is designed based on a flat address space, whereas the Intel 80x86 family is based on segmentation. Both are merging memory models toward a mixture of paging and segmentation. It is possible to combine these two schemes to improve on each.

This combination is best illustrated by two different architectures - the innovative but not widely used MULTICS system and the Intel 386.

6.7.1 MULTICS

In the MULTICS system, a logical address is formed from an l&bit segment number and a 16-bit offset. Although this scheme creates a 34-bit address space, the segment-table overhead is tolerable; we need only as many segment-table entries as we have segments, as there need not be empty segment-table entries.

However, with segments of 64K words, each of which consists of 36 bits, the average segment size could be large and external fragmentation could be a problem.

Even if external fragmentation is not a problem, the search time to allocate a segment, using first-fit or best-fit, could be long. Thus, we may waste memory due to external fragmentation, or waste time due to lengthy searches, or both.

The solution adopted was to *page* the *segments.* Paging eliminates external fragmentation and makes the allocation problem trivial: any empty frame can be used for a desired page. Each page in MULTICS consists of 1K words. Thus, the segment offset (16 bits) is broken into a 6-bit page number and a 10-bit page offset.

The page number indexes into the page table to give the frame number. Finally, the frame number is combined with the page offset to form a physical address. The translation scheme is shown in Figure 8.26. Notice that the difference between this solution and pure segmentation is that the segmenttable entry contains not the base address of the segment, but rather the base address of a *page table* for this segment.

We must now have a separate page table for each segment. However, because each segment is limited in length by its segment-table entry, the page table does not need to be full sized. It requires only as many entries as are actually needed.

As with paging, the last page of each segment generally will not be completely full. Thus, we will have, on the average, one-half page of *internal* fragmentation per segment. Consequently, although we have eliminated external

fragmentation, we have introduced internal fragmentation and increased table-space overhead.

In truth, even the paged-segmentation view of MULTICS just presented is simplistic. Because the segment number is an 18-bit quantity, we could have upto 262,144 segments, requiring an excessively large segment table.

To ease this problem, MULTICS pages the segment table! The segment number (18 bits) is broken into an &bit page number and a 10-bit page offset. Hence, the segment table is represented by a page table consisting of up to 28 entries. Thus, in general, a logical address in MULTICS is as follows:

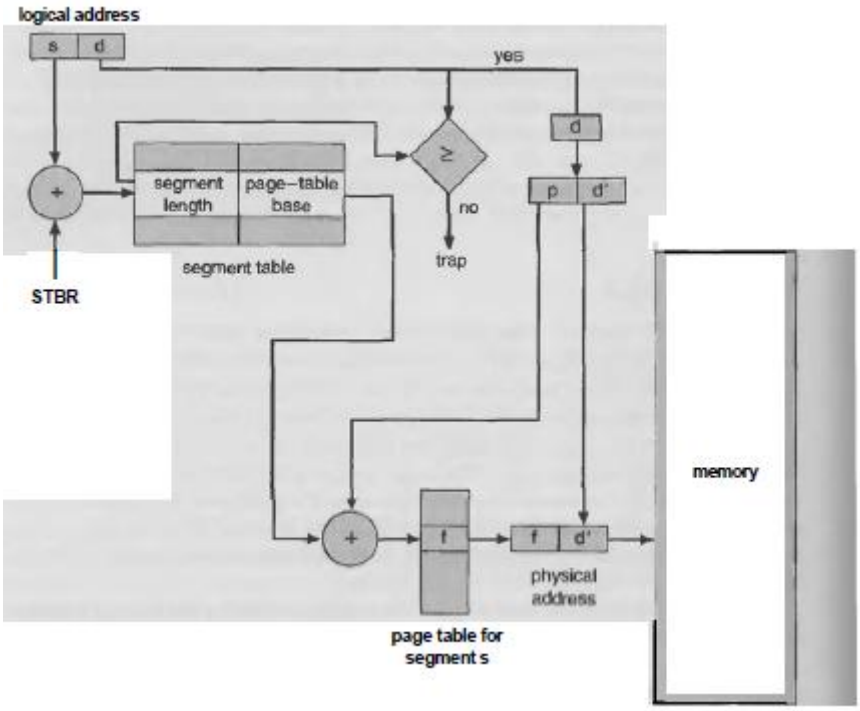


Figure *6.26* Paged segmentation on the GE 645 (MULTICS).

segment number		offset	
s_1	s_2	d_1	d_2
8	10	6	10

where ***sl*** is an index into the page table of the segment table and ***s2*** is the displacement within the page of the segment table. Now we have found the page containing the segment table we want.

Then, ***dl*** is a displacement into the page table of the desired segment, and finally, ***d2*** is a displacement into the page containing the word to be accessed (see Figure 6.27).

To ensure reasonable performance, 16 associative registers are available that contain the address of the 16 most recently referred pages. Each register consists of two parts: a key and a value. The key is a 24-bit field that is the concatenation of a segment number and a page number. The value is the frame number.

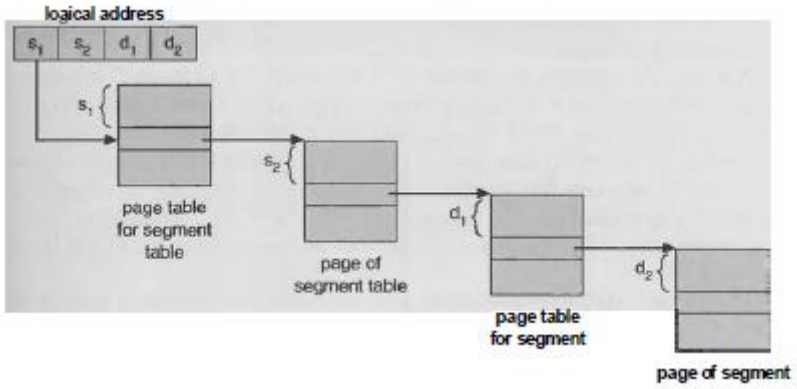


Figure 6.27 Address translation in MULTICS.