various portions of main memory in order to seize and relinquish control alternately. Certain other hardware features are also desirable:

- **Memory protection:** While the user program is executing, it must not alter the memory area containing the monitor. If such an attempt is made, the processor hardware should detect an error and transfer control to the monitor. The monitor would then abort the job, print out an error message, and load the next job.
- **Timer:** A timer is used to prevent a single job from monopolizing the system. The timer is set at the beginning of each job. If the timer expires, an interrupt occurs, and control returns to the monitor.
- **Privileged instructions:** Certain instructions are designated privileged and can be executed only by the monitor. If the processor encounters such an instruction while executing a user program, an error interrupt occurs. Among the privileged instructions are I/O instructions, so that the monitor retains control of all I/O devices. This prevents, for example, a user program from accidentally reading job control instructions from the next job. If a user program wishes to perform I/O, it must request that the monitor perform the operation for it. If a privileged instruction is encountered by the processor while it is executing a user program, the processor hardware considers this an error and transfers control to the monitor.
- **Interrupts:** Early computer models did not have this capability. This feature gives the operating system more flexibility in relinquishing control to and regaining control from user programs.

Processor time alternates between execution of user programs and execution of the monitor. There have been two sacrifices: Some main memory is now given over to the monitor and some processor time is consumed by the monitor. Both of these are forms of overhead. Even with this overhead, the simple batch system improves utilization of the computer.

Multiprogrammed Batch Systems Even with the automatic job sequencing provided by a simple batch operating system, the processor is often idle. The problem is that I/O devices are slow compared to the processor. Figure 8.4 details a representative calculation. The calculation concerns a program that processes a file of records and performs, on average, 100 processor instructions per record. In this example the computer spends over 96% of its time waiting for I/O devices to finish transferring data! Figure 8.5a illustrates this situation. The processor spends a certain amount of time executing, until it reaches an I/O instruction. It must then wait until that I/O instruction concludes before proceeding.

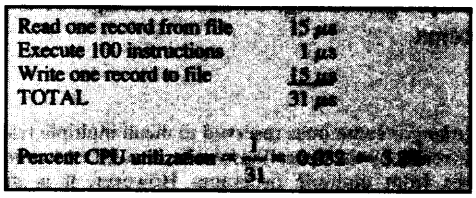


Figure 8.4 System Utilization Example

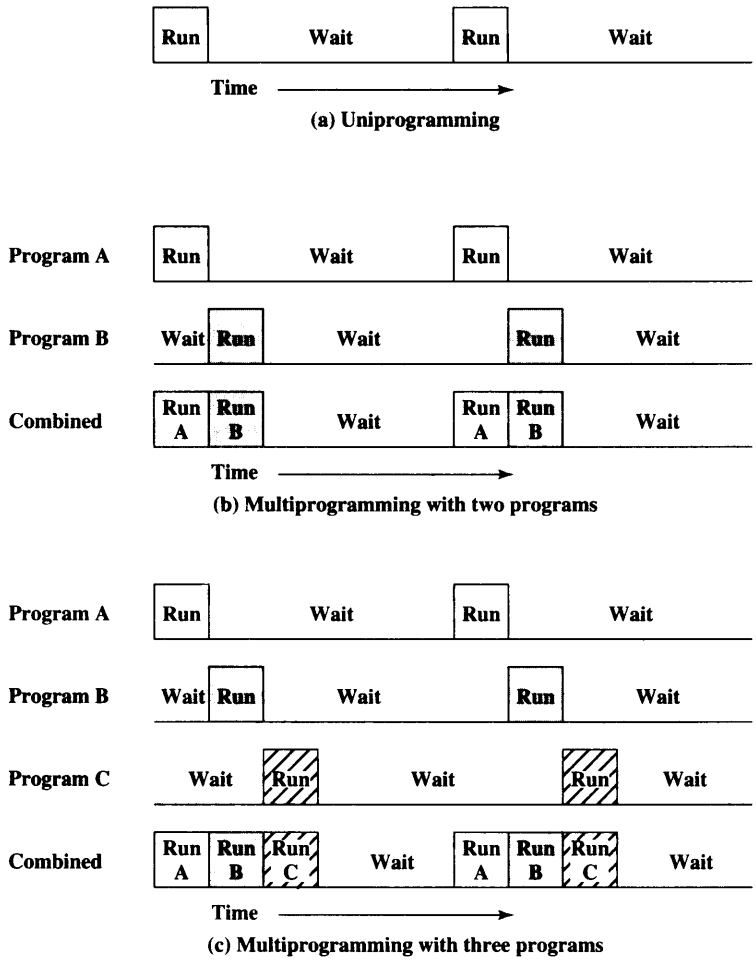


Figure 8.5 Multiprogramming Example

This inefficiency is not necessary. We know that there must be enough memory to hold the operating system (resident monitor) and one user program. Suppose that there is room for the operating system and two user programs. Now, when one job needs to wait for I/O, the processor can switch to the other job, which likely is not waiting for I/O (Figure 8.5b). Furthermore, we might expand memory to hold three, four, or more programs and switch among all of them (Figure 8.5c). This technique is known as **multiprogramming**, or **multitasking**.[1] It is the central theme of modern operating systems.

[1]The term *multitasking* is sometimes reserved to mean multiple tasks within the same program that may be handled concurrently by the operating system, in contrast to *multiprogramming*, which would refer to multiple processes from multiple programs. However, it is more common to equate the terms *multitasking* and *multiprogramming*, as is done in most standards dictionaries (e.g., IEEE Std 100-1992, *The New IEEE Standard Dictionary of Electrical and Electronics Terms*).

Example 8.1 This example illustrates the benefit of multiprogramming. Consider a computer with 250 Mbytes of available memory (not used by the operating system), a disk, a terminal, and a printer. Three programs, JOB1, JOB2, and JOB3, are submitted for execution at the same time, with the attributes listed in Table 8.1. We assume minimal processor requirements for JOB2 and JOB3 and continuous disk and printer use by JOB3. For a simple batch environment, these jobs will be executed in sequence. Thus, JOB1 completes in 5 minutes. JOB2 must wait until the 5 minutes is over, and then completes 15 minutes after that. JOB3 begins after 20 minutes and completes at 30 minutes from the time it was initially submitted. The average resource utilization, throughput, and response times are shown in the uniprogramming column of Table 8.2. Device-by-device utilization is illustrated in Figure 8.6a. It is evident that there is gross underutilization for all resources when averaged over the required 30-minute time period.

Now suppose that the jobs are run concurrently under a multiprogramming operating system. Because there is little resource contention between the jobs, all three can run in nearly minimum time while coexisting with the others in the computer (assuming that JOB2 and JOB3 are allotted enough processor time to keep their

Table 8.1 Sample Program Execution Attributes

	JOB1	JOB2	JOB3
Type of job	Heavy compute	Heavy I/O	Heavy I/O
Duration	5 min	15 min	10 min
Memory required	50 M	100 M	80 M
Need disk?	No	No	Yes
Need terminal?	No	Yes	No
Need printer?	No	No	Yes

Table 8.2 Effects of Multiprogramming on Resource Utilization

	Uniprogramming	Multiprogramming
Processor use	20%	40%
Memory use	33%	67%
Disk use	33%	67%
Printer use	33%	67%
Elapsed time	30 min	15 min
Throughput rate	6 jobs/hr	12 jobs/hr
Mean response time	18 min	10 min

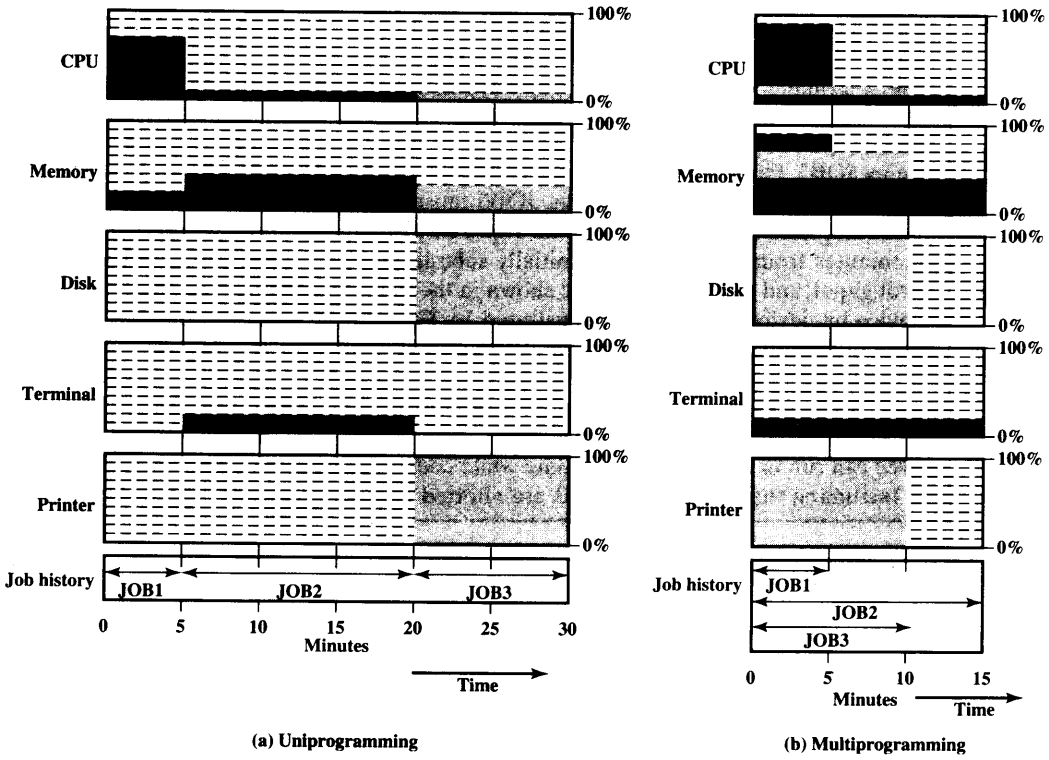


Figure 8.6 Utilization Histograms

input and output operations active). JOB1 will still require 5 minutes to complete but at the end of that time, JOB2 will be one-third finished, and JOB3 half finished. All three jobs will have finished within 15 minutes. The improvement is evident when examining the multiprogramming column of Table 8.2, obtained from the histogram shown in Figure 8.6b.

As with a simple batch system, a multiprogramming batch system must rely on certain computer hardware features. The most notable additional feature that is useful for multiprogramming is the hardware that supports I/O interrupts and DMA. With interrupt-driven I/O or DMA, the processor can issue an I/O command for one job and proceed with the execution of another job while the I/O is carried out by the device controller. When the I/O operation is complete, the processor is interrupted and control is passed to an interrupt-handling program in the operating system. The operating system will then pass control to another job.

Table 8.3 Batch Multiprogramming versus Time Sharing

	Batch Multiprogramming	Time Sharing
Principal objective	Maximize processor use	Minimize response time
Source of directives to operating system	Job control language commands provided with the job	Commands entered at the terminal

Multiprogramming operating systems are fairly sophisticated compared to single-program, or **uniprogramming**, systems. To have several jobs ready to run, the jobs must be kept in main memory, requiring some form of **memory management**. In addition, if several jobs are ready to run, the processor must decide which one to run, which requires some algorithm for scheduling. These concepts are discussed later in this chapter.

Time-Sharing Systems With the use of multiprogramming, batch processing can be quite efficient. However, for many jobs, it is desirable to provide a mode in which the user interacts directly with the computer. Indeed, for some jobs, such as transaction processing, an interactive mode is essential.

Today, the requirement for an interactive computing facility can be, and often is, met by the use of a dedicated microcomputer. That option was not available in the 1960s, when most computers were big and costly. Instead time sharing was developed.

Just as multiprogramming allows the processor to handle multiple batch jobs at a time, multiprogramming can be used to handle multiple interactive jobs. In this latter case, the technique is referred to as time sharing, because the processor's time is shared among multiple users. In a time-sharing system, multiple users simultaneously access the system through terminals, with the operating system interleaving the execution of each user program in a short burst or quantum of computation. Thus, if there are n users actively requesting service at one time, each user will only see on the average $1/n$ of the effective computer speed, not counting operating system overhead. However, given the relatively slow human reaction time, the response time on a properly designed system should be comparable to that on a dedicated computer.

Both batch multiprogramming and time sharing use multiprogramming. The key differences are listed in Table 8.3.

8.2 SCHEDULING

The key to multiprogramming is scheduling. In fact, four types of scheduling are typically involved (Table 8.4). We will explore these presently. But first, we introduce the concept of **process**. This term was first used by the designers of the Multics

Table 8.4 Types of Scheduling

Long-term scheduling	The decision to add to the pool of processes to be executed
Medium-term scheduling	The decision to add to the number of processes that are partially or fully in main memory
Short-term scheduling	The decision as to which available process will be executed by the processor
I/O scheduling	The decision as to which process's pending I/O request shall be handled by an available I/O device

operating system in the 1960s. It is a somewhat more general term than *job.* Many definitions have been given for the term *process,* including

- A program in execution
- The "animated spirit" of a program
- That entity to which a processor is assigned

This concept should become clearer as we proceed.

Long-Term Scheduling

The long-term scheduler determines which programs are admitted to the system for processing. Thus, it controls the degree of multiprogramming (number of processes in memory). Once admitted, a job or user program becomes a process and is added to the queue for the short-term scheduler. In some systems, a newly created process begins in a swapped-out condition, in which case it is added to a queue for the medium-term scheduler.

In a batch system, or for the batch portion of a general-purpose operating system, newly submitted jobs are routed to disk and held in a batch queue. The long-term scheduler creates processes from the queue when it can. There are two decisions involved here. First, the scheduler must decide that the operating system can take on one or more additional processes. Second, the scheduler must decide which job or jobs to accept and turn into processes. The criteria used may include priority, expected execution time, and I/O requirements.

For interactive programs in a time-sharing system, a process request is generated when a user attempts to connect to the system. Time-sharing users are not simply queued up and kept waiting until the system can accept them. Rather, the operating system will accept all authorized comers until the system is saturated, using some predefined measure of saturation. At that point, a connection request is met with a message indicating that the system is full and the user should try again later.

Medium-Term Scheduling

Medium-term scheduling is part of the swapping function, described in Section 8.3. Typically, the swapping-in decision is based on the need to manage the degree of multiprogramming. On a system that does not use virtual memory, memory management is also an issue. Thus, the swapping-in decision will consider the memory requirements of the swapped-out processes.

Short-Term Scheduling

The long-term scheduler executes relatively infrequently and makes the coarse-grained decision of whether or not to take on a new process, and which one to take. The short-term scheduler, also known as the **dispatcher**, executes frequently and makes the fine-grained decision of which job to execute next.

Process States To understand the operation of the short-term scheduler, we need to consider the concept of a **process state**. During the lifetime of a process, its status will change a number of times. Its status at any point in time is referred to as a *state*. The term *state* is used because it connotes that certain information exists that defines the status at that point. At minimum, there are five defined states for a process (Figure 8.7):

- **New:** A program is admitted by the high-level scheduler but is not yet ready to execute. The operating system will initialize the process, moving it to the ready state.
- **Ready:** The process is ready to execute and is awaiting access to the processor.
- **Running:** The process is being executed by the processor.
- **Waiting:** The process is suspended from execution waiting for some system resource, such as I/O.
- **Halted:** The process has terminated and will be destroyed by the operating system.

For each process in the system, the operating system must maintain information indicating the state of the process and other information necessary for process execution. For this purpose, each process is represented in the operating system by a **process control block** (Figure 8.8), which typically contains the following:

- **Identifier:** Each current process has a unique identifier.
- **State:** The current state of the process (new, ready, and so on).
- **Priority:** Relative priority level.
- **Program counter:** The address of the next instruction in the program to be executed.

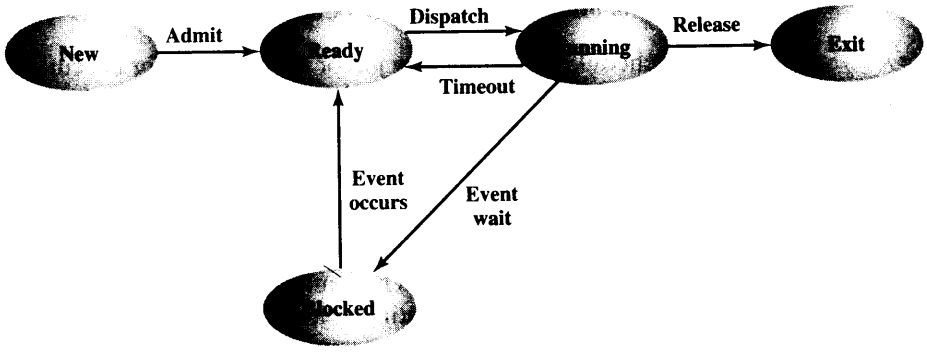


Figure 8.7 Five-State Process Model

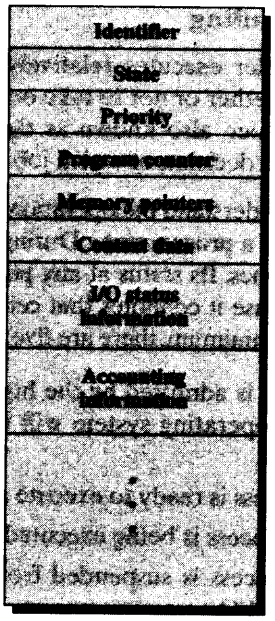
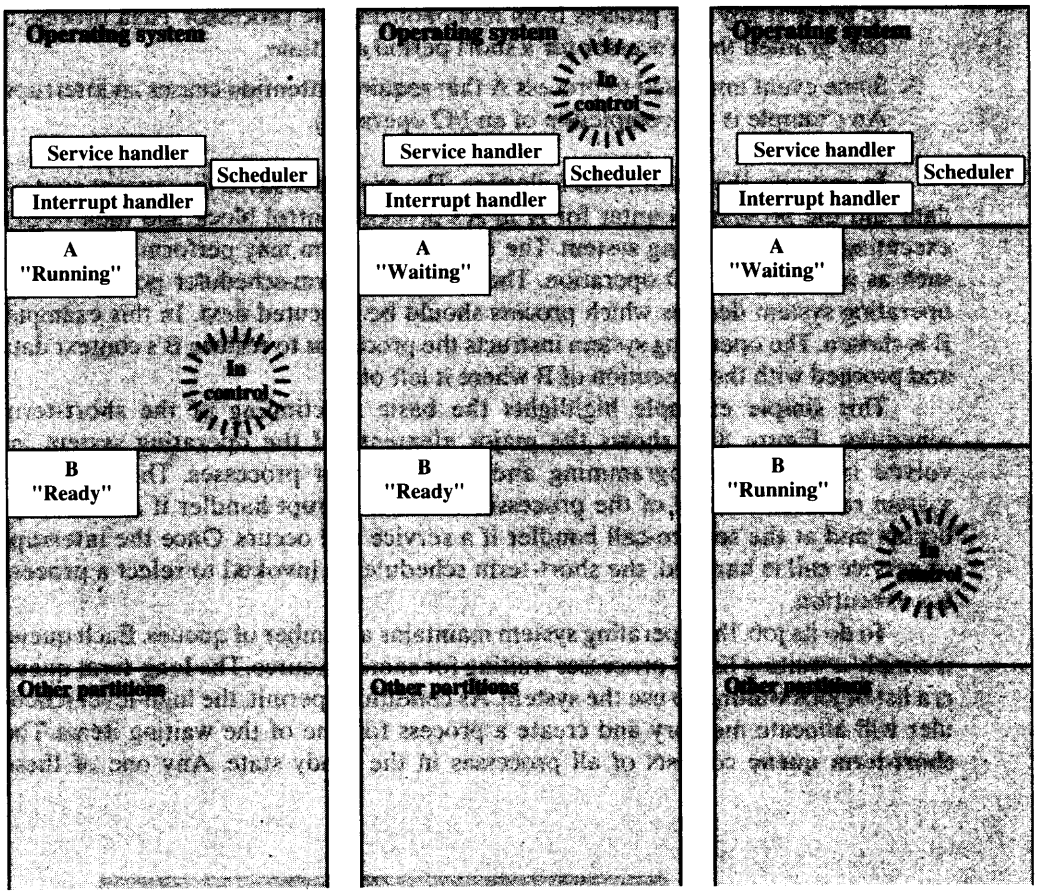


Figure 8.8 Process Control Block

- **Memory pointers:** The starting and ending locations of the process in memory.
- **Context data:** These are data that are present in registers in the processor while the process is executing, and they will be discussed in Part Three. For now, it is enough to say that these data represent the "context" of the process. The context data plus the program counter are saved when the process leaves the ready state. They are retrieved by the processor when it resumes execution of the process.
- **I/O status information:** Includes outstanding I/O requests, I/O devices (e.g., tape drives) assigned to this process, a list of files assigned to the process, and so on.
- **Accounting information:** May include the amount of processor time and clock time used, time limits, account numbers, and so on.

When the scheduler accepts a new job or user request for execution, it creates a blank process control block and places the associated process in the new state. After the system has properly filled in the process control block, the process is transferred to the ready state.

Scheduling Techniques To understand how the operating system manages the scheduling of the various jobs in memory, let us begin by considering the simple example in Figure 8.9. The figure shows how main memory is partitioned at a given point in time. The kernel of the operating system is, of course, always resident. In addition, there are a number of active processes, including A and B, each of which is allocated a portion of memory.



(a) (b) (c)

Figure 8.9 Scheduling Example

We begin at a point in time when process A is running. The processor is executing instructions from the program contained in A's memory partition. At some later point in time, the processor ceases to execute instructions in A and begins executing instructions in the operating system area. This will happen for one of three reasons:

1. Process A issues a service call (e.g., an I/O request) to the operating system. Execution of A is suspended until this call is satisfied by the operating system.
2. Process A causes an *interrupt.* An interrupt is a hardware-generated signal to the processor. When this signal is detected, the processor ceases to execute A and transfers to the interrupt handler in the operating system. A variety of events related to A will cause an interrupt. One example is an error, such as attempting to execute a privileged instruction. Another example is a timeout;

to prevent any one process from monopolizing the processor, each process is only granted the processor for a short period at a time.

3. Some event unrelated to process A that requires attention causes an interrupt. An example is the completion of an I/O operation.

In any case, the result is the following. The processor saves the current context data and the program counter for A in A's process control block and then begins executing in the operating system. The operating system may perform some work, such as initiating an I/O operation. Then the short-term-scheduler portion of the operating system decides which process should be executed next. In this example, B is chosen. The operating system instructs the processor to restore B's context data and proceed with the execution of B where it left off.

This simple example highlights the basic functioning of the short-term scheduler. Figure 8.10 shows the major elements of the operating system involved in the multiprogramming and scheduling of processes. The operating system receives control of the processor at the interrupt handler if an interrupt occurs and at the service-call handler if a service call occurs. Once the interrupt or service call is handled, the short-term scheduler is invoked to select a process for execution.

To do its job, the operating system maintains a number of queues. Each queue is simply a waiting list of processes waiting for some resource. The **long-term queue** is a list of jobs waiting to use the system. As conditions permit, the high-level scheduler will allocate memory and create a process for one of the waiting items. The **short-term queue** consists of all processes in the ready state. Any one of these

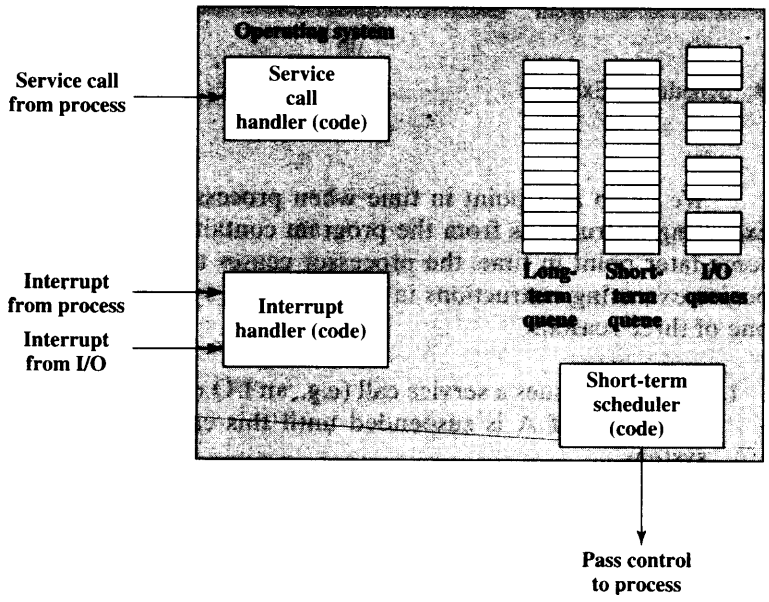


Figure 8.10 Key Elements of an Operating System for Multiprogramming

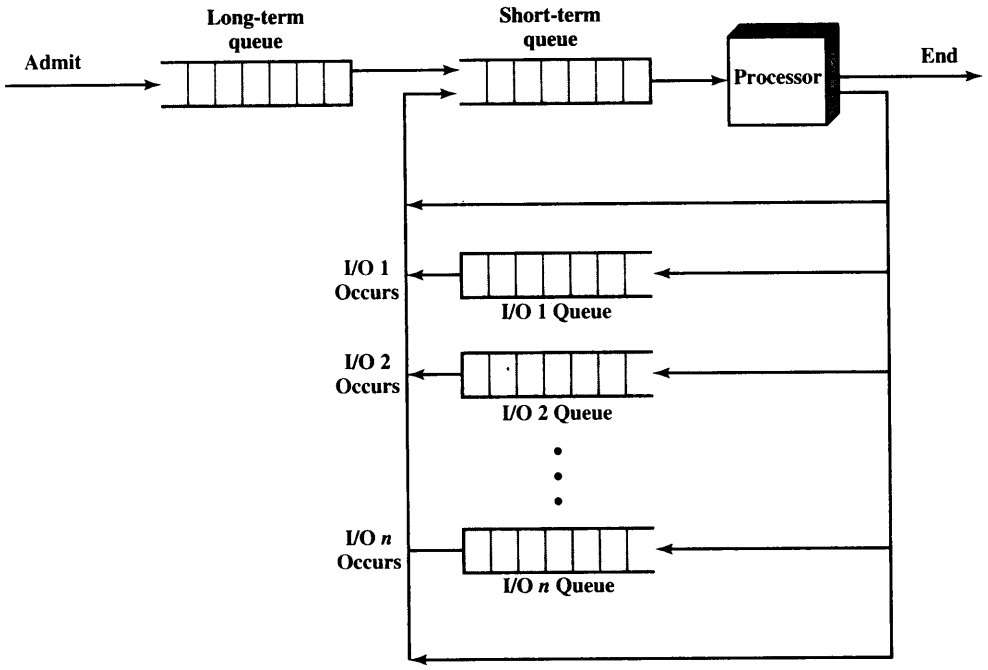


Figure 8.11 Queuing Diagram Representation of Processor Scheduling

processes could use the processor next. It is up to the short-term scheduler to pick one. Generally, this is done with a round-robin algorithm, giving each process some time in turn. Priority levels may also be used. Finally, there is an **I/O queue** for each I/O device. More than one process may request the use of the same I/O device. All processes waiting to use each device are lined up in that device's queue.

Figure 8.11 suggests how processes progress through the computer under the control of the operating system. Each process request (batch job, user-defined interactive job) is placed in the long-term queue. As resources become available, a process request becomes a process and is then placed in the ready state and put in the short-term queue. The processor alternates between executing operating system instructions and executing user processes. While the operating system is in control, it decides which process in the short-term queue should be executed next. When the operating system has finished its immediate tasks, it turns the processor over to the chosen process.

As was mentioned earlier, a process being executed may be suspended for a variety of reasons. If it is suspended because the process requests I/O, then it is placed in the appropriate I/O queue. If it is suspended because of a timeout or because the operating system must attend to pressing business, then it is placed in the ready state and put into the short-term queue.

Finally, we mention that the operating system also manages the I/O queues. When an I/O operation is completed, the operating system removes the satisfied process from that I/O queue and places it in the short-term queue. It then selects another waiting process (if any) and signals for the I/O device to satisfy that process's request.

8.3 MEMORY MANAGEMENT

In a uniprogramming system, main memory is divided into two parts: one part for the operating system (resident monitor) and one part for the program currently being executed. In a multiprogramming system, the "user" part of memory is subdivided to accommodate multiple processes. The task of subdivision is carried out dynamically by the operating system and is known as **memory management**.

Effective memory management is vital in a multiprogramming system. If only a few processes are in memory, then for much of the time all of the processes will be waiting for I/O and the processor will be idle. Thus, memory needs to be allocated efficiently to pack as many processes into memory as possible.

Swapping

Referring back to Figure 8.11, we have discussed three types of queues: the long-term queue of requests for new processes, the short-term queue of processes ready to use the processor, and the various I/O queues of processes that are not ready to use the processor. Recall that the reason for this elaborate machinery is that I/O activities are much slower than computation and therefore the processor in a uniprogramming system is idle most of the time.

But the arrangement in Figure 8.11 does not entirely solve the problem. It is true that, in this case, memory holds multiple processes and that the processor can move to another process when one process is waiting. But the processor is so much faster than I/O that it will be common for *all* the processes in memory to be waiting on I/O. Thus, even with multiprogramming, a processor could be idle most of the time.

What to do? Main memory could be expanded, and so be able to accommodate more processes. But there are two flaws in this approach. First, main memory is expensive, even today. Second, the appetite of programs for memory has grown as fast as the cost of memory has dropped. So larger memory results in larger processes, not more processes.

Another solution is *swapping,* depicted in Figure 8.12. We have a long-term queue of process requests, typically stored on disk. These are brought in, one at a time, as space becomes available. As processes are completed, they are moved out of main memory. Now the situation will arise that none of the processes in memory are in the ready state (e.g., all are waiting on an I/O operation). Rather than remain idle, the processor *swaps* one of these processes back out to disk into an *intermediate queue.* This is a queue of existing processes that have been temporarily kicked out of memory. The operating system then brings in another process from the intermediate queue, or it honors a new process request from the long-term queue. Execution then continues with the newly arrived process.

Swapping, however, is an I/O operation, and therefore there is the potential for making the problem worse, not better. But because disk I/O is generally the fastest I/O on a system (e.g., compared with tape or printer I/O), swapping will usually enhance performance. A more sophisticated scheme, involving virtual memory,

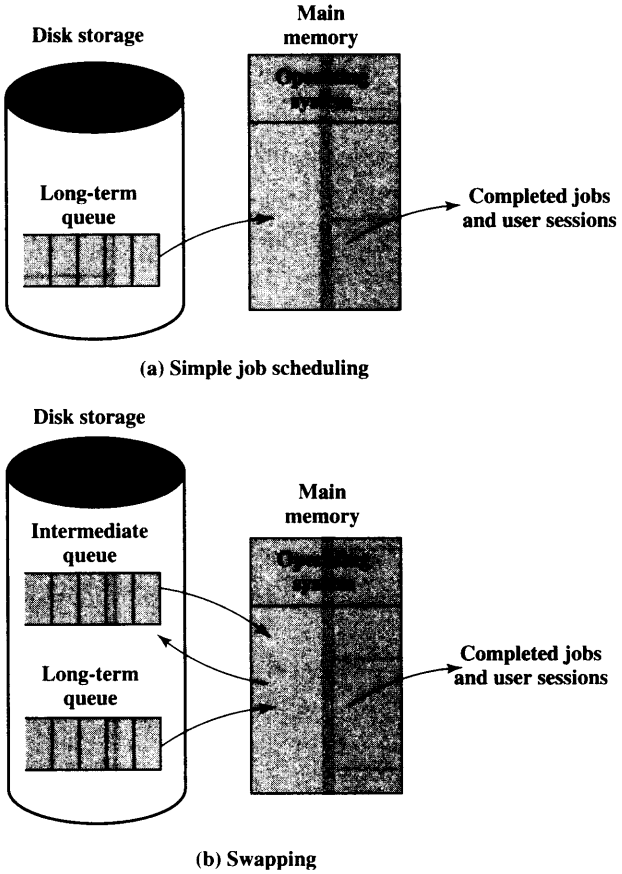


Figure 8.12 The Use of Swapping

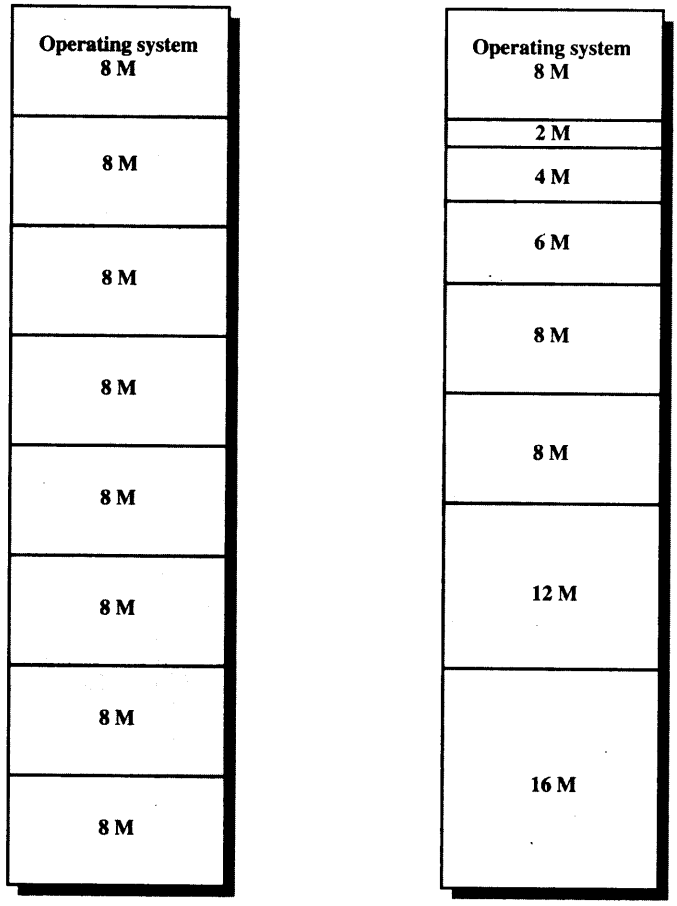
improves performance over simple swapping. This will be discussed shortly. But first, we must prepare the ground by explaining partitioning and paging.

Partitioning

The simplest scheme for partitioning available memory is to use *fixed-size partitions*, as shown in Figure 8.13. Note that, although the partitions are of fixed size, they need not be of equal size. When a process is brought into memory, it is placed in the smallest available partition that will hold it.

Even with the use of unequal fixed-size partitions, there will be wasted memory. In most cases, a process will not require exactly as much memory as provided by the partition. For example, a process that requires 3M bytes of memory would be placed in the 4M partition of Figure 8.13b, wasting 1M that could be used by another process.

A more efficient approach is to use *variable-size partitions*. When a process is brought into memory, it is allocated exactly as much memory as it requires and no more.



(a) Equal-size partitions

(b) Unequal-size partitions

Figure 8.13 Example of Fixed Partitioning of a 64-Mbyte Memory

Example 8.2 An example, using 64 Mbytes of main memory, is shown in Figure 8.14. Initially, main memory is empty, except for the operating system (a). The first three processes are loaded in, starting where the operating system ends and occupying just enough space for each process (b, c, d). This leaves a "hole" at the end of memory that is too small for a fourth process. At some point, none of the processes in memory is ready. The operating system swaps out process 2 (e), which leaves sufficient room to load a new process, process 4 (f). Because process 4 is smaller than process 2, another small hole is created. Later, a point is reached at which none of the processes in main memory is ready, but process 2, in the Ready-Suspend state, is available. Because there is insufficient room in memory for process 2, the operating system swaps process 1 out (g) and swaps process 2 back in (h).

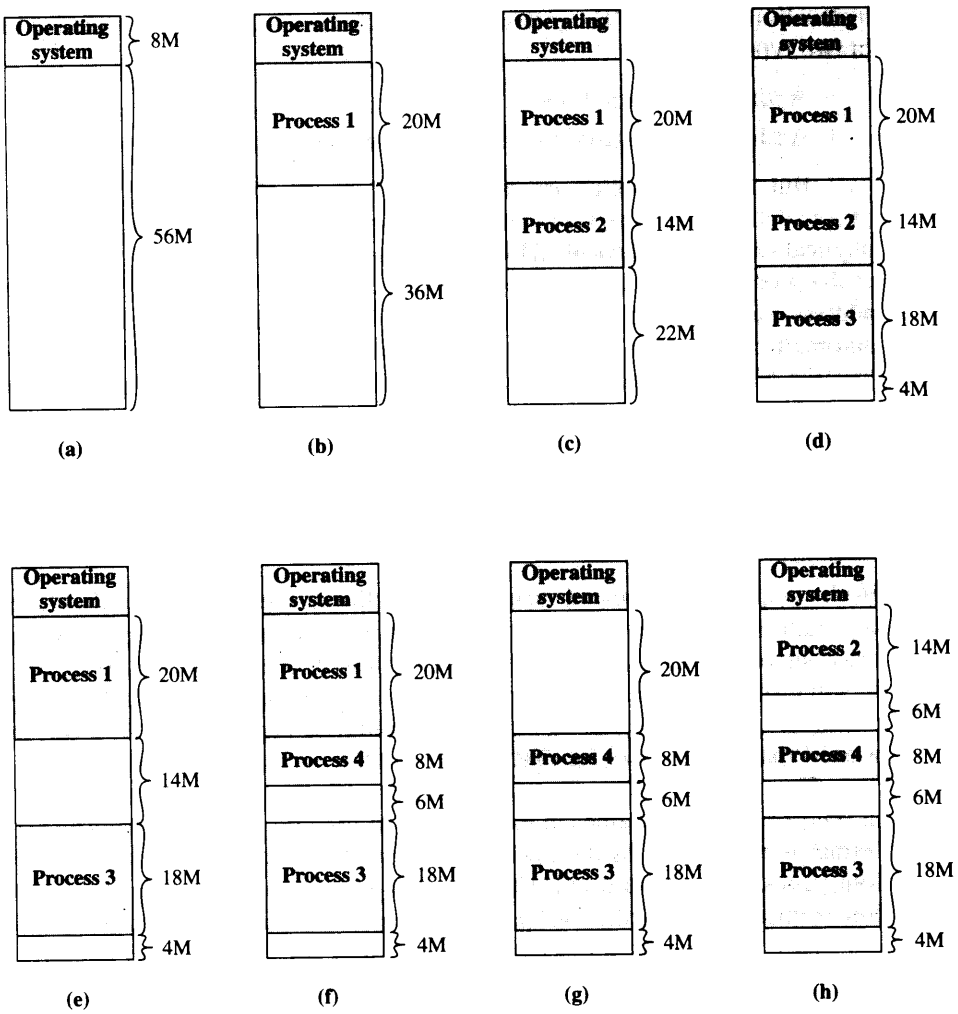


Figure 8.14 The Effect of Dynamic Partitioning

As this example shows, this method starts out well, but eventually it leads to a situation in which there are a lot of small holes in memory. As time goes on, memory becomes more and more fragmented, and memory utilization declines. One technique for overcoming this problem is **compaction**: From time to time, the operating system shifts the processes in memory to place all the free memory together in one block. This is a time-consuming procedure, wasteful of processor time.

Before we consider ways of dealing with the shortcomings of partitioning, we must clear up one loose end. If the reader considers Figure 8.14 for a moment, it should become obvious that a process is not likely to be loaded into the same place in main memory each time it is swapped in. Furthermore, if compaction is used, a process may be shifted while in main memory. A process in memory consists of

instructions plus data. The instructions will contain addresses for memory locations of two types:

- Addresses of data items
- Addresses of instructions, used for branching instructions

But these addresses are not fixed. They will change each time a process is swapped in. To solve this problem, a distinction is made between logical addresses and physical addresses. A **logical address** is expressed as a location relative to the beginning of the program. Instructions in the program contain only logical addresses. A **physical address** is an actual location in main memory. When the processor executes a process, it automatically converts from logical to physical address by adding the current starting location of the process, called its **base address**, to each logical address. This is another example of a processor hardware feature designed to meet an operating system requirement. The exact nature of this hardware feature depends on the memory management strategy in use. We will see several examples later in this chapter.

Paging

Both unequal fixed-size and variable-size partitions are inefficient in the use of memory. Suppose, however, that memory is partitioned into equal fixed-size chunks that are relatively small, and that each process is also divided into small fixed-size chunks of some size. Then the chunks of a program, known as **pages**, could be assigned to available chunks of memory, known as **frames**, or page frames. At most, then, the wasted space in memory for that process is a fraction of the last page.

Figure 8.15 shows an example of the use of pages and frames. At a given point in time, some of the frames in memory are in use and some are free. The list of free frames is maintained by the operating system. Process A, stored on disk, consists of four pages. When it comes time to load this process, the operating system finds four free frames and loads the four pages of the process A into the four frames.

Now suppose, as in this example, that there are not sufficient unused contiguous frames to hold the process. Does this prevent the operating system from loading A? The answer is no, because we can once again use the concept of logical address. A simple base address will no longer suffice. Rather, the operating system maintains a **page table** for each process. The page table shows the frame location for each page of the process. Within the program, each logical address consists of a page number and a relative address within the page. Recall that in the case of simple partitioning, a logical address is the location of a word relative to the beginning of the program; the processor translates that into a physical address. With paging, the logical-to-physical address translation is still done by processor hardware. The processor must know how to access the page table of the current process. Presented with a logical address (page number, relative address), the processor uses the page table to produce a physical address (frame number, relative address). An example is shown in Figure 8.16.

This approach solves the problems raised earlier. Main memory is divided into many small equal-size frames. Each process is divided into frame-size pages: Smaller processes require fewer pages, larger processes require more. When a process is brought in, its pages are loaded into available frames, and a page table is set up.

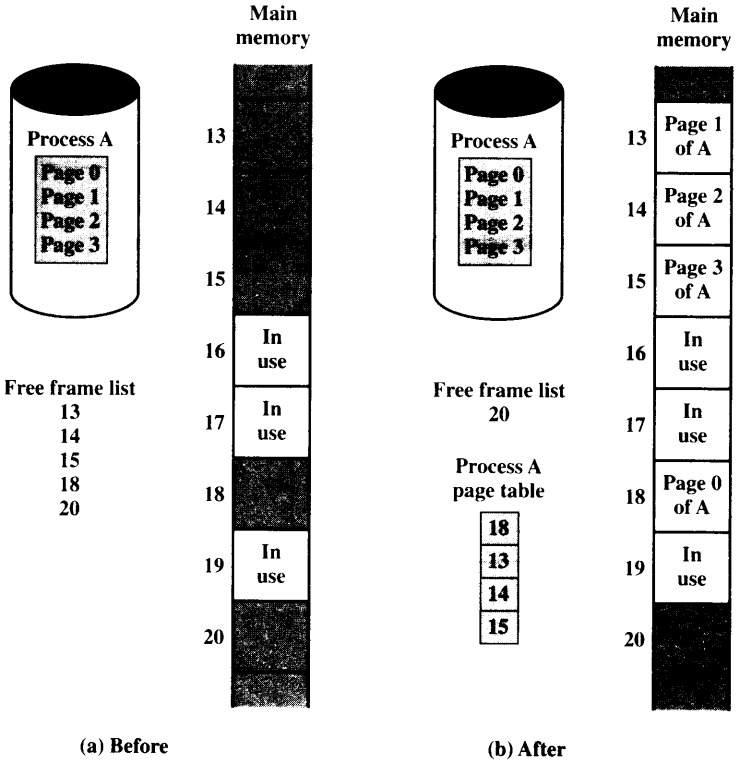


Figure 8.15 Allocation of Free Frames

Virtual Memory

Demand Paging With the use of paging, truly effective multiprogramming systems came into being. Furthermore, the simple tactic of breaking a process up into pages led to the development of another important concept: virtual memory.

To understand virtual memory, we must add a refinement to the paging scheme just discussed. That refinement is **demand paging**, which simply means that each page of a process is brought in only when it is needed, that is, on demand.

Consider a large process, consisting of a long program plus a number of arrays of data. Over any short period of time, execution may be confined to a small section of the program (e.g., a subroutine), and perhaps only one or two arrays of data are being used. This is the principle of locality, which we introduced in Appendix 4A. It would clearly be wasteful to load in dozens of pages for that process when only a few pages will be used before the program is suspended. We can make better use of memory by loading in just a few pages. Then, if the program branches to an instruction on a page not in main memory, or if the program references data on a page not in memory, a **page fault** is triggered. This tells the operating system to bring in the desired page.

Thus, at any one time, only a few pages of any given process are in memory, and therefore more processes can be maintained in memory. Furthermore, time is saved because unused pages are not swapped in and out of memory. However, the operating system must be clever about how it manages this scheme. When it brings one page in, it

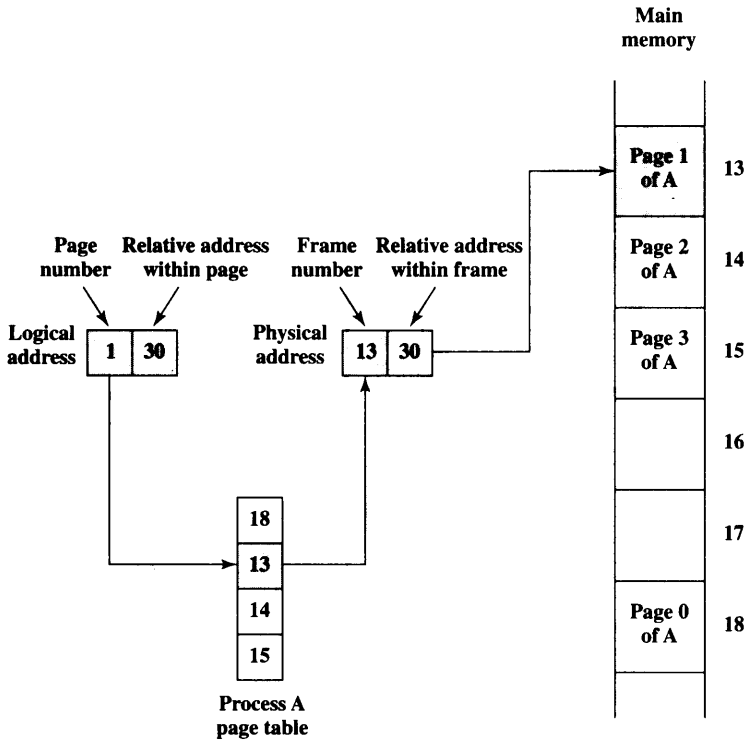


Figure 8.16 Logical and Physical Addresses

must throw another page out; this is known as **page replacement**. If it throws out a page just before it is about to be used, then it will just have to go get that page again almost immediately. Too much of this leads to a condition known as **thrashing**: The processor spends most of its time swapping pages rather than executing instructions. The avoidance of thrashing was a major research area in the 1970s and led to a variety of complex but effective algorithms. In essence, the operating system tries to guess, based on recent history, which pages are least likely to be used in the near future.

With demand paging, it is not necessary to load an entire process into main memory. This fact has a remarkable consequence: *It is possible for a process to be larger than all of main memory.* One of the most fundamental restrictions in programming has been lifted. Without demand paging, a programmer must be acutely aware of how much memory is available. If the program being written is too large, the programmer must devise ways to structure the program into pieces that can be loaded one at a time. With demand paging, that job is left to the operating system and the hardware. As far as the programmer is concerned, he or she is dealing with a huge memory, the size associated with disk storage.

Because a process executes only in main memory, that memory is referred to as **real memory**. But a programmer or user perceives a much larger memory—that which is allocated on the disk. This latter is therefore referred to as **virtual memory**. Virtual memory allows for very effective multiprogramming and relieves the user of the unnecessarily tight constraints of main memory.

Page Table Structure The basic mechanism for reading a word from memory involves the translation of a virtual, or logical, address, consisting of page number and offset, into a physical address, consisting of frame number and offset, using a page table. Because the page table is of variable length, depending on the size of the process, we cannot expect to hold it in registers. Instead, it must be in main memory to be accessed. Figure 8.16 suggests a hardware implementation of this scheme. When a particular process is running, a register holds the starting address of the page table for that process. The page number of a virtual address is used to index that table and look up the corresponding frame number. This is combined with the offset portion of the virtual address to produce the desired real address.

In most systems, there is one page table per process. But each process can occupy huge amounts of virtual memory. For example, in the VAX architecture, each process can have up to 2^{31} = 2 GBytes of virtual memory. Using 2^9 = 512-byte pages, that means that as many as 2^{22} page table entries are required *per process*. Clearly, the amount of memory devoted to page tables alone could be unacceptably high. To overcome this problem, most virtual memory schemes store page tables in virtual memory rather than real memory. This means that page tables are subject to paging just as other pages are. When a process is running, at least a part of its page table must be in main memory, including the page table entry of the currently executing page. Some processors make use of a two-level scheme to organize large page tables. In this scheme, there is a page directory, in which each entry points to a page table. Thus, if the length of the page directory is X, and if the maximum length of a page table is Y, then a process can consist of up to $X \times Y$ pages. Typically, the maximum length of a page table is restricted to be equal to one page. We will see an example of this two-level approach when we consider the Pentium II later in this chapter.

An alternative approach to the use of one- or two-level page tables is the use of an inverted page table structure (Figure 8.17). Variations on this approach are used on the PowerPC, UltraSPARC, and the IA-64 architecture. An implementation of the Mach operating system on the RT-PC also uses this technique.

In this approach, the page number portion of a virtual address is mapped into a hash value using a simple hashing function.[2] The hash value is a pointer to the inverted page table, which contains the page table entries. There is one entry in the inverted page table for each real memory page frame rather than one per virtual page. Thus a fixed proportion of real memory is required for the tables regardless of the number of processes or virtual pages supported. Because more than one virtual address may map into the same hash table entry, a chaining technique is used for managing the overflow. The hashing technique results in chains that are typically short—between one and two entries. The page table's structure is called *inverted* because it indexes page table entries by frame number rather than by virtual page number.

---

[2]A hash function maps numbers in the range 0 through M into numbers in the range 0 through N, where $M > N$. The output of the hash function is used as an index into the hash table. Since more than one input maps into the same output, it is possible for an input item to map to a hash table entry that is already occupied. In that case, the new item must *overflow* into another hash table location. Typically, the new item is placed in the first succeeding empty space, and a pointer from the original location is provided to chain the entries together. A document at this book's Web site provides more information on hash functions.

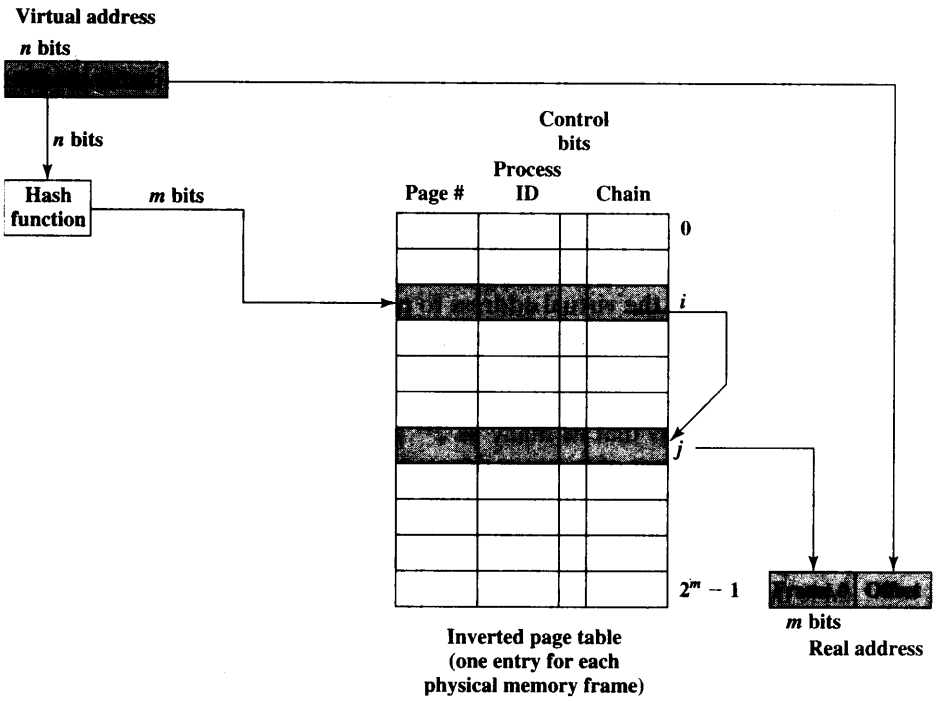


Figure 8.17 Inverted Page Table Structure

Translation Lookaside Buffer

In principle, then, every virtual memory reference can cause two physical memory accesses: one to fetch the appropriate page table entry, and one to fetch the desired data. Thus, a straightforward virtual memory scheme would have the effect of doubling the memory access time. To overcome this problem, most virtual memory schemes make use of a special cache for page table entries, usually called a translation lookaside buffer (TLB). This cache functions in the same way as a memory cache and contains those page table entries that have been most recently used. Figure 8.18 is a flowchart that shows the use of the TLB. By the principle of locality, most virtual memory references will be to locations in recently used pages. Therefore, most references will involve page table entries in the cache. Studies of the VAX TLB have shown that this scheme can significantly improve performance [CLAR85, SATY81].

Note that the virtual memory mechanism must interact with the cache system (not the TLB cache, but the main memory cache). This is illustrated in Figure 8.19. A virtual address will generally be in the form of a page number, offset. First, the memory system consults the TLB to see if the matching page table entry is present. If it is, the real (physical) address is generated by combining the frame number with the offset. If not, the entry is accessed from a page table. Once the real address is generated, which is in the form of a tag and a remainder, the cache is consulted to see if the block containing that word is present (see Figure 4.5). If so, it is returned to the processor. If not, the word is retrieved from main memory.

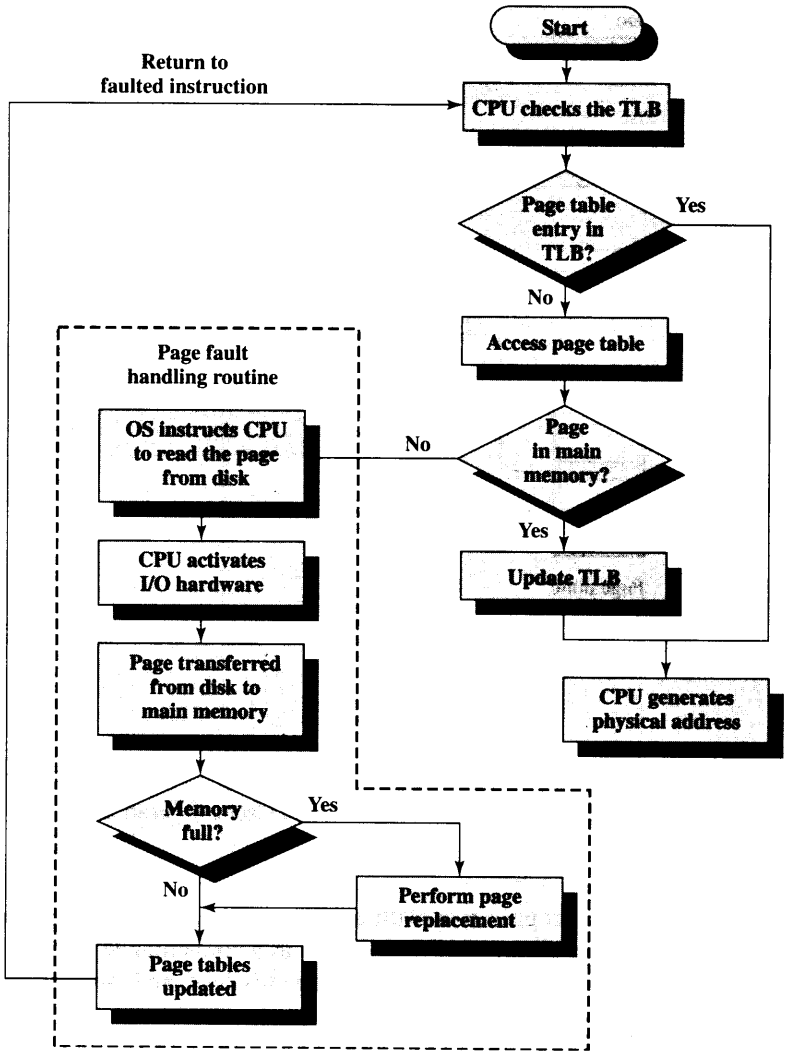


Figure 8.18 Operation of Paging and Translation Lookaside Buffer (TLB) [FURH87]

The reader should be able to appreciate the complexity of the processor hardware involved in a single memory reference. The virtual address is translated into a real address. This involves reference to a page table, which may be in the TLB, in main memory, or on disk. The referenced word may be in cache, in main memory, or on disk. In the latter case, the page containing the word must be loaded into main memory and its block loaded into the cache. In addition, the page table entry for that page must be updated.

Segmentation

There is another way in which addressable memory can be subdivided, known as *segmentation*. Whereas paging is invisible to the programmer and serves the purpose

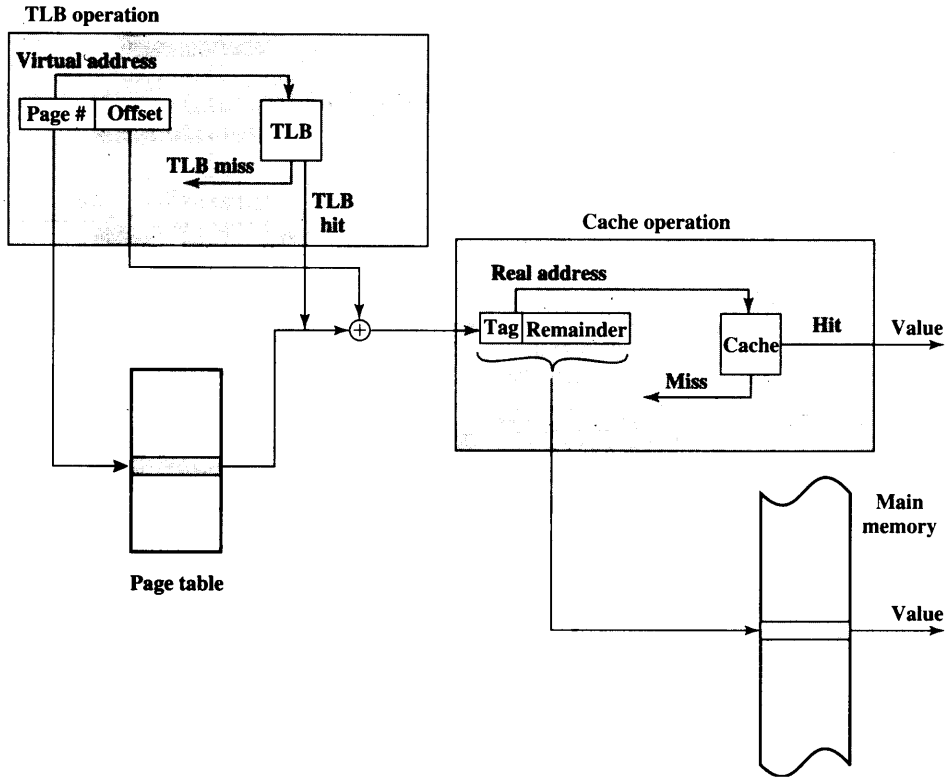


Figure 8.19 Translation Lookaside Buffer and Cache Operation

of providing the programmer with a larger address space, segmentation is usually visible to the programmer and is provided as a convenience for organizing programs and data and as a means for associating privilege and protection attributes with instructions and data.

Segmentation allows the programmer to view memory as consisting of multiple address spaces or segments. Segments are of variable, indeed dynamic, size. Typically, the programmer or the operating system will assign programs and data to different segments. There may be a number of program segments for various types of programs as well as a number of data segments. Each segment may be assigned access and usage rights. Memory references consist of a (segment number, offset) form of address.

This organization has a number of advantages to the programmer over a non-segmented address space:

1. It simplifies the handling of growing data structures. If the programmer does not know ahead of time how large a particular data structure will become, it is not necessary to guess. The data structure can be assigned its own segment, and the operating system will expand or shrink the segment as needed.

2. It allows programs to be altered and recompiled independently without requiring that an entire set of programs be relinked and reloaded. Again, this is accomplished using multiple segments.
3. It lends itself to sharing among processes. A programmer can place a utility program or a useful table of data in a segment that can be addressed by other processes.
4. It lends itself to protection. Because a segment can be constructed to contain a well-defined set of programs or data, the programmer or a system administrator can assign access privileges in a convenient fashion.

These advantages are not available with paging, which is invisible to the programmer. On the other hand, we have seen that paging provides for an efficient form of memory management. To combine the advantages of both, some systems are equipped with the hardware and operating system software to provide both.

8.4 PENTIUM II AND POWERPC MEMORY MANAGEMENT

Pentium II Memory-Management Hardware

Since the introduction of the 32-bit architecture, microprocessors have evolved sophisticated memory management schemes that build on the lessons learned with medium- and large-scale systems. In many cases, the microprocessor versions are superior to their larger-system antecedents. Because the schemes were developed by the microprocessor hardware vendor and may be employed with a variety of operating systems, they tend to be quite general purpose. A representative example is the scheme used on the Pentium II. The Pentium II memory management hardware is essentially the same as that used in the Intel 80386 and 80486 processors, with some refinements.

Address Spaces The Pentium II includes hardware for both segmentation and paging. Both mechanisms can be disabled, allowing the user to choose from four distinct views of memory:

- **Unsegmented unpaged memory:** In this case, the virtual address is the same as the physical address. This is useful, for example, in low-complexity, high-performance controller applications.
- **Unsegmented paged memory:** Here memory is viewed as a paged linear address space. Protection and management of memory is done via paging. This is favored by some operating systems (e.g., Berkeley UNIX).
- **Segmented unpaged memory:** Here memory is viewed as a collection of logical address spaces. The advantage of this view over a paged approach is that it affords protection down to the level of a single byte, if necessary. Furthermore, unlike paging, it guarantees that the translation table needed (the segment table) is on-chip when the segment is in memory. Hence, segmented unpaged memory results in predictable access times.

- **Segmented paged memory:** Segmentation is used to define logical memory partitions subject to access control, and paging is used to manage the allocation of memory within the partitions. Operating systems such as UNIX System V favor this view.

Segmentation When segmentation is used, each virtual address (called a logical address in the Pentium II documentation) consists of a 16-bit segment reference and a 32-bit offset. Two bits of the segment reference deal with the protection mechanism, leaving 14 bits for specifying a particular segment. Thus, with unsegmented memory, the user's virtual memory is 2^{32} = 4 GBytes. With segmented memory, the total virtual memory space as seen by a user is 2^{46} = 64 terabytes (TBytes). The physical address space employs a 32-bit address for a maximum of 4 GBytes.

The amount of virtual memory can actually be larger than the 64 TBytes. This is because the processor's interpretation of a virtual address depends on which process is currently active. Virtual address space is divided into two parts. One-half of the virtual address space (8K segments × 4 GBytes) is global, shared by all processes; the remainder is local and is distinct for each process.

Associated with each segment are two forms of protection: privilege level and access attribute. There are four privilege levels from most protected (level 0) to least protected (level 3). The privilege level associated with a data segment is its "classification"; the privilege level associated with a program segment is its "clearance." An executing program may only access data segments for which its clearance level is lower than (more privileged) or equal to (same privilege) the privilege level of the data segment.

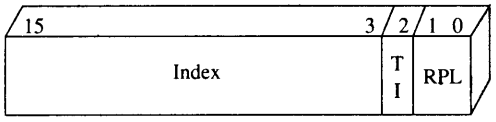
The hardware does not dictate how these privilege levels are to be used; this depends on the operating system design and implementation. It was intended that privilege level 1 would be used for most of the operating system, and level 0 would be used for that small portion of the operating system devoted to memory management, protection, and access control. This leaves two levels for applications. In many systems, applications will reside at level 3, with level 2 being unused. Specialized application subsystems that must be protected because they implement their own security mechanisms are good candidates for level 2. Some examples are database management systems, office automation systems, and software engineering environments.

In addition to regulating access to data segments, the privilege mechanism limits the use of certain instructions. Some instructions, such as those dealing with memory-management registers, can only be executed in level 0. I/O instructions can only be executed up to a certain level that is designated by the operating system; typically, this will be level 1.

The access attribute of a data segment specifies whether read/write or read-only accesses are permitted. For program segments, the access attribute specifies read/execute or read-only access.

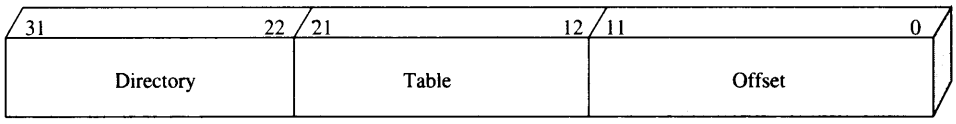
The address translation mechanism for segmentation involves mapping a virtual address into what is referred to as a linear address (Figure 8.20b). A virtual address consists of the 32-bit offset and a 16-bit segment selector (Figure 8.20a). The segment selector consists of the following fields:

- **Table Indicator (TI):** Indicates whether the global segment table or a local segment table should be used for translation.

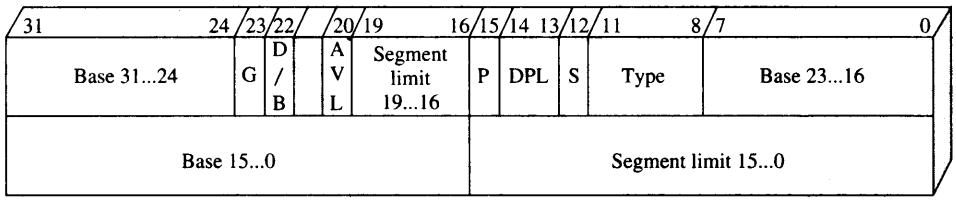


TI = Table indicator
RPL = Requestor privilege level

(a) Segment selector

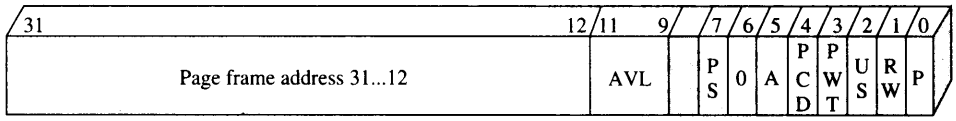


(b) Linear address



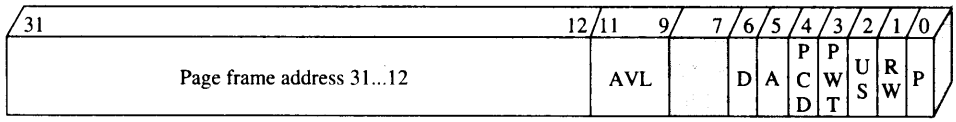
AVL = Available for use by system software
Base = Segment base address
D/B = Default operation size
DPL = Descriptor privilege size
G = Granularity
Limit = Segment limit
P = Segment present
Type = Segment type
S = Descriptor type
□ = Reserved

(c) Segment descriptor (segment table entry)



AVL = Available for systems programmer use
PS = Page size
A = Accessed
PCD = Cache disable
PWT = Write through
US = User/supervisor
RW = Read–write
P = Present

(d) Page directory entry



D = Dirty

(e) Page table entry

Figure 8.20 Pentium Memory-Management Formats

- **Segment Number:** The number of the segment. This serves as an index into the segment table.
- **Requested Privilege level (RPL):** The privilege level requested for this access.

Each entry in a segment table consists of 64 bits, as shown in Figure 8.20c. The fields are defined in Table 8.5.

Paging Segmentation is an optional feature and may be disabled. When segmentation is in use, addresses used in programs are virtual addresses and are converted into linear addresses, as just described. When segmentation is not in use, linear addresses are used in programs. In either case, the following step is to convert that linear address into a real 32-bit address.

To understand the structure of the linear address, you need to know that the Pentium II paging mechanism is actually a two-level table lookup operation. The first level is a page directory, which contains up to 1024 entries. This splits the 4-GByte linear memory space into 1024 page groups, each with its own page table, and each 4 MBytes in length. Each page table contains up to 1024 entries; each entry corresponds to a single 4-KByte page. Memory management has the option of using one page directory for all processes, one page directory for each process, or some combination of the two. The page directory for the current task is always in main memory. Page tables may be in virtual memory.

Figure 8.20 shows the formats of entries in page directories and page tables, and the fields are defined in Table 8.5. Note that access control mechanisms can be provided on a page or page group basis.

The Pentium II also makes use of a translation lookaside buffer. The buffer can hold 32 page table entries. Each time that the page directory is changed, the buffer is cleared.

Figure 8.21 illustrates the combination of segmentation and paging mechanisms. For clarity, the translation lookaside buffer and memory cache mechanisms are not shown.

Finally, the Pentium II includes a new extension not found on the 80386 or 80486, the provision for two page sizes. If the PSE (page size extension) bit in control register 4 is set to 1, then the paging unit permits the operating system programmer to define a page as either 4 KByte or 4 MByte in size.

When 4-MByte pages are used, there is only one level of table lookup for pages. When the hardware accesses the page directory, the page directory entry (Figure 8.20d) has the PS bit set to 1. In this case, bits 9 through 21 are ignored and bits 22 through 31 define the base address for a 4-MByte page in memory. Thus, there is a single page table.

The use of 4-MByte pages reduces the memory-management storage require-ments for large main memories. With 4-KByte pages, a full 4-GByte main memory requires about 4 MBytes of memory just for the page tables. With 4-MByte pages, a single table, 4 KBytes in length, is sufficient for page memory management.