

# Memory Management and Processor Management

WHEN YOU FINISH READING THIS CHAPTER YOU SHOULD BE ABLE TO:

- Distinguish between resident and transient routines.
- Briefly explain concurrency.
- Explain several memory management techniques including fixed-partition memory management, dynamic memory management, segmentation, paging, and segmentation and paging.
- Briefly explain dynamic address translation.
- Explain why memory protection is necessary.
- Explain how virtual memory works.
- Define multiprogramming.
- Explain how a multiprogramming operating system's dispatcher manages the processor. Explain the role of interrupts in the dispatching process.
- Distinguish between multiprogramming and time-sharing.
- Explain how the queuing routine and the scheduler work together to load application programs.
- Define spooling.
- Briefly explain deadlock.

## ■ Memory Management

**Memory management** is concerned with managing the computer's available pool of memory, allocating space to application routines, and making sure that they do not interfere with each other.

### Resident and Transient Routines

The operating system is a collection of software routines. Some routines, such as the ones that control physical I/O, directly support application programs as they run and thus must be **resident**. Others, such as the routine that formats disks, are used only occasionally. These **transient** routines are stored on disk and read into memory only when needed.

Generally, the operating system occupies low memory beginning with address 0. Key control information comes first (Figure 6.1), followed by the various resident operating system routines. The remaining memory, called the **transient area**, is where application programs and transient operating system routines are loaded.

### Concurrency

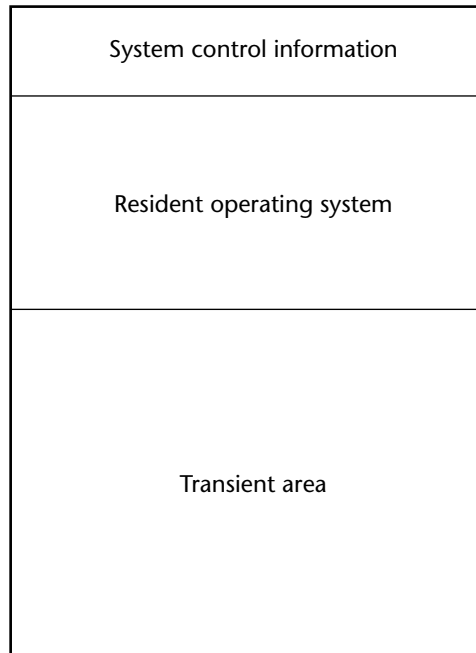
Given the speed disparity between a computer and its peripherals, input and output operations significantly impact efficiency. For example, picture a computer with a single program in memory. The program cannot process data it does not yet have, and success cannot be assumed until an output operation is finished, so the program waits for input or output. Since the program controls the computer, the computer waits, too. Typically, given the speed disparity between the processor and its peripheral devices, a program spends far more time waiting for I/O than processing data.

Why not put two programs in memory? Then, when program A is waiting for data, the processor can turn its attention to program B. And why stop at two programs? With three, even more otherwise wasted time is utilized (Figure 6.2). Generally, the more programs in memory, the greater the utilization of the processor.

The processor fetches and executes a single instruction during each machine cycle. If the processor can execute only one *instruction* at a time, it cannot possibly execute two or more programs at a time. Consequently, although multiple programs can share memory, only one can be active. Simultaneous means "at the same instant." No single processor can execute two or more programs simultaneously. Concurrent means "over the same time period." A processor can certainly execute two or more programs concurrently.

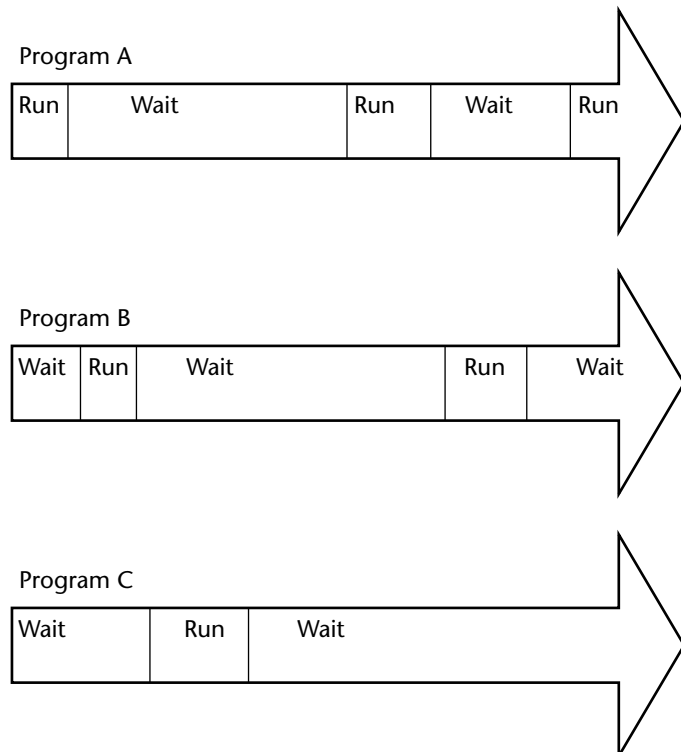
**Figure 6.1**

Generally, the operating system occupies low memory. The remaining memory, called the transient area, is where application programs and transient operating system routines are loaded.



**Figure 6.2**

On many systems, multiple programs are loaded into memory and executed concurrently.



## Partitions and Regions

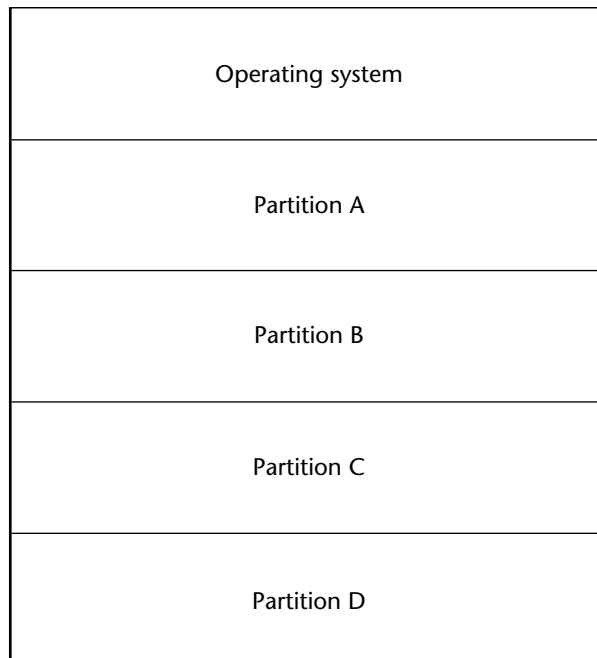
The simplest approach to managing memory for multiple, concurrent programs, **fixed-partition memory management** (Figure 6.3), divides the available space into fixed-length **partitions**, each of which holds one program. Partition sizes are generally set when the system is initially started, so the memory allocation decision is made before the actual amount of space needed by a given program is known. Because the size of a partition must be big enough to hold the largest program that is likely to be loaded, fixed-partition memory management wastes space. Its major advantage is simplicity.

Under **dynamic memory management**, the transient area is treated as a pool of unstructured free space. When the system decides to load a particular program, a **region** of memory just sufficient to hold the program is allocated from the pool. Because a program gets only the space it needs, relatively little is wasted.

Dynamic memory management does not completely solve the wasted space problem, however. Assume, for example, that a 640K program has just finished

**Figure 6.3**

Fixed-partition memory management divides the available space into fixed-length partitions each of which holds one program.



executing (Figure 6.4). If there are no 640K programs available, the system might load a 250K program and a 300K program, but note that 90K remains unallocated. If there are no 90K or smaller programs available, the space will simply not be used. Over time, little chunks of unused space will be spread throughout memory, creating a **fragmentation** problem.

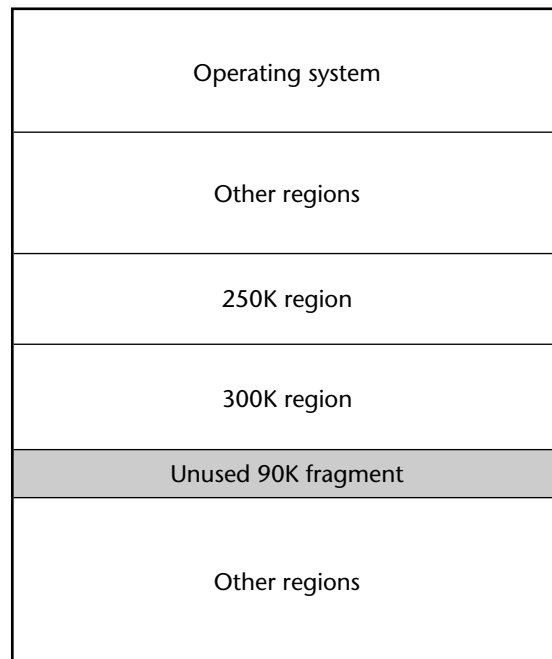
## Segmentation

One reason for the fragmentation problem is that both fixed-partition and dynamic memory management assume that a given program must be loaded into contiguous memory. With **segmentation**, programs are divided into independently addressed segments and stored in noncontiguous memory (Figure 6.5).

Segmentation requires adding a step to the address translation process. When a program is loaded into memory, the operating system builds a **segment table** listing the (absolute) entry point address of each of the program's segments (Figure 6.6). (Note that there is one segment table for each active program.) Later, when the

**Figure 6.4**

Under dynamic memory management, the transient area is treated as a pool of unstructured free space and a region of memory just sufficient to hold the program is allocated from the pool when the program is loaded. Fragmentation (bits of unused space spread throughout memory) is a possible problem.



operating system starts a given program, it loads the address of that program's segment table into a special register.

As the program runs, addresses must be translated from relative to absolute form because programmers still write the same code and compilers still generate base-plus-displacement addresses. After fetching an instruction, the instruction control unit expands each operand address by adding the base register and the displacement. Traditionally, the expanded address was an absolute address. On a segmented system, however, the expanded address consists of two parts: a segment number and a displacement (Figure 6.6).

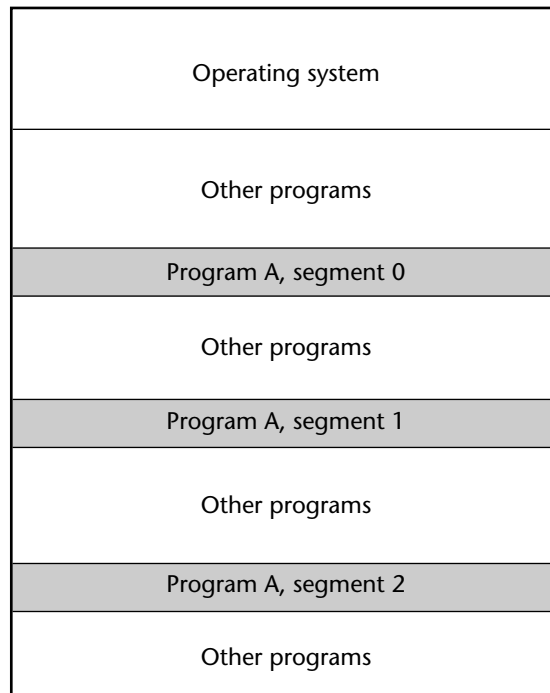
To convert the segment/displacement address to an absolute address, hardware:

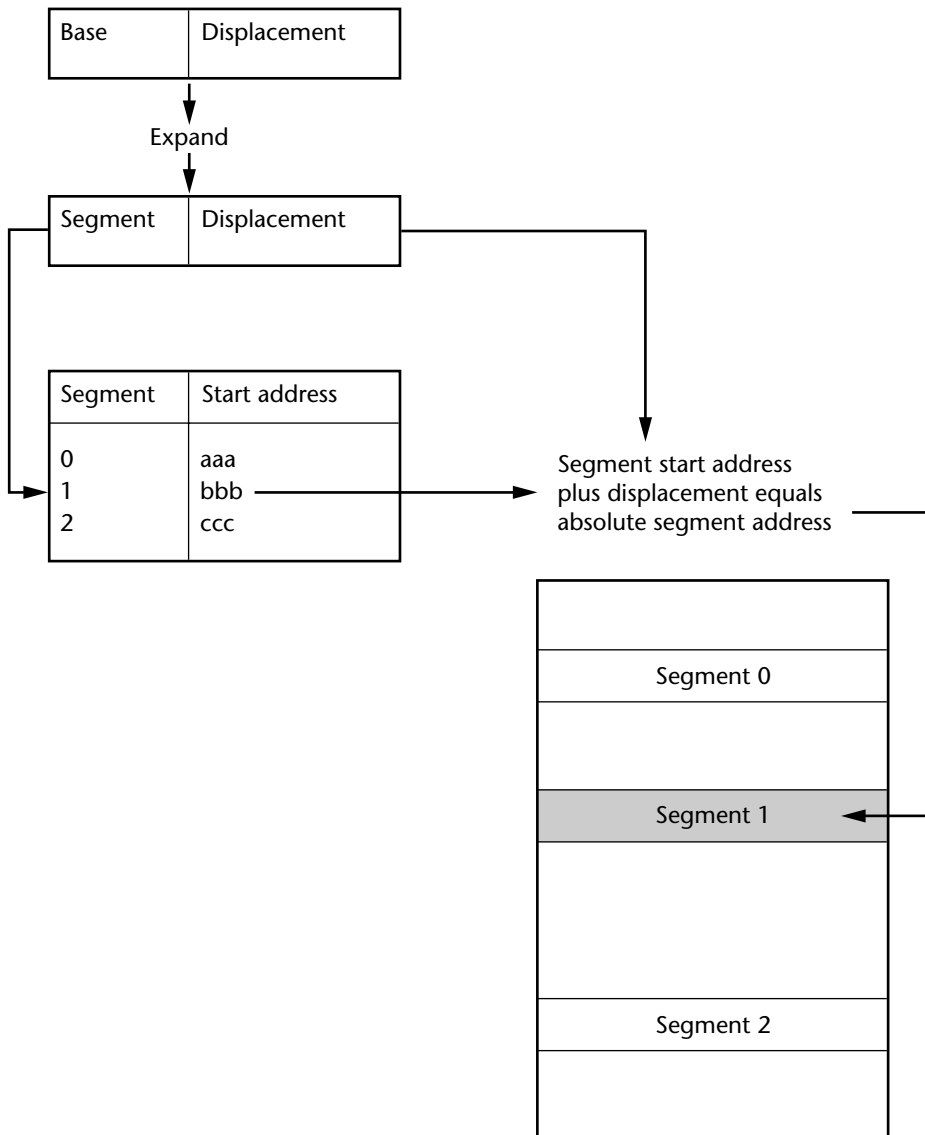
1. checks the special register to find the program's segment table,
2. extracts the segment number from the expanded address,
3. uses the segment number to search the program's segment table,
4. finds the segment's absolute entry point address, and
5. adds the displacement to the entry point address to get an absolute address.

This process (Figure 6.6) is called **dynamic address translation**.

### Figure 6.5

With segmentation, programs are divided into independently addressed segments and stored in noncontiguous memory.





**Figure 6.6**

To dynamically translate a segment address to an absolute address, break the address into segment and displacement portions, use the segment number to find the segment's absolute entry point address in a program segment table, and add the displacement to the entry point address.

## Paging

A program's segments can vary in length. Under **paging**, a program is broken into fixed-length pages. Page size is generally small (perhaps 2K to 4K), and chosen with hardware efficiency in mind.

Like segments, a program's pages are loaded into noncontiguous memory. Addresses consist of two parts (Figure 6.7), a page number in the high-order positions and a displacement in the low-order bits. Addresses are dynamically translated as the program runs. When an instruction is fetched, its base-plus-displacement addresses are expanded to absolute addresses by hardware. Then the page's base address is looked up in a program **page table** (like the segment table, maintained by the operating system) and added to the displacement.

## Segmentation *and* Paging

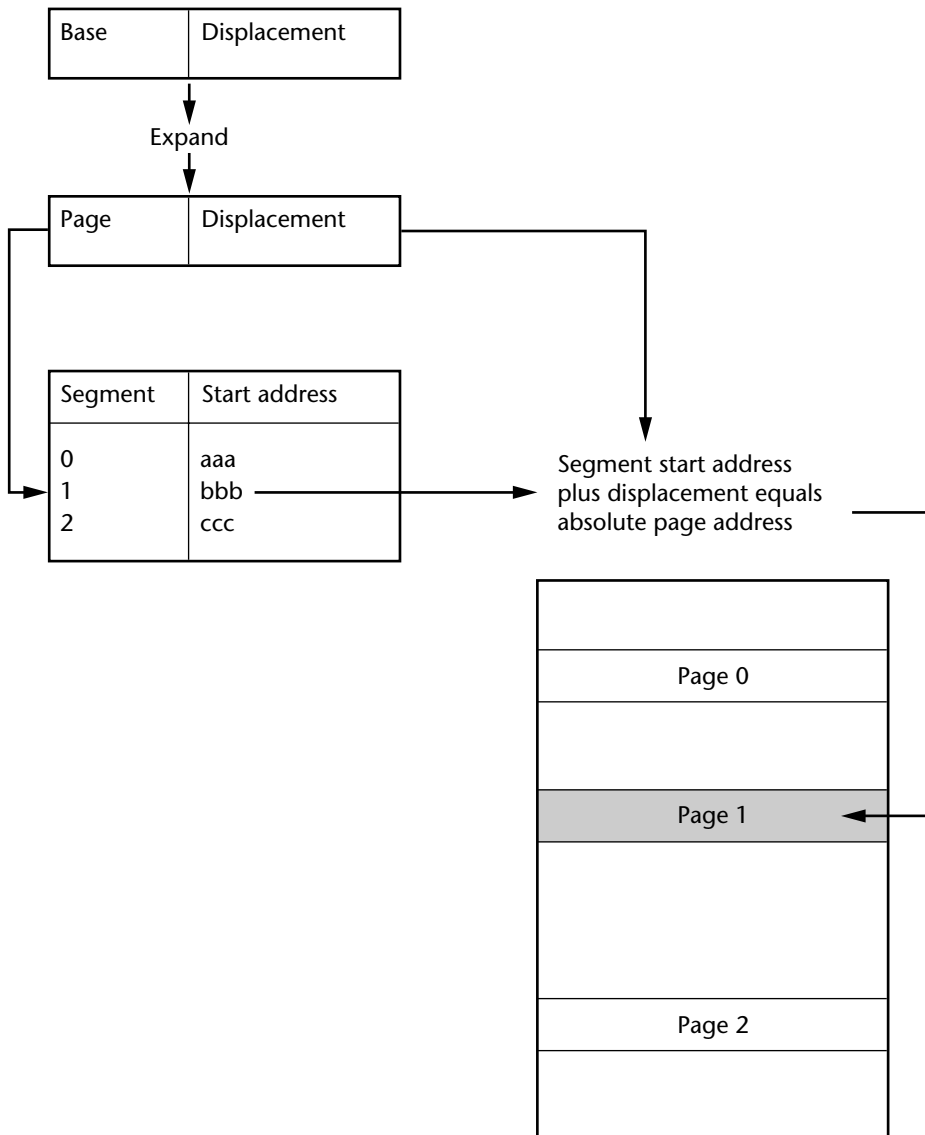
With **segmentation *and* paging**, addresses are divided into a segment number, a page number within that segment, and a displacement within that page (Figure 6.8). After the instruction control unit expands the relative address, dynamic address translation begins. First, the program's segment table is searched for the segment number, which yields the address of the segment's page table. The page table is then searched for the page's base address, which is added to the displacement to get an absolute address.

## Memory Protection

There is more to memory management than simply allocating space. The contents of RAM are easily changed. With multiple programs sharing memory, it is possible for one program to destroy the contents of memory space belonging to another, so the active programs must be protected from each other. Generally, the operating system keeps track of the space assigned to each program. If a program attempts to modify (or, sometimes, even to read) the contents of memory locations that do not belong to it, the operating system's **memory protection** routine intervenes and (usually) terminates the program.

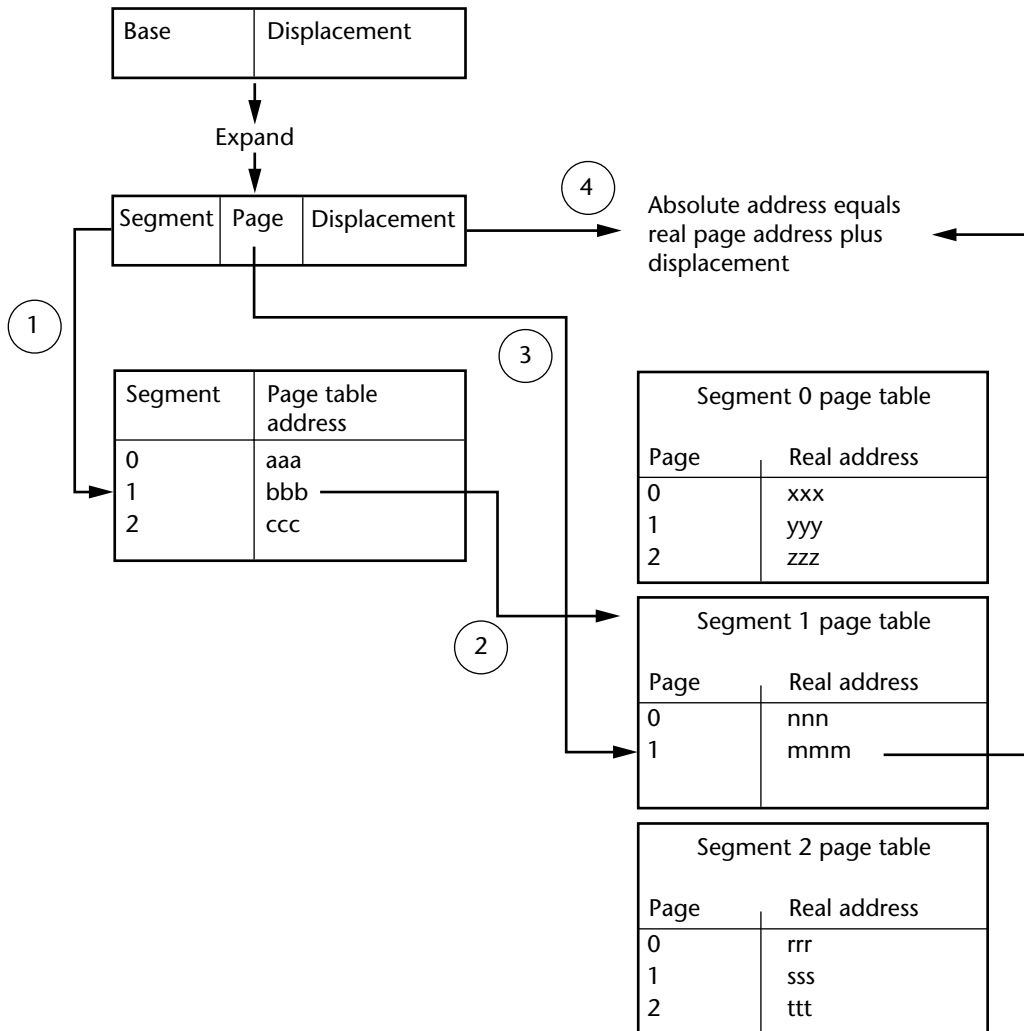
## ■ Overlay Structures

The amount of space a program needs can change as it runs, so some programs make use of **overlay** structures. The idea of overlays developed during the second generation when the amount of available memory was quite limited. The problem, in a nutshell, was how to fit a 32K program onto a 16K machine. Memory is not nearly so limited on modern computers, but overlays are still used.



**Figure 6.7**

To dynamically translate a page address to an absolute address, break the address into page and displacement portions, use the page number to find the page's absolute entry point address in a program page table, and add the displacement to the entry point address.

**Figure 6.8**

Under segmentation *and* paging, addresses are divided into three parts: a segment, a page, and a displacement. Dynamic address translation involves two table look-up operations.

The idea is to break the program into logically independent modules. For example, imagine a program with four 128K modules (Figure 6.9a). Module 1 holds the main control logic and key data common to the entire program. Module 2 processes valid input data. Occasionally, errors or unusual data values call for the logic in module 3. Module 4 generates end-of-program statistics, so it is needed only when the program terminates.

Clearly, module 1 must remain in memory at all times. If no errors are encountered, there is no need for module 3. If an error occurs, module 3's logic must be executed, but modules 2 and 4 are superfluous. Thus, as the program begins, modules 1 and 2 are in memory (Figure 6.9b). When an error is encountered, module 3 overlays module 2 (Figure 6.9c). It stays in memory until the next valid set of data is read, at which time module 2 replaces it. Finally, just before the program ends, module 4 overlays 2 or 3 (Figure 6.9d) and generates its statistics.

## ■ Virtual Memory

If a processor can execute only one instruction at a time, why is it necessary for every instruction in a program to be in memory before that program can begin executing? It isn't. You have already considered overlay structures in which only portions of a program are in memory at any given time. Those early overlay structures were precursors of modern **virtual memory** systems.

The word virtual means "not in actual fact." To the programmer or user, virtual memory acts just like real memory, but it isn't real memory.

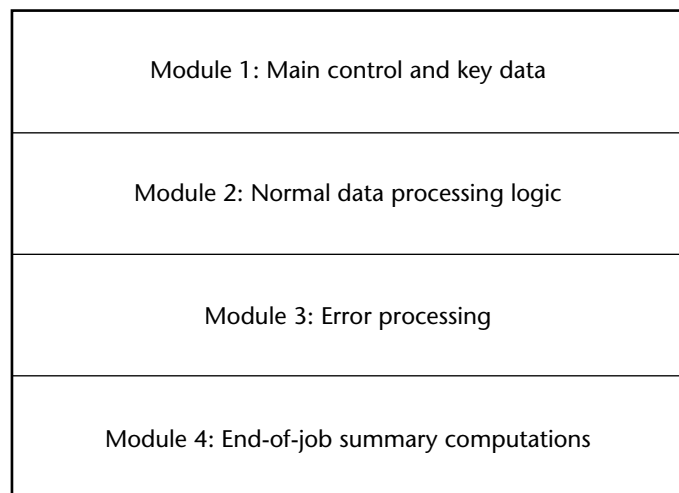
### Implementing Virtual Memory

Figure 6.10 illustrates a common approach to implementing virtual memory. It shows three levels of storage—virtual memory, the external paging device, and real memory. **Real memory** is good, old-fashioned main memory, directly addressable by the processor. The **external paging device** is usually disk. Virtual memory is a

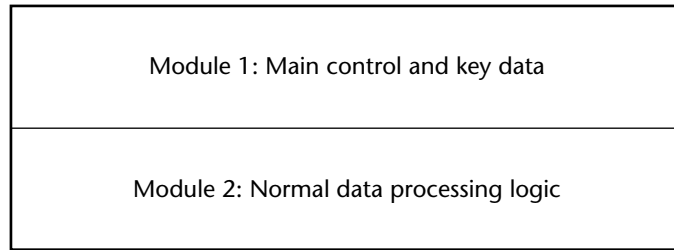
#### Figure 6.9

With overlay structures, only the necessary portions of a program are loaded into memory:

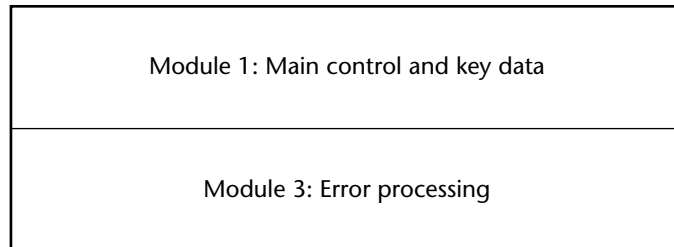
**a.** The complete program consists of four modules.



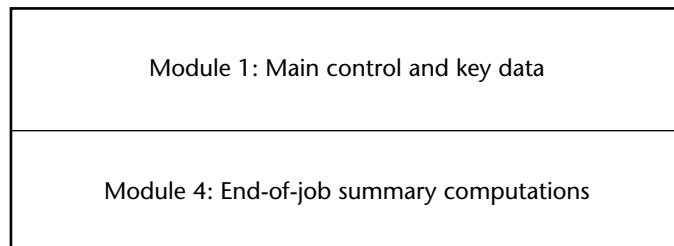
**b.** Under normal conditions, only modules 1 and 2 are in memory.



**c.** When errors occur, module 3 overlays module 2.



**d.** At end-of-job, only modules 1 and 4 are needed.

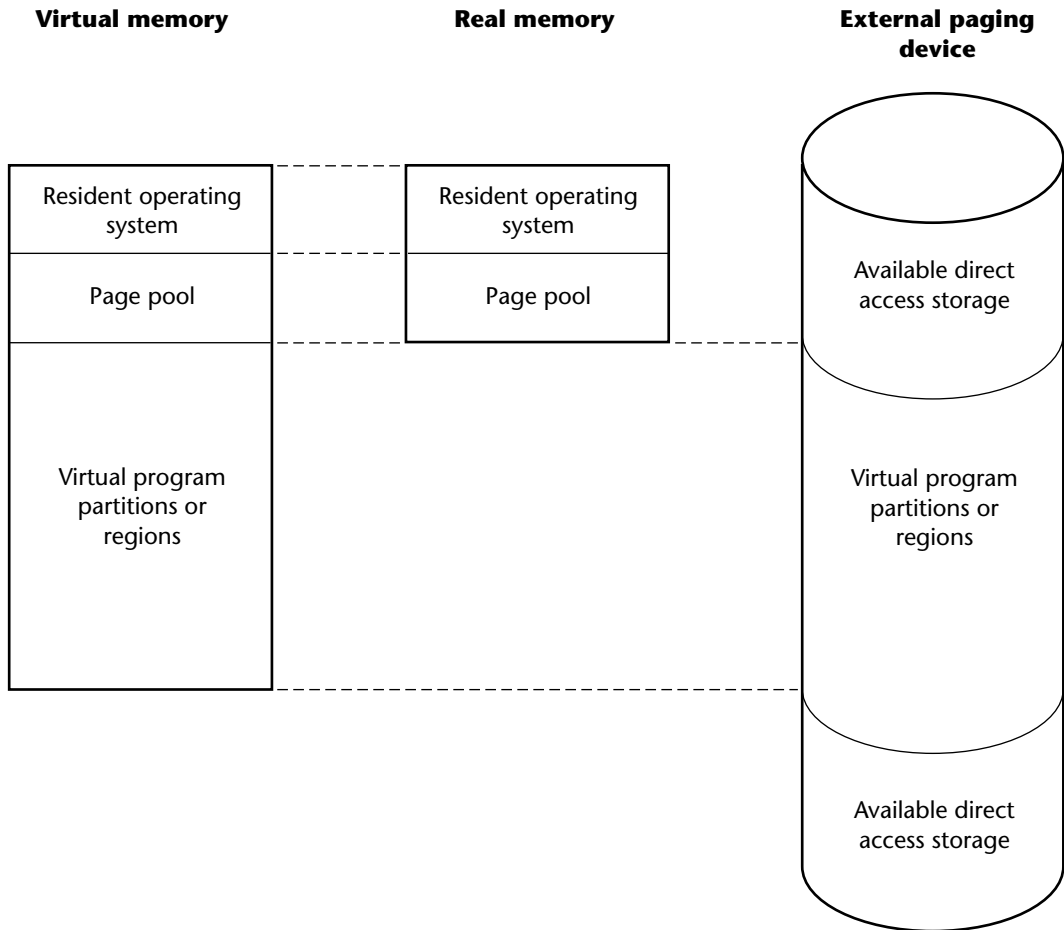


model that simplifies address translation. It “contains” the operating system and all the application programs, but it does not physically exist anywhere. Its contents are physically stored in real memory and on the external paging device.

Virtual memory is divided into two components. The first is exactly equal to the amount of real memory on the system and is physically stored in real memory. It contains the resident operating system and the transient program area (called the **page pool**).

The second component of virtual memory consists of space over and above real memory capacity. It is physically stored on the external paging device and contains the application programs. The operating system is loaded into real memory. Application programs are loaded onto the external paging device. Selected pages are then swapped between the real memory page pool and the external paging device (Figure 6.11).

Traditionally, the operating system’s memory management routine was concerned with allocating real memory space. On a virtual memory system, an equiv-



**Figure 6.10**

Virtual memory is a model that holds space for the operating system and several application programs. The operating system and selected pages occupy real memory. Application programs are stored on an external paging device.

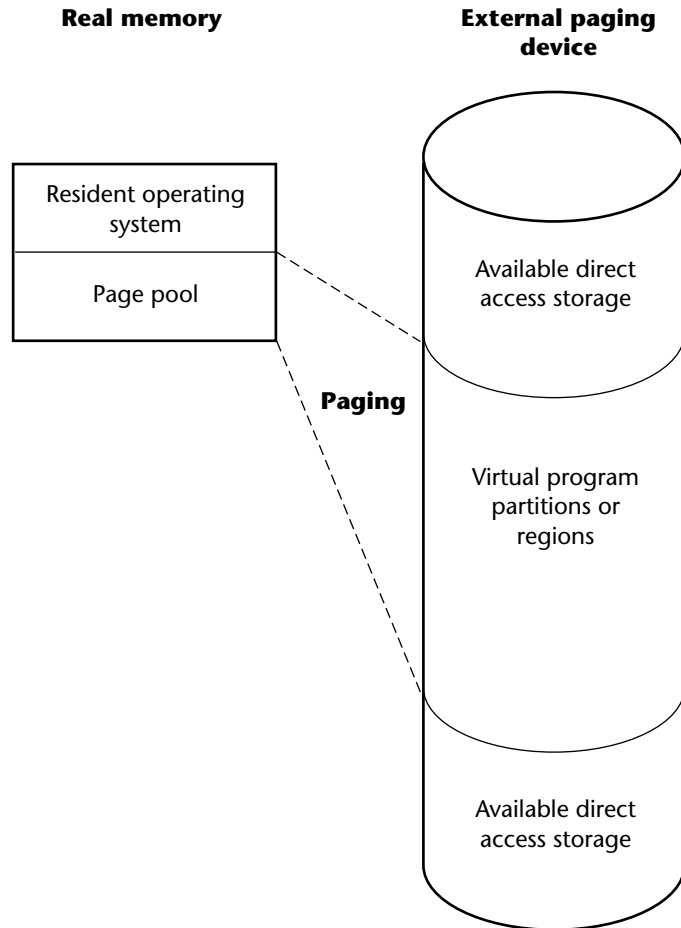
alent module allocates space on the external paging device. This space can be divided into fixed-length partitions, variable-length regions, segments, pages, or any other convenient unit. Swapping pages between the external paging device and real memory is a system function and thus is transparent to the user.

### Addressing Virtual Memory

The instructions that run on a virtual memory system are identical to the instructions that run on a regular system. The operands hold relative (base-plus-displacement) addresses. Thus, immediately after an instruction is fetched, the instruction

**Figure 6.11**

Pages are swapped between the external paging device and the real-memory page pool.



control unit expands the address by adding the displacement to the contents of a base register. On a regular system, the base register holds the program's load point in real memory. On a virtual system, however, the base register holds the program's load point in virtual memory, so the computed address reflects the page's virtual memory location.

The dynamic address translation process (which resembles segmentation and paging addressing, see Figure 6.8) starts when the program is loaded into virtual memory. The operating system allocates space on the external paging device and notes the virtual addresses of the program's segments and pages in the program's segment and page tables. Later, when a given page is swapped into real memory, the page's real address is noted in the page table.

Note that a page must be in real memory for the processor to execute its instructions. When an instruction executes, the instruction control unit (in its usual way) adds the base register and the displacement to get an address in virtual memory. To convert a virtual address to a real address, hardware:

1. accesses the program's segment table using the high-order bits of the virtual address as a key,
2. locates the program's page table using the pointer in the segment table,
3. accesses the page table to find the page's real base address using the middle bits in the virtual address as a key, and
4. adds the displacement found in the low-order bits of the virtual address to the page's real memory base address.

On most systems, the process is streamlined through the use of special registers and other hardware.

## Page Faults

When a virtual address points to a page that is not in real memory, a **page fault** is recognized and a page-in (or swap-in) operation begins. If no real memory is available for the new page, some other page must be swapped out. Often the "least currently accessed" page (the page that has gone the longest time without being referenced) is selected.

Bringing pages into memory only when they are referenced is called **demand paging**. An option called **pre-paging** involves predicting the demand for a new page and swapping it into memory before it is actually needed. Many pre-paging algorithms assume that segments hold logically related code, so if the instructions on page 1 are currently executing, the chances are that the instructions on page 2 will be executed next. While far from perfect, such techniques can significantly speed up program execution.

## Thrashing

When real memory is full, a demand for a new page means that another page must be swapped out. If this happens frequently, the system can find itself spending so much time swapping pages into and out from memory that little time is left for useful work.

This problem is called **thrashing**, and it can seriously degrade system performance. The short-term solution is removing a program or two from real memory until the system settles down. The long-term solution is to improve the real-to-virtual ratio, usually by adding more real memory.

## ■ Multiprogramming

**Multiprogramming** is a common approach to processor management when two or more programs occupy memory and execute concurrently. Originally developed to support batch-processing applications, multiprogramming takes advantage of the extreme speed disparity between a computer and its peripheral

devices. Traditionally, the key measures of effectiveness are throughput (run time divided by elapsed time) and turnaround (the time between job submission and job completion).

## The Dispatcher

Imagine two programs concurrently occupying memory. Some time ago, program A requested data from disk (Figure 6.12). Because it was unable to continue until the input operation was completed, it dropped into a **wait state** and the processor turned to program B. Assume the input operation has just been completed. Both programs are now in a **ready state**; in other words, both are ready to resume processing. Which one goes first? Computers are so fast that a human operator cannot effectively make such realtime choices. Instead, the decision is made by an operating system routine called the **dispatcher**.

Consider a system with two partitions: foreground and background. The dispatcher typically checks the program in the foreground partition first. If the program is ready, the dispatcher restarts it. Only if the foreground program is still in a wait state does the dispatcher check the background partition. The foreground has high priority; the background has low priority.

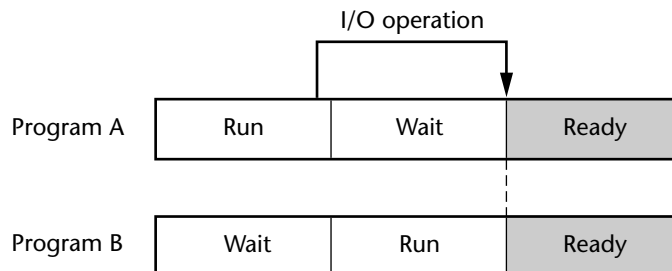
This idea can be extended to larger systems with multiple concurrent programs in memory. The dispatcher checks partitions in a fixed order until a ready state program is found. The first partition checked has highest priority; the last has lowest priority. The only way the low-priority program can execute is if all the higher priority partitions are in a wait state.

## Control Blocks

There are several control fields that must be maintained in support of each active program. Often, a **control block** is created to hold a partition's key control flags, constants, and variables (Figure 6.13). The control blocks (one per partition) are linked to form a linked list. The dispatcher typically determines which program is to start by following the chain of pointers from control block to control block. A given control block's relative position in the linked list might be determined by its priority or

**Figure 6.12**

When two or more programs are in a ready state, the operating system's dispatcher decides which one executes first.



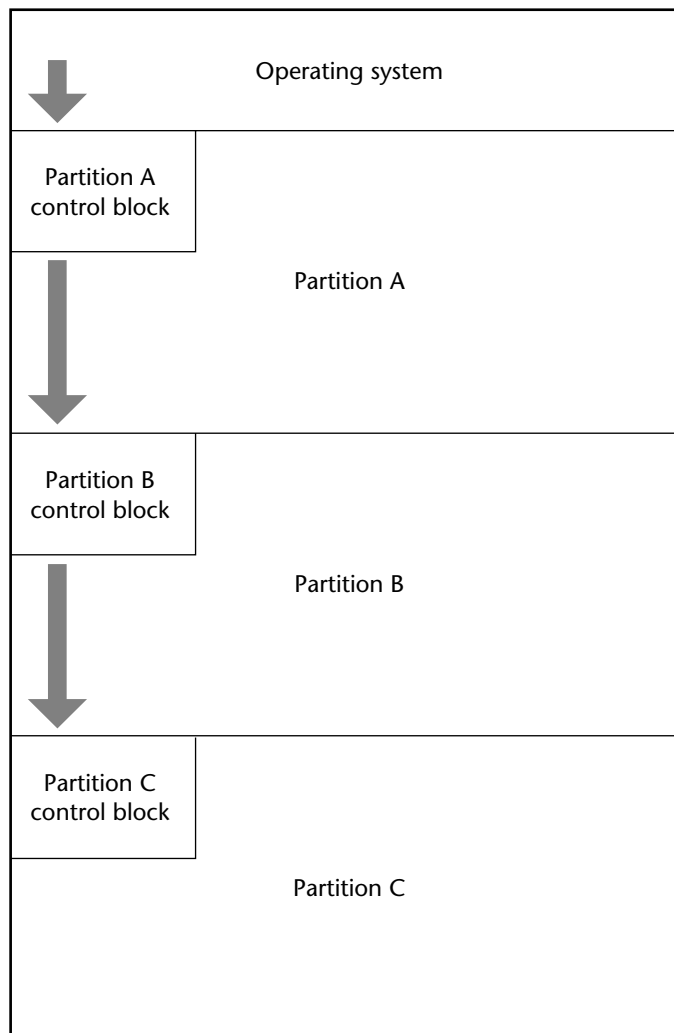
computed dynamically, perhaps taking into account such factors as program size, time in memory, peripheral device requirements, and other measures of the program's impact on system resources.

## Interrupts

A program normally surrenders control of the processor when it requests an I/O operation and is eligible to continue when that I/O operation is completed. Consequently, the key to multiprogramming is recognizing when input or output opera-

**Figure 6.13**

Information about each program is stored in the program's control block. The dispatcher determines which program to start next by following a linked list of control blocks.

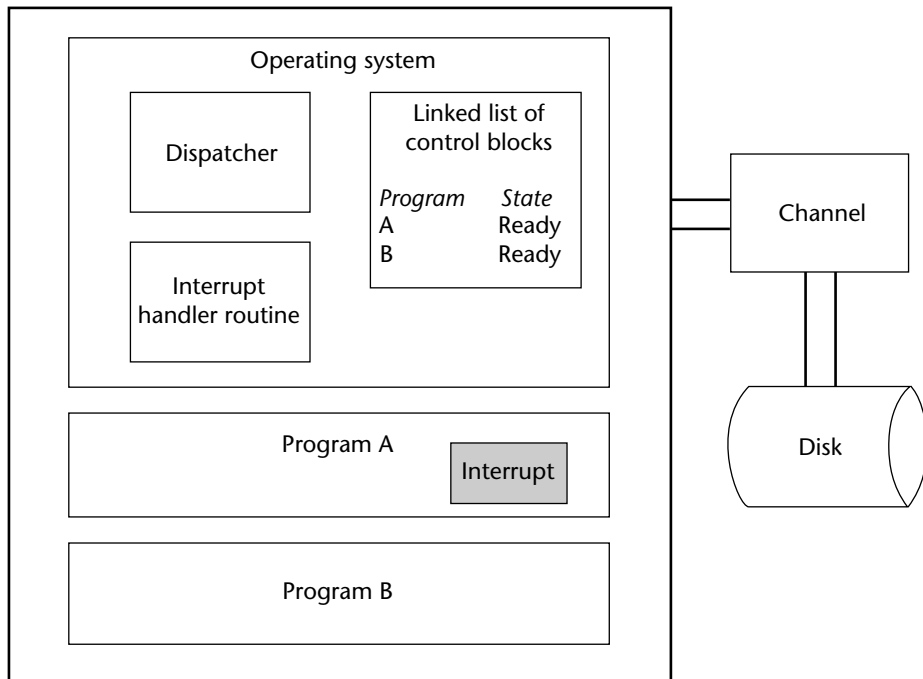


tions begin or end. The operating system knows when these events occur because they are marked by **interrupts**.

An interrupt is an electronic signal. Hardware senses the signal, saves key control information for the currently executing program, and starts the operating system's **interrupt handler** routine. At that instant, the interrupt ends. The operating system then handles the interrupt. Subsequently, after the interrupt is processed, the dispatcher starts an application program. Eventually, the program that was executing at the time of the interrupt resumes processing.

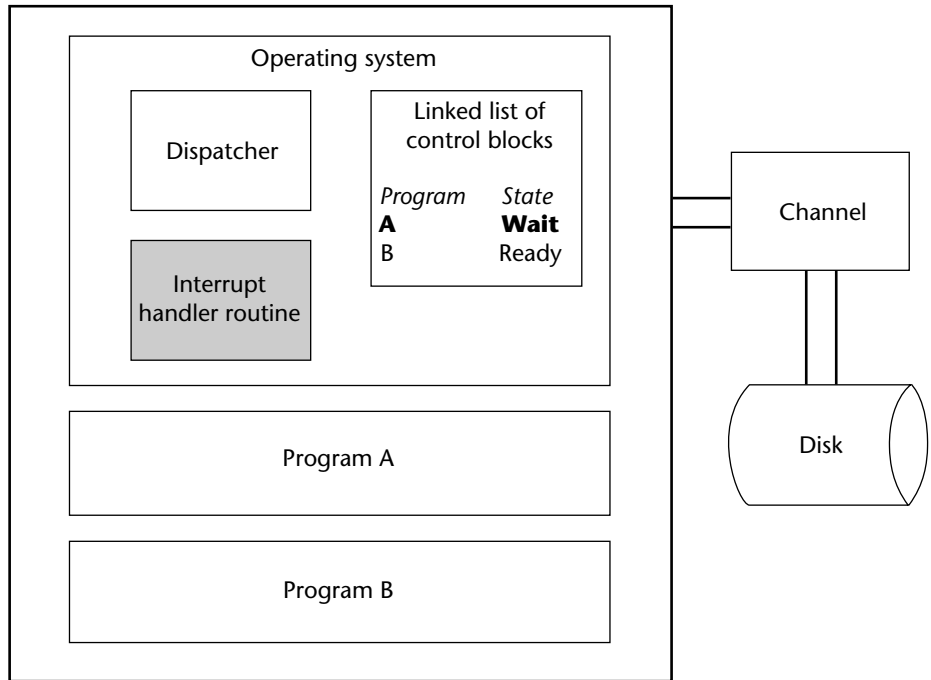
For example, follow the steps in Figure 6.14. When an application program needs data, it issues an interrupt (Figure 6.14a). In response, hardware starts the interrupt handler routine, which saves key control information, drops the application program into a wait state (Figure 6.14b), and calls the input/output control system to start the I/O operation. Finally, control flows to the dispatcher, which starts a different program (Figure 6.14c).

Later, when the I/O operation is finished, the channel issues an interrupt (Figure 6.14d). Once again the interrupt handler routine begins executing (Figure 6.14e). After verifying that the I/O operation was successfully completed, it resets the program that initially requested the data (program A) to a ready state. Then it calls the dispatcher, which starts the highest priority "ready" application program

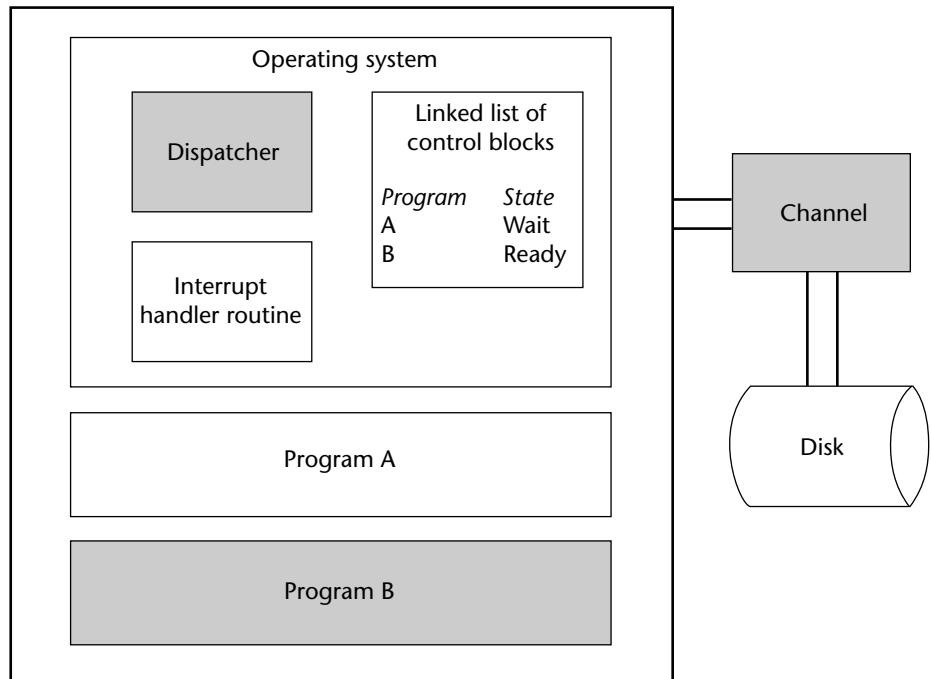


**Figure 6.14**

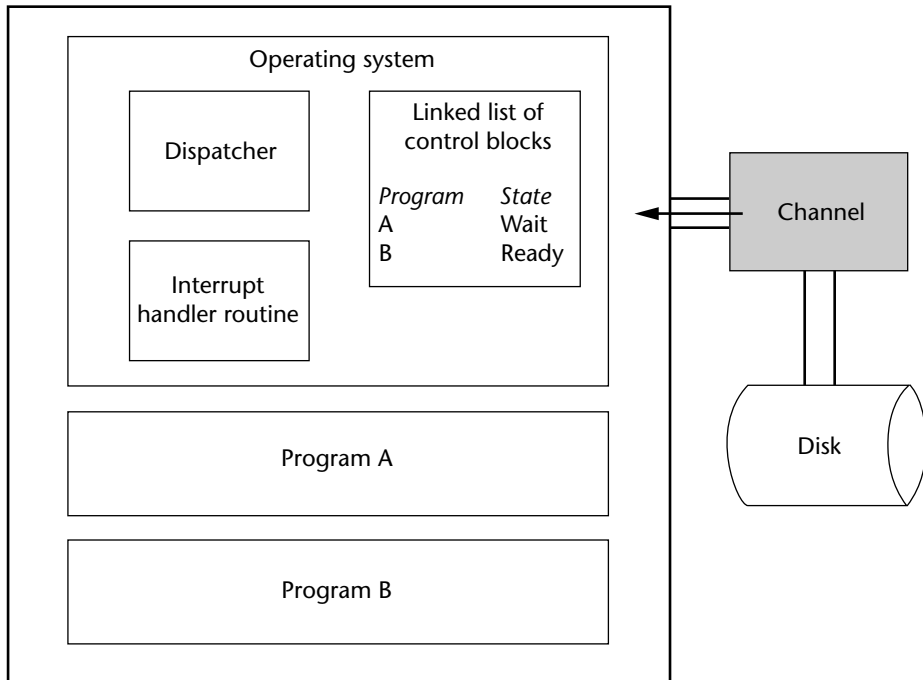
The beginning and end of each input or output operation are signaled by an interrupt: **a.** The program requests the operating system's support by issuing an interrupt.



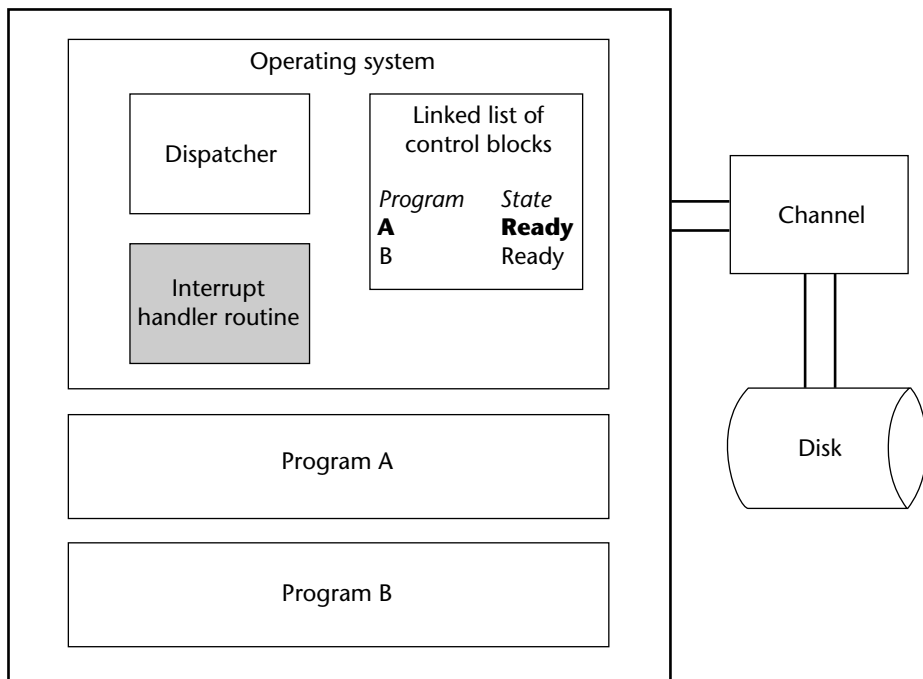
**b.** Following the interrupt, the interrupt handler routine sets the program to a wait state.



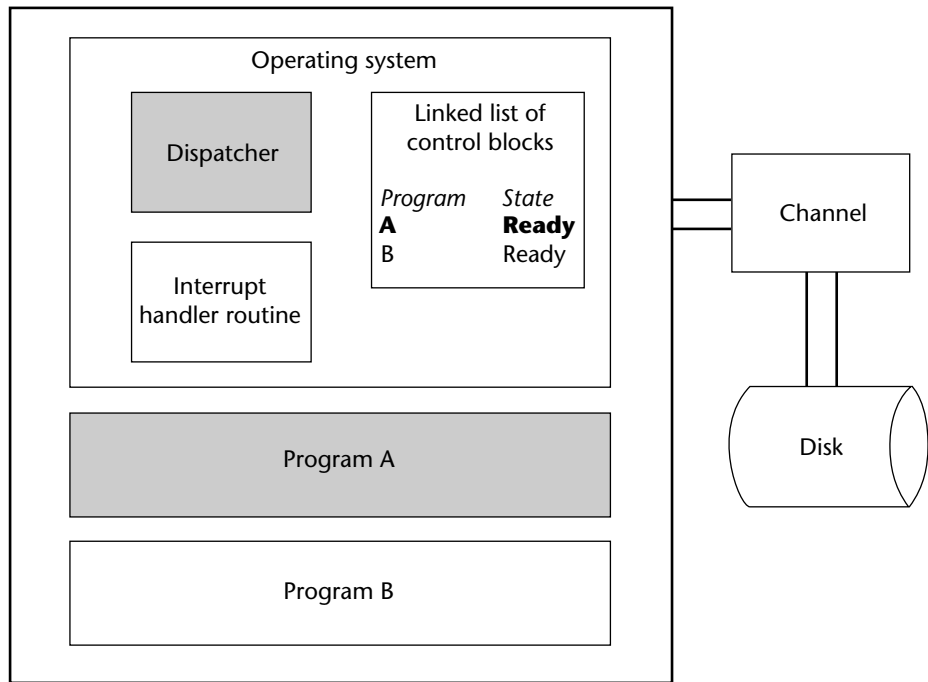
**c.** After the interrupt handler routine starts the requested input or output operation, the dispatcher starts another application program.



d. Several microseconds later, the channel signals the end of the I/O operation by sending the processor an interrupt.



e. Following the interrupt, the interrupt handler routine resets program A to a ready state.



f. After the interrupt is processed, the dispatcher selects an application program and starts it.

(Figure 6.14f). In this example, note that program A goes next even though program B was running at the time the interrupt occurred.

Interrupts can originate with either hardware or software. A program issues an interrupt to request the operating system's support (for example, to start an I/O operation). Hardware issues an interrupt to notify the processor that an asynchronous event (such as the completion of an I/O operation or a hardware failure) has occurred. Other types of interrupts might signal an illegal operation (a zero divide) or the expiration of a preset time interval.

Interrupts mark events. In response, hardware starts the interrupt handler routine, which performs the appropriate logical functions and, if necessary, resets the affected program's state (wait, ready). Following each interrupt, the dispatcher starts the highest priority ready program. That, in a nutshell, is the essence of multiprogramming.

## ■ Time-Sharing

**Time-sharing** is a different approach to managing multiple concurrent users designed with interactive processing in mind. The most important measure of effectiveness is response time, the elapsed time between entering a transaction

and seeing the system's response appear on the screen. Note, however, that time-sharing and multiprogramming are not mutually exclusive. In fact, it is not uncommon for an interactive, time-sharing system to run in the high priority partition on a large, multiprogramming mainframe.

## Roll-In/Roll-Out

Picture a typical time-sharing application. A series of brief transactions (single program statements, single lines of input data, single commands) is typed through a keyboard. In most cases, very little actual processing is required to support each transaction. Typing is (relatively speaking) slow, perhaps two transactions per minute.

To the computer, each user represents a string of brief, widely spaced processing demands. Consequently, as a given transaction is processed, the system knows that considerable time will pass before that user's next transaction arrives, so the workspace can be rolled out to secondary storage, making room for another application in memory. Later, when the first user's next transaction arrives, his or her workspace is rolled back in. Most timesharing systems use such **roll-in/roll-out** techniques to manage memory space.

## Time-Slicing

Imagine that you have just spent 20 minutes typing the data for a statistical analysis program. Each line of data was one brief transaction; your work to this point is a typical time-sharing application. Your last transaction is different, however. It is a command that tells the system to process the data, and that command initiates a computational routine that can easily run for several minutes. While your transaction is being processed, the other users on the system will have to wait, and given the objective of maintaining good response time, that is unacceptable.

The solution is **time-slicing**. Each program is restricted to a maximum "slice" of time, perhaps 0.001 second. Once a program begins executing, it runs until one of two things happens. If the program requires input or output before exhausting its time slice, it calls the operating system and "voluntarily" drops into a wait state, just like a multiprogramming application. If, however, the program uses up its entire time slice, a timer interrupt transfers control to the operating system and the time-sharing dispatcher starts the next program.

## Polling

Often, a **polling** algorithm is used to determine which program is activated next. Imagine a table of program control blocks (Figure 6.15). Starting at the top of the table, the dispatcher checks program 1's status. If program 1 is ready, the dispatcher

starts it. If not, program 2's status is checked. Assume that program 2 is ready and the dispatcher starts it.

One time slice later, program 2 is forced to surrender control of the processor. Because the last program executed was number 2, the dispatcher resumes polling with the *third* table entry; note that program 2 is now at the end of the line. Eventually, the dispatcher works its way through the entire table. At this point, it returns to the top and repeats the process. Program 2 will get another shot only after every other program has a chance.

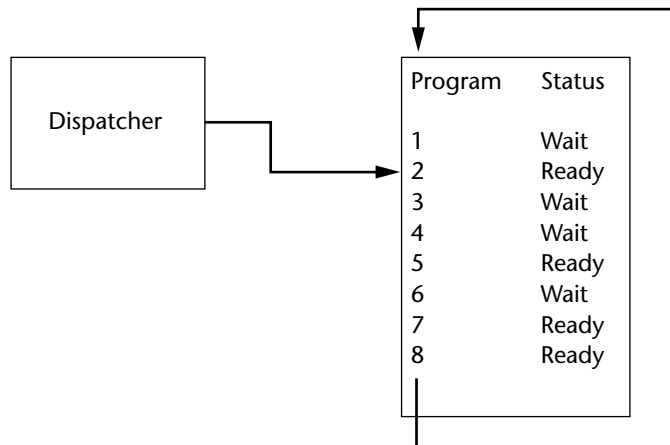
There are alternatives to simple round-robin polling. Two (or even more) tables can be maintained, with high-priority programs on the first one and background or low-priority routines on the second. Another option is to place multiple references to a crucial program on the table, thus giving it several shots at the processor on each polling cycle. Some systems use a priority scheme and recompute priorities every second or two. A common priority algorithm is dividing actual run time by elapsed residency time. The limit of this computation is 1, and that is considered low priority. The more processor time a program uses, the worse its priority becomes, so compute-bound tasks tend to drop to the end of the line.

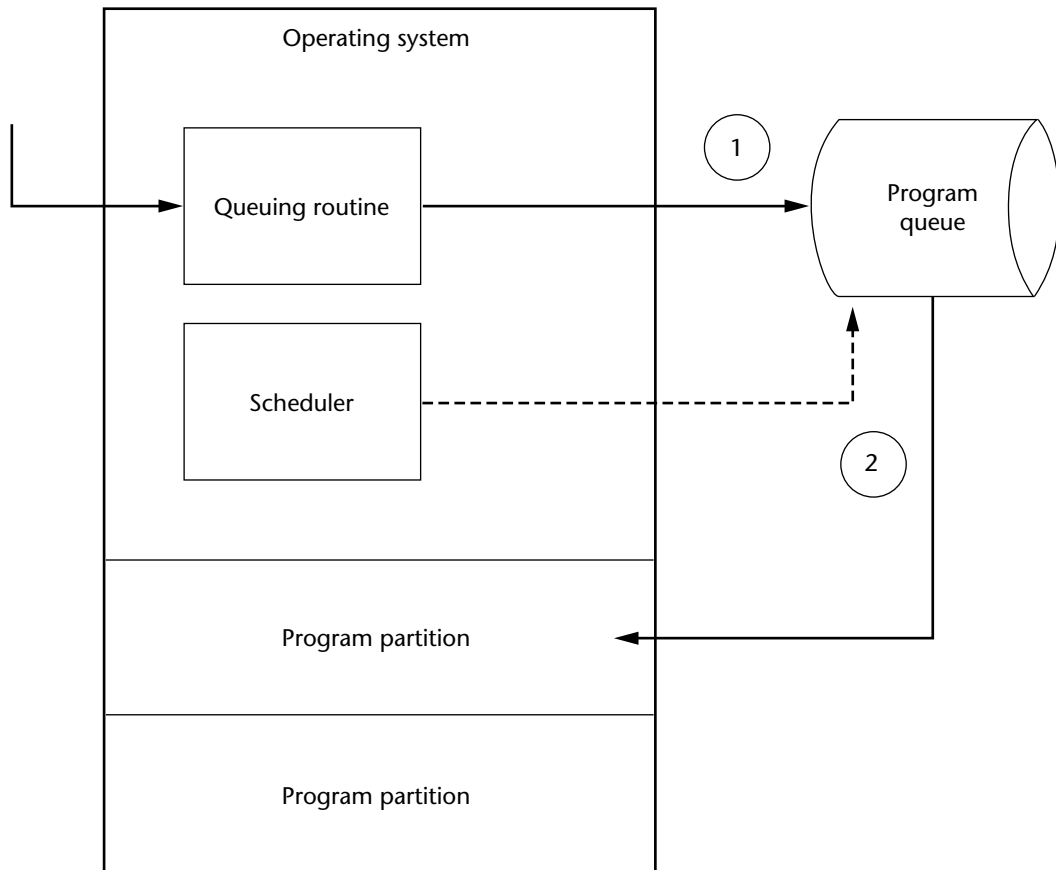
## ■ Scheduling and Queuing

Processor management is concerned with the *internal* priorities of programs already in memory. A program's *external* priority is a different issue. As one program finishes processing and space becomes available, which program is loaded into memory next? This decision typically involves two separate modules, a **queuing routine** and a **scheduler**. As programs enter the system, they are placed on a

**Figure 6.15**

Many time-sharing dispatchers rely on a polling algorithm to select the next program to start.





**Figure 6.16**

As programs enter the system, they are placed on a queue by the queuing routine. When space becomes available, the scheduler selects a program from the queue and loads it into memory.

queue by the queuing routine (Figure 6.16). When space becomes available, the scheduler selects a program from the queue and loads it into memory.

Clearly distinguish between a program's internal and external priorities. Once a program is in memory, the dispatcher uses its *internal* priority to determine its right to access the processor. In contrast, the program's *external* priority has to do with loading it into memory in the first place. Until the program is in memory, it has no internal priority. Once in memory, its external priority is no longer relevant.

## ■ Spooling

Imagine a program that generates payroll for 10,000 employees. Printing 10,000 checks takes several minutes. Why not write the checks to disk and print them later, in the background? That way, the memory allocated to the payroll program is freed for another program much more quickly.

That is the essential idea behind **spooling**. Even with multiprogramming, it is common for all application programs to be waiting for I/O. During these idle periods, the operating system's spooling module reads data from such slow devices as terminal keyboards and stores them on a high-speed medium such as disk. Later, when the program is loaded, its input data can be read from the high-speed disk. On output, data are spooled to disk and later dumped to the printer during idle periods. Because the application program deals only with high-speed I/O, it finishes processing and thus frees space for another program much more quickly.

## ■ Deadlock

**Deadlock** is one possible consequence of poor resource management. Imagine, for example, that two programs need data from the same disk. Program A issues a seek command and drops into a wait state. Subsequently, program B begins executing and issues its own seek command. Eventually, the first seek operation is completed and the dispatcher starts program A, which issues a read command. Unfortunately, the second seek command has moved the access mechanism, so A must reissue its seek command and once again drop into a wait state. Soon B issues its read command, discovers that the access mechanism is in the wrong place, and reissues its seek command.

Consider the outcome of this nightmare. Program A positions the access mechanism. Program B moves it. Program A repositions it; program B does the same thing. Picture the access mechanism moving rapidly back and forth across the disk's surface. No data are read or written. Neither program can proceed. The result is deadlock.

Deadlock is not limited to peripheral devices. It happens when two (or more) programs each control *any* resource needed by the other. Neither program can continue until the other "gives in," and if neither is willing to give in, the system, almost literally, "spins its wheels." At best, that leads to inefficiency. At worst, it can bring the entire system to a halt.

One solution is prevention; some operating systems will not load a program unless all its resource needs can be guaranteed. Other operating systems allow some deadlocks to occur, sense them, and take corrective action.

---

## Summary

Memory management is concerned with managing the computer's available pool of memory, allocating space to application routines and making sure that they do not interfere with each other. Some operating system routines directly support application programs as they run and thus must be resident. Other transient routines are stored on disk and read into memory only when needed. Generally, the operating system occupies low memory. The remaining memory, called the transient area, is where application programs and transient operating system routines are loaded.

On many modern systems, two or more programs are loaded into memory and execute concurrently. Fixed-partition memory management divides memory into fixed-length partitions and loads one program into each one. Greater efficiency can be achieved by using dynamic memory management and dividing the transient area into variable-length regions, but the inability to fully utilize available space can lead to fragmentation.

Under segmentation, a program is broken into variable-length segments that are independently loaded into noncontiguous memory. Segmented addresses are converted to absolute form by using a segment table through a process called dynamic address translation. Paging is similar to segmentation except that pages are fixed in length and dynamic address translation uses a page table. With segmentation and paging, programs are broken into logical segments and the segments subdivided into pages.

Most operating systems incorporate memory protection features that prevent one program from destroying the contents of another's memory space. With overlay structures, a program is broken into logically independent modules, and only those modules that are actually active are loaded into memory.

On a virtual memory system, programs are loaded on the external paging device (usually disk), and individual pages are paged-in to real memory as needed. Virtual memory is a logical model that contains everything traditionally stored in main memory, including the operating system and application programs. Real memory holds the operating system and a page pool. Virtual memory space above and beyond the available real memory contains the application programs. Physically, they are stored on an external paging device, which is divided into partitions or regions.

Pages are swapped between the real memory page pool and the external paging device. If a referenced page is not in real memory, a page fault is recognized and the needed page is swapped in. Bringing pages into memory only when they are referenced is called demand paging. An option called pre-paging involves predicting the demand for a new page and swapping it into memory before it is actually needed. Excessive paging can lead to thrashing, which degrades system performance.

Multiprogramming is one technique for managing the processor when multiple, concurrent programs occupy memory. Programs that are waiting for system

resources (for example, waiting for the completion of an I/O operation) are placed (by the operating system) into a wait state. Programs that are ready to process are placed into a ready state. An operating system routine called the dispatcher selects the next ready program, often by following a linked list of program control blocks. Interrupts mark the beginning and end of input and output operations, and the appropriate program's state is reset by the interrupt handler routine. Following each interrupt, the dispatcher starts the highest priority ready state program.

Time-sharing is used for interactive applications. Because the time between successive transactions is relatively lengthy, memory space can be managed by roll-in/roll-out techniques. To eliminate the risk of one program tying up the system and forcing all other users to wait, time-slicing is used to manage the processor's time. Often, the dispatcher follows a polling algorithm to determine which program to start next.

When a program first enters a system, it might be stored on a queue by a queuing routine. Later, when space becomes available, a scheduler selects the next program from the queue and loads it into memory. With spooling, data intended for a slow output device such as a printer are written instead to a high-speed device such as disk and later dumped to the slow device when the processor has time. Input data can be spooled to disk before being processed, too.

Deadlock occurs when two programs each control a resource needed by the other but neither is willing to give up its resource. Some operating systems are designed to prevent deadlock. Others sense deadlock and take corrective action.

## ■ Key Words

control block	polling
deadlock	pre-paging
demand paging	queuing routine
dispatcher	ready state
dynamic address translation	real memory
dynamic memory management	region
external paging device	resident
fixed-partition memory	roll-in/roll-out
management	scheduler
fragmentation	segment table
interrupt	segmentation
interrupt handler	segmentation and paging
memory management	spooling
memory protection	thrashing
multiprogramming	time-sharing
overlay	time-slicing
page fault	transient
page pool	transient area
page table	virtual memory
paging	wait state
partition	

## ■ Exercises

1. Distinguish between resident and transient routines. What is the transient area?
2. Distinguish between concurrent and simultaneous. A single processor can execute two or more programs concurrently but not simultaneously. Why?
3. Distinguish fixed-partition memory management and dynamic memory management.
4. Segmentation and/or paging can help to minimize fragmentation because programs are loaded into noncontiguous memory. Briefly explain what that means.
5. Distinguish between segmentation and paging.
6. Explain segmentation *and* paging.
7. Briefly explain dynamic address translation.
8. Why is memory protection necessary?
9. What is an overlay structure?
10. How does a virtual memory system work? Distinguish between virtual memory, the external paging device, and real memory.
11. What is a page fault? Distinguish between demand paging and pre-paging.
12. Briefly explain thrashing.
13. What is multiprogramming?
14. Briefly explain how a multiprogramming operating system's dispatcher manages the processor. What are control blocks and why are they necessary?
15. What is an interrupt? What is the interrupt handler routine? Briefly explain how a multiprogramming system's dispatcher relies on interrupts to manage the processor.
16. Distinguish between multiprogramming and time-sharing.
17. Briefly explain roll-in/roll-out, time-slicing, and polling.
18. Explain how the queuing routine and the scheduler work together to load application programs. Distinguish between a program's internal and external priorities.
19. What is spooling?
20. What is deadlock?