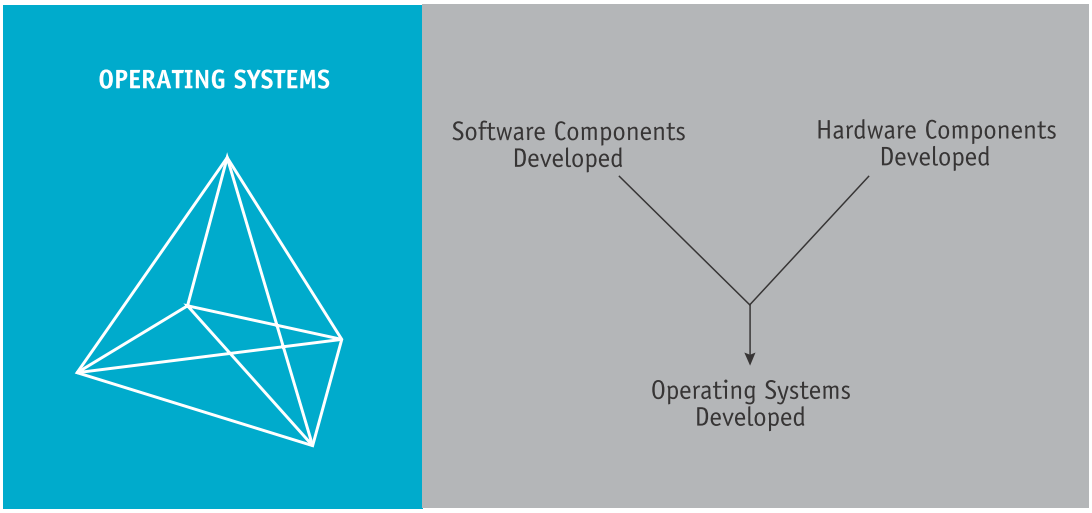
Chapter 1 Introducing Operating Systems



"I think there is a world market for maybe five computers."

—attributed to Thomas J. Watson (1874–1956; chairman of IBM 1949–1956)

Learning Objectives

After completing this chapter, you should be able to describe:

- Innovations in operating systems development
- The basic role of an operating system
- The major operating system software subsystem managers and their functions
- The types of machine hardware on which operating systems run
- The differences among batch, interactive, real-time, hybrid, and embedded operating systems
- Design considerations of operating systems designers

Introduction

To understand an operating system is to begin to understand the workings of an entire computer system, because the operating system software manages each and every piece of hardware and software. In the pages that follow, we explore what operating systems are, how they work, what they do, and why.

This chapter briefly describes the workings of operating systems on the simplest scale. The following chapters explore each component in more depth and show how its function relates to the other parts of the operating system. In other words, you see how the pieces work together harmoniously to keep the computer system working smoothly.

What Is an Operating System?

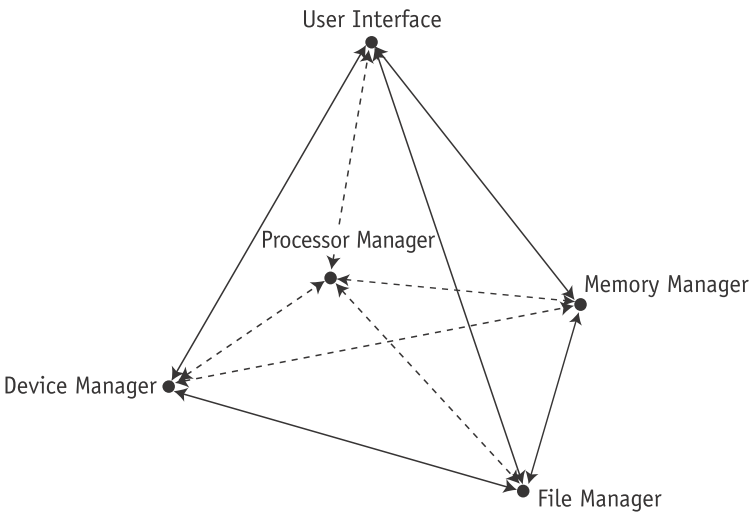
A computer system typically consists of software (programs) and hardware (the tangible machine and its electronic components). The operating system software is the chief piece of software, the portion of the computing system that manages all of the hardware and all of the other software. To be specific, it controls every file, every device, every section of main memory, and every moment of processing time. It controls who can use the system and how. In short, the operating system is the boss.

Therefore, each time the user sends a command, the operating system must make sure that the command is executed, or if it's not executed, it must arrange for the user to get a message explaining the error. Remember: this doesn't necessarily mean that the operating system executes the command or sends the error message—but it does control the parts of the system that do.

Operating System Software

The pyramid shown in Figure 1.1 is an abstract representation of the operating system in its simplest form and demonstrates how its major components typically work together.

At the base of the pyramid are the four essential managers of every major operating system: the **Memory Manager, Processor Manager, Device Manager,** and **File Manager.** These managers and their interactions are discussed in detail in Chapters 1 through 8 of this book. Each manager works closely with the other managers as each one performs its unique role. At the top of the pyramid is the User Interface, which allows the user to issue commands to the operating system. Because this component has specific elements, in both form and function, it is often very different from one operating system to the next—sometimes even between different versions of the same operating system.

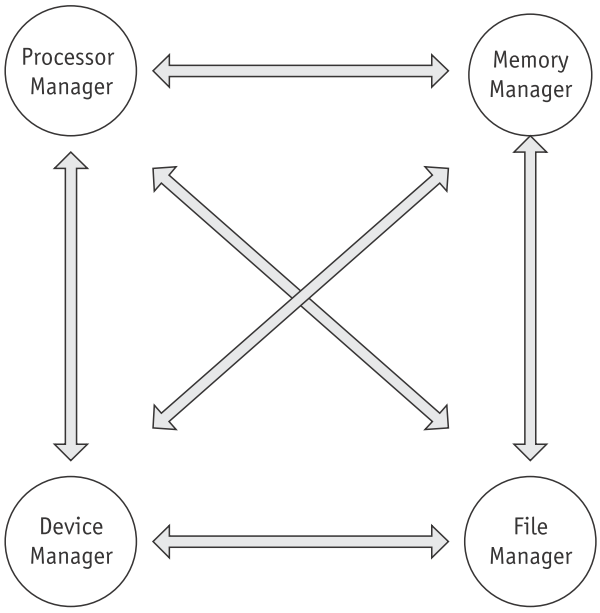


(figure 1.1)

This pyramid represents an operating system on a stand-alone computer unconnected to a network. It shows the four subsystem managers and the user interface.

Regardless of the size or configuration of the system, the four managers illustrated in Figure 1.2 must, at a minimum, perform the following tasks while collectively keeping the system working smoothly:

- Monitor the system's resources
- Enforce the policies that determine what component gets what resources, when, and how much
- Allocate the resources when appropriate
- Deallocate the resources when appropriate



(figure 1.2)

Each manager at the base of the pyramid takes responsibility for its own tasks while working harmoniously with every other manager.

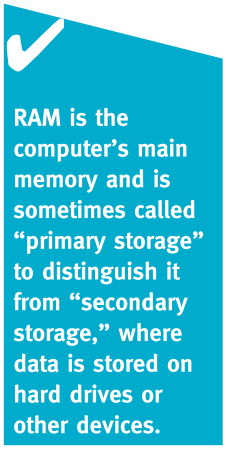
For example, the Memory Manager must keep track of the status of the computer system's main memory space, allocate the correct amount of it to incoming processes, and deallocate that space when appropriate—all while enforcing the policies that were established by the designers of the operating system.

An additional management task, networking, was not always an integral part of operating systems. Today the vast majority of major operating systems incorporate a **Network Manager** to coordinate the services required for multiple systems to work cohesively together. For example, the Network Manager must coordinate the workings of the networked resources, which might include shared access to memory space, processors, printers, databases, monitors, applications, and more. This can be a complex balancing act as the number of resources increases, as it often does.

Main Memory Management

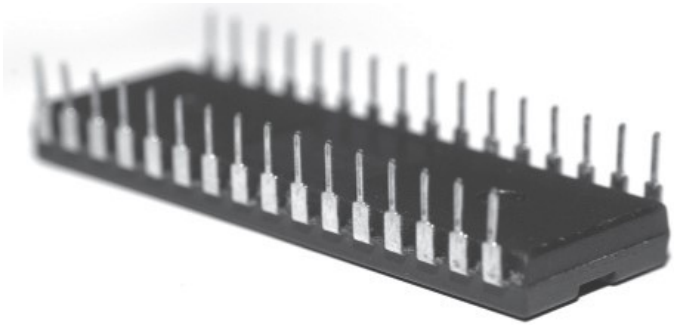
The Memory Manager (the subject of Chapters 2 and 3) is in charge of main memory, widely known as **RAM** (short for random access memory). The Memory Manager checks the validity of each request for memory space, and if it is a legal request, allocates a portion of memory that isn't already in use. If the memory space becomes fragmented, this manager might use policies established by the operating systems designers to reallocate memory to make more useable space available for other jobs that are waiting. Finally, when the job or process is finished, the Memory Manager deallocates its allotted memory space.

A key feature of RAM chips—the hardware that comprises computer memory—is that they depend on the constant flow of electricity to hold data. When the power fails or is turned off, the contents of RAM is wiped clean. This is one reason why computer system designers attempt to build elegant shutdown procedures, so the contents of RAM can be stored on a nonvolatile device, such as a hard drive, before the main memory chips lose power during computer shutdown.



A critical responsibility of the Memory Manager is to protect all of the space in main memory, particularly that occupied by the operating system itself—it can't allow any part of the operating system to be accidentally or intentionally altered because that would lead to instability or a system crash.

Another kind of memory that's critical when the computer is powered on is **Read-Only Memory** (often shortened to **ROM**), shown in Figure 1.3. This ROM chip holds software called **firmware**, the programming code that is used to start the computer and perform other necessary tasks. To put it in simplest form, it describes in prescribed steps when and how to load each piece of the operating system after the power is turned on and until the computer is ready for use. The contents of the ROM chip are nonvolatile, meaning that they are not erased when the power is turned off, unlike the contents of RAM.



(figure 1.3)

A computer's relatively small ROM chip contains the firmware (unchanging software) that prescribes system initialization when the system powers on.

Processor Management

The Processor Manager (discussed in Chapters 4 through 6) decides how to allocate the central processing unit (CPU); an important function of the Processor Manager is to keep track of the status of each job, process, thread, and so on. We will discuss all of these in the chapters that follow, but for this overview, let's limit our discussion to **processes** and define them as a program's "instance of execution." A simple example could be a request to solve a mathematical equation: this would be a single job consisting of several processes, with each process performing a part of the overall equation.

The Processor Manager is required to monitor the computer's CPU to see if it's busy executing a process or sitting idle as it waits for some other command to finish execution. Generally, systems are more efficient when their CPUs are kept busy. The Processor Manager handles each process's transition, from one state of execution to another, as it moves from the starting queue, through the running state, and finally to the finish line (where it then tends to the next process). Therefore, this manager can be compared to a traffic controller. When the process is finished, or when the maximum amount of computation time has expired, the Processor Manager reclaims the CPU so it can allocate it to the next waiting process. If the computer has multiple CPUs, as in a multicore system, the Process Manager's responsibilities are greatly complicated.

Device Management

A flash memory device is an example of secondary storage because it doesn't lose data when its power is turned off.

The Device Manager (the subject of Chapter 7) is responsible for connecting with every device that's available on the system and for choosing the most efficient way to allocate each of these printers, ports, disk drives, and more, based on the device scheduling policies selected by the designers of the operating system.

Good device management requires that this part of the operating system uniquely identify each device, start its operation when appropriate, monitor its progress, and finally deallocate the device to make the operating system available to the next waiting process. This isn't as easy as it sounds because of the exceptionally wide range of devices

that can be attached to any system. For example, let's say you're adding a printer to your system. There are several kinds of printers commonly available (laser, inkjet, inkless thermal, etc.) and they're made by manufacturers that number in the hundreds or thousands. To complicate things, some devices can be shared, while some can be used by only one user or one job at a time. Designing an operating system to manage such a wide range of printers (as well as monitors, keyboards, pointing devices, disk drives, cameras, scanners, and so on) is a daunting task. To do so, each device has its own software, called a **device driver**, which contains the detailed instructions required by the operating system to start that device, allocate it to a job, use the device correctly, and deallocate it when it's appropriate.

File Management

The File Manager (described in Chapter 8), keeps track of every file in the system, including data files, program files, utilities, compilers, applications, and so on. By following the access policies determined by the system designers, the File Manager enforces restrictions on who has access to which files. Many operating systems allow authorized individuals to change those permissions and restrictions. The File Manager also controls the range of actions that each user is allowed to perform with files after they access them. For example, one user might have read-only access to a critical database, while the systems administrator might hold read-and-write access and the authority to create and delete files in the same database. Access control is a key part of good file management and is tightly coupled with system security software.

When the File Manager allocates space on a secondary storage device (such as a hard drive, flash drive, archival device, and so on), it must do so knowing the technical requirements of that device. For example, if it needs to store an archival copy of a large file, it needs to know if the device stores it more efficiently as one large block or in several smaller pieces that are linked through an index. This information is also necessary for the file to be retrieved correctly later. Later, if this large file must be modified after it has been stored, the File Manager must be capable of making those modifications accurately and as efficiently as possible.

Network Management

Operating systems with networking capability have a fifth essential manager called the Network Manager (the subject of Chapters 9 and 10) that provides a convenient way for authorized users to share resources. To do so, this manager must take overall responsibility for every aspect of network connectivity, including the requirements of the available devices as well as files, memory space, CPU capacity, transmission connections, and types of encryption (if necessary). Networks with many available

resources require management of a vast range of alternative elements, which enormously complicates the tasks required to add network management capabilities.

Networks can range from a small wireless system that connects a game system to the Internet, to a private network for a small business, to one that connects multiple computer systems, devices, and mobile phones to the Internet. Regardless of the size and complexity of the network, these operating systems must be prepared to properly manage the available memory, CPUs, devices, and files.

User Interface

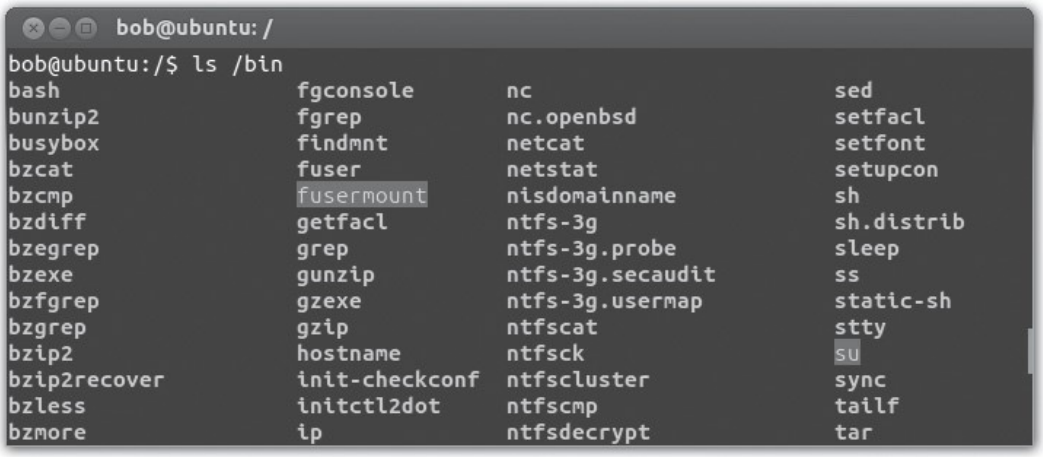
The user interface—the portion of the operating system that users interact with directly—is one of the most unique components of an operating system. Two primary types are the **graphical user interface** (GUI) shown in Figure 1.4 and the **command line interface.** The GUI relies on input from a pointing device such as a mouse or your finger. Specific menu options, desktops, and formats often vary widely from one operating system to another (and sometimes from one version to another).

The alternative to a GUI is a command line interface, which responds to specific commands typed on a keyboard and displayed on the monitor, as shown in Figure 1.5. These interfaces accept typed commands and offer skilled users powerful additional control because typically the commands can be linked together (concatenated) to perform complex tasks with a single multifunctional command that would require many mouse clicks to duplicate using a graphical interface.

While a command structure offers powerful functionality, it has strict requirements for every command: each must be typed accurately, each must be formed in correct syntax, and combinations of commands must be assembled correctly. In addition, users need to know how to recover gracefully from any errors they encounter. These command line interfaces were once standard for operating systems and are still favored by power users but have largely been supplemented with simple, forgiving graphical user interfaces.

(figure 1.4)

An example of the graphical user interface (GUI) for Ubuntu Linux.



(figure 1.5)

Using the Linux command line interface to show a partial list of valid commands.

Cooperation Issues

None of the elements of an operating system can perform its individual tasks in isolation—each must also work harmoniously with every other manager. To illustrate this using a very simplified example, let's follow the steps as someone chooses a menu option to open a program. The following series of major steps are typical of the discrete actions that would occur in fractions of a second as a result:

1. The Device Manager receives the electrical impulse caused by a click of the mouse, decodes the command by calculating the location of the cursor, and sends that information through the User Interface, which identifies the requested command. Immediately, it sends the command to the Processor Manager.
2. The Processor Manager then sends an acknowledgment message (such as "waiting" or "loading") to be displayed on the monitor so the user knows that the command has been sent successfully.
3. The Processor Manager determines whether the user request requires that a file (in this case a program file) be retrieved from storage or whether it is already in memory.
4. If the program is in secondary storage (perhaps on a disk), the File Manager calculates its exact location on the disk and passes this information to the Device Manager, which retrieves the program and sends it to the Memory Manager.
5. If necessary, the Memory Manager finds space for the program file in main memory and records its exact location. Once the program file is in memory, this manager keeps track of its location in memory.
6. When the CPU is ready to run it, the program begins execution by the Processor Manager. When the program has finished executing, the Processor Manager relays this information to the other managers.
7. The Processor Manager reassigns the CPU to the next program waiting in line. If the file was modified, the File Manager and Device Manager cooperate to

store the results in secondary storage. (If the file was not modified, there's no need to change the stored version of it.)

8. The Memory Manager releases the program's space in main memory and gets ready to make it available to the next program to require memory.
9. Finally, the User Interface displays the results and gets ready to take the next command.

Although this is a vastly oversimplified demonstration of a very fast and complex operation, it illustrates some of the incredible precision required for an operating system to work smoothly. The complications increase greatly when networking capability is added. Although we'll be discussing each manager in isolation for much of this text, remember that no single manager could perform its tasks without the active cooperation of every other manager.

Cloud Computing

One might wonder how **cloud computing** (in simplest terms, this is the practice of using Internet-connected resources to perform processing, storage, or other operations) changes the role of operating systems. Generally, cloud computing allows the operating systems to accommodate remote access to system resources and provide increased security for these transactions. However, at its roots, the operating system still maintains responsibility for managing all local resources and coordinating data transfer to and from the cloud. And the operating system that is managing the far-away resources is responsible for the allocation and deallocation of all its resources—this time on a massive scale. Companies, organizations, and individuals are moving a wide variety of resources to the cloud, including data management, file storage, applications, processing, printing, security, and so on, and one can expect this trend to continue. But regardless of where the resource is located—in the box, under the desk, or the cloud, the role of the operating system is the same—to access those resources and manage the system as efficiently as possible.

An Evolution of Computing Hardware

To appreciate the role of the operating system (which is software), it may help to understand the computer system's **hardware**, which is the physical machine and its electronic components, including memory chips, the central processing unit (CPU), the input/output devices, and the storage devices.

- **Main memory** (**RAM**) is where the data and instructions must reside to be processed.
- The **central processing unit** (**CPU**) is the "brains" of the computer. It has the circuitry to control the interpretation and execution of instructions. All storage

Platform	Operating System
Telephones, tablets	Android, iOS, Windows
Laptops, desktops	Linux, Mac OS X, UNIX, Windows
Workstations, servers	Linux, Mac OS X Server, UNIX, Windows Server
Mainframe computers	Linux, UNIX, Windows, IBM z/OS
Supercomputers	Linux, UNIX

(table 1.1)

A brief list of platforms and a few of the operating systems designed to run on them, listed in alphabetical order.

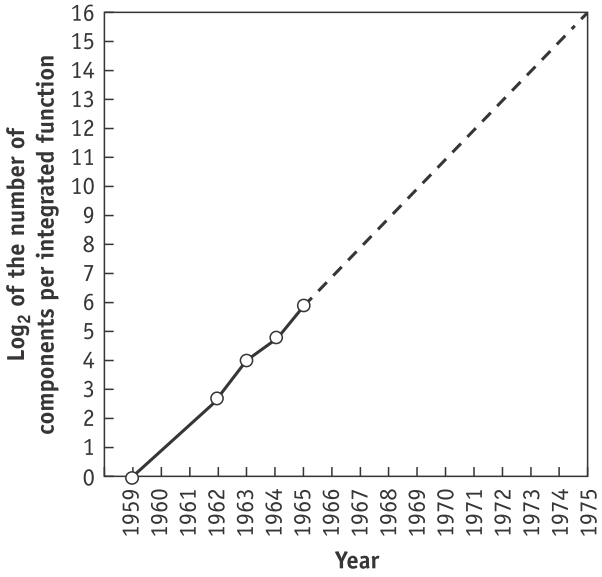
references, data manipulations, and input/output operations are initiated or performed by the CPU.

- Devices, sometimes called I/O devices for input/output devices, include every peripheral unit attached to the computer system, from printers and monitors to magnetic disk and optical disc drives, , flash memory, keyboards, and so on.

At one time, computers were classified by memory capacity, which is no longer the case. A few of the operating systems that can be used on a variety of platforms are shown in Table 1.1.

Mac OS X, HP-UX, and Sun Solaris are only three of many operating systems that are based on UNIX.

In 1965, Intel executive Gordon Moore observed that each new processor chip contained roughly twice as much capacity as its predecessor (number of components per integrated function), and that each chip was released within 18–24 months of the previous chip. His original paper included a graph (shown in Figure 1.6) predicting that the trend would cause computing power to rise exponentially over relatively brief periods of time, and it has. Now known as Moore's Law, the trend has continued and



Number of components per integrated function for minimum cost per component extrapolated vs time.

(figure 1.6)

Gordon Moore's 1965 paper included the prediction that the number of transistors incorporated in a chip will approximately double every 24 months [Moore, 1965].

Courtesy of Intel Corporation.

is still remarkably accurate. Moore's Law is often cited by industry observers when making their chip capacity forecasts.

Types of Operating Systems

Operating systems fall into several general categories distinguished by the speed of their response and the method used to enter data into the system. The five categories are batch, interactive, real-time, hybrid, and embedded systems.

Batch systems feature jobs that are entered as a whole and in sequence. That is, only one job can be entered at a time, and once a job begins processing, then no other job can start processing until the resident job is finished. These systems date from early computers, when each job consisted of a stack of cards—or reels of magnetic tape—for input and were entered into the system as a unit, called a batch. The efficiency of a batch system is measured in **throughput**, which is the number of jobs completed in a given amount of time (usually measured in minutes, hours, or days.)

Interactive systems allow multiple jobs to begin processing and return results to users with better response times than batch systems, but interactive systems are slower than the real-time systems we talk about next. Early versions of these operating systems allowed each user to interact directly with the computer system via commands entered from a typewriter-like terminal, and the operating system used complex algorithms to share processing power (often with a single processor) among the jobs awaiting processing. Interactive systems offered huge improvements in response over batch-only systems with **turnaround times** in seconds or minutes instead of hours or days.

Real-time systems are used in time-critical environments where reliability is critical and data must be processed within a strict time limit. This time limit need not be ultra-fast (though it often is), but system response time must meet the deadline because there are significant consequences of not doing so. They also need to provide contingencies to fail gracefully—that is, preserve as much of the system's capabilities and data as possible to facilitate recovery. Examples of real-time systems are those used for spacecraft, airport traffic control, fly-by-wire aircraft, critical industrial processes, and medical systems, to name a few. There are two types of real-time systems depending on the consequences of missing the deadline: hard and soft systems.

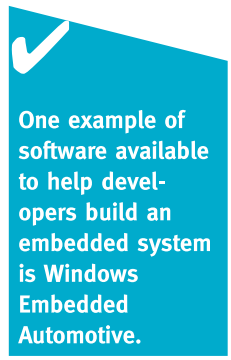
- Hard real-time systems risk total system failure if the predicted time deadline is missed.
- Soft real-time systems suffer performance degradation, but not total system failure, as a consequence of a missed deadline.

Although it's theoretically possible to convert a general-purpose operating system into a real-time system by merely establishing a deadline, the need to be extremely

predictable is not part of the design criteria for most operating systems so they can't provide the guaranteed response times that real-time performance requires. Therefore, most embedded systems (described below) and real-time environments require operating systems that are specially designed to meet real-time needs.

Hybrid systems, widely used today, are a combination of batch and interactive. They appear to be interactive because individual users can enter multiple jobs or processes into the system and get fast responses, but these systems actually accept and run batch programs in the background when the interactive load is light. A hybrid system takes advantage of the free time between high-demand usage of the system and low-demand times.

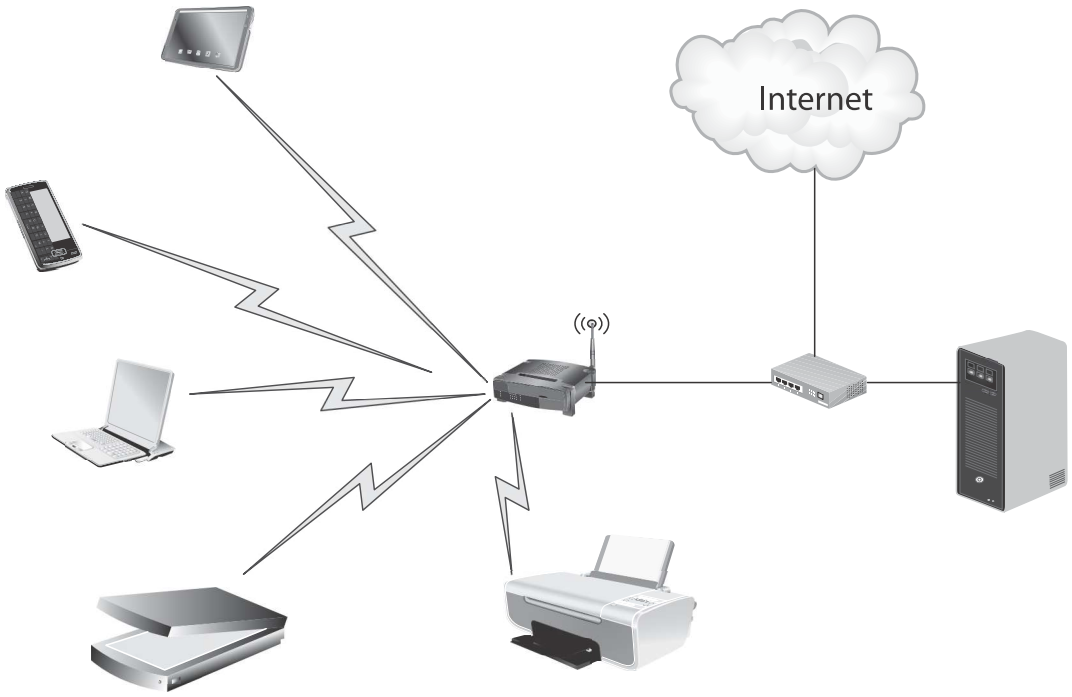
Networks allow users to manipulate resources that may be located over a wide geographical area. Network operating systems were originally similar to single-processor operating systems in that each machine ran its own local operating system and served its own local user group. Now network operating systems make up a special class of software that allows users to perform their tasks using few, if any, local resources. One example of this phenomenon is cloud computing.



One example of software available to help developers build an embedded system is Windows Embedded Automotive.

As shown in Figure 1.7, wireless networking capability is a standard feature in many computing devices: cell phones, tablets, and other handheld Web browsers.

Embedded systems are computers that are physically placed inside the products that they operate to add very specific features and capabilities. For example, embedded systems can be found in automobiles, digital music players, elevators, and pacemakers, to



(figure 1.7)

Example of a simple network. The server is connected by cable to the router and other devices connect wirelessly.

name a few. Computers embedded in automobiles facilitate engine performance, braking, navigation, entertainment systems, and engine diagnostic details.

Operating systems for embedded computers are very different from those for general computer systems. Each one is designed to perform a set of specific programs, which are not interchangeable among systems. This permits the designers to make the operating system more efficient and take advantage of the computer's limited resources (typically slower CPUs and smaller memory resources), to their maximum.

Before a general-purpose operating system, such as Linux, UNIX, or Windows, can be used in an embedded system, the system designers must select which operating system components are required in that particular environment and which are not. The final version of this operating system generally includes redundant safety features and only the necessary elements; any unneeded features or functions are dropped. Therefore, operating systems with a small kernel (the core portion of the software) and other functions that can be mixed and matched to meet the embedded system requirements have potential in this market.

Grace Murray Hopper (1906–1992)

Grace Hopper developed one of the world's first compilers (intermediate programs that translate human-readable instructions into zeros and ones that are the language of the target computer) and compiler-based programming languages. A mathematician, she joined the U.S. Navy Reserves in 1943. She was assigned to work on computer systems at Harvard, where she did her groundbreaking efforts which included her work developing the COBOL language, which was widely adopted. In 1969, the annual Data Processing Management Association awarded her its "Man of the Year Award," and in 1973, she became the first woman of any nationality and the first person from the United States to be made a Distinguished Fellow of the British Computer Society.

For more information, see *www.computerhistory.org*.

National Medal of Technology and Innovation (1991): "For her pioneering accomplishments in the development of computer programming languages that simplified computer technology and opened the door to a significantly larger universe of users."

Brief History of Operating Systems Development

The evolution of operating system software parallels the evolution of the computer hardware it was designed to control.

1940s

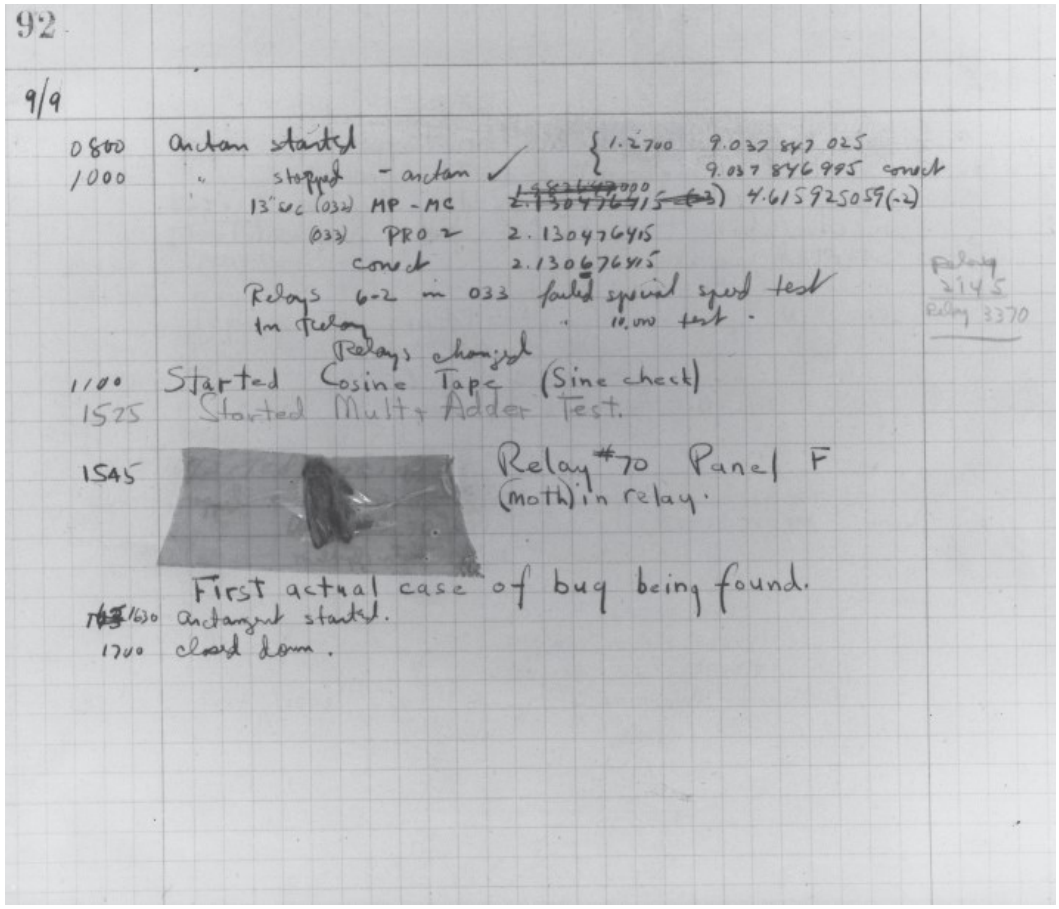
Computers from this time were operated by the programmers presiding from the main console—this was a hands-on process. In fact, to fix an error in a program, the programmer would stop the processor, read the contents of each register, make the corrections in memory, and then resume operation. To run programs on these systems, the programmers would reserve the entire machine for the entire time they estimated it would take for the computer to execute their program, and the computer sat idle between reservations. As a result, the machine was poorly utilized because the processor, the CPU, was active for only a fraction of the time it was reserved and didn't work at all between reservations.

There were a lot of variables that could go wrong with these early computers. For example, when Harvard's Mark I computer stopped working one day in 1945, technicians investigating the cause for the interruption discovered that a moth had short-circuited Relay 70 in Panel F, giving its life in the process. The researcher, Grace Murray Hopper, duly placed the dead insect in the system log as shown in Figure 1.8 noting, "First actual case of bug being found." The incident spawned the industry-wide use of the word "bug" to indicate that a system is not working correctly, and the term is still commonly used today.

1950s

Two improvements were widely adopted during this decade: the task of running programs was assigned to professional computer operators (instead of individual programmers) who were assigned to maximize the computer's operation and schedule the incoming jobs as efficiently as possible. Job scheduling required that waiting jobs be put into groups with similar requirements (for example, by grouping all jobs running with a certain language compiler) so the entire batch of jobs could run faster. But even with these batching techniques, there were still extensive time lags between the CPUs that were fast and the I/O devices that ran much slower. Eventually, several factors helped improve the performance of the CPU and the system.

- The speeds of I/O devices, such as tape drives and disks, gradually increased.
- To use more of the available storage area in these devices, records were grouped into blocks before they were retrieved or stored. (This is called "blocking," meaning that several logical records are grouped within one physical record and is discussed in detail in Chapter 7.)



(figure 1.8)

Dr. Grace Hopper's research journal included the first computer bug, the remains of a moth that became trapped in the computer's relays, causing the system to crash. [Photo © 2002 IEEE]

- To reduce the discrepancy in speed between the I/O and the CPU, an interface called the control unit was placed between them to act as a buffer. A buffer is an interim storage area that works as a temporary holding place. As the slow input device reads one record, the control unit places each character of the record into the buffer. When the buffer is full, the entire record is quickly transmitted to the CPU. The process is just the opposite for output devices: the CPU places the entire record into the buffer, which is then passed on by the control unit at the slower rate required by the output device.

  The buffers of this generation were conceptually similar to those now used routinely by Web browsers to make video and audio playback smoother, as shown in Figure 1.9.

(figure 1.9)

Typical buffer indicator showing progress. About one third of this file has already been displayed and a few seconds more are ready in the buffer.

During the second generation, programs were still assigned to the processor one-at-a-time in sequence. The next step toward better use of the system's resources was the move to shared processing.

1960s

Computers in the mid-1960s were designed with faster CPUs, but they still had problems interacting directly with the relatively slow printers and other I/O devices. The solution was called multiprogramming, which introduced the concept of loading many programs at one time and allowing them to share the attention of the single CPU.

The most common mechanism for implementing multiprogramming was the introduction of the concept of the interrupt, whereby the CPU was notified of events needing operating systems services. For example, when a program issued a print command, called input/output (I/O) command, it generated an interrupt, which signaled the release of the CPU from one job so it could begin execution of the next job. This was called *passive multiprogramming* because the operating system didn't control the interrupts, but instead, it waited for each job to end on its own. This was less than ideal because if a job was CPU-bound (meaning that it performed a great deal of nonstop CPU processing before issuing an interrupt), it could monopolize the CPU for a long time while all other jobs waited, even if they were more important.

To counteract this effect, computer scientists designed *active multiprogramming*, allowing the operating system a more active role. Each program was initially allowed to use only a preset slice of CPU time. When time expired, the job was interrupted by the operating system so another job could begin its execution. The interrupted job then had to wait until it was allowed to resume execution at some later time. Soon, this idea, called time slicing, became common in many interactive systems.

1970s

During this decade, computers were built with faster CPUs, creating an even greater disparity between their rapid processing speed and slower I/O times. However, multiprogramming schemes to increase CPU use were limited by the physical capacity of the main memory, which was a limited resource and very expensive. For example, the first Cray supercomputer, shown in Figure 1.10, was installed at Los Alamos National Laboratory in 1976 and had only 8 megabytes (MB) of main memory, significantly less than many computing devices today.

A solution to this physical limitation was the development of virtual memory, which allowed portions of multiple programs to reside in memory at the same time. In other words, a virtual memory system could divide each program into parts and keep those parts in secondary storage, bringing each part into memory only as it was needed.

(figure 1.10)

The Cray I supercomputer boasted 8 megabytes of main memory and a world-record speed of 160 million floating-point operations per second. Its circular design allowed no cable to be more than 4 feet (1.2 meters) in length.

Virtual memory soon became standard in operating systems of all sizes and paved the way to much better use of the CPU.

1980s

Hardware during this time became more flexible, with logical functions that were built on easily replaceable circuit boards. And because it had become cheaper to create these circuit boards, more operating system functions were made part of the hardware itself, giving rise to a new concept—**firmware**, a word used to indicate that a program is permanently held in read only memory (ROM), as opposed to being held in secondary storage.

Eventually the industry moved to multiprocessing (having more than one processor), and more complex languages were designed to coordinate the activities of the multiple processors servicing a single job. As a result, it became possible to execute two programs at the same time (in parallel), and eventually operating systems for computers of every size were routinely expected to accommodate multiprocessing.

The evolution of personal computers and high-speed communications sparked the move to networked systems and distributed processing, enabling users in remote locations to share hardware and software resources. These systems required a new kind of operating system—one capable of managing multiple sets of subsystem managers, as well as hardware that might reside half a world away.

On the other hand, with distributed operating systems, users could think they were working with a system using one processor, when in fact they were connected to a cluster of many processors working closely together. With these systems, users didn't need to know which processor was running their applications or which devices were storing their files. These details were all handled transparently by the operating system—something that required more than just adding a few lines of code to a uniprocessor operating system. The disadvantage of such a complex operating system was the requirement for more complex processor-scheduling algorithms.

1990s

The overwhelming demand for Internet capability in the mid-1990s sparked the proliferation of networking capability. The World Wide Web was first described in a paper by Tim Berners-Lee; his original concept is shown in Figure 1.11. Based on this research, he designed the first Web server and browser, making it available to the general public in 1991. While his innovations sparked increased connectivity, it also increased demand for tighter security to protect system assets from Internet threats.

The decade also introduced a proliferation of multimedia applications demanding additional power, flexibility, and device compatibility for most operating systems, as well as large amounts of storage capability (in addition to longer battery life and cooler operation). These technological advances required commensurate advances by the operating system.

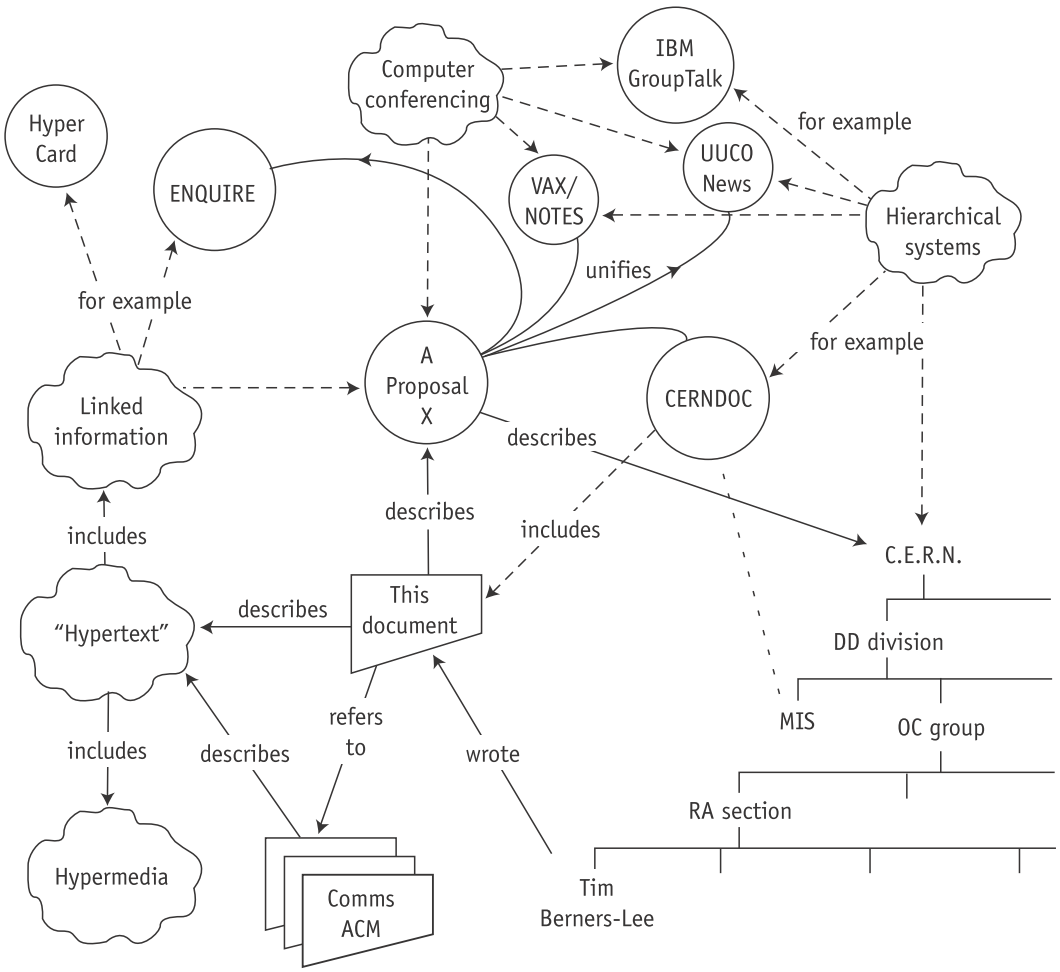
2000s

The new century emphasized the need for improved flexibility, reliability, and speed. The concept of virtual machines was expanded to allow computers to accommodate multiple operating systems that ran at the same time and shared resources. One example is shown in Figure 1.12.

Virtualization allowed separate partitions of a single server to support a different operating system. In other words, it turned a single physical server into multiple virtual servers, often with multiple operating systems. Virtualization required the operating system to have an intermediate manager, to oversee the access of each operating system to the server's physical resources.

(figure 1.11)

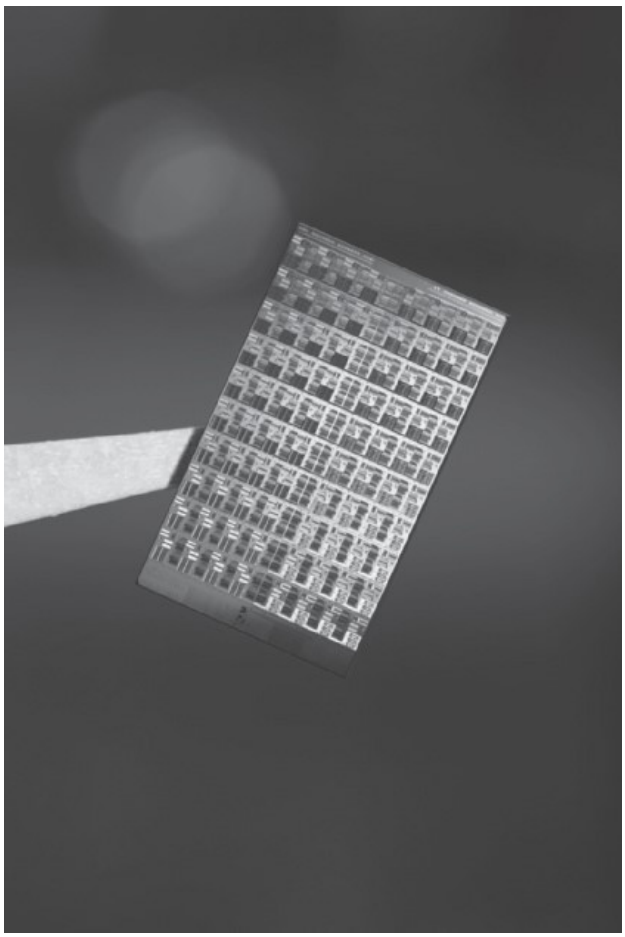
Illustration from the 1989 proposal by Tim Berners-Lee describing his revolutionary "linked information system."



(figure 1.12)

With virtualization, different operating systems can run on a single computer.

Courtesy of Parallels, Inc.



(Figure 1.13)

This single piece of silicon can hold 80 cores, which (to put it in simplest terms) can perform 80 calculations at one time.

(Courtesy of Intel Corporation)

Processing speed enjoyed a similar advancement with the commercialization of multicore processors, which can contain from two to many cores. For example, a chip with two CPUs (sometimes called a dual-core chip) allows two sets of calculations to run at the same time, which sometimes leads to faster job completion. It's almost as if the user has two separate computers, and thus two processors, cooperating on a single task. Designers have created chips that have dozens of cores, as shown in Figure 1.13.

Does this hardware innovation affect the operating system software? Absolutely—because the operating system must now manage the work of each of these processors and be able to schedule and manage the processing of their multiple tasks.

2010s

Increased mobility and wireless connectivity spawned a proliferation of dual-core, quad-core, and other multicore CPUs with more than one processor (also called a core) on a computer chip. Multicore engineering was driven by the problems caused by nano-sized transistors and their ultra-close placement on a computer chip. Although chips with millions of transistors that were very close together helped increase system

performance dramatically, the close proximity of these transistors also allowed current to "leak," which caused the buildup of heat, as well as other issues. With the development of multi-core technology, a single chip (one piece of silicon) was equipped with two or more processor cores. In other words, they replaced a single large processor with two half-sized processors (called dual core), four quarter-sized processors (quad core), and so on. This design allowed the same sized chip to produce less heat and offered the opportunity to permit multiple calculations to take place at the same time.

Design Considerations

The people who write operating systems are faced with many choices that can affect every part of the software and the resources it controls. Before beginning, designers typically start by asking key questions, using the answers to guide them in their work. The most common overall goal is to maximize use of the system's resources (memory, processing, devices, and files) and minimize downtime, though certain proprietary systems may have other priorities. Typically, designers include the following factors into their developmental efforts: the minimum and maximum RAM resources, the number and type of CPUs available, the variety of devices likely to be connected, the range of files, networking capability, security requirements, default user interfaces available, assumed user capabilities, and so on.

For example, a mobile operating system for a tablet might have a single CPU and need to manage that CPU to minimize the heat it generates. Likewise, if the operating system manages a real-time system, where deadlines are urgent, designers need to manage the memory, processor time, devices, and files so urgent deadlines will be met. For these reasons, operating systems are often complex pieces of software that juggle numerous applications, access to networked resources, several users, and multiple CPUs in an effort to keep the system running effectively.

As you might expect, no single operating system is perfect for every environment. Some systems can be best served with a UNIX system, others benefit from the structure of a Windows system, and still others work best using Linux, Mac OS, or Android, or even a custom-built operating system.

Conclusion

In this chapter, we looked at the overall function of operating systems and how they have evolved to run increasingly complex computers and computer systems. Like any complicated subject, there's much more detail to explore. As we'll see in the remainder of this text, there are many ways to perform every task, and it's up to the designer of the operating system to choose the policies that best match the system's environment.

In the following chapters, we'll explore in detail how each portion of the operating system works, as well as its features, functions, benefits, and costs. We'll begin with the part of the operating system that's the heart of every computer: the module that manages main memory.

Key Terms

batch system: a type of computing system that executes programs, each of which is submitted in its entirety, can be grouped into batches, and executed without external intervention.

central processing unit (CPU): a component with circuitry to control the interpretation and execution of instructions.

cloud computing: a multifaceted technology that allows computing, data storage and retrieval and other computer functions to take place over a large network, typically the Internet.

Device Manager: the section of the operating system responsible for controlling the use of devices. It monitors every device, channel, and control unit and chooses the most efficient way to allocate all of the system's devices.

embedded computer system: a dedicated computer system that often is part of a larger physical system, such as jet aircraft or automobiles. Often, it must be small, fast, and able to work with real-time constraints, fail-safe execution, and nonstandard I/O devices.

File Manager: the section of the operating system responsible for controlling the use of files.

firmware: software instructions or data that are stored in a fixed or "firm" way, usually implemented on some type of read only memory (ROM).

hardware: the tangible machine and its components, including main memory, I/O devices, I/O channels, direct access storage devices, and the central processing unit.

hybrid system: a computer system that supports both batch and interactive processes.

interactive system: a system that allows each user to interact directly with the operating system.

kernel: the primary part of the operating system that remains in random access memory (RAM) and is charged with performing the system's most essential tasks, such as managing main memory and disk access.

main memory: the memory unit that works directly with the CPU and in which the data and instructions must reside in order to be processed. Also called *primary storage, RAM,* or *internal memory*.

mainframe: the historical name given to a large computer system characterized by its large size, high cost, and relatively fast performance.

Memory Manager: the section of the operating system responsible for controlling the use of memory. It checks the validity of each request for memory space, and if it's a legal request, allocates the amount needed to execute the job.

multiprocessing: when two or more CPUs share the same main memory, most I/O devices, and the same control program routines. They service the same job stream and execute distinct processing programs concurrently.

multiprogramming: a technique that allows a single processor to process several programs residing simultaneously in main memory and interleaving their execution by overlapping I/O requests with CPU requests.

network: a system of interconnected computer systems and peripheral devices that exchange information with one another.

operating system: the primary software on a computing system that manages its resources, controls the execution of other programs, and manages communications and data storage.

Processor Manager: a composite of two submanagers, the Job Scheduler and the Process Scheduler, which decides how to allocate the CPU.

RAM: random access memory. See *main memory*.

real-time system: a computing system used in time-critical environments that require guaranteed response times, such as navigation systems, rapid transit systems, and industrial control systems.

server: a node that provides to clients various network services, such as file retrieval, printing, or database access services.

storage: the place where data is stored in the computer system. Primary storage is main memory. Secondary storage is nonvolatile media, such as disks and flash memory.

Interesting Searches

For more background on a few of the topics discussed in this chapter, begin a search with these terms.

- NASA - Computers Aboard the Space Shuttles
- IBM Computer History Archive
- History of the UNIX Operating System
- History of Microsoft Windows Products
- Computer History Museum

Exercises

Research Topics

Whenever you research computer technology, make sure your resources are timely. Note the date when the research was published, as this field changes quickly. Also be sure to validate the authenticity of your sources. Avoid any that may be questionable, such as blogs and publicly edited online sources.

A. New research on topics concerning operating systems and computer science are published by professional societies. Research the Internet or current literature to identify at least two prominent professional societies with peer-reviewed computer science journals, then list the advantages of becoming a member, the dues for students, and a one-sentence summary of a published paper concerning operating systems. Cite your sources.

B. Write a one-page review of an article about the subject of operating systems that appeared in a recent computing magazine or academic journal. Give a summary of the article, including the primary topic, your own summary of the information presented, and the author's conclusion. Give your personal evaluation of the article, including topics that made the article interesting to you (or not) and its relevance to your own experiences. Be sure to cite your source.

Exercises

1. Gordon Moore predicted the dramatic increase in transistors per chip in 1965 and his prediction has held for decades. Some industry analysts insist that Moore's Law has been a predictor of chip design, but others say it is a motivator for designers of new chips. In your opinion, who is correct? Explain your answer.
2. Give an example of an organization that might find batch-mode processing useful and explain why.
3. Name five current operating systems (other than those mentioned in Table 1.1) and identify the devices that each is designed to run on.
4. List three situations that might demand a real-time operating system and explain in detail the system characteristics that persuade you to do so.
5. Many people confuse main memory and secondary storage. Explain why this happens, and describe how you would explain the differences to classmates so they would no longer confuse the two.
6. Name the five key concepts about an operating system that you think a user needs to know and understand.
7. Explain the impact of the evolution of computer hardware and the accompanying evolution of operating systems software.

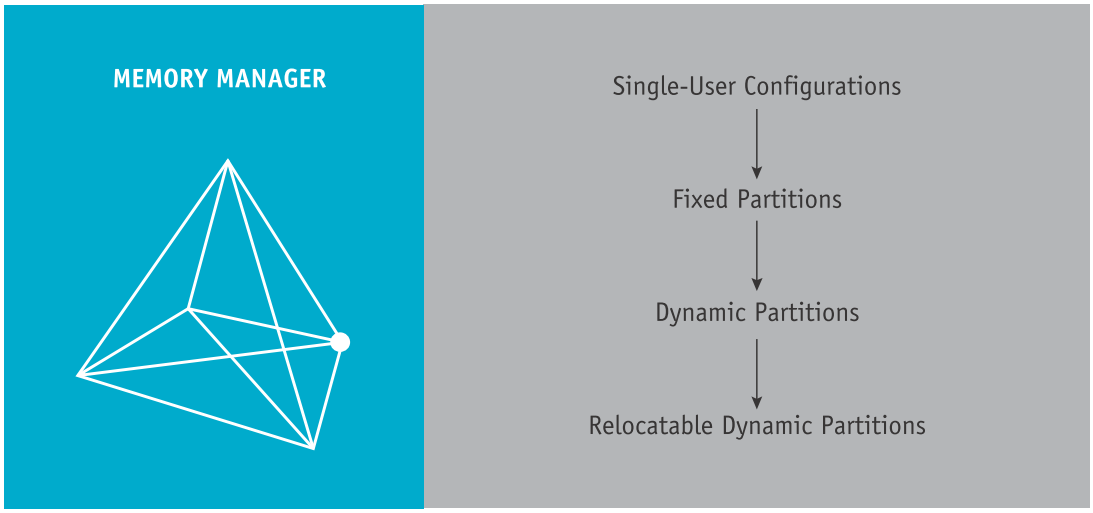
8. Give real-world examples of interactive, batch, real-time, and embedded systems and explain the fundamental differences among them.
9. Briefly compare active and passive multiprogramming and give examples of each.
10. Give at least two reasons why a regional bank might decide to buy six networked servers instead of one mainframe.
11. Select two of the following professionals: an insurance adjuster, a delivery person for a courier service, a newspaper reporter, a general practitioner doctor, or a manager in a supermarket. Suggest at least two ways that each person might use a mobile computer to work efficiently.
12. Compare the development of two operating systems described in Chapters 13-16 of this text, including design goals and evolution. Name the operating system each was based on, if any. Which one do you believe is more efficient for your needs? Explain why.

Advanced Exercises

Advanced Exercises are appropriate for readers with supplementary knowledge of operating systems.

13. In computing literature, the value represented by the prefixes kilo-, mega-, giga-, and so on can vary depending on whether they are describing many bytes of main memory or many bits of data transmission speed. Calculate the number of bytes in a megabyte (MB) and compare it to the number of bits in a megabit (Mb). If there is a difference, explain why that is the case. Cite your sources.
14. Draw a system flowchart illustrating the steps performed by an operating system as it executes the instruction to back up a disk on a single-user computer system. Begin with the user typing the command on the keyboard or choosing an option from a menu, and conclude with the result being displayed on the monitor.
15. In a multiprogramming and time-sharing environment, several users share a single system at the same time. This situation can result in various security problems. Name two such problems. Can we ensure the same degree of security in a time-share machine as we have in a dedicated machine? Explain your answer.
16. The boot sequence is the series of instructions that enable the operating system to get installed and running when you power on a computer. For an operating system of your choice, describe in your own words the role of firmware and the boot process.
17. A “dual boot” system gives users the opportunity to choose from among a list of operating systems when powering on a computer. Describe how this process works. Explain whether or not there is a risk that one operating system might intrude on the space reserved for another operating system.

Chapter 2 Memory Management: Simple Systems



"Memory is the primary and fundamental power, without which there could be no other intellectual operation."

—Samuel Johnson (1709–1784)

Learning Objectives

After completing this chapter, you should be able to describe:

- The basic functionality of the four memory allocation schemes presented in this chapter: single user, fixed partitions, dynamic partitions, and relocatable dynamic partitions
- Best-fit memory allocation as well as first-fit memory allocation
- How a memory list keeps track of available memory
- The importance of memory deallocation
- The importance of the bounds register in memory allocation schemes
- The role of compaction and how it can improve memory allocation efficiency

The management of **main memory** is critical. In fact, for many years, the performance of the *entire* system was directly dependent on two things: How much memory was available and how that memory was optimized while jobs were being processed.

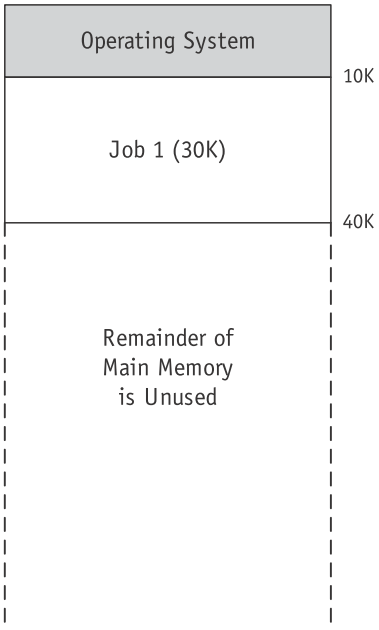
This chapter introduces the role of main memory (also known as random access memory or **RAM**, core memory, or primary storage) and four types of memory allocation schemes: single-user systems, fixed partitions, dynamic partitions, and relocatable dynamic partitions. Let's begin with the simplest memory management scheme—the one used with the simplest computer systems.

Single-User Contiguous Scheme

This memory allocation scheme works like this: before execution can begin, each job or program is loaded in its entirety into memory and allocated as much contiguous space in memory as it needs, as shown in Figure 2.1. The key words here are *entirety* and *contiguous*. If the program is too large to fit into the available memory space, it cannot begin execution.

A single-user scheme supports only one job at a time. It's not possible to share main memory.

This scheme demonstrates a significant limiting factor of all computers—they have only a finite amount of memory. If a program doesn't fit, then either the size of the main memory must be increased, or the program must be modified to fit, often by revising it to be smaller.



(figure 2.1)

Only one program can fit into memory at a time, even if there is room to accommodate other waiting jobs.

Single-user systems in a non-networked environment allocate, to each user, access to all available main memory for each job, and jobs are processed sequentially, one

after the other. To allocate memory, the amount of work required from the operating system's Memory Manager is minimal, as described in these steps:

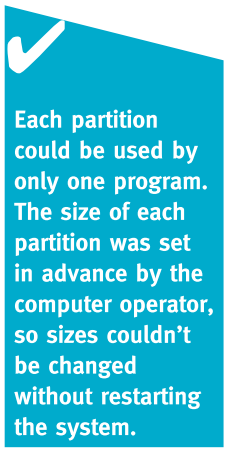
1. Evaluate the incoming process to see if it is small enough to fit into the available space. If it is, load it into memory; if not, reject it and evaluate the next incoming process,
2. Monitor the occupied memory space. When the resident process ends its execution and no longer needs to be in memory, make the entire amount of main memory space available and return to Step 1, evaluating the next incoming process.

An "Algorithm to Load a Job in a Single-User System" using pseudocode and demonstrating these steps can be found in Appendix A.

Once the program is entirely loaded into memory, it begins its execution and remains there until execution is complete, either by finishing its work or through the intervention of the operating system, such as when an error is detected.

One major problem with this type of memory allocation scheme is that it doesn't support multiprogramming (multiple jobs or processes occupying memory at the same time); it can handle only one at a time. When these single-user configurations were first made available commercially in the late 1940s and early 1950s, they were used in research institutions but proved unacceptable for the business community—it wasn't cost effective to spend almost $200,000 for a piece of equipment that could be used by only one person at a time. The next scheme introduced memory partitions.

Fixed Partitions



Each partition could be used by only one program. The size of each partition was set in advance by the computer operator, so sizes couldn't be changed without restarting the system.

The first attempt to allow for multiprogramming used **fixed partitions** (also known as static partitions) within main memory—each partition could be assigned to one job. A system with four partitions could hold four jobs in memory at the same time. One fact remained the same, however—these partitions were static, so the systems administrator had to turn off the entire system to reconfigure their sizes, and any job that couldn't fit into the largest partition could not be executed.

An important factor was introduced with this scheme: protection of the job's memory space. Once a partition was assigned to a job, the jobs in other memory partitions had to be prevented from invading its boundaries, either accidentally or intentionally. This problem of partition intrusion didn't exist in single-user contiguous allocation schemes because only one job was present in main memory at any given time—only the portion of main memory that held the operating system had to be protected. However, for the fixed partition allocation schemes, protection was mandatory for each partition in main memory. Typically this was the joint responsibility of the hardware of the computer and of the operating system.

The algorithm used to store jobs in memory requires a few more steps than the one used for a single-user system because the size of the job must be matched with the size of the available partitions to make sure it fits completely. ("An Algorithm to Load a Job in a Fixed Partition" is in Appendix A.) Remember, this scheme also required that the entire job be loaded into memory before execution could begin.

To do so, the Memory Manager could perform these steps in a two-partition system:

1. Check the incoming job's memory requirements. If it's greater than the size of the largest partition, reject the job and go to the next waiting job. If it's less than the largest partition, go to Step 2.
2. Check the job size against the size of the first available partition. If the job is small enough to fit, see if that partition is free. If it is available, load the job into that partition. If it's busy with another job, go to Step 3.
3. Check the job size against the size of the second available partition. If the job is small enough to fit, check to see if that partition is free. If it is available, load the incoming job into that partition. If not, go to Step 4.
4. Because neither partition is available now, place the incoming job in the waiting queue for loading at a later time. Return to Step 1 to evaluate the next incoming job.

This partition scheme is more flexible than the single-user scheme because it allows more than one program to be in memory at the same time. However, it still requires that the *entire* program be stored *contiguously* and *in memory* from the beginning to the end of its execution.

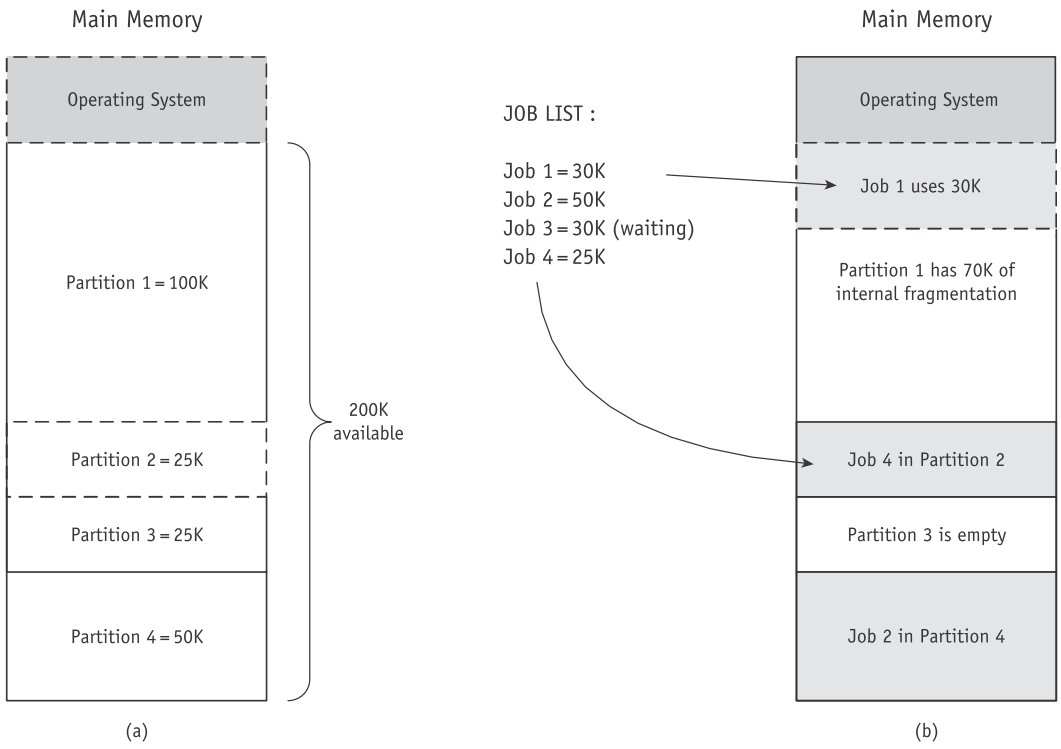
In order to allocate memory spaces to jobs, the Memory Manager must maintain a table which shows each memory partition's size, its **address**, its access restrictions, and its current status (free or busy). Table 2.1 shows a simplified version for the system illustrated in Figure 2.2. Note that Table 2.1 and the other tables in this chapter have been simplified. More detailed discussions of these tables and their contents are presented in Chapter 8, "File Management."

Partition Size	Memory Address	Access	Partition Status
100K	200K	Job 1	Busy
25K	300K	Job 4	Busy
25K	325K		Free
50K	350K	Job 2	Busy

(table 2.1)

A simplified fixed-partition memory table with the free partition shaded.

When each resident job terminates, the status of its memory partition is changed from busy to free to make it available to an incoming job.



(figure 2.2) *As the jobs listed in Table 2.1 are loaded into the four fixed partitions, Job 3 must wait even though Partition 1 has 70K of available memory. Jobs are allocated space on the basis of "first available partition of required size."*

The fixed partition scheme works well if all of the jobs that run on the system are of similar size or if the sizes are known ahead of time and don't vary between reconfigurations. Ideally, that would require accurate, advance knowledge of all the jobs waiting to be run on the system in the coming hours, days, or weeks. However, under most circumstances, the operator chooses partition sizes in an arbitrary fashion, and thus not all incoming jobs fit in them.

There are significant consequences if the partition sizes are too small; large jobs will need to wait if the large partitions are already booked, and they will be rejected if they're too big to fit into the largest partition.

On the other hand, if the partition sizes are too big, memory is wasted. Any job that occupies less than the entire partition (the vast majority will do so) will cause unused memory in the partition to remain idle. Remember that each partition is an indivisible unit that can be allocated to only one job at a time. Figure 2.3 demonstrates one such circumstance: Job 3 is kept waiting because it's too big for Partition 3, even though there is more than enough unused space for it in Partition 1, which is claimed by Job 1.

This phenomenon of less-than-complete use of memory space in a fixed partition is called **internal fragmentation** (because it's inside a partition) and is a major drawback to this memory allocation scheme.

Jay W. Forrester (1918–)

Jay Forrester led the groundbreaking MIT Whirlwind computer project (1947-1953) that developed the first practical and reliable, high-speed random access memory for digital computers. At the time it was called "coincident current magnetic core storage." This invention became the standard memory device for digital computers and provided the foundation for main memory development through the 1970s. Forrester is the recipient of numerous awards, including the IEEE Computer Pioneer Award (1982), the U.S. National Medal of Technology and Innovation (1989), and induction into the Operational Research Hall of Fame (2006).

For more information:
http://www.computerhistory.org/fellowawards/hall/bios/Jay,Forrester/

Received the IEEE Computer Pioneer Award: "For exceptional advances in the digital computer through his invention and application of the magnetic-core random-access memory, employing coincident current addressing."

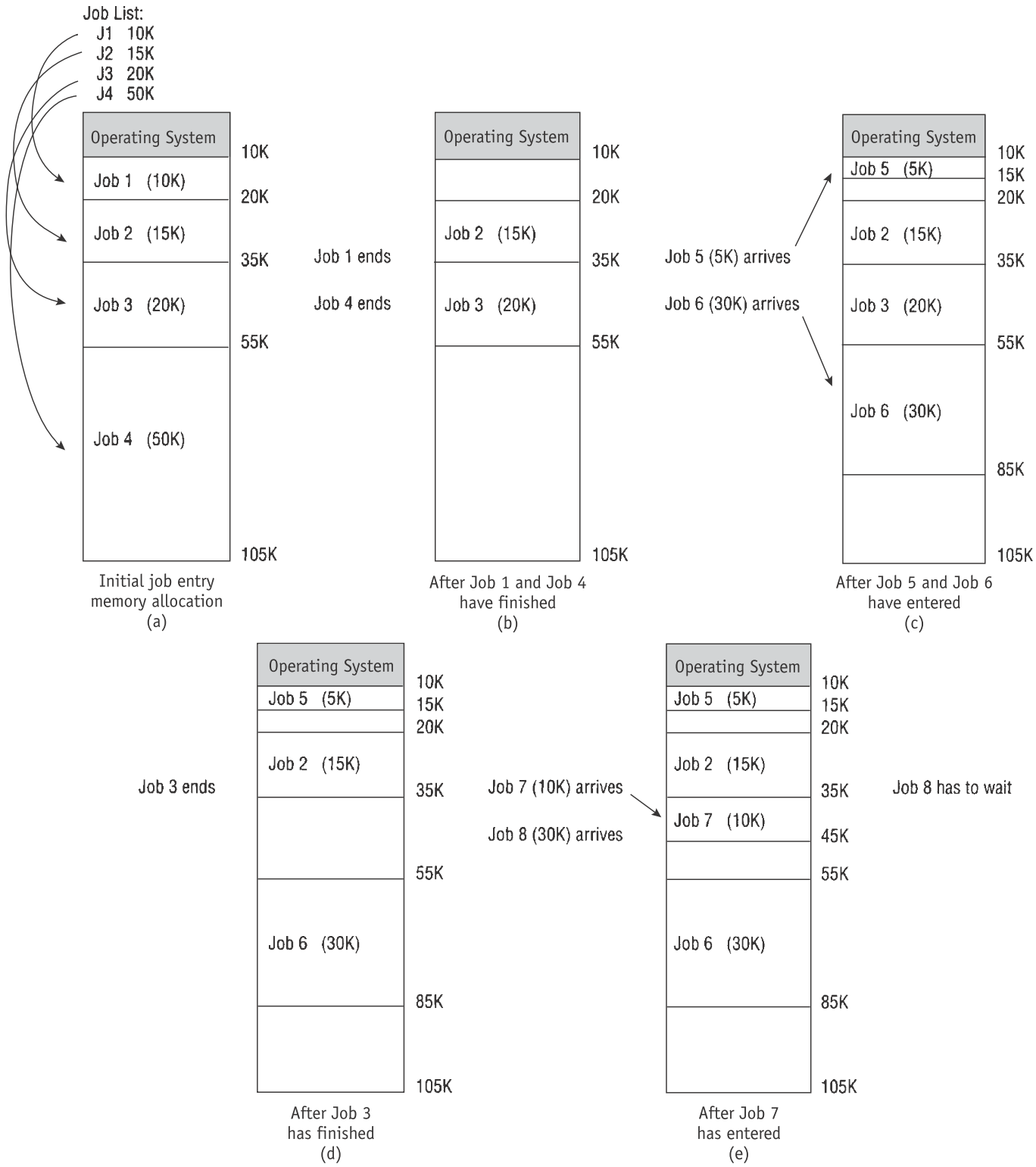
Dynamic Partitions

With the introduction of the **dynamic partition** allocation scheme, memory is allocated to an incoming job in one contiguous block, and each job is given only as much memory as it requests when it is loaded for processing. Although this is a significant improvement over fixed partitions because memory is no longer wasted inside each partition, it introduces another problem.

There are two types of fragmentation: internal and external. The type depends on the location of the wasted space.

It works well when the first jobs are loaded. As shown in Figure 2.4, a dynamic partition scheme allocates memory efficiently as each of the first few jobs are loaded, but when those jobs finish and new jobs enter the system (which are not the same size as those that just vacated memory), the newer jobs are allocated space in the available partition spaces on a priority basis. Figure 2.4 demonstrates first-come, first-served priority—that is, each job is loaded into the first available partition. Therefore, the subsequent allocation of memory creates fragments of free memory *between* partitions of allocated memory. This problem is called **external fragmentation** and, like internal fragmentation, allows memory to be wasted.

In Figure 2.3, notice the difference between the first snapshot (a), where the entire amount of main memory is used efficiently, and the last snapshot, (e), where there are



(figure 2.3)

Main memory use during dynamic partition allocation. Five snapshots (a-e) of main memory as eight jobs are submitted for processing and allocated space on the basis of "first come, first served." Job 8 has to wait (e) even though there's enough free memory between partitions to accommodate it.

three free partitions of 5K, 10K, and 20K—35K in all—enough to accommodate Job 8, which requires only 30K. However, because the three memory blocks are separated by partitions, Job 8 cannot be loaded in a contiguous manner. Therefore, this scheme forces Job 8 to wait.

Before we go to the next allocation scheme, let's examine two ways that an operating system can allocate free sections of memory.

Best-Fit and First-Fit Allocation

For both fixed and dynamic memory allocation schemes, the operating system must keep lists of each memory location, noting which are free and which are busy. Then, as new jobs come into the system, the free partitions must be allocated fairly, according to the policy adopted by the programmers who designed and wrote the operating system.

Memory partitions may be allocated on the basis of first-fit memory allocation or best-fit memory allocation. For both schemes, the Memory Manager keeps detailed lists of the free and busy sections of memory either by size or by location. The **best-fit allocation method keeps** the free/busy lists in order by size, from smallest to largest. The **first-fit allocation method** keeps the free/busy lists organized by memory locations, from low-order memory to high-order memory. Each has advantages depending on the needs of the particular allocation scheme. Best-fit usually makes the best use of memory space; first-fit is faster.

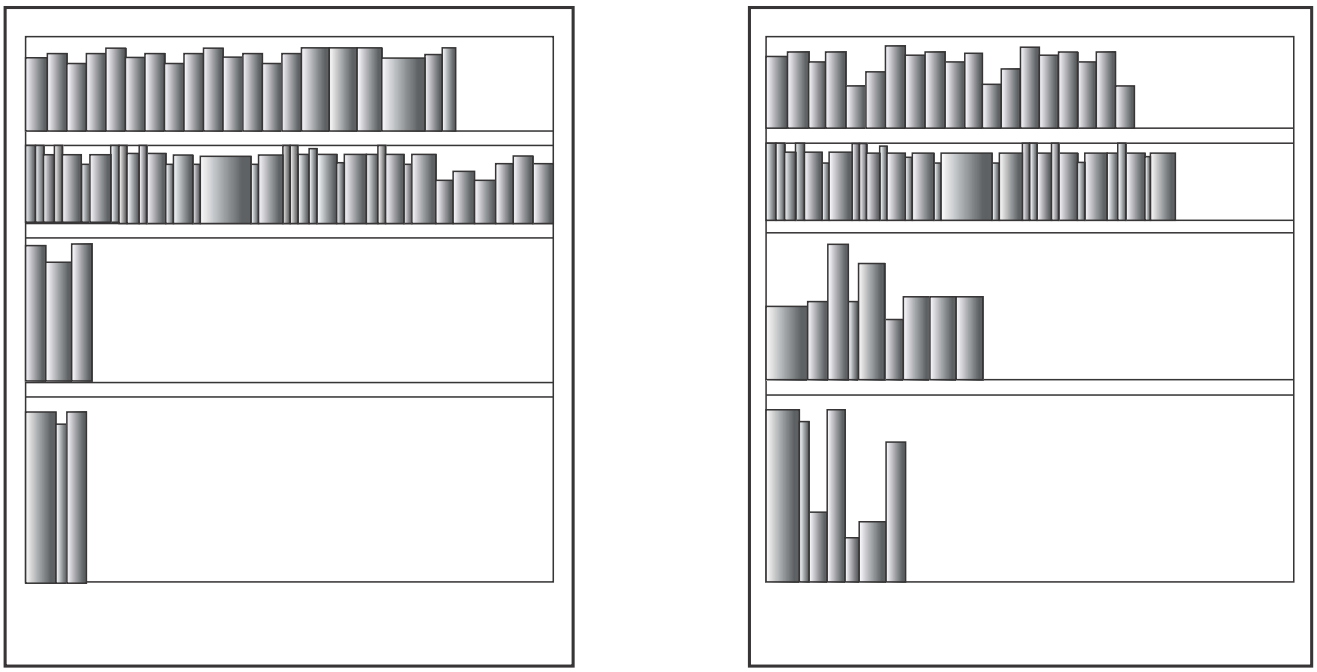
To understand the trade-offs, imagine that you are in charge of a small local library. You have books of all shapes and sizes, and there's a continuous stream of people taking books out and bringing them back—someone is always waiting. It's clear that your library is a popular place and you'll always be busy, without any time to rearrange the books on the shelves.

You need a system. Your shelves have fixed partitions with a few tall spaces for oversized books, several shelves for paperbacks and DVDs, and lots of room for textbooks. You'll need to keep track of which spaces on the shelves are full and where you have spaces for more. For the purposes of our example, we'll keep two lists: a free list showing all the available spaces, and a busy list showing all the occupied spaces. Each list will include the size and location of each space.

As each book is removed from its place on the shelf, you'll update both lists by removing the space from the busy list and adding it to the free list. Then as your books are returned and placed back on a shelf, the two lists will be updated again.

There are two ways to organize your lists: by size or by location. If they're organized by size, the spaces for the smallest books are at the top of the list and those for the

largest are at the bottom, as shown in Figure 2.4. When they're organized by location, the spaces closest to your lending desk are at the top of the list and the areas farthest away are at the bottom. Which option is best? It depends on what you want to optimize: space or speed.



(figure 2.4)

The bookshelf on the left uses the best-fit allocation. The one on the right is first-fit allocation, where some small items occupy space that could hold larger ones.

If the lists are organized by size, you're optimizing your shelf space—as books arrive, you'll be able to put them in the spaces that fit them best. This is a best-fit scheme. If a paperback is returned, you'll place it on a shelf with the other paperbacks or at least with other small books. Similarly, oversized books will be shelved with other large books. Your lists make it easy to find the smallest available empty space where the book can fit. The disadvantage of this system is that you're wasting time looking for the best space. Your other customers have to wait for you to put each book away, so you won't be able to process as many customers as you could with the other kind of list.

In the second case, a list organized by shelf location, you're optimizing the time it takes you to put books back on the shelves by using a first-fit scheme. This system ignores the size of the book that you're trying to put away. If the same paperback book arrives, you can quickly find it an empty space. In fact, any nearby empty space will suffice if it's large enough—and if it's close to your desk. In this case you are optimizing the time it takes you to shelve the books.

This is a fast method of shelving books, and if speed is important, it's the better of the two alternatives. However, it isn't a good choice if your shelf space is limited or if many large books are returned, because large books must wait for the large spaces. If all of your large spaces are filled with small books, the customers returning large books must wait until a suitable space becomes available. (Eventually you'll need time to rearrange the books and compact your collection.)

Figure 2.5 shows how a large job can have problems with a first-fit memory allocation list. Jobs 1, 2, and 4 are able to enter the system and begin execution; Job 3 has to wait even though there would be more than enough room to accommodate it if all of the fragments of memory were added together. First-fit offers fast allocation, but it isn't always efficient. (On the other hand, the same job list using a best-fit scheme would use memory more efficiently, as shown in Figure 2.6. In this particular case, a best-fit scheme would yield better memory utilization.)

Job List:

Job number	Memory requested
J1	10K
J2	20K
J3	30K*
J4	10K

Memory List:

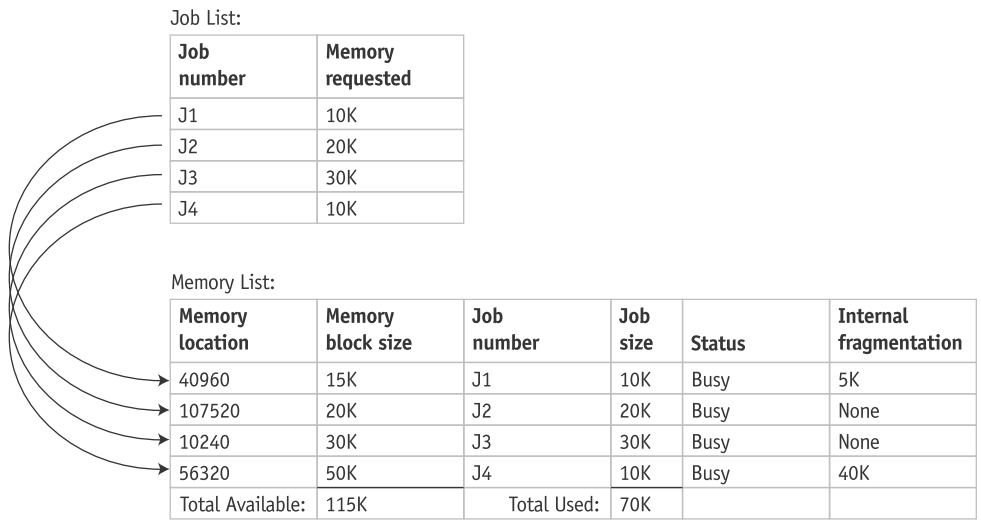
Memory location	Memory block size	Job number	Job size	Status	Internal fragmentation
10240	30K	J1	10K	Busy	20K
40960	15K	J4	10K	Busy	5K
56320	50K	J2	20K	Busy	30K
107520	20K			Free	
Total Available:	115K	Total Used:	40K		

(figure 2.5)

Using a first-fit scheme, Job 1 claims the first available space. Job 2 then claims the first partition large enough to accommodate it, but by doing so it takes the last block large enough to accommodate Job 3. Therefore, Job 3 (indicated by the asterisk) must wait until a large block becomes available, even though there's 75K of unused memory space (internal fragmentation). Notice that the memory list is ordered according to memory location.

Memory use has been increased, but the memory allocation process takes more time. What's more, while internal fragmentation has been diminished, it hasn't been completely eliminated.

The first-fit algorithm (included in Appendix A) assumes that the Memory Manager keeps two lists, one for free memory blocks and one for busy memory blocks. The operation consists of a simple loop that compares the size of each job to the size of each memory block until a block is found that's large enough to fit the job. Then the job is stored into that block of memory, and the Memory Manager fetches the next job from the entry queue. If the entire list is searched in vain, then the job is placed into a waiting queue. The Memory Manager then fetches the next job and repeats the process.



(figure 2.6)

Best-fit free scheme. Job 1 is allocated to the closestfitting free partition, as are Job 2 and Job 3. Job 4 is allocated to the only available partition although it isn't the best-fitting one. In this scheme, all four jobs are served without waiting. Notice that the memory list is ordered according to memory size. This scheme uses memory more efficiently but it's slower to implement.

In Table 2.2, a request for a block of 200 spaces has just been given to the Memory Manager. (The spaces may be words, bytes, or any other unit the system handles.) Using the first-fit scheme and starting from the top of the list, the Memory Manager locates the first block of memory large enough to accommodate the job, which is at location 6785. The job is then loaded, starting at location 6785 and occupying the next 200 spaces. The next step is to adjust the free list to indicate that the block of free memory now starts at location 6985 (not 6785 as before) and that it contains only 400 spaces (not 600 as before).

(table 2.2)

These two snapshots of memory show the status of each memory block before and after 200 spaces are allocated at address 6785, using the first-fit algorithm. (Note: All values are in decimal notation unless otherwise indicated.)

Status Before Request		Status After Request	
Beginning Address	**Free Memory Block Size**	**Beginning Address**	**Free Memory Block Size**
4075	105	4075	105
5225	5	5225	5
6785	600	*6985	400
7560	20	7560	20
7600	205	7600	205
10250	4050	10250	4050
15125	230	15125	230
24500	1000	24500	1000

The best-fit algorithm is slightly more complex because the goal is to find the smallest memory block into which the job will fit. One of the problems with it is that the entire table must be searched before an allocation can be made because the memory blocks are physically stored in sequence according to their location in memory (and not by memory block sizes as shown in Figure 2.6). The system could execute an

algorithm to continuously rearrange the list in ascending order by memory block size, but that would add more overhead and might not be an efficient use of processing time in the long run.

The best-fit algorithm (included in Appendix A) is illustrated showing only the list of free memory blocks. Table 2.3 shows the free list before and after the best-fit block has been allocated to the same request presented in Table 2.2.

Status Before Request		Status After Request	
Beginning Address	**Free Memory Block Size**	**Beginning Address**	**Free Memory Block Size**
4075	105	4075	105
5225	5	5225	5
6785	600	6785	600
7560	20	7560	20
7600	205	*7800	5
10250	4050	10250	4050
15125	230	15125	230
24500	1000	24500	1000

(table 2.3)

These two snapshots of memory show the status of each memory block before and after 200 spaces are allocated at address 7600, using the best-fit algorithm.

In Table 2.3, a request for a block of 200 spaces has just been given to the Memory Manager. Using the best-fit algorithm and starting from the top of the list, the Memory Manager searches the entire list and locates a block of memory starting at location 7600, which is the block that's the smallest one that's also large enough to accommodate the job. The choice of this block minimizes the wasted space (only 5 spaces are wasted, which is less than in the four alternative blocks). The job is then stored, starting at location 7600 and occupying the next 200 spaces. Now the free list must be adjusted to show that the block of free memory starts at location 7800 (not 7600 as before) and that it contains only 5 spaces (not 205 as before). But it doesn't eliminate wasted space. Note that while the best-fit resulted in a better fit, it also resulted (as it does in the general case) in a smaller free space (5 spaces), which is known as a sliver.

Which is best—first-fit or best-fit? For many years there was no way to answer such a general question because performance depends on the job mix. In recent years, access times have become so fast that the scheme that saves the more valuable resource, memory space, may be the best. Research continues to focus on finding the optimum allocation scheme.

In the exercises at the end of this chapter, two other hypothetical allocation schemes are explored: next-fit, which starts searching from the last allocated block for the next available block when a new job arrives; and worst-fit (the opposite of best-fit), which allocates the largest free available block to the new job. Although it's a good way to explore the theory of memory allocation, it might not be the best choice for a real system.

Deallocation

Until now, we've considered only the problem of how memory blocks are allocated, but eventually there comes a time for the release of memory space, called **deallocation**.

When memory is deallocated, it creates an opportunity for external fragmentation.

For a fixed partition system, the process is quite straightforward. When the job is completed, the Memory Manager immediately deallocates it by resetting the status of the entire memory block from "busy" to "free." Any code—for example, binary values with 0 indicating free and 1 indicating busy—may be used, so the mechanical task of deallocating a block of memory is relatively simple.

A dynamic partition system uses a more complex algorithm (shown in Appendix A) because it tries to combine free areas of memory whenever possible. Therefore, the system must be prepared for three alternative situations:

- Case 1: When the block to be deallocated is adjacent to another free block
- Case 2: When the block to be deallocated is between two free blocks
- Case 3: When the block to be deallocated is isolated from other free blocks

Case 1: Joining Two Free Blocks

Table 2.4 shows how deallocation occurs in a dynamic memory allocation system when the job to be deallocated is next to one free memory block.

After deallocation, the free list looks like the one shown in Table 2.5.

(table 2.4)

This is the original free list before deallocation for Case 1. The asterisk indicates the free memory block (of size 5) that's adjacent to the soon-to-be-free memory block (of size 200) that's shaded.

Beginning Address	Memory Block Size	Status
4075	105	Free
5225	5	Free
6785	600	Free
7560	20	Free
(7600)	(200)	(Busy)[1]
*7800	5	Free
10250	4050	Free
15125	230	Free
24500	1000	Free

[1]Although the numbers in parentheses don't appear in the free list, they've been inserted here for clarity. The job size is 200 and its beginning location is 7600.

Beginning Address	Memory Block Size	Status
4075	105	Free
5225	5	Free
6785	600	Free
7560	20	Free
*7600	205	Free
10250	4050	Free
15125	230	Free
24500	1000	Free

(table 2.5)

Case 1. This is the free list after deallocation. The shading indicates the location where changes were made to indicate the free memory block (of size 205).

Using the deallocation algorithm, the system sees that the memory to be released is next to a free memory block, which starts at location 7800. Therefore, the list must be changed to reflect the starting address of the new free block, 7600, which was the address of the first instruction of the job that just released this block. In addition, the memory block size for this new free space must be changed to show its new size, which is the combined total of the two free partitions (200 + 5).

Case 2: Joining Three Free Blocks

When the deallocated memory space is between two free memory blocks, the process is similar, as shown in Table 2.6.

Using the deallocation algorithm, the system learns that the memory to be deallocated is between two free blocks of memory. Therefore, the sizes of the three free partitions (20 + 20 + 205) must be combined and the total stored with the smallest beginning address, 7560.

Beginning Address	Memory Block Size	Status
4075	105	Free
5225	5	Free
6785	600	Free
*7560	20	Free
(7580)	(20)	(Busy)[1]
*7600	205	Free
10250	4050	Free
15125	230	Free
24500	1000	Free

[1] Although the numbers in parentheses don't appear in the free list, they have been inserted here for clarity.

(table 2.6)

Case 2. This is the Free List before deallocation. The asterisks indicate the two free memory blocks that are adjacent to the soon-to-be-free memory block.

Because the entry at location 7600 has been combined with the previous entry, we must empty out this entry. We do that by changing the status to **null entry**, with no beginning address and no memory block size, as indicated by an asterisk in Table 2.7. This negates the need to rearrange the list at the expense of memory.

(table 2.7)

Case 2. The free list after a job has released memory. The revised entry is shaded.

Beginning Address	Memory Block Size	Status
4075	105	Free
5225	5	Free
6785	600	Free
7560	245	Free
*		(null entry)
10250	4050	Free
15125	230	Free
24500	1000	Free

Case 3: Deallocating an Isolated Block

The third alternative is when the space to be deallocated is isolated from all other free areas. For this example, we need to know more about how the busy memory list is configured. To simplify matters, let's look at the busy list for the memory area between locations 7560 and 10250. Remember that, starting at 7560, there's a free memory block of 245, so the busy memory area includes everything from location 7805 (7560 + 245) to 10250, which is the address of the next free block. The free list and busy list are shown in Table 2.8 and Table 2.9.

(table 2.8)

Case 3. Original free list before deallocation. The soon-to-be-free memory block (at location 8805) is not adjacent to any blocks that are already free.

Beginning Address	Memory Block Size	Status
4075	105	Free
5225	5	Free
6785	600	Free
7560	245	Free
		(null entry)
10250	4050	Free
15125	230	Free
24500	1000	Free

Beginning Address	Memory Block Size	Status
7805	1000	Busy
*8805	445	Busy
9250	1000	Busy

(table 2.9)

Case 3. Busy memory list before deallocation. The job to be deallocated is of size 445 and begins at location 8805. The asterisk indicates the soon-to-be-free memory block.

Using the deallocation algorithm, the system learns that the memory block to be released is not adjacent to any free blocks of memory; instead it is between two other busy areas. Therefore, the system must search the table for a null entry.

The scheme presented in this example creates null entries in both the busy and the free lists during the process of allocation or deallocation of memory. An example of a null entry occurring as a result of deallocation was presented in Case 2. A null entry in the busy list occurs when a memory block between two other busy memory blocks is returned to the free list, as shown in Table 2.10. This mechanism ensures that all blocks are entered in the lists according to the beginning address of their memory location from smallest to largest.

Beginning Address	Memory Block Size	Status
7805	1000	Busy
*		(null entry)
9250	1000	Busy

(table 2.10)

Case 3. This is the busy list after the job has released its memory. The asterisk indicates the new null entry in the busy list.

When the null entry is found, the beginning memory location of the terminating job is entered in the beginning address column, the job size is entered under the memory block size column, and the status is changed from "null entry" to free to indicate that a new block of memory is available, as shown in Table 2.11.

Beginning Address	Memory Block Size	Status
4075	105	Free
5225	5	Free
6785	600	Free
7560	245	Free
*8805	445	Free
10250	4050	Free
15125	230	Free
24500	1000	Free

(table 2.11)

Case 3. This is the free list after the job has released its memory. The asterisk indicates the new free block entry replacing the null entry.

Relocatable Dynamic Partitions

All of the memory allocation schemes described thus far shared some unacceptable fragmentation characteristics that had to be resolved as the number of waiting jobs became unwieldy and demand increased to use all the slivers of memory often left unused.

When you use a defragmentation utility, you are causing memory to be compacted and file segments to be relocated so they can be retrieved faster.

The solution to both problems was the development of **relocatable dynamic partitions**. With this memory allocation scheme, the Memory Manager relocates programs to gather together all of the empty blocks and compact them to make one block of memory large enough to accommodate some or all of the jobs waiting to get in.

The **compaction of memory**, sometimes referred to as memory defragmentation, is performed by the operating system to reclaim fragmented space. Remember our earlier example of the lending library? If you stopped lending books for a few moments and rearranged the books in the most effective order, you would be compacting your collection. But this demonstrates its disadvantage—this is an overhead process that can take place only while everything else waits.

Compaction isn't an easy task. Most or all programs in memory must be relocated so they're contiguous, and then every address, and every reference to an address, within each program must be adjusted to account for the program's new location in memory. However, all other values within the program (such as data values) must be left alone. In other words, the operating system must distinguish between addresses and data values, and these distinctions are not obvious after the program has been loaded into memory.

To appreciate the complexity of **relocation**, let's look at a typical program. Remember, all numbers are stored in memory as binary values (ones and zeros), and in any given program instruction it's not uncommon to find addresses as well as data values. For example, an assembly language program might include the instruction to add the integer 1 to I. The source code instruction looks like this:

```
ADDI I, 1
```

However, after it has been translated into actual code it could be represented like this (for readability purposes the values are represented here in octal code, not binary code):

```
000007 271 01 0 00 000001
```

Question: Which elements are addresses and which are instruction codes or data values? It's not obvious at first glance. In fact, the address is the number on the left (000007). The instruction code is next (271), and the data value is on the right (000001). Therefore, if this instruction is relocated 200 places, then the address would

be adjusted (added to or subtracted) by 200, but the instruction code and data value would not be.

The operating system can tell the function of each group of digits by its location in the line and the operation code. However, if the program is to be moved to another place in memory, each address must be identified, or flagged. This becomes particularly important when the program includes loop sequences, decision sequences, and branching sequences, as well as data references. If, by chance, every address was not adjusted by the same value, the program would branch to the wrong section of the program or to a part of another program, or it would reference the wrong data.

The program shown in Figure 2.7 and Figure 2.8 shows how the operating system flags the addresses so that they can be adjusted if and when a program is relocated.

Internally, the addresses are marked with a special symbol (indicated in Figure 2.8 by apostrophes) so the Memory Manager will know to adjust them by the value stored in the relocation register. All of the other values (data values) are not marked and won't be changed after relocation. Other numbers in the program, those indicating instructions, registers, or constants used in the instruction, are also left alone.

Figure 2.9 illustrates what happens to a program in memory during compaction and relocation.

This discussion of compaction raises three questions:

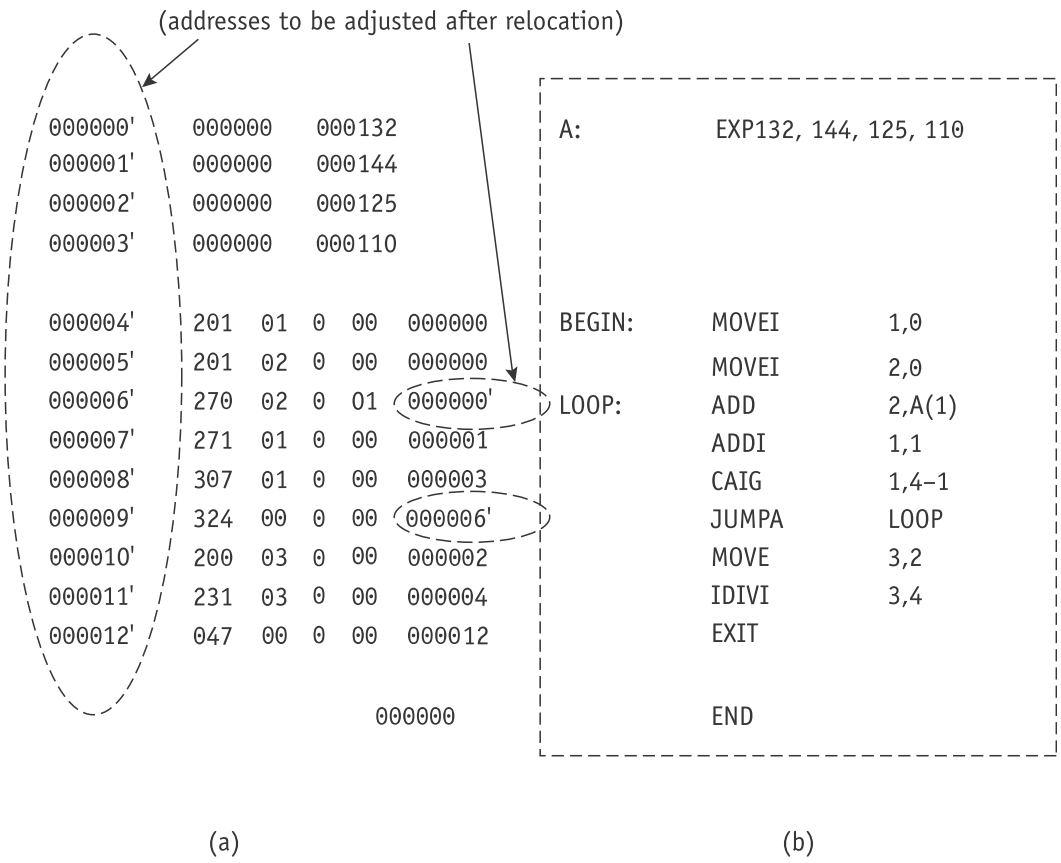
1. What goes on behind the scenes when relocation and compaction take place?
2. What keeps track of how far each job has moved from its original storage area?
3. What lists have to be updated?

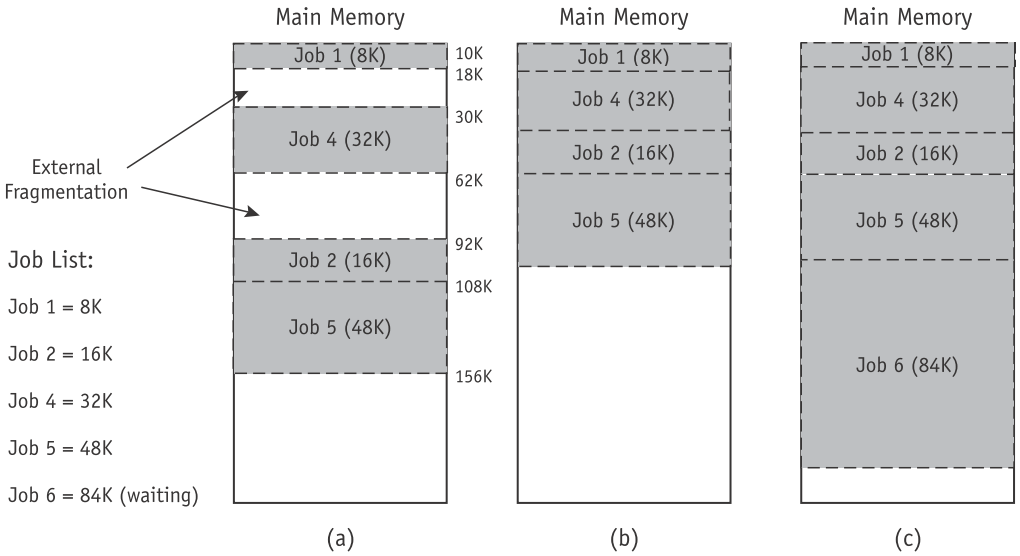
The last question is easiest to answer. After relocation and compaction, both the free list and the busy list are updated. The free list is changed to show the partition for

```
A       EXP 132, 144, 125, 110      ;the data values
BEGIN:  MOVEI           1,0         ;initialize register 1
        MOVEI           2,0         ;initialize register 2
LOOP:   ADD             2,A(1)      ;add (A + reg 1) to reg 2
        ADDI            1,1         ;add 1 to reg 1
        CAIG            1,4-1       ;is register 1 > 4-1?
        JUMPA           LOOP        ;if not, go to Loop
        MOVE            3,2         ;if so, move reg 2 to reg 3
        IDIVI           3,4         ;divide reg 3 by 4,
                                    ;remainder to register 4
        EXIT                        ;end
        END
```

(figure 2.7)

An assembly language program that performs a simple incremental operation. This is what the programmer submits to the assembler. The commands are shown on the left and the comments explaining each command are shown on the right after the semicolons.





(figure 2.8)

The original assembly language program after it has been processed by the assembler, shown on the right (a). To run the program, the assembler translates it into machine readable code (b) with all addresses marked by a special symbol (shown here as an apostrophe) to distinguish addresses from data values. All addresses (and no data values) must be adjusted after relocation.

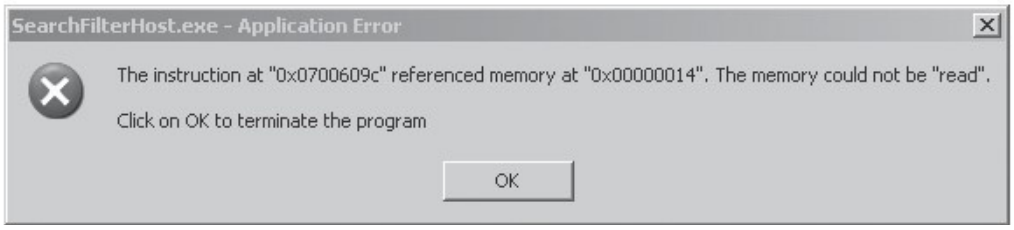
(figure 2.9)

Three snapshots of memory before and after compaction with the operating system occupying the first 10K of memory. When Job 6 arrives requiring 84K, the initial memory layout in (a) shows external fragmentation totaling 96K of space. Immediately after compaction (b), external fragmentation has been eliminated, making room for Job 6 which, after loading, is shown in (c).

the new block of free memory: the one formed as a result of compaction that will be located in memory starting after the last location used by the last job. The busy list is changed to show the new locations for all of the jobs already in progress that were relocated. Each job will have a new address except for those that were already residing at the lowest memory locations. For example, if the entry in the busy list starts at location 7805 (as shown in Table 2.9) and is relocated 762 spaces, then the location would be changed to 7043 (but the memory block size would not change—only its location).

To answer the other two questions we must learn more about the hardware components of a computer—specifically the registers. Special-purpose registers are used to help with the relocation. In some computers, two special registers are set aside for this purpose: the bounds register and the relocation register.

The **bounds register** is used to store the highest (or lowest, depending on the specific system) location in memory accessible by each program. This ensures that during execution, a program won't try to access memory locations that don't belong to it—that is, those that are out of bounds. The result of one such instance is reflected in the error message shown in Figure 2.10.



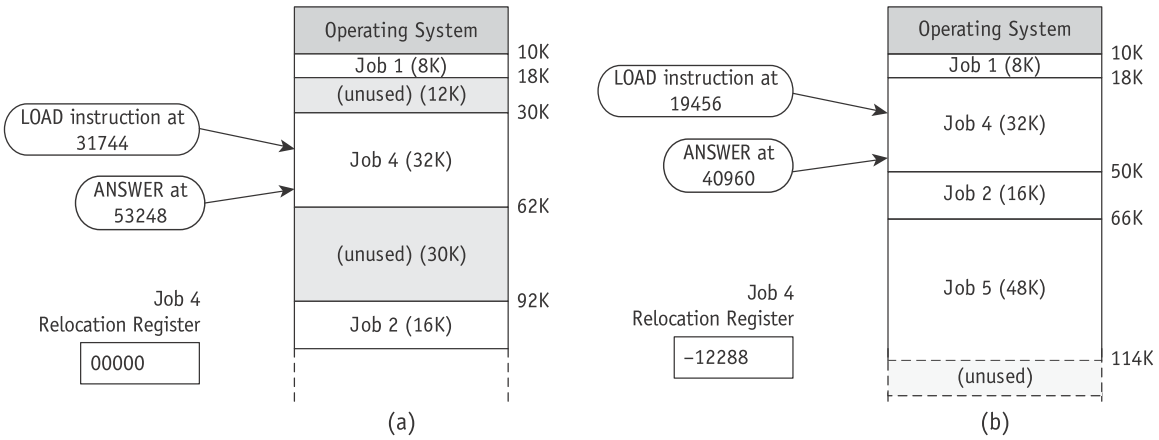
(figure 2.10)

This message indicates that a program attempted to access memory space that is out of bounds. This was a fatal error resulting in the program's termination.

The **relocation register** contains the value that must be added to each address referenced in the program so that the system will be able to access the correct memory addresses after relocation. If a program isn't relocated, the value stored in the program's relocation register is zero. Figure 2.11 illustrates what happens during relocation by using the relocation register (all values are shown in decimal form).

Originally, Job 4 was loaded into memory starting at memory location 30K. (1K equals 1,024 bytes. Therefore, the exact starting address is: 30 * 1024 = 30720.) It required a block of memory of 32K (or 32 * 1024 = 32,768) addressable locations. Therefore, when it was originally loaded, the job occupied the space from memory location 30720 to memory location 63488-1.

Now, suppose that within the program, at memory location 31744, there's this instruction: LOAD 4, ANSWER. (This assembly language command asks that the data value



(figure 2.11)

Contents of the relocation register for Job 4 before the job's relocation and compaction (a) and after (b).

known as ANSWER be loaded into Register 4 for later computation. In this example, Register 4 is a working/computation register, which is distinct from either the relocation or the bounds registers mentioned earlier.) Similarly, the variable known as ANSWER, the value 37, is stored at memory location 53248 before compaction and relocation.

After relocation, Job 4 resides at a new starting memory address of 18K (to be exact, it is 18432, which is 18 * 1024). The job still has the same number of addressable locations, but those locations are in a different physical place in memory. Job 4 now occupies memory from location 18432 to location 51200-1 and, thanks to the relocation register, all of the addresses will be adjusted accordingly when the program is executed.

What does the relocation register contain? In this example, it contains the value –12288. As calculated previously, 12288 is the size of the free block that has been moved toward the high addressable end of memory where it can be combined with other free blocks of memory. The sign is negative because Job 4 has been moved back, closer to the low addressable end of memory, as shown at the top of Figure 2.10(b).

If the addresses were not adjusted by the value stored in the relocation register, then even though memory location 31744 is still part of Job 4's accessible set of memory locations, it would not contain the LOAD command. Not only that, but the other location, 53248, would be out of bounds. The instruction that was originally at 31744 has been moved to location 19456. That's because all of the instructions in this program have been moved back by 12K (12 * 1024 = 12,288), which is the size of the free block. Therefore, location 53248 has been displaced by –12288 and ANSWER, the data value 37, is now located at address 40960.

However, the precise LOAD instruction that was part of the original job has not been changed. The instructions inside the job are not revised in any way. And the original address where ANSWER had been stored, 53248, remains unchanged in the program no matter how many times Job 4 is relocated. Before the instruction is executed, the true address is computed by adding the value stored in the relocation register to the address found at that instruction. By changing only the value in the relocation register each time the job is moved in memory, this technique allows every job address to be properly adjusted.

In effect, by compacting and relocating, the Memory Manager optimizes the use of memory and thus improves throughput—an important measure of system performance. An unfortunate side effect is that this memory allocation scheme requires more overhead than with the previous schemes. The crucial factor here is the timing of the compaction—when and how often it should be done. There are three options.

- One approach is to perform compaction when a certain percentage of memory becomes busy—say 75 percent. The disadvantage of this approach is that the system would incur unnecessary overhead if no jobs were waiting to use the remaining 25 percent.
- A second approach is to compact memory only when there are jobs waiting to get in. This would entail constant checking of the entry queue, which might result in unnecessary overhead and thus slow down the processing of jobs already in the system.
- A third approach is to compact memory after a prescribed amount of time has elapsed. If the amount of time chosen is too small, however, then the system will spend more time on compaction than on processing. If it's too large, too many jobs will congregate in the waiting queue and the advantages of compaction are lost.

Each option has its good and bad points. The best choice for any system is decided by the operating system designers who, based on the job mix and other factors, try to optimize both processing time and memory use while keeping overhead as low as possible.

Conclusion

Four memory management techniques were presented in this chapter: single-user systems, fixed partitions, dynamic partitions, and relocatable dynamic partitions. Although they have differences, they all share the requirement that the entire program (1) be loaded into memory, (2) be stored contiguously, and (3) remain in memory until the entire job is completed. Consequently, each puts severe restrictions on the size of executable jobs because each one can be only as large as the biggest partition in memory. These schemes laid the groundwork for more complex memory management techniques that allowed users to interact more directly and more often with the system.

In the next chapter, we examine several memory allocation schemes that had two new things in common. First, they removed the requirement that jobs be loaded into contiguous memory locations. This allowed jobs to be subdivided so they could be loaded anywhere in memory (if space allowed) as long as the pieces were linked to each other in some way. The second requirement no longer needed was that the entire job reside in memory throughout its execution. By eliminating these mandates, programs were able to grow in size and complexity, opening the door to virtual memory.

Key Terms

address: a number that designates a particular memory location.

best-fit memory allocation: a main memory allocation scheme that considers all free blocks and selects for allocation the one that will result in the least amount of wasted space.

bounds register: a register used to store the highest location in memory legally accessible by each program.

compaction of memory: the process of collecting fragments of available memory space into contiguous blocks by relocating programs and data in a computer's memory.

deallocation: the process of freeing an allocated resource, whether memory space, a device, a file, or a CPU.

dynamic partitions: a memory allocation scheme in which jobs are given as much memory as they request when they are loaded for processing, thus creating their own partitions in main memory.

external fragmentation: a situation in which the dynamic allocation of memory creates unusable fragments of free memory between blocks of busy, or allocated, memory.

first-fit memory allocation: a main memory allocation scheme that searches from the beginning of the free block list and selects for allocation the first block of memory large enough to fulfill the request.

fixed partitions: a memory allocation scheme in which main memory is sectioned with one partition assigned to each job.

internal fragmentation: a situation in which a partition is only partially used by the program; the remaining space within the partition is unavailable to any other job and is therefore wasted.

main memory: the unit that works directly with the CPU and in which the data and instructions must reside in order to be processed. Also called *random access memory (RAM)*, *primary storage*, or *internal memory*.

null entry: an empty entry in a list.

RAM: another term for *main memory*.

relocatable dynamic partitions: a memory allocation scheme in which the system relocates programs in memory to gather together all empty blocks and compact them to make one block of memory that's large enough to accommodate some or all of the jobs waiting for memory.

relocation: (1) the process of moving a program from one area of memory to another; or (2) the process of adjusting address references in a program, by either software or hardware means, to allow the program to execute correctly when loaded in different sections of memory.

relocation register: a register that contains the value that must be added to each address referenced in the program so that the Memory Manager will be able to access the correct memory addresses.

Interesting Searches

For more background on a few of the topics discussed in this chapter, begin a search with these terms.

- Core Memory Technology
- technikum29 Museum of Computer and Communication Technology
- How RAM Works
- Static vs. Dynamic Partitions
- Internal vs. External Fragmentation

Exercises

Research Topics

A. Identify a computer (a desktop or laptop computer) that can accommodate an upgrade or addition of internal memory. Research the brand or type of memory chip that could be used in that system. Do not perform the installation. Instead, list the steps you would take to install the memory. Be sure to cite your sources.

B. For a platform of your choice, research the growth in the size of main memory from the time the platform was developed to the present day. Create a chart showing milestones in memory growth and the approximate date. Choose one from laptops, desktops, midrange computers, and mainframes. Be sure to mention the organization that performed the research and cite your sources.

Exercises

1. Describe a present-day computing environment that might use each of the memory allocation schemes (single user, fixed, dynamic, and relocatable dynamic) described in the chapter. Defend your answer by describing the advantages and disadvantages of the scheme in each case.
2. How often should memory compaction/relocation be performed? Describe the advantages and disadvantages of performing it even more often than recommended.
3. Using your own words, explain the role of the bounds register.
4. Describe in your own words the role of the relocation register.
5. Give an example of computing circumstances that would favor first-fit allocation over best-fit. Explain your answer.
6. Given the following information:

Job List:		**Memory Block List:**	
Job Number	**Memory Requested**	**Memory Block**	**Memory Block Size**
Job A	690K	Block 1	900K (low-order memory)
Job B	275K	Block 2	910K
Job C	760K	Block 3	300K (high-order memory)

   a. Use the first-fit algorithm to indicate which memory blocks are allocated to each of the three arriving jobs.
   b. Use the best-fit algorithm to indicate which memory blocks are allocated to each of the three arriving jobs.

7. Given the following information:

Job List:		**Memory Block List:**	
Job Number	**Memory Requested**	**Memory Block**	**Memory Block Size**
Job A	57K	Block 1	900K (low-order memory)
Job B	920K	Block 2	910K
Job C	50K	Block 3	200K
Job D	701K	Block 4	300K (high-order memory)

   a. Use the best-fit algorithm to indicate which memory blocks are allocated to each of the three arriving jobs.
   b. Use the first-fit algorithm to indicate which memory blocks are allocated to each of the three arriving jobs.

8. Next-fit is an allocation algorithm that starts by using the first-fit algorithm but keeps track of the partition that was last allocated. Instead of restarting the search with Block 1, it starts searching from the most

recently allocated block when a new job arrives. Using the following information:

Job List:		Memory Block List:	
Job Number	**Memory Requested**	**Memory Block**	**Memory Block Size**
Job A	590K	Block 1	100K (low-order memory)
Job B	50K	Block 2	900K
Job C	275K	Block 3	280K
Job D	460K	Block 4	600K (high-order memory)

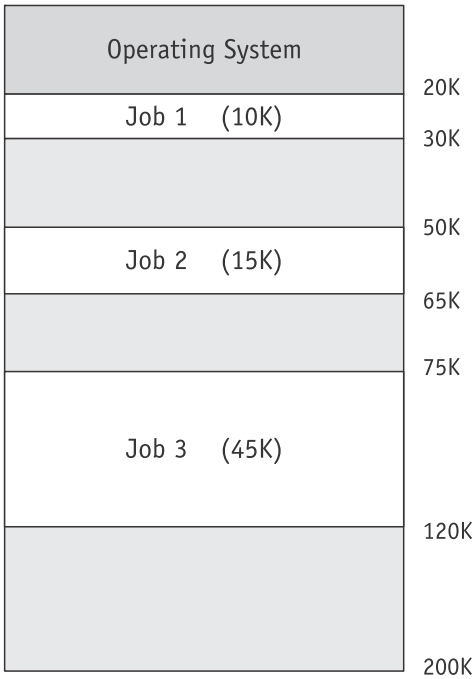
Indicate which memory blocks are allocated to each of the three arriving jobs, and explain in your own words what advantages the next-fit algorithm could offer.

9. Worst-fit is an allocation algorithm that allocates the largest free block to a new job. It is the opposite of the best-fit algorithm. Compare it to next-fit conditions given in Exercise 8 and explain in your own words what advantages the worst-fit algorithm could offer.
10. Imagine an operating system that cannot perform memory deallocation. Name at least three effects on overall system performance that might result and explain your answer.
11. In a system using the relocatable dynamic partitions scheme, given the following situation (and using decimal form): Job Q is loaded into memory starting at memory location 42K.
    a. Calculate the exact starting address for Job Q in bytes.
    b. If the memory block has 3K in fragmentation, calculate the size of the memory block.
    c. Is the resulting fragmentation internal or external? Explain your reasoning.
12. In a system using the relocatable dynamic partitions scheme, given the following situation (and using decimal form): Job W is loaded into memory starting at memory location 5000K.
    a. Calculate the exact starting address for Job W in bytes.
    b. If the memory block has 3K in fragmentation, calculate the size of the memory block.
    c. Is the resulting fragmentation internal or external? Explain your reasoning.
13. If the relocation register holds the value -83968, was the relocated job moved toward the lower or higher addressable end of main memory? By how many kilobytes was it moved? Explain your conclusion.
14. In a system using the fixed partitions memory allocation scheme, given the following situation (and using decimal form): After Job J is loaded into a partition of size 50K, the resulting fragmentation is 7168 bytes. Perform the following:
    a. What is the size of Job J in bytes?
    b. What type of fragmentation is caused?

15. In a system using the dynamic partition memory allocation scheme, given the following situation (and using decimal form): After Job C of size 70K is loaded into a partition resulting in 7K of fragmentation, calculate the size (in bytes) of its partition and identify the type of fragmentation that is caused. Explain your answer.

Advanced Exercises

16. The relocation example presented in the chapter implies that compaction is done entirely in memory, without secondary storage. Can all free sections of memory be merged into one contiguous block using this approach? Why or why not?
17. In this chapter we described one way to compact memory. Some people suggest an alternate method: all jobs could be copied to a secondary storage device and then reloaded (and relocated) contiguously into main memory, thus creating one free block after all jobs have been recopied into memory. Is this viable? Could you devise a better way to compact memory? Write your algorithm and explain why it is better.
18. Given the memory configuration in Figure 2.12, answer the following questions. At this point, Job 4 arrives requesting a block of 100K.
    a. Can Job 4 be accommodated? Why or why not?
    b. If relocation is used, what are the contents of the relocation registers for Job 1, Job 2, and Job 3 after compaction?
    c. What are the contents of the relocation register for Job 4 after it has been loaded into memory?



(figure 2.12)

Memory configuration for Exercise 18.

    d. An instruction that is part of Job 1 was originally loaded into memory location 22K. What is its new location after compaction?
    e. An instruction that is part of Job 2 was originally loaded into memory location 55K. What is its new location after compaction?
    f. An instruction that is part of Job 3 was originally loaded into memory location 80K. What is its new location after compaction?
    g. If an instruction was originally loaded into memory location 110K, what is its new location after compaction?

Programming Exercises

19. Here is a long-term programming project. Use the information that follows to complete this exercise.

Job List

Job Stream Number	Time	Job Size
1	5	5760
2	4	4190
3	8	3290
4	2	2030
5	2	2550
6	6	6990
7	8	8940
8	10	740
9	7	3930
10	6	6890
11	5	6580
12	8	3820
13	9	9140
14	10	420
15	10	220
16	7	7540
17	3	3210
18	1	1380
19	9	9850
20	3	3610
21	7	7540
22	2	2710
23	8	8390
24	5	5950
25	10	760

Memory List

Memory Block	Size
1	9500
2	7000
3	4500
4	8500
5	3000
6	9000
7	1000
8	5500
9	1500
10	500

At one large batch-processing computer installation, the management wants to decide what storage placement strategy will yield the best possible performance. The installation runs a large real storage computer (as opposed to "virtual" storage, which is covered in Chapter 3) under fixed partition multiprogramming. Each user program runs in a single group of contiguous storage locations. Users state their storage requirements and time units for CPU usage on their Job Control Card (it used to, and still does, work this way, although cards may not be used). The operating system allocates to each user the appropriate partition and starts up the user's job. The job remains in memory until completion. A total of 50,000 memory locations are available, divided into blocks as indicated in the previous table.

a. Write (or calculate) an event-driven simulation to help you decide which storage placement strategy should be used at this installation. Your program would use the job stream and memory partitioning as indicated previously. Run the program until all jobs have been executed with the memory as is (in order by address). This will give you the first-fit type performance results.

b. Sort the memory partitions by size and run the program a second time; this will give you the best-fit performance results. For both parts a. and b., you are investigating the performance of the system using a typical job stream by measuring:

    1. Throughput (how many jobs are processed per given time unit)
    2. Storage utilization (percentage of partitions never used, percentage of partitions heavily used, and so on)
    3. Waiting queue length
    4. Waiting time in queue
    5. Internal fragmentation

Given that jobs are served on a first-come, first-served basis:

c. Explain how the system handles conflicts when jobs are put into a waiting queue and there are still jobs entering the system—which job goes first?

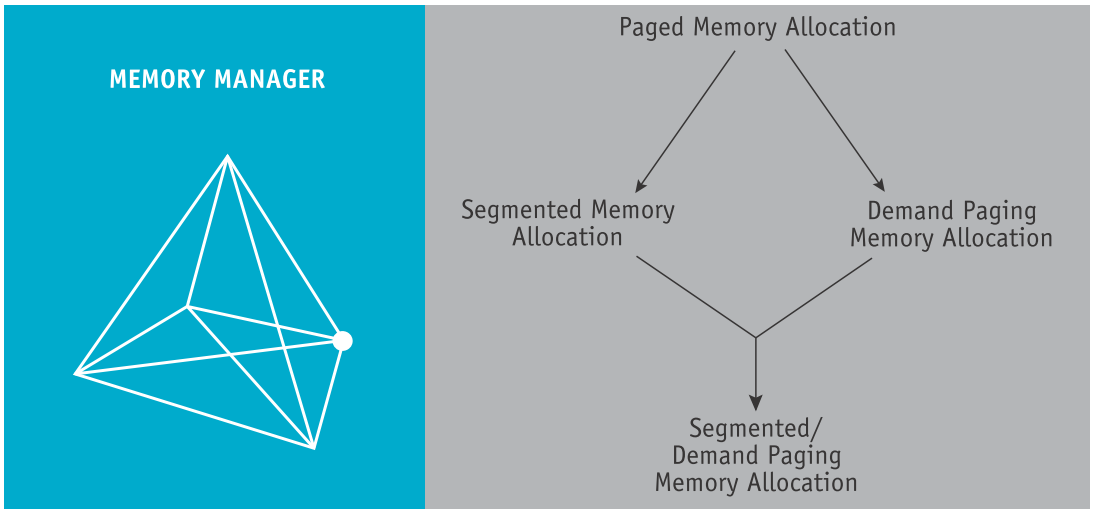
d. Explain how the system handles the "job clocks," which keep track of the amount of time each job has run, and the "wait clocks," which keep track of how long each job in the waiting queue has to wait.

e. Since this is an event-driven system, explain how you define "event" and what happens in your system when the event occurs.

f. Look at the results from the best-fit run and compare them with the results from the first-fit run. Explain what the results indicate about the performance of the system for this job mix and memory organization. Is one method of partitioning better than the other? Why or why not? Could you recommend one method over the other given your sample run? Would this hold in all cases? Write some conclusions and recommendations.

20. Suppose your system (as explained in Exercise 19) now has a "spooler" (a storage area in which to temporarily hold jobs), and the job scheduler can choose which will be served from among 25 resident jobs. Suppose also that the first-come, first-served policy is replaced with a "faster-job, first-served" policy. This would require that a sort by time be performed on the job list before running the program. Does this make a difference in the results? Does it make a difference in your analysis? Does it make a difference in your conclusions and recommendations? The program should be run twice to test this new policy with both best-fit and first-fit.
21. Suppose your spooler (as described in Exercise 19) replaces the previous policy with one of "smallest-job, first-served." This would require that a sort by job size be performed on the job list before running the program. How do the results compare to the previous two sets of results? Will your analysis change? Will your conclusions change? The program should be run twice to test this new policy with both best-fit and first-fit.

Chapter 3 Memory Management: Virtual Memory Systems



"Nothing is so much strengthened by practice, or weakened by neglect, as memory."

—Quintillian *(A.D. 35–100)*

Learning Objectives

After completing this chapter, you should be able to describe:

- The basic functionality of the memory allocation methods covered in this chapter: paged, demand paging, segmented, and segmented/demand paged memory allocation
- The influence that these page allocation methods have had on virtual memory
- The difference between a first-in first-out page replacement policy, a least-recently-used page replacement policy, and a clock page replacement policy
- The mechanics of paging and how a memory allocation scheme determines which pages should be swapped out of memory
- The concept of the working set and how it is used in memory allocation schemes
- Cache memory and its role in improving system response time

In this chapter we'll follow the evolution of memory management with four new memory allocation schemes. They remove the restriction of storing the programs contiguously, and most of them eliminate the requirement that the entire program reside in memory during its execution. Our concluding discussion of cache memory will show how its use improves the performance of the Memory Manager.

Paged Memory Allocation

Paged memory allocation is based on the concept of dividing jobs into units of equal size and each unit is called a **page.** Some operating systems choose a page size that is the exact same size as a section of main memory, which is called a **page frame.** Likewise, the sections of a magnetic disk are called **sectors** or blocks. The paged memory allocation scheme works quite efficiently when the pages, sectors, and page frames are all the same size. The exact size (the number of bytes that can be stored in each of them) is usually determined by the disk's sector size. Therefore, one sector will hold one page of job instructions (or data) and fit into one page frame of memory, but because this is the smallest addressable chunk of disk storage, it isn't subdivided even for very tiny jobs.

Before executing a program, a basic Memory Manager prepares it by:

1. Determining the number of pages in the program
2. Locating enough empty page frames in main memory
3. Loading all of the program's pages into those frames

When the program is initially prepared for loading, its pages are in logical sequence—the first pages contain the first instructions of the program and the last page has the last instructions. We refer to the program's instructions as "bytes" or "words."

The loading process is different from the schemes we studied in Chapter 2 because the pages do not have to be loaded in adjacent memory blocks. With the paged memory allocation, they can be loaded in noncontiguous page frames. In fact, each page can be stored in any available page frame anywhere in main memory.

The primary advantage of storing programs in noncontiguous page frames is that main memory is used more efficiently because an empty page frame can be used by any page of any job. In addition, the compaction scheme used for relocatable partitions is eliminated because there is no external fragmentation between page frames.

However, with every new solution comes a new problem. Because a job's pages can be located anywhere in main memory, the Memory Manager now needs a mechanism to keep track of them—and that means enlarging the size, complexity, and overhead of the operating system software.

In our examples, the first page is Page 0, and the second is Page 1, and so on. Page frames are numbered the same way, starting with zero.

The simplified example in Figure 3.1 shows how the Memory Manager keeps track of a program that is four pages long. To simplify the arithmetic, we've arbitrarily set the page size at 100 bytes. Job 1 is 350 bytes long and is being readied for execution. (The Page Map Table for the first job is shown later in Table 3.2.)

Notice in Figure 3.1 that the last page frame (Page Frame 11) is not fully utilized because Page 3 is less than 100 bytes—the last page uses only 50 of the 100 bytes available. In fact, very few jobs perfectly fill all of the pages, so internal fragmentation is still a problem, but only in the job's last page frame.

In Figure 3.1 (with seven free page frames), the operating system can accommodate an incoming job of up to 700 bytes because it can be stored in the seven empty page frames. But a job that is larger than 700 bytes cannot be accommodated until Job 1 ends its execution and releases the four page frames it occupies. And any job larger than 1100 bytes will never fit into the memory of this tiny system. Therefore, although paged memory allocation offers the advantage of noncontiguous storage, it still requires that the entire job be stored in memory during its execution.

(figure 3.1)

In this example, each page frame can hold 100 bytes. This job, at 350 bytes long, is divided among four page frames with internal fragmentation in the last page frame.

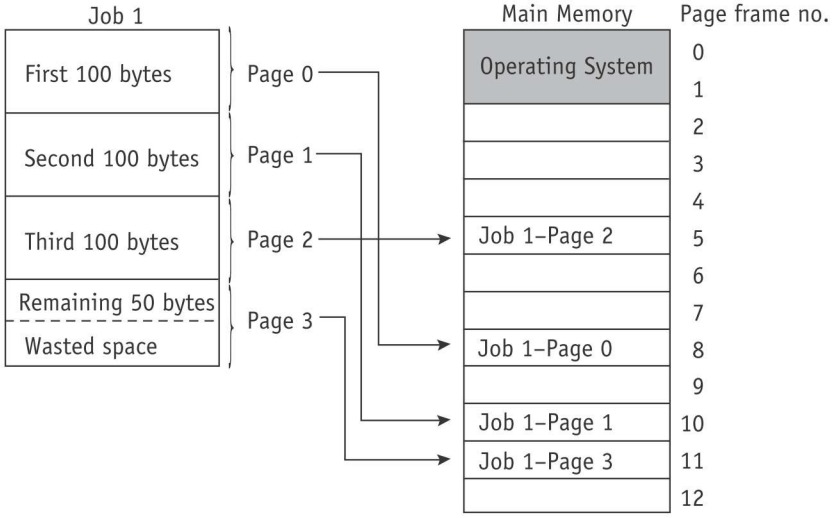


Figure 3.1 uses arrows and lines to show how a job's pages fit into page frames in memory, but the Memory Manager uses tables to keep track of them. There are essentially three tables that perform this function: the Job Table, the Page Map Table, and the Memory Map Table. Although different operating systems may have different names for them, the tables provide the same service regardless of the names they are given. All three tables reside in the part of main memory that is reserved for the operating system.

As shown in Table 3.1, the **Job Table** (**JT**) contains two values for each active job: the size of the job (shown on the left) and the memory location where its Page Map Table is stored (on the right). For example, the first job has a size of 400 and is at location 3096 in memory. The Job Table is a dynamic list that grows as jobs are loaded into the system and shrinks, as shown in (b) in Table 3.1, as they are later completed.

Job Table	
Job Size	PMT Location
400	3096
200	3100
500	3150

(a)

Job Table	
Job Size	PMT Location
400	3096
500	3150

(b)

Job Table	
Job Size	PMT Location
400	3096
700	3100
500	3150

(c)

(table 3.1)

This section of the Job Table initially has one entry for each job (a). When the second job ends (b), its entry in the table is released and then replaced by the entry for the next job (c).

Each active job has its own **Page Map Table (PMT)**, which contains the vital information for each page—the page number and its corresponding memory address of the page frame. Actually, the PMT includes only one entry per page. The page numbers are sequential (Page 0, Page 1, Page 2, and so on), so it isn't necessary to list each page number in the PMT. The first entry in the PMT always lists the page frame memory address for Page 0, the second entry is the address for Page 1, and so on.

A simple **Memory Map Table (MMT)** has one entry for each page frame and shows its location and its free/busy status.

At compilation time, every job is divided into pages. Using Job 1 from Figure 3.1, we can see how this works:

- Page 0 contains the first hundred bytes.
- Page 1 contains the second hundred bytes.
- Page 2 contains the third hundred bytes.
- Page 3 contains the last 50 bytes.

As you can see, the program has 350 bytes; but when they are stored, the system numbers them starting from 0 through 349. Therefore, the system refers to them as Byte 0 through Byte 349.

How far away is a certain byte from the beginning of its page frame? This value is called the **displacement** (also called the offset) and it is a relative factor that's used to locate one certain byte within its page frame.

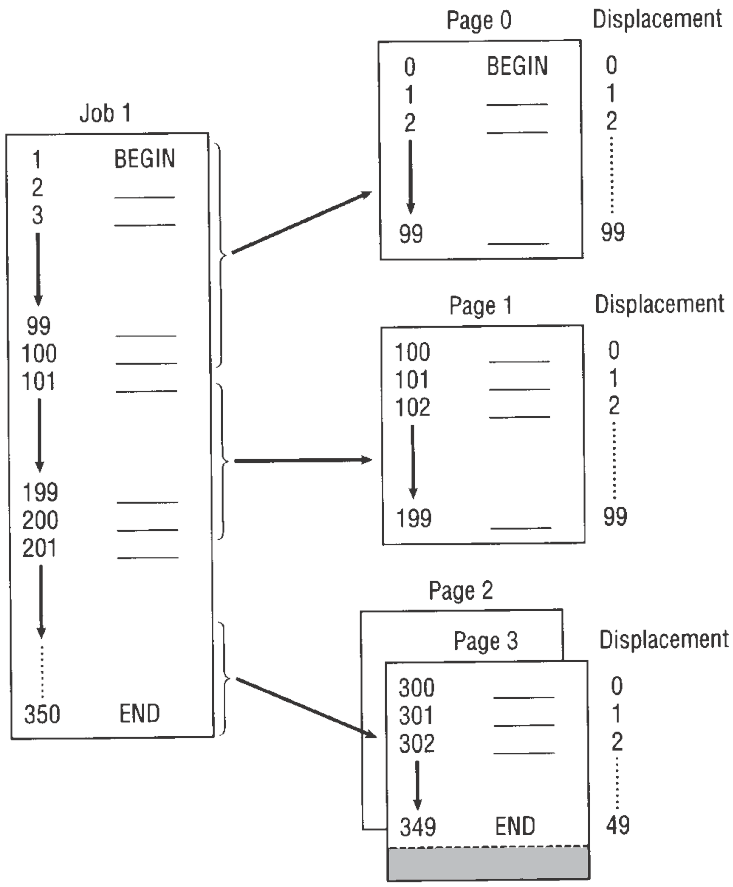
Here's how it works. In the simplified example shown in Figure 3.2, Bytes 0, 100, 200, and 300 are the first bytes for pages 0, 1, 2, and 3, respectively, so each has a displacement of zero. To state it another way, the distance from the beginning of the page frame to Byte 200 is zero bytes. Likewise, if the Memory Manager needs to access Byte 314, it can first go to the page frame containing Page 3 and then go to Byte 14 (the fifteenth) in that page frame.

Because the first byte of each page has a displacement of zero, and the last byte has a displacement of 99, the operating system can access the correct bytes by using its relative position from the beginning of its page frame.

While the computer hardware performs the calculations, the operating system is responsible for maintaining the tables that track the allocation and deallocation of storage.

(figure 3.2)

This job is 350 bytes long and is divided into four pages of 100 bytes each that are loaded into four page frames in memory.



In this example, it is easy for us to see intuitively that all bytes numbered less than 100 will be on Page 0, all bytes numbered greater than or equal to 100 but less than 200 will be on Page 1, and so on. (That is the advantage of choosing a convenient fixed page size, such as 100 bytes.) The operating system uses an algorithm to calculate the page and displacement using a simple arithmetic calculation:

BYTE_NUMBER_TO_BE_LOCATED / PAGE_SIZE = PAGE_NUMBER

To do the same calculation using long division, it would look like this:

```
                    page number
page size ) byte number to be located
            xxx
              xxx
              xxx
                displacement
```

For example, if we use 100 bytes as the page size, the page number and the displacement (the location within that page) of Byte 276 can be calculated as:

276 / 100 = 2 (with a remainder of 76)

The quotient (2) is the page number, and the remainder (76) is the displacement. Therefore, we know that the byte is located on Page 2, at Byte 76, which is exactly 77 bytes from the top of the page (remember that the first byte is located at zero).

Let's try another example with a more common page size of 4096 bytes. If we are seeking the location of Byte 6144, then we divide 6144 by 4096, and the result is 1.5 (which is 1 with a remainder of 2048). Therefore, the byte we are seeking is located midway on the second page (Page 1). To find the exact location, multiply the page size (4096) by the decimal (0.5) to discover that the byte we're seeking is located at Byte 2048 of Page 1 (which is exactly 2049 bytes from the start of the page).

Using the concepts just presented, and using the same parameters from the first example, answer these questions:

1. Could the operating system (or the hardware) get a page number that is greater than 3 if the program was searching for Byte 276?
2. If it did, what should the operating system do?
3. Could the operating system get a remainder/displacement of more than 99?
4. What is the smallest remainder/displacement possible?

Here are the answers:

1. No.
2. Send an error message and stop processing the program (because the page is out of bounds).
3. No.
4. Zero.

This procedure gives the location of an instruction with respect to the job's pages. However, these pages numbers are logical, not physical. Each page is actually stored in a page frame that can be located anywhere in available main memory. Therefore, the paged memory allocation algorithm needs to be expanded to find the exact location of the byte in main memory. To do so, we need to correlate each of the job's pages with its page frame number using the job's Page Map Table.

For example, if we look at the PMT for Job 1 from Figure 3.1, we see that it looks like the data in Table 3.2.

Page Number	Page Frame Number
0	8
1	10
2	5
3	11

(table 3.2)

Page Map Table for Job 1 in Figure 3.1.

In the first division example, we were looking for an instruction with a displacement of 76 on Page 2. To find its exact location in memory, the operating system (or the hardware) conceptually has to perform the following four steps.

STEP 1 Do the arithmetic computation just described to determine the page number and displacement of the requested byte: 276/100 = 2 with remainder of 76.

- Page number = the integer resulting from the division of the job space address by the page size = 2
- Displacement = the remainder from the page number division = 76

STEP 2 Refer to this job's PMT (shown in Table 3.2) and find out which page frame contains Page 2.

PAGE 2 is located in PAGE FRAME 5.

STEP 3 Get the address of the beginning of the page frame by multiplying the page frame number (5) by the page frame size (100).

```
ADDRESS_PAGE-FRAME = NUMBER_PAGE-FRAME * SIZE_PAGE-FRAME
ADDRESS_PAGE-FRAME = 5 * 100 = 500
```

STEP 4 Now add the displacement (calculated in Step 1) to the starting address of the page frame to compute the precise location in memory of the instruction:

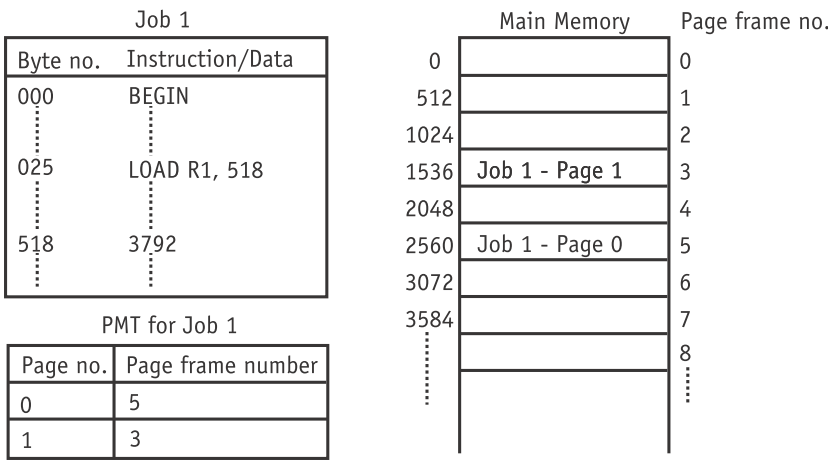
```
INSTR_ADDRESS_IN_MEMORY = ADDRESS_PAGE-FRAME + DISPLACEMENT
INSTR_ADDRESS_IN_MEMORY = 500 + 76 = 576
```

The result of these calculations tells us exactly where (location 576) the requested byte (Byte 276) is located in main memory.

Here is another example. It follows the hardware and the operating system as an assembly language program is run with a LOAD instruction (the instruction LOAD R1, 518 tells the system to load into Register 1 the value found at Byte 518).

In Figure 3.3, the page frame size is set at 512 bytes each and the page size is also 512 bytes for this system. From the PMT we can see that this job has been divided into two pages. To find the exact location of Byte 518 (where the system will find the value to load into Register 1), the system will do the following:

1. Compute the page number and displacement—the page number is 1, and the displacement is 6: (518/512) = 1 with a remainder of 6.
2. Go to the job's Page Map Table and retrieve the appropriate page frame number for Page 1. (Page 1 can be found in Page Frame 3).
3. Compute the starting address of the page frame by multiplying the page frame number by the page frame size: (3 * 512 = 1536).



(figure 3.3)

This system has page frame and page sizes of 512 bytes each. The PMT shows where the job's two pages are loaded into available page frames in main memory.

4. Calculate the exact address of the instruction in main memory by adding the displacement to the starting address: (1536 + 6 = 1542). Therefore, memory address 1542 holds the value we are looking for that should be loaded into Register 1.

As you can see, this is not a simple operation. Every time an instruction is executed, or a data value is used, the operating system (or the hardware) must translate the job space address, which is logical, into its physical address, which is absolute. By doing so, we are resolving the address, called **address resolution** or address translation. All of this processing is overhead, which takes processing capability away from the running jobs and those waiting to run. (Technically, in most systems, the hardware does most of the address resolution, which reduces some of this overhead.)

A huge advantage of a paging scheme is that it allows jobs to be allocated in noncontiguous memory locations, allowing memory to be used more efficiently. (Recall that the simpler memory management systems described in Chapter 2 required all of a job's pages to be stored contiguously—one next to the other.) However, there are disadvantages— overhead is increased and internal fragmentation is still a problem, although it occurs only in the last page of each job. The key to the success of this scheme is the size of the page. A page size that is too small will generate very long PMTs (with a corresponding increase in overhead), while a page size that is too large will result in excessive internal fragmentation. Determining the best page size is an important policy decision—there are no hard and fast rules that will guarantee optimal use of resources, and it is a problem we'll see again as we examine other paging alternatives. The best size depends on the nature of the jobs being processed and on the constraints placed on the system.

Demand Paging Memory Allocation

Demand paging introduced the concept of loading only a part of the program into memory for processing. It was the first widely used scheme that removed the restriction of having the entire job in memory from the beginning to the end of its processing.

With demand paging, the pages are loaded as each is requested. This requires high-speed access to the pages.

With demand paging, jobs are still divided into equally-sized pages that initially reside in secondary storage. When the job begins to run, its pages are brought into memory only as they are needed, and if they're never needed, they're never loaded.

Demand paging takes advantage of the fact that programs are written sequentially so that while one section, or module, is processed, other modules may be idle. Not all the pages are accessed at the same time, or even sequentially. For example:

- Instructions to handle errors are processed only when a specific error is detected during execution. For instance, these instructions can indicate that input data was incorrect or that a computation resulted in an invalid answer. If no error occurs, and we hope this is generally the case, these instructions are never processed and never need to be loaded into memory.
- Many program modules are mutually exclusive. For example, while the program is being loaded (when the input module is active), then the processing module is inactive because it is generally not performing calculations during the input stage. Similarly, if the processing module is active, then the output module (such as printing) may be idle.
- Certain program options are either mutually exclusive or not always accessible. For example, when a program gives the user several menu choices, as shown in Figure 3.4, it allows the user to make only one selection at a time. If the user selects the first option (FILE), then the module with those program instructions is the only one that is being used, so that is the only module that needs to be in memory at this time. The other modules remain in secondary storage until they are "called." Many tables are assigned a large fixed amount of address space even though only a fraction of the table is actually used. For example, a symbol table might be prepared to handle 100 symbols. If only 10 symbols are used, then 90 percent of the table remains unused.

One of the most important innovations of demand paging was that it made virtual memory feasible. Virtual memory is discussed later in this chapter.

(figure 3.4)

When you choose one option from the menu of an application program such as this one, the other modules that aren't currently required (such as Help) don't need to be moved into memory immediately.

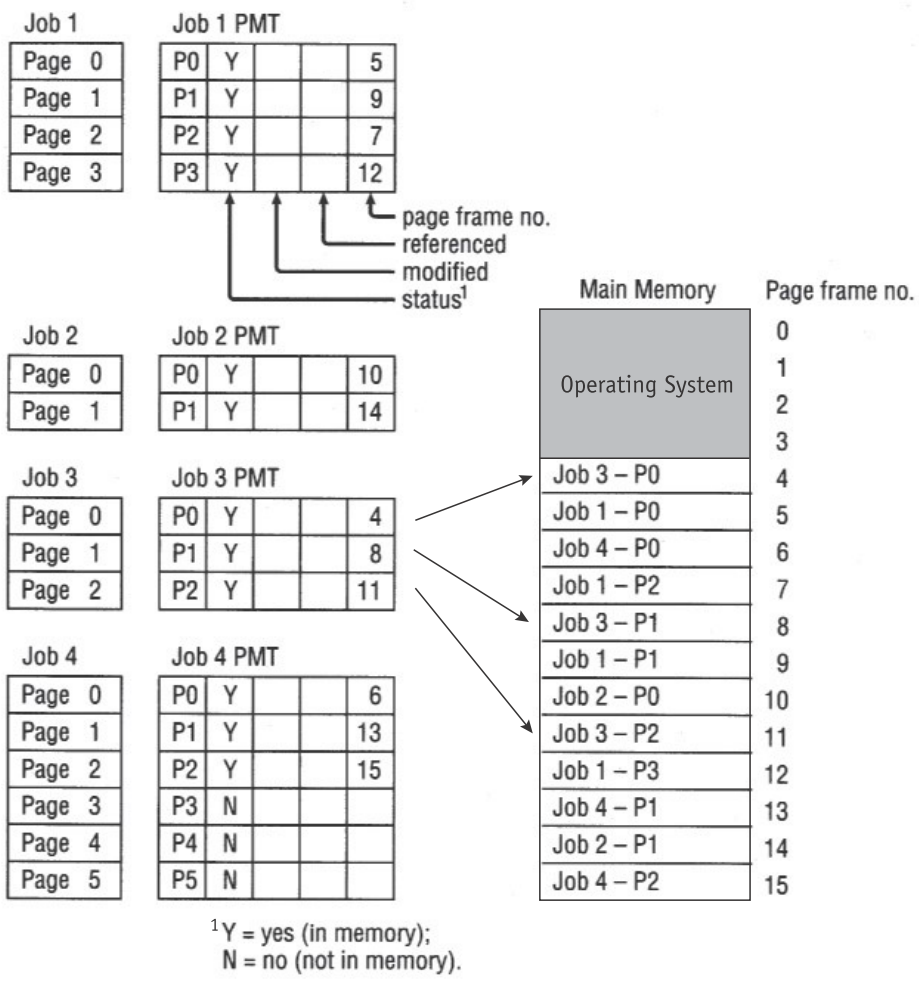
The demand paging scheme allows the user to run jobs with less main memory than is required if the operating system is using the paged memory allocation scheme described earlier. In fact, a demand paging scheme can give the appearance of vast amounts of physical memory when, in reality, physical memory is significantly less than vast.

The key to the successful implementation of this scheme is the use of a high-speed direct access storage device (often shortened to DASD), which will be discussed in Chapter 7. (Examples of DASDs include hard drives or flash memory.) High speed

access is vital because this scheme requires pages to be passed quickly from secondary storage to main memory and back again as they are needed.

How and when the pages are passed (also called swapped) between main memory and secondary storage depends on predefined policies that determine when to make room for needed pages and how to do so. The operating system relies on tables (such as the Job Table, the Page Map Tables, and the Memory Map Table) to implement the algorithm. These tables are basically the same as for paged memory allocation. With demand paging, there are three new fields for each page in each PMT: one to determine if the page being requested is already in memory, a second to determine if the page contents have been modified while in memory, and a third to determine if the page has been referenced most recently. The fourth field, which we discussed earlier in this chapter, shows the page frame number, as shown at the top of Figure 3.5.

The memory field tells the system where to find each page. If it is already in memory (shown here with a Y), the system will be spared the time required to bring it from secondary storage. As one can imagine, it's faster for the operating system to scan a table



(figure 3.5)

Demand paging requires that the Page Map Table for each job keep track of each page as it is loaded or removed from main memory. Each PMT tracks the status of the page, whether it has been modified, whether it has been recently referenced, and the page frame number for each page currently in main memory. (Note: For this illustration, the Page Map Tables have been simplified. See Table 3.3 for more detail.)

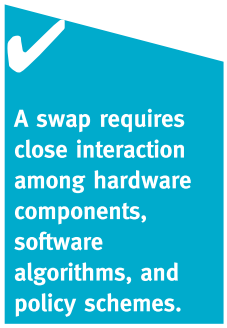
located in main memory than it is to retrieve a page from a disk or other secondary storage device.

The modified field, noting whether or not the page has been changed, is used to save time when pages are removed from main memory and returned to secondary storage. If the contents of the page haven't been modified, then the page doesn't need to be rewritten to secondary storage. The original, the one that is already there, is correct.

The referenced field notes any recent activity and is used to determine which pages show the most processing activity and which are relatively inactive. This information is used by several page-swapping policy schemes to determine which pages should remain in main memory and which should be swapped out when the system needs to make room for other pages being requested.

The last field shows the page frame number for that page.

For example, in Figure 3.5 the number of total job pages is 15, and the number of total page frames available to jobs is 12. (The operating system occupies the first four of the 16 page frames in main memory.)



A swap requires close interaction among hardware components, software algorithms, and policy schemes.

Assuming the processing status illustrated in Figure 3.5, what happens when Job 4 requests that Page 3 be brought into memory, given that there are no empty page frames available?

To move in a new page, one of the resident pages must be swapped back into secondary storage. Specifically, that includes copying the resident page to the disk (if it was modified) and writing the new page into the newly available page frame.

The hardware components generate the address of the required page, find the page number, and determine whether it is already in memory. Refer to the "Hardware Instruction Processing Algorithm" shown in Appendix A for more information.

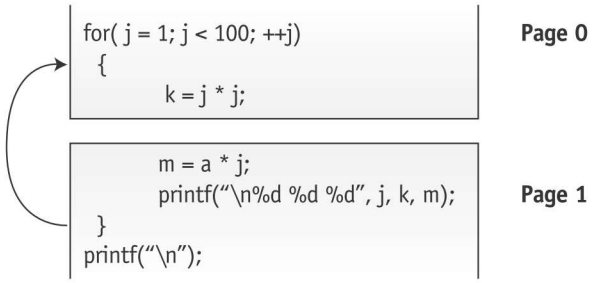
If the Algorithm finds that the page is not in memory, then the operating system software takes over. The section of the operating system that resolves these problems is called the **page fault handler,** and an algorithm to perform that task is also described in Appendix A. The page fault handler determines whether there are empty page frames in memory so that the requested page can be immediately copied from secondary storage. If all page frames are busy, then the page fault handler must decide which page will be swapped out. (This decision is determined by the predefined policy for page removal.) Then the swap is made.

How many tables are changed because of this page swap? Immediately, three tables must be updated: the Page Map Tables for both jobs (the PMT with the page that was swapped out and the PMT with the page that was swapped in) as well as the Memory Map Table. Finally, the instruction that was interrupted is resumed, and processing continues.

Although demand paging is a good solution to improve memory use, it is not free of problems. When there is an excessive amount of **page swapping** between main memory and secondary storage, the operation becomes inefficient. This phenomenon is called **thrashing.**

Thrashing is similar to the problem that students face when comparing explanations of a complex problem in two different textbooks. The amount of time you spend switching back and forth between the two books could cause you to spend more time figuring out where you left off in each book than you spend actually solving the issue at hand.

Caused when pages are frequently removed from memory but are called back shortly thereafter, thrashing uses a great deal of the computer's time and accomplishes very little. Thrashing can occur across jobs, when a large number of jobs are vying for a relatively low number of free pages (that is, the ratio of job pages to free memory page frames is high), or it can happen within a job—for example, in loops that cross page boundaries. We can demonstrate this with a simple example. Suppose that a loop to perform a certain sequence of steps begins at the bottom of one page and is completed at the top of the next page, as demonstrated in the C program in Figure 3.6. This requires that the command sequence go from Page 0 to Page 1 to Page 0 to Page 1 to Page 0, again and again, until the loop is finished.



(figure 3.6)

An example of demand paging that causes a page swap each time the loop is executed and results in thrashing. If only a single page frame is available, this program will have one page fault each time the loop is executed.

If there is only one empty page frame available, the first page is loaded into memory and execution begins. But, in this example, after executing the last instruction on Page 0, that page is swapped out to make room for Page 1. Now execution can continue with the first instruction on Page 1, but at the "}" symbol, Page 1 must be swapped out so Page 0 can be brought back in to continue the loop. Before this program is completed, swapping will have occurred 199 times (unless another page frame becomes free so both pages can reside in memory at the same time). A failure to find a page in memory is often called a **page fault;** this example would generate 199 page faults (and swaps).

Thrashing increases wear and tear on the hardware and slows data access.

In such extreme cases, the rate of useful computation could be degraded by at least a factor of 100. Ideally, a demand paging scheme is most efficient when programmers are aware of the page size used by their operating system and are careful to design their programs to keep page faults to a minimum; but in reality, this is not often feasible.

Page Replacement Policies and Concepts

As we just learned, the policy that selects the page to be removed, the **page replacement policy**, is crucial to the efficiency of the system, and the algorithm to do that must be carefully selected.

Several such algorithms exist, and it is a subject that enjoys a great deal of theoretical attention and research. Two of the most well-known algorithms are first-in first-out and least recently used. The **first-in first-out (FIFO) policy** is based on the assumption that the best page to remove is the one that has been in memory the longest. The **least recently used (LRU) policy** chooses the page least recently accessed to be swapped out.

To illustrate the difference between FIFO and LRU, let us imagine a dresser drawer filled with your favorite sweaters. The drawer is full, but that didn't stop you from buying a new sweater. Now you have to put it away. Obviously it won't fit in your sweater drawer unless you take something out—but which sweater should you move to the storage closet? Your decision will be based on a "sweater removal policy."

You could take out your oldest sweater (the one that was first in), figuring that you probably won't use it again—hoping you won't discover in the following days that it is your most used, most treasured possession. Or, you could remove the sweater that you haven't worn recently and has been idle for the longest amount of time (the one that was least recently used). It is readily identifiable because it is at the bottom of the drawer. But just because it hasn't been used recently doesn't mean that a once-a-year occasion won't demand its appearance soon!

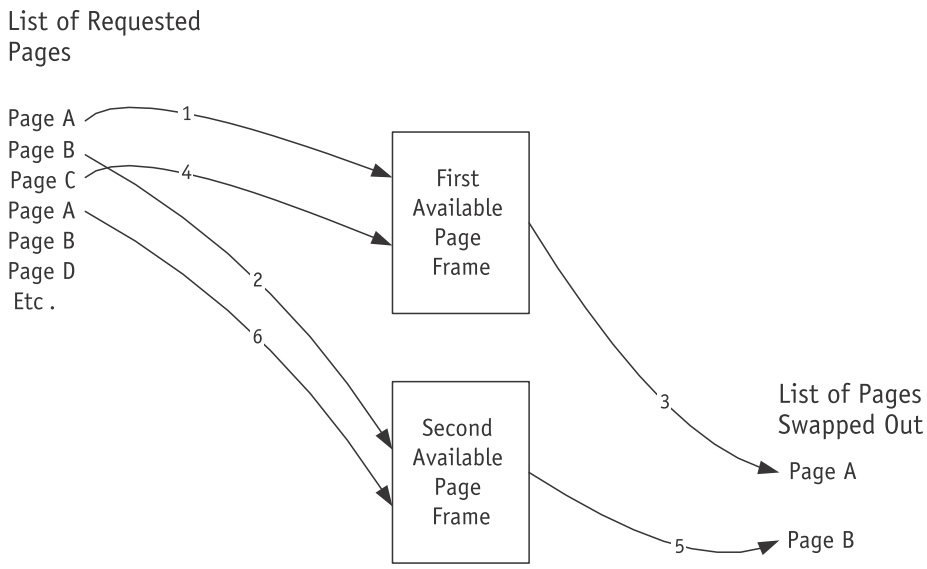
What guarantee do you have that once you have made your choice you won't be trekking to the storage closet to retrieve the sweater you stored there yesterday? You could become a victim of thrashing, going back and forth to swap out sweaters.

Which is the best policy? It depends on the weather, the wearer, and the wardrobe. Of course, one option is to get another drawer. For an operating system (or a computer), this is the equivalent of adding more accessible memory. We explore that option after we discover how to more effectively use the memory we already have.

First-In First-Out

The first-in first-out (FIFO) page replacement policy will remove the pages that have been in memory the longest, that is, those that were "first in." The process of swapping pages is illustrated in Figure 3.7.

- Step 1: Page A is moved into the first available page frame.
- Step 2: Page B is moved into the second available page frame.
- Step 3: Page A is swapped into secondary storage.



(figure 3.7)

First, Pages A and B are loaded into the two available page frames. When Page C is needed, the first page frame is emptied so C can be placed there. Then Page B is swapped out so Page A can be loaded there.

- Step 4: Page C is moved into the first available page frame.
- Step 5: Page B is swapped into secondary storage.
- Step 6: Page A is moved into the second available page frame.

Figure 3.8 demonstrates how the FIFO algorithm works by following a job with four pages (A, B, C, and D) as it is processed by a system with only two available page frames. The job needs to have its pages processed in the following order: A, B, A, C, A, B, D, B, A, C, D.

When both page frames are occupied, each new page brought into memory will cause an interrupt and a page swap into secondary storage. A page interrupt, which we identify with an asterisk (*), is generated anytime a page needs to be loaded into memory, whether a page is swapped out or not. Then we count the number of page interrupts and compute the failure rate and the success rate.

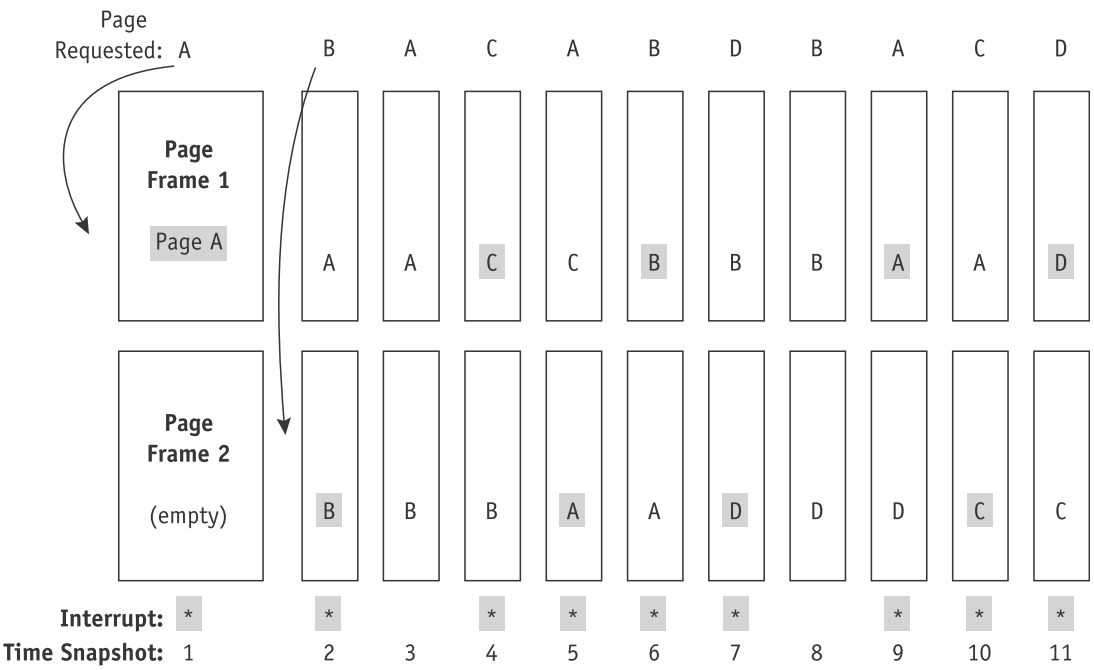
The efficiency of this configuration is dismal—due to the limited number of page frames available, there are 9 page interrupts out of 11 page requests. To calculate the failure rate, we divide the number of interrupts by the number of page requests:

Failure-Rate = Number_of_Interrupts / Page_Requests_Made

The failure rate of this system is 9/11, which is 82 percent. Stated another way, the success rate is 2/11, or 18 percent. A failure rate this high is usually unacceptable.

We are not saying FIFO is bad. We chose this example to show how FIFO works, not to diminish its appeal as a page replacement policy. The high failure rate here is caused by both the limited amount of memory available and the order in which pages are requested by the program. Because the job dictates the page order, that order can't be changed by

In Figure 3.8, using FIFO, Page A is swapped out when a newer page arrives, even though it is used the most often.



(figure 3.8)

Using a FIFO policy, this page trace analysis shows how each page requested is swapped into the two available page frames. When the program is ready to be processed, all four pages are in secondary storage. When the program calls a page that isn't already in memory, a page interrupt is issued, as shown by the gray boxes and asterisks. This program resulted in nine page interrupts.

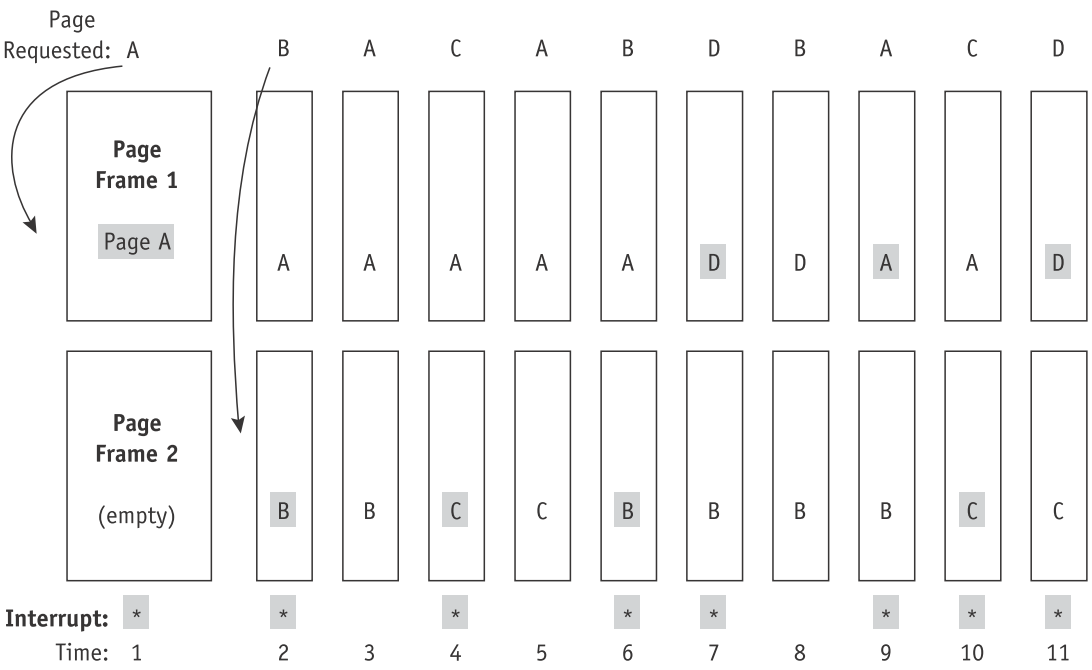
the system, although the size of main memory can be changed. However, buying more memory may not always be the best solution—especially when you have many users and each one wants an unlimited amount of memory. As explained later in this chapter, there is no guarantee that buying more memory will always result in better performance.

Least Recently Used

The least recently used (LRU) page replacement policy swaps out the pages that show the least recent activity, figuring that these pages are the least likely to be used again in the immediate future. Conversely, if a page is used, it is likely to be used again soon; this is based on the theory of locality, explained later in this chapter.

To see how it works, let us follow the same job in Figure 3.8 but using the LRU policy. The results are shown in Figure 3.9. To implement this policy, a queue of requests is kept in FIFO order, a time stamp of when the page entered the system is saved, or a mark in the job's PMT is made periodically.

The efficiency of the configuration for this example is only slightly better than with FIFO. Here, there are 8 page interrupts out of 11 page requests, so the failure rate is 8/11, or 73 percent. In this example, an increase in main memory by one page frame



Using LRU in Figure 3.9, Page A stays in memory longer because it is used most often.

(figure 3.9)

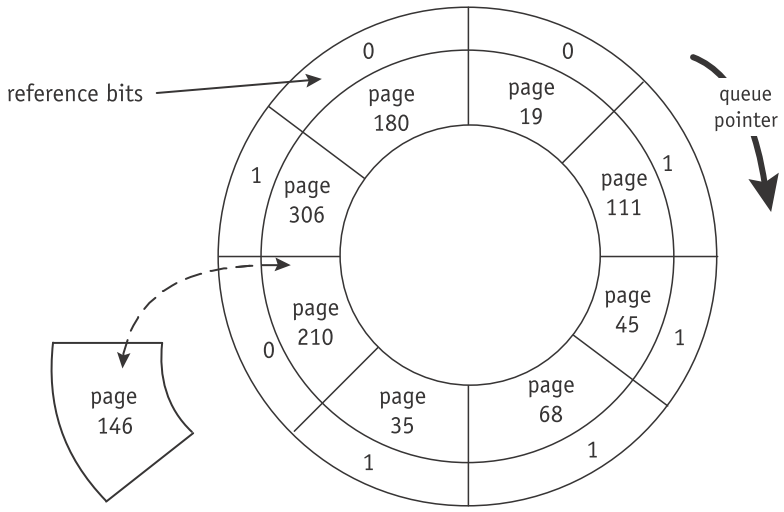
Memory management using an LRU page removal policy for the program shown in Figure 3.8. Throughout the program, 11 page requests are issued, but they cause only 8 page interrupts.

would increase the success rate of both FIFO and LRU. However, we can't conclude on the basis of only one example that one policy is better than the other. In fact, LRU will never cause an increase in the number of page interrupts.

The same cannot be said about FIFO. It has been shown that *under certain circumstances* adding more memory can, *in rare cases*, actually cause an increase in page interrupts when using a FIFO policy. This was first demonstrated by Laszlo Belady in 1969 and is called the **FIFO Anomaly** or the **Belady Anomaly**. Although this is an unusual occurrence, the fact that it exists, coupled with the fact that pages are removed regardless of their activity (as was the case in Figure 3.8), has removed FIFO from the most favored policy position.

Clock Replacement Variation

A variation of the LRU page replacement algorithm is known as the **clock page replacement policy** because it is implemented with a circular queue and uses a pointer to step through the reference bits of the active pages, simulating a clockwise motion. The algorithm is paced according to the computer's **clock cycle**, which is the time span between two ticks in its system clock. The algorithm checks the reference bit for each page. If the bit is one (indicating that it was recently referenced), the bit is reset to zero and the bit for the next page is checked. However, if the reference bit is zero (indicating



(figure 3.10)

A circular queue, which contains the page number and its reference bit. The pointer seeks the next candidate for removal and replaces page 210 with a new page, 146.

that the page has not recently been referenced), that page is targeted for removal. If all the reference bits are set to one, then the pointer must cycle through the entire circular queue again, giving each page a second and perhaps a third or fourth chance. Figure 3.10 shows a circular queue containing the reference bits for eight pages currently in memory. The pointer indicates the page that would be considered next for removal. Figure 3.10 shows what happens to the reference bits of the pages that have been given a second chance. When a new page, 146, has to be allocated to a page frame, it is assigned to the next space that has a reference bit of zero, the space previously occupied by Page 210.

Bit Shifting Variation

A second variation of LRU uses an 8-bit reference byte and a bit-shifting technique to track the usage of each page currently in memory. When the page is first copied into memory, the leftmost bit of its reference byte is set to 1, and all bits to the right of the one are set to zero. See Figure 3.11. At specific time intervals of the clock cycle, the Memory Manager shifts every page's reference bytes to the right by one bit, dropping their rightmost bit. Meanwhile, each time a page is referenced, the leftmost bit of its reference byte is set to 1.

This process of shifting bits to the right and resetting the leftmost bit to 1 when a page is referenced gives a history of each page's usage. For example, a page that has not been used for the last eight time intervals would have a reference byte of 00000000, while one that has been referenced once every time interval will have a reference byte of 11111111.

When a page fault occurs, the LRU policy selects the page with the smallest value in its reference byte because that would be the one least recently used. Figure 3.11 shows how the reference bytes for six active pages change during four snapshots of usage. In (a), the six pages have been initialized; this indicates that all of them have been

referenced once. In (b), Pages 1, 3, 5, and 6 have been referenced again (marked with 1), but Pages 2 and 4 have not (now marked with 0 in the leftmost position). In (c), Pages 1, 2, and 4 have been referenced. In (d), Pages 1, 2, 4, and 6 have been referenced. In (e), Pages 1 and 4 have been referenced.

As shown in Figure 3.11, the values stored in the reference bytes are not unique: Page 3 and Page 5 have the same value. In this case, the LRU policy may opt to swap out all of the pages with the smallest value or may select one among them based on other criteria such as FIFO, priority, or whether the contents of the page have been modified.

Two other page removal algorithms, MRU (most recently used) and LFU (least frequently used), are discussed in exercises at the end of this chapter.

Page Number	Time Snapshot 0	Time Snapshot 1	Time Snapshot 2	Time Snapshot 3	Time Snapshot 4
1	10000000	11000000	11100000	11110000	11111000
2	10000000	01000000	10100000	11010000	01101000
3	10000000	11000000	01100000	00110000	00011000
4	10000000	01000000	10100000	11010000	11101000
5	10000000	11000000	01100000	00110000	00011000
6	10000000	11000000	01100000	10110000	01011000

(a) (b) (c) (d) (e)

(figure 3.11)

Notice how the reference bit for each page is updated with every time tick. Arrows (a) through (e) show how the initial bit shifts to the right with every tick of the clock.

The Mechanics of Paging

Before the Memory Manager can determine which pages will be swapped out, it needs specific information about each page in memory—information included in the Page Map Tables.

For example, in Figure 3.5, the Page Map Table for Job 1 includes three bits: the status bit, the referenced bit, and the modified bit (these are the three middle columns: the two empty columns and the Y/N column represents "in memory"). But the representation of the table shown in Figure 3.5 is simplified for illustration purposes. It actually looks something like the one shown in Table 3.3.

Each Page Map Table must track each page's status, modifications, and references. It does so with three bits, each of which can be either 0 or 1.

Page No.	Status Bit	Modified Bit	Referenced Bit	Page Frame No.
0	1	1	1	5
1	1	0	0	9
2	1	0	0	7
3	1	0	1	12

(table 3.3)

Page Map Table for Job 1 shown in Figure 3.5. A 1 = Yes and 0 = No.

As we said before, the status bit indicates whether the page is currently in memory. The referenced bit indicates whether the page has been called (referenced) recently. This bit is important because it is used by the LRU algorithm to determine which pages should be swapped out.

The modified bit indicates whether the contents of the page have been altered; if so, the page must be rewritten to secondary storage when it is swapped out before its page frame is released. (A page frame with contents that have not been modified can be overwritten directly, thereby saving a step.) That is because when a page is swapped into memory it isn't removed from secondary storage. The page is merely copied—the original remains intact in secondary storage. Therefore, if the page isn't altered while it is in main memory (in which case the modified bit remains unchanged at zero), the page needn't be copied back to secondary storage when it is swapped out of memory—the page that is already there is correct. However, if modifications were made to the page, the new version of the page must be written over the older version—and that takes time. Each bit can be either 0 or 1, as shown in Table 3.4.

(table 3.4)

The meaning of these bits used in the Page Map Table.

Status Bit		Modified Bit		Referenced Bit	
Value	Meaning	Value	Meaning	Value	Meaning
0	*not* in memory	0	*not* modified	0	*not* called
1	*resides* in memory	1	*was* modified	1	*was* called

The status bit for all pages in memory is 1. A page must be in memory before it can be swapped out so all of the candidates for swapping have a 1 in this column. The other two bits can be either 0 or 1, making four possible combinations of the referenced and modified bits, as shown in Table 3.5.

(table 3.5)

Four possible combinations of modified and referenced bits and the meaning of each.

	Modified?	Referenced?	What it Means
Case 1	0	0	Not modified AND not referenced
Case 2	0	1	Not modified BUT was referenced
Case 3	1	0	Was modified BUT not referenced [Impossible?]
Case 4	1	1	Was modified AND was referenced

The FIFO algorithm uses only the modified and status bits when swapping pages (because it doesn't matter to FIFO how recently they were referenced), but the LRU looks at all three before deciding which pages to swap out.

Which pages would the LRU policy choose first to swap? Of the four cases described in Table 3.5, it would choose pages in Case 1 as the ideal candidates for removal

because they've been neither modified nor referenced. That means they wouldn't need to be rewritten to secondary storage, and they haven't been referenced recently. So the pages with zeros for these two bits would be the first to be swapped out.

What is the next most likely candidate? The LRU policy would choose Case 3 next because the other two, Case 2 and Case 4, were recently referenced. The bad news is that Case 3 pages have been modified; therefore, the new contents will be written to secondary storage, and that means it will take more time to swap them out. By process of elimination, then we can say that Case 2 pages would be the third choice and Case 4 pages would be those least likely to be removed.

You may have noticed that Case 3 presents an interesting situation: apparently these pages have been modified without being referenced. How is that possible? The key lies in how the referenced bit is manipulated by the operating system. When the pages are brought into memory, each is usually referenced at least once. That means that all of the pages soon have a referenced bit of 1. Of course the LRU algorithm would be defeated if every page indicated that it had been referenced. Therefore, to make sure the referenced bit actually indicates *recently* referenced, the operating system periodically resets it to 0. Then, as the pages are referenced during processing, the bit is changed from 0 to 1 and the LRU policy is able to identify which pages actually are frequently referenced. As you can imagine, there is one brief instant, just after the bits are reset, in which all of the pages (even the active pages) have reference bits of 0 and are vulnerable. But as processing continues, the most-referenced pages soon have their bits reset to 1, so the risk is minimized.

The Working Set

One innovation that improved the performance of demand paging schemes was the concept of the **working set**, which is the collection of pages residing in memory that can be accessed directly without incurring a page fault. Typically, a job's working set changes as the job moves through the system: one working set could be used to initialize the job, another could work through repeated calculations, another might interact with output devices, and a final set could close the job.

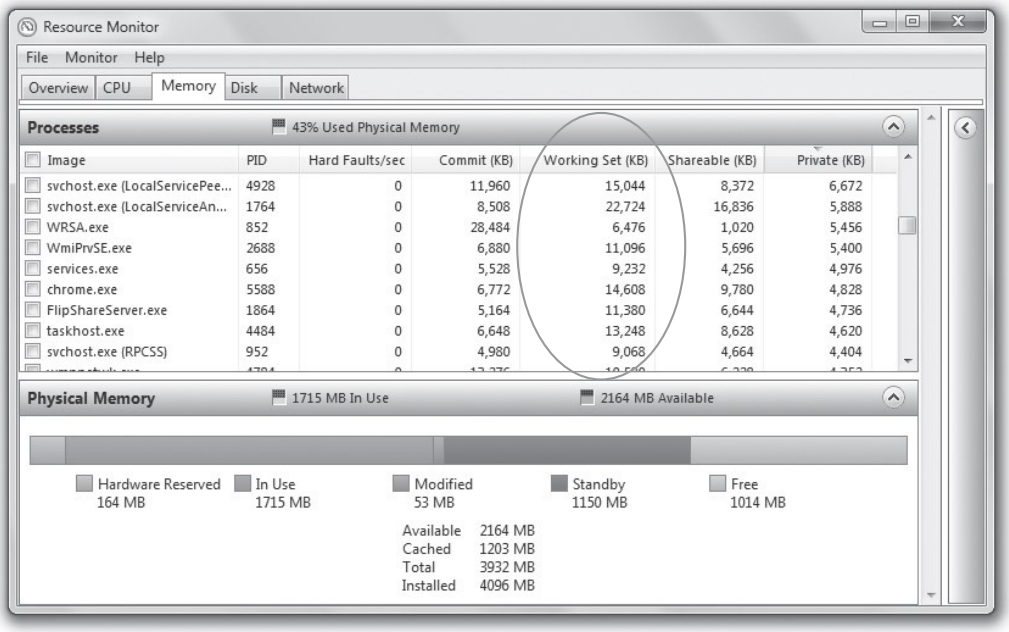
For example, when a user requests execution of a program, the first page is loaded into memory and execution continues as more pages are loaded: those containing variable declarations, others containing instructions, others containing data, and so on. After a while, most programs reach a fairly stable state, and processing continues smoothly with very few additional page faults. At this point, one of the job's working sets is in memory, and the program won't generate many page faults until it gets to another phase requiring a different set of pages to do the work—a different working set. An example of working sets in Windows is shown in Figure 3.12.

On the other hand, it is possible that a poorly structured program could require that every one of its pages be in memory before processing can begin.

(figure 3.12)

Using a Windows operating system, this user searched for and opened the Resource Monitor and clicked on the Memory tab to see the working set for each open application.

Captured from Windows operating system.

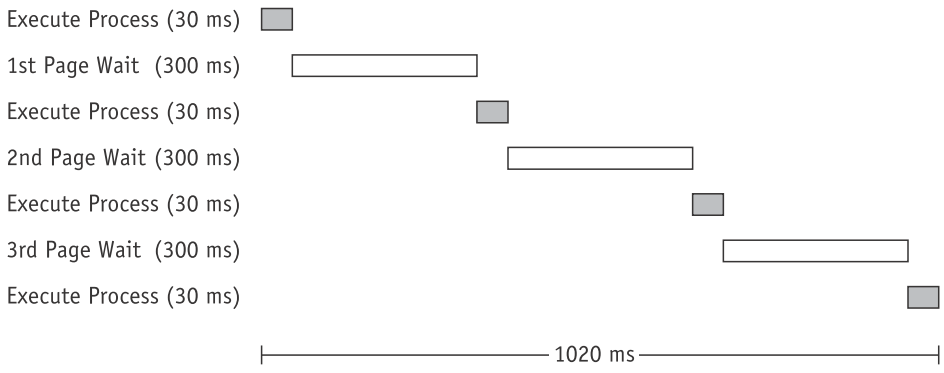


Fortunately, most programs are structured, and this leads to a **locality of reference** during the program's execution, meaning that during any phase of its execution, the program references only a small fraction of its pages. If a job is executing a loop, then the instructions within the loop are referenced extensively while those outside the loop may not be used at all until the loop is completed—that's locality of reference. The same concept applies to sequential instructions, subroutine calls (within the subroutine), stack implementations, access to variables acting as counters or sums, or multidimensional variables such as arrays and tables, where only a few of the pages are needed to handle the references.

It would be convenient if all of the pages in a job's working set were loaded into memory at one time to minimize the number of page faults and to speed up processing, but that is easier said than done. To do so, the system needs definitive answers to some difficult questions: How many pages comprise the working set? What is the maximum number of pages the operating system will allow for a working set?

The second question is particularly important in networked or time-sharing systems, which regularly swap jobs, or pages of jobs, into memory and back to secondary storage to accommodate the needs of many users. The problem is this: every time a job is reloaded back into memory (or has pages swapped), it has to generate several page faults until its working set is back in memory and processing can continue. It is a time-consuming task for the CPU, as shown in Figure 3.13.

One solution adopted by many paging systems is to begin by identifying each job's working set and then loading it into memory in its entirety before allowing execution to begin. In a time-sharing or networked system, this means the operating system must keep track of the size and identity of every working set, making sure that the



(figure 3.13)

Time line showing the amount of time required to process page faults for a single program. The program in this example takes 120 milliseconds (ms) to execute but an additional 900 ms to load the necessary pages into memory. Therefore, job turnaround is 1020 ms.

jobs destined for processing at any one time won't exceed the available memory. Some operating systems use a variable working set size and either increase it when necessary (the job requires more processing) or decrease it when necessary. This may mean that the number of jobs in memory will need to be reduced if, by doing so, the system can ensure the completion of each job and the subsequent release of its memory space.

We have looked at several examples of demand paging memory allocation schemes. Demand paging had two advantages. It was the first scheme in which a job was no

Peter J. Denning (1942–)

Dr. Denning published his groundbreaking paper in 1967, in which he described the working set model for program behavior. He helped establish virtual memory as a permanent part of operating systems. Because of his work, the concepts of working sets, locality of reference, and thrashing became standard in the curriculum for operating systems. In the 1980s, he was one of the four founding principal investigators of the Computer Science Network, a precursor to today's Internet. Denning's honors include fellowships in three professional societies: the Institute of Electrical and Electronics Engineers (IEEE, 1982), American Association for the Advancement of Science (AAAS, 1984), and Association of Computing Machinery (ACM, 1993).

For more information:

http://www.computerhistory.org/events/events.php?spkid=0&ssid=1173214027

Dr. Denning won the Association for Computing Machinery Special Interest Group Operating Systems Hall of Fame Award 2005: "The working set model became the ideal for designing the memory manager parts of operating systems. Virtual storage became a standard part of operating systems."

longer constrained by the size of physical memory (and it introduced the concept of virtual memory, discussed later in this chapter). The second advantage was that it utilized memory more efficiently than the previous schemes because the sections of a job that were used seldom or not at all (such as error routines) weren't loaded into memory unless they were specifically requested. Its disadvantage was the increased overhead caused by the tables and the page interrupts. The next allocation scheme built on the advantages of both paging and dynamic partitions.

Segmented Memory Allocation

The concept of segmentation is based on the common practice by programmers of structuring their programs in modules—logical groupings of code. With **segmented memory allocation**, each job is divided into several **segments** of different sizes, one for each module that contains pieces that perform related functions. A **subroutine** is an example of one such logical group. Segmented memory allocation was designed to reduce page faults that resulted from having a segment's loop split over two or more pages, for example. This is fundamentally different from a paging scheme, which divides the job into several pages all of the same size, each of which often contains pieces from more than one program module.

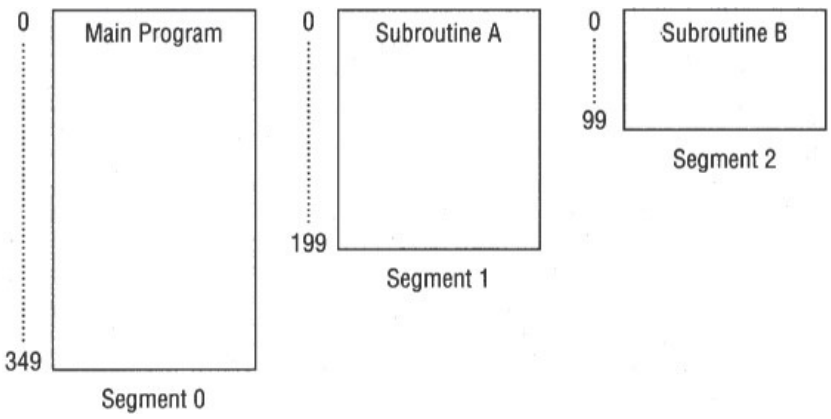
The Segment Map Table functions the same way as a Page Map Table but it manages segments instead of pages.

A second important difference with segmented memory allocation is that main memory is no longer divided into page frames, because the size of each segment is different ranging from quite small to large. Therefore, as with the dynamic partition allocation scheme discussed in Chapter 2, memory is allocated in a dynamic manner.

When a program is compiled or assembled, the segments are set up according to the program's structural modules. Each segment is numbered and a **Segment Map Table** (SMT) is generated for each job; it contains the segment numbers, their lengths, access rights, status, and (when each is loaded into memory) its location in memory. Figures 3.14 and 3.15 show the same job that's composed of a main program and two subroutines (for example, one subroutine calculates the normal pay rate, and a second one calculates

(figure 3.14)

Segmented memory allocation. Job 1 includes a main program and two subroutines. It is a single job that is structurally divided into three segments of different sizes.



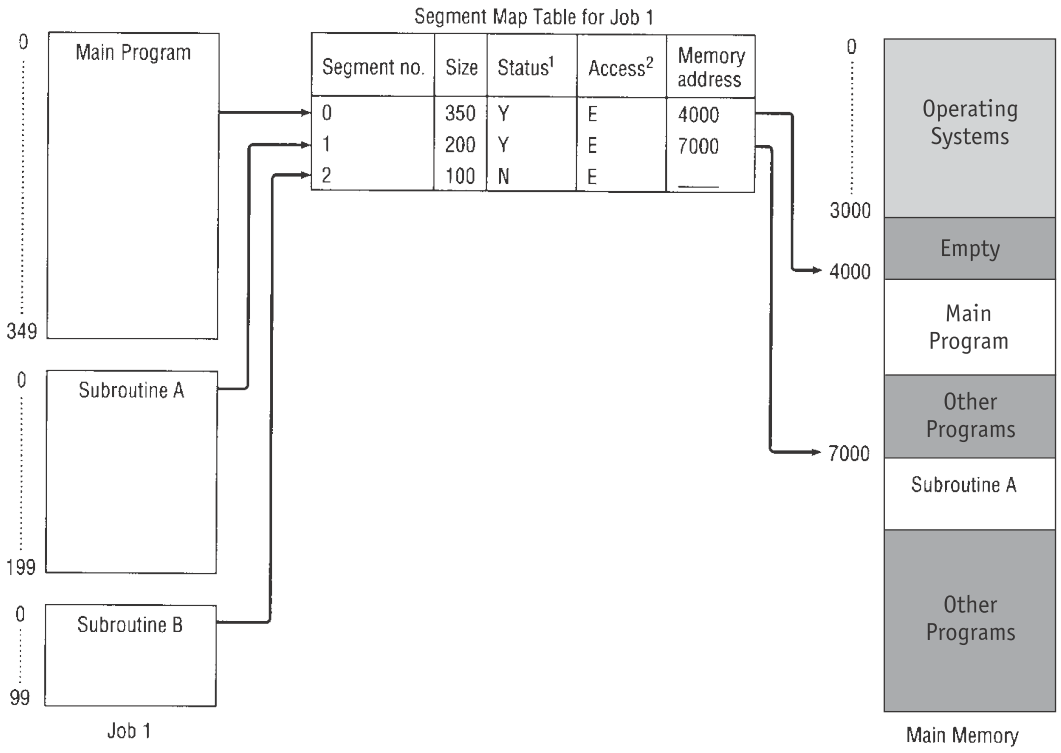
the overtime pay or commissions) with its Segment Map Table and main memory allocation. Notice that the segments need not be located adjacent or even near each other in main memory. (As in demand paging, the referenced and modified bits are used in segmentation and appear in the SMT even though they aren't included in Figure 3.15.)

The Memory Manager needs to keep track of the segments in memory. This is done with three tables, combining aspects of both dynamic partitions and demand paging memory management:

- The Job Table lists every job being processed (one for the whole system).
- The Segment Map Table lists details about each segment (one for each job).
- The Memory Map Table monitors the allocation of main memory (one for the whole system).

Like demand paging, the instructions within each segment are ordered sequentially, but the segments don't need to be stored contiguously in memory. We just need to know where each segment is stored. The contents of the segments themselves are contiguous in this scheme.

To access a specific location within a segment, we can perform an operation similar to the one used for paged memory management. The only difference is that we



(figure 3.15)

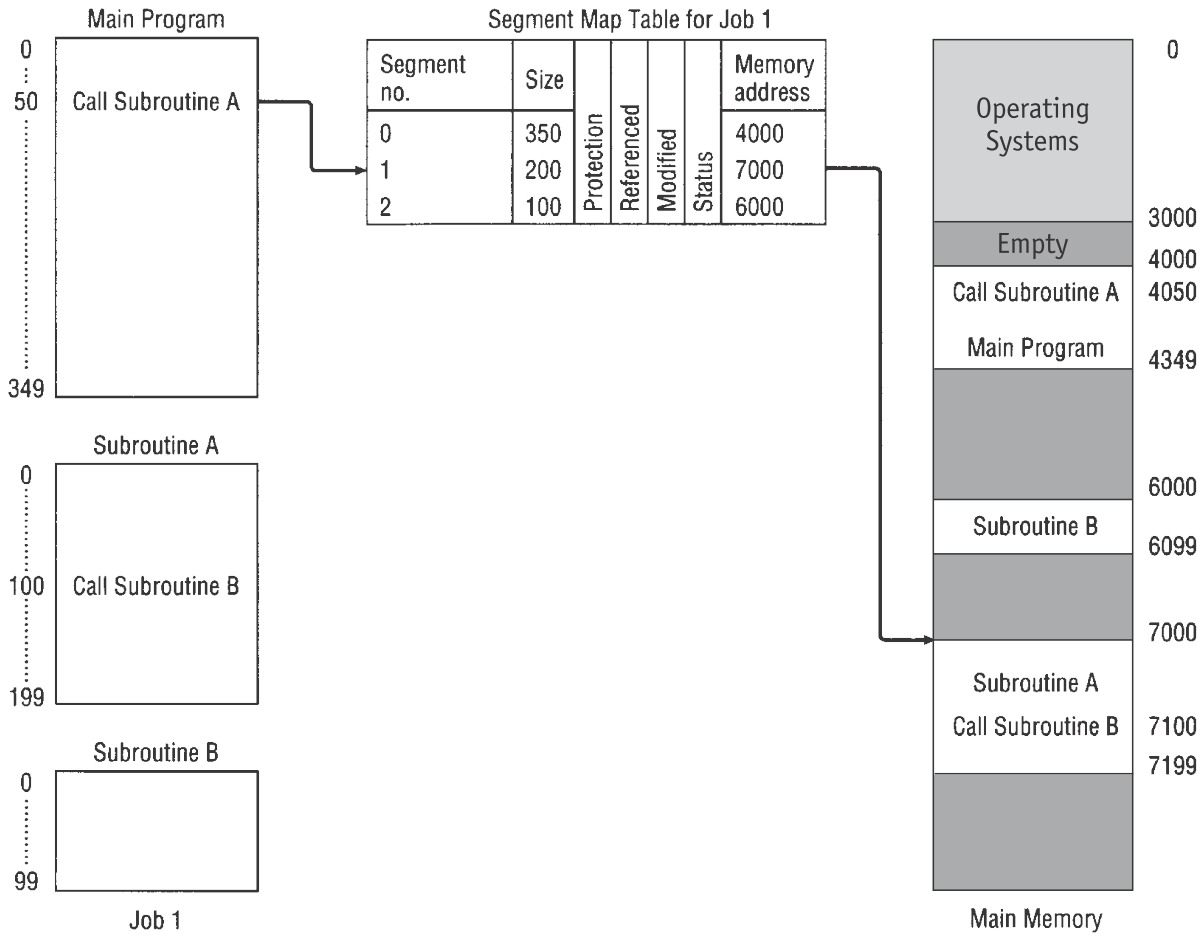
The Segment Map Table tracks each segment for this job. Notice that Subroutine B has not yet been loaded into memory.

work with segments instead of pages. The addressing scheme requires the segment number and the displacement within that segment; and because the segments are of different sizes, the displacement must be verified to make sure it isn't outside of the segment's range.

In Figure 3.16, Segment 1 includes all of Subroutine A, so the system finds the beginning address of Segment 1, 7000, and it begins there.

If the instruction requested that processing begin at byte 100 of Subroutine A (which is possible in languages that support multiple entries into subroutines) then, to locate that item in memory, the Memory Manager would need to add 100 (the displacement) to 7000 (the beginning address of Segment 1).

Can the displacement be larger than the size of the segment? No, not if the program is coded correctly; however, accidents and attempted exploits do happen, and



(figure 3.16)

During execution, the main program calls Subroutine A, which triggers the SMT to look up its location in memory.

the Memory Manager must always guard against this possibility by checking the displacement against the size of the segment, verifying that it is not out of bounds.

To access a location in memory, when using either paged or segmented memory management, the address is composed of two values: the page or segment number and the displacement. Therefore, it is a two-dimensional addressing scheme—that is, it has two elements: SEGMENT_NUMBER and DISPLACEMENT.

The disadvantage of any allocation scheme in which memory is partitioned dynamically is the return of external fragmentation. Therefore, if that schema is used, recompaction of available memory is necessary from time to time.

As you can see, there are many similarities between paging and segmentation, so they are often confused. The major difference is a conceptual one: pages are physical units that are invisible to the user's program and consist of fixed sizes; segments are logical units that are visible to the user's program and consist of variable sizes.

Segmented/Demand Paged Memory Allocation

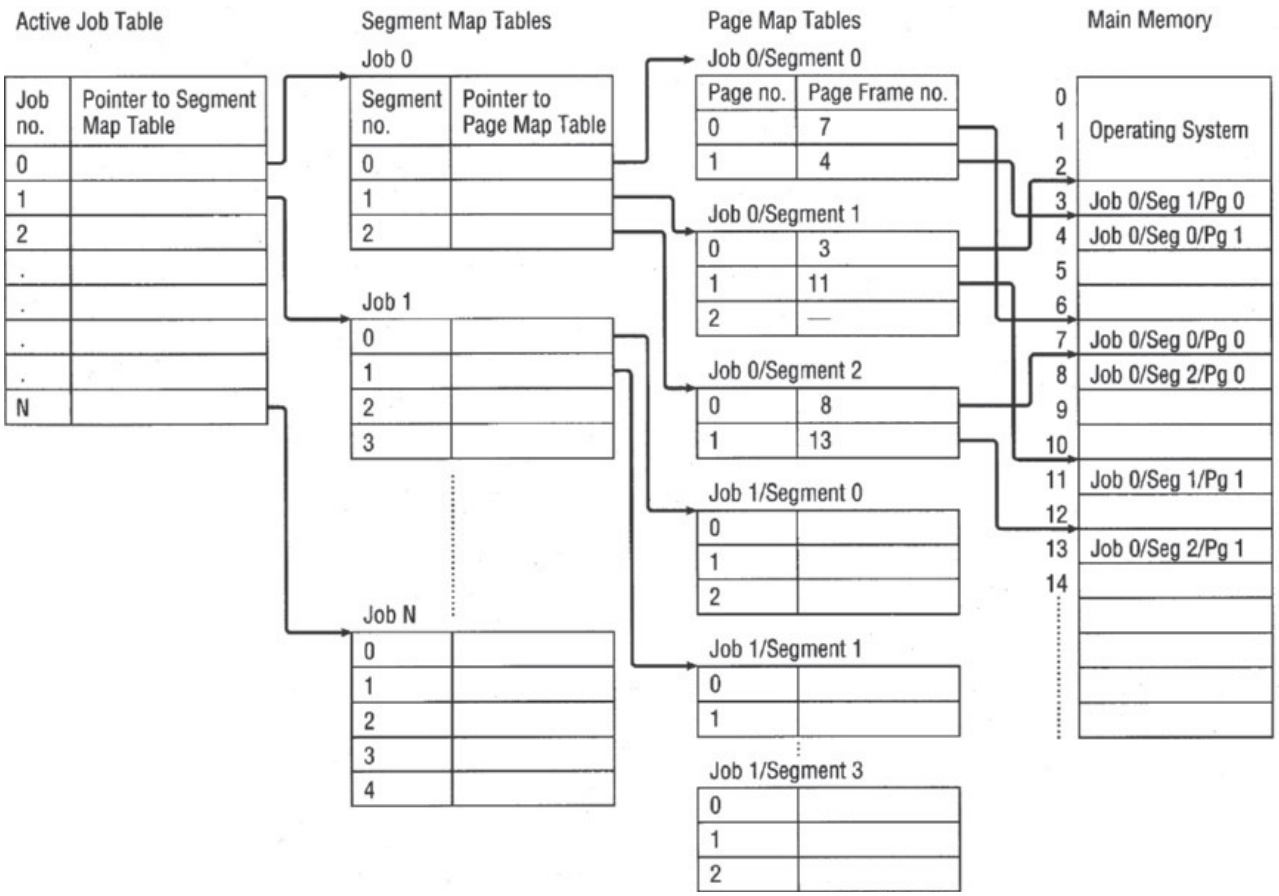
The **segmented/demand paged memory allocation** scheme evolved from the two we have just discussed. It is a combination of segmentation and demand paging, and it offers the logical benefits of segmentation, as well as the physical benefits of paging. The logic isn't new. The algorithms used by the demand paging and segmented memory management schemes are applied here with only minor modifications.

This allocation scheme doesn't keep each segment as a single contiguous unit, but subdivides it into pages of equal size that are smaller than most segments and more easily manipulated than whole segments. Therefore, many of the problems of segmentation (compaction, external fragmentation, and secondary storage handling) are removed because the pages are of fixed length.

This scheme, illustrated in Figure 3.17, requires four types of tables:

- The Job Table lists every job in process (there's one JT for the whole system).
- The Segment Map Table lists details about each segment (one SMT for each job).
- The Page Map Table lists details about every page (one PMT for each segment).
- The Memory Map Table monitors the allocation of the page frames in main memory (there's one for the whole system).

Notice that the tables in Figure 3.17 have been simplified. The SMT actually includes additional information regarding protection (such as the authority to read, write, and execute parts of the file), and it also determines which specific users or processes are allowed to access that segment. In addition, the PMT includes indicators of the page's status, last modification, and last reference.



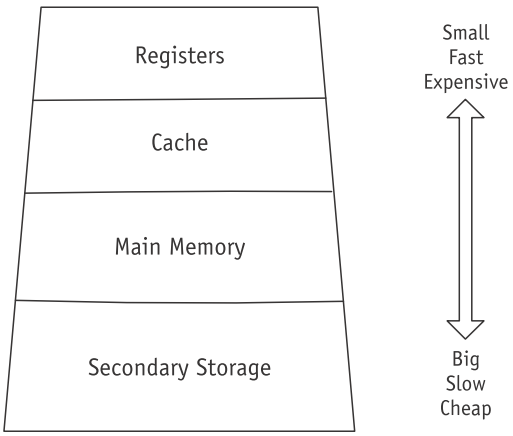
(figure 3.17)

How the Job Table, Segment Map Table, Page Map Table, and main memory interact in a segment/paging scheme.

Here's how it works: To access a certain location in memory, the system locates the address, which is composed of three entries: the segment number, the page number within that segment, and the displacement within that page. Therefore, it is a three-dimensional addressing scheme with the SEGMENT_NUMBER, the PAGE_NUMBER, and the DISPLACEMENT.

The major disadvantages of this memory allocation scheme are twofold: the overhead that is required to manage the tables (Segment Map Tables and the Page Map Tables), and the time required to reference them. To minimize the number of references, many systems take advantage of associative memory to speed up the process.

Associative memory is a name given to several registers that are allocated to each job that is active. See Figure 3.18. Their task is to associate several segment and page numbers belonging to the job being processed with their main memory addresses. These associative registers are hardware, and the exact number of registers varies from system to system.



(figure 3.18)

Hierarchy of data storage options, from most expensive at the top to cheapest at the bottom (adapted from Stuart, 2009).

To appreciate the role of associative memory, it is important to understand how the system works with segments and pages. In general, when a job is allocated to the CPU, its Segment Map Table is loaded into main memory, while its Page Map Tables are loaded only as they are needed. When pages are swapped between main memory and secondary storage, all tables are updated.

Whenever a page request is issued, two searches can take place at the same time: one through associative memory and one through the SMT and its PMTs.

Here is a typical procedure:

1. When a page is first requested, the job's SMT is searched to locate its PMT.
2. The PMT is loaded (if necessary) and searched to determine the page's location in memory.
   a. If the page isn't in memory, then a page interrupt is issued, the page is brought into memory, and the table is updated. (As the example indicates, loading the PMT can cause a page interrupt, or fault, as well.) This process is just as tedious as it sounds, but it gets easier.
   b. Since this segment's PMT (or part of it) now resides in memory, any other requests for pages within this segment can be quickly accommodated because there is no need to bring the PMT into memory. However, accessing these tables (SMT and PMT) is time consuming.

This illustrates the problem addressed by associative memory, which stores the information related to the most-recently-used pages. Then when a page request is issued, two searches begin at once—one through the segment and page map tables and one through the contents of the associative registers.

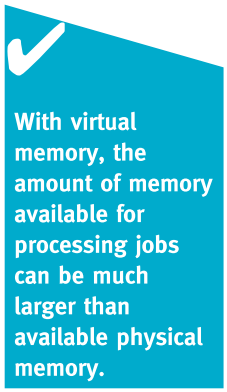
If the search of the associative registers is successful, then the search through the tables is abandoned and the address translation is performed using the information in the associative registers. However, if the search of associative memory fails, no time is lost because the search through the SMT and PMTs had already begun (in this schema). When this search is successful and the main memory address from the PMT has been determined, the address is used to continue execution of the program while the reference is also stored in one of the associative registers. If all of the associative registers

are full, then an LRU (or other) algorithm is used and the least-recently-referenced associative register holds the information on this requested page.

For example, a system with eight associative registers per job will use them to store the SMT entries and PMT entries for the last eight pages referenced by that job. When an address needs to be translated from segment and page numbers to a memory location, the system will look first in the eight associative registers. If a match is found, the memory location is taken from the associative register; if there is no match, then the SMT and PMTs will continue to be searched and the new information will be stored in one of the eight registers as a result. It's worth noting that in some systems the searches do not run in parallel, but the search of the SMT and PMTs is performed after a search of the associative registers fails.

If a job is swapped out to secondary storage during its execution, then all of the information stored in its associative registers is saved, as well as the current PMT and SMT, so the displaced job can be resumed quickly when the CPU is reallocated to it. The primary advantage of a large associative memory is increased speed. The disadvantage is the high cost of the complex hardware required to perform the parallel searches.

Virtual Memory



With virtual memory, the amount of memory available for processing jobs can be much larger than available physical memory.

Virtual memory became possible with the capability of moving pages at will between main memory and secondary storage, and it effectively removed restrictions on maximum program size. With virtual memory, even though only a portion of each program is stored in memory at any given moment, by swapping pages into and out of memory, it gives users the appearance that their programs are completely loaded into main memory during their entire processing time—a feat that would require an incredibly large amount of main memory.

Virtual memory can be implemented with both paging and segmentation, as seen in Table 3.6.

(table 3.6)

Comparison of the advantages and disadvantages of virtual memory with paging and segmentation.

Virtual Memory with Paging	Virtual Memory with Segmentation
Allows internal fragmentation within page frames	Doesn't allow internal fragmentation
Doesn't allow external fragmentation	Allows external fragmentation
Programs are divided into equal-sized pages	Programs are divided into unequal-sized segments that contain logical groupings of code
The absolute address is calculated using page number and displacement	The absolute address is calculated using segment number and displacement
Requires Page Map Table (PMT)	Requires Segment Map Table (SMT)

Segmentation allows users to share program code. The shared segment contains: (1) an area where unchangeable code (called **reentrant code**) is stored, and (2) several data areas, one for each user. In this case, users share the code, which cannot be modified, but they can modify the information stored in their own data areas as needed without affecting the data stored in other users' data areas.

Before virtual memory, sharing meant that copies of files were stored in each user's account. This allowed them to load their own copy and work on it at any time. This kind of sharing created a great deal of unnecessary system cost—the I/O overhead in loading the copies and the extra secondary storage needed. With virtual memory, those costs are substantially reduced because shared programs and subroutines are loaded on demand, satisfactorily reducing the storage requirements of main memory (although this is accomplished at the expense of the Memory Map Table).

The use of virtual memory requires cooperation between the Memory Manager (which tracks each page or segment) and the processor hardware (which issues the interrupt and resolves the virtual address). For example, when a page that is not already in memory is needed, a page fault is issued and the Memory Manager chooses a page frame, loads the page, and updates entries in the Memory Map Table and the Page Map Tables.

Virtual memory works well in a multiprogramming environment because most programs spend a lot of time waiting—they wait for I/O to be performed; they wait for pages to be swapped in or out; and they wait when their turn to use the processor is expired. In a multiprogramming environment, the waiting time isn't wasted because the CPU simply moves to another job.

Virtual memory has increased the use of several programming techniques. For instance, it aids the development of large software systems, because individual pieces can be developed independently and linked later on.

Virtual memory management has several advantages:

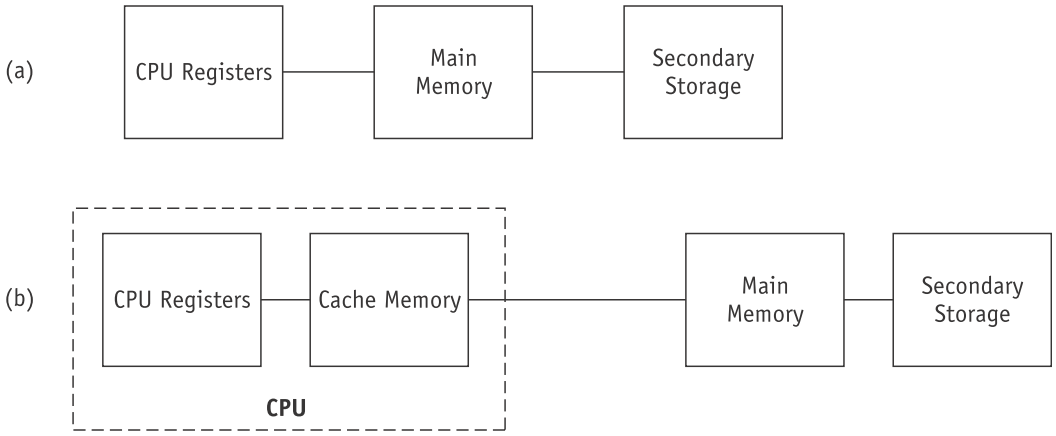
- A job's size is no longer restricted to the size of main memory (or worse, to the free space available within main memory).
- Memory is used more efficiently because the only sections of a job stored in memory are those needed immediately, while those not needed remain in secondary storage.
- It allows an unlimited amount of multiprogramming, which can apply to many jobs, as in dynamic and static partitioning, or to many users.
- It allows the sharing of code and data.
- It facilitates dynamic linking of program segments.

The advantages far outweigh these disadvantages:

- Increased processor hardware costs.
- Increased overhead for handling paging interrupts.
- Increased software complexity to prevent thrashing.

Cache Memory

Cache memory is based on the concept of using a small, fast, and expensive memory to supplement the workings of main memory. Because the cache is usually small in capacity (compared to main memory), it can use more expensive memory chips. These are five to ten times faster than main memory and match the speed of the CPU. Therefore, if data or instructions that are frequently used are stored in cache memory, memory access time can be cut down significantly and the CPU can execute those instructions faster, thus raising the overall performance of the computer system. (It's similar to the role of a frequently called list of telephone numbers in a telephone. By keeping those numbers in an easy-to-reach place, they can be called much faster than those in a long contact list.)



(figure 3.19) *Comparison of (a) the traditional path used by early computers between main memory and the CPU and (b) the path used by modern computers to connect the main memory and the CPU via cache memory.*

Cache size is a significant contributor to overall response and is an important element in system design.

As shown in Figure 3.19(a), early computers were designed to have data and instructions transferred from secondary storage to main memory and then to special-purpose registers for processing—this path necessarily increased the amount of time needed to process those instructions. However, because the same instructions are used repeatedly in most programs, computer system designers thought it would be more efficient if the system would not use a complete memory cycle every time an instruction or data value is required. Designers found that this could be done if they placed repeatedly used data in general-purpose registers instead of in main memory.

To solve this problem, computer systems automatically store frequently used data in an intermediate memory unit called **cache memory**. This adds a middle layer to the original hierarchy. Cache memory can be thought of as an intermediary between main memory and the special-purpose registers, which are the domain of the CPU, as shown in Figure 3.17(b).

A typical microprocessor has two or more levels of caches, such as Level 1 (L1), Level 2 (L2), and Level 3 (L3), as well as specialized caches.

In a simple configuration with only two cache levels, information enters the processor through the bus interface unit, which immediately sends one copy to the Level 2 cache, which is an integral part of the microprocessor and is directly connected to the CPU. A second copy is sent to one of two Level 1 caches, which are built directly into the chip. One of these Level 1 caches stores instructions, while the other stores data to be used by the instructions. If an instruction needs additional data, the instruction is put on hold while the processor looks for the missing data—first in the data Level 1 cache, and then in the larger Level 2 cache before searching main memory. Because the Level 2 cache is an integral part of the microprocessor, data moves two to four times faster between the CPU and the cache than between the CPU and main memory.

To understand the relationship between main memory and cache memory, consider the relationship between the size of the Web and the size of your private browser bookmark file. Your bookmark file is small and contains only a tiny fraction of all the available addresses on the Web; but the chance that you will soon visit a Web site that is in your bookmark file is high. Therefore, the purpose of your bookmark file is to keep your most frequently accessed addresses easy to reach so you can access them quickly. Similarly, the purpose of cache memory is to keep handy the most recently accessed data and instructions so that the CPU can access them repeatedly without wasting time.

The movement of data or instructions from main memory to cache memory uses a method similar to that used in paging algorithms. First, cache memory is divided into blocks of equal size called slots. Then, when the CPU first requests an instruction or data from a location in main memory, the requested instruction and several others around it are transferred from main memory to cache memory, where they are stored in one of the free slots. Moving a block at a time is based on the principle of locality of reference, which states that it is very likely that the next CPU request will be physically close to the one just requested. In addition to the block of data transferred, the slot also contains a label that indicates the main memory address from which the block was copied. When the CPU requests additional information from that location in main memory, cache memory is accessed first; and if the contents of one of the labels in a slot matches the address requested, then access to main memory is not required.

The algorithm to execute one of these "transfers from main memory" is simple to implement (a pseudocode algorithm can be found in Appendix A).

These steps become more complicated when there are no free cache slots, which can occur because the size of cache memory is smaller than that of main memory—in this case individual slots cannot be permanently allocated to blocks. To address this contingency, the system needs a policy for block replacement, which could be similar to those used in page replacement.

When designing cache memory, one must take into consideration the following four factors:

- *Cache size*. Studies have shown that having any cache, even a small one, can substantially improve the performance of the computer system.
- *Block size*. Because of the principle of locality of reference, as block size increases, the ratio of number of references found in the cache to the total number of references will tend to increase.
- *Block replacement algorithm*. When all the slots are busy and a new block has to be brought into the cache, a block that is least likely to be used in the near future should be selected for replacement. However, as we saw in paging, this is nearly impossible to predict. A reasonable course of action is to select a block that has not been used for a long time. Therefore, LRU is the algorithm that is often chosen for block replacement, which requires a hardware mechanism to determine the least recently used slot.
- *Rewrite policy*. When the contents of a block residing in cache are changed, it must be written back to main memory before it is replaced by another block. A rewrite policy must be in place to determine when this writing will take place. One alternative is to do this every time that a change occurs, which would increase the number of memory writes, possibly increasing overhead. A second alternative is to do this only when the block is replaced or the process is finished, which would minimize overhead but would leave the block in main memory in an inconsistent state. This would create problems in multiprocessor environments and in cases where I/O modules can access main memory directly.

The optimal selection of cache size and replacement algorithm can result in 80 to 90 percent of all requests being in the cache, making for a very efficient memory system. This measure of efficiency, called the cache hit ratio, is used to determine the performance of cache memory and, when shown as a percentage, represents the percentage of total memory requests that are found in the cache. One formula is this:

$$\text{HitRatio} = \frac{\text{number of requests found in the cache}}{\text{total number of requests}} * 100$$

For example, if the total number of requests is 10, and 6 of those are found in cache memory, then the hit ratio is 60 percent, which is reasonably good and suggests improved system performance.

```
HitRatio = (6/10) * 100 = 60%
```

Likewise, if the total number of requests is 100, and 9 of those are found in cache memory, then the hit ratio is only 9 percent.

```
HitRatio = (9/100) * 100 = 9%
```

Another way to measure the efficiency of a system with cache memory, assuming that the system always checks the cache first, is to compute the average memory access time (Avg_Mem_AccTime) using the following formula:

```
Avg_Mem_AccTime=Avg_Cache_AccessTime+(1-HitRatio)*Avg_MainMem_AccTime
```

For example, if we know that the average cache access time is 200 nanoseconds (nsec) and the average main memory access time is 1000 nsec, then a system with a hit ratio of 60 percent will have an average memory access time of 600 nsec:

```
AvgMemAccessTime = 200 + (1 - 0.60) * 1000 = 600 nsec
```

A system with a hit ratio of 9 percent will show an average memory access time of 1110 nsec:

```
AvgMemAccessTime = 200 + (1 - 0.09) * 1000 = 1110 nsec
```

Because of its role in improving system performance, cache is routinely added to a wide variety of main memory configurations as well as devices.

Conclusion

The memory management portion of the operating system assumes responsibility for allocating memory storage (in main memory, cache memory, and registers) and, equally important, for deallocating it when execution is completed.

The design of each scheme that we discuss in Chapters 2 and 3 addressed a different set of pressing problems. As we have seen, when some problems are solved, others were often created. Table 3.7 shows how these memory allocation schemes compare.

The Memory Manager is only one of several managers that make up the operating system. After the jobs are loaded into memory using a memory allocation scheme, the Processor Manager takes responsibility to allocate processor time to each job, and every process and thread in that job, in the most efficient manner possible. We will see how that is done in the next chapter.

(table 3.7)

The big picture. Comparison of the memory allocation schemes discussed in Chapters 2 and 3.

Scheme	Problem Solved	Problem Created	Key Software Changes
Single-user contiguous	Not applicable	Job size limited to physical memory size; CPU often idle	Not applicable
Fixed partitions	Idle CPU time	Internal fragmentation; job size limited to partition size	Add Processor Scheduler; add protection handler
Dynamic partitions	Internal fragmentation	External fragmentation	Algorithms to manage partitions
Relocatable dynamic partitions	External fragmentation	Compaction overhead; job size limited to physical memory size	Algorithms for compaction
Paged	Need for compaction	Memory needed for tables; Job size limited to physical memory size; internal fragmentation returns	Algorithms to manage tables
Demand paged	Job size limited to memory size; inefficient memory use	Large number of tables; possibility of thrashing; overhead required by page interrupts; paging hardware added	Algorithm to replace pages; algorithm to search for pages in secondary storage
Segmented	Internal fragmentation	Difficulty managing variable-length segments in secondary storage; external fragmentation	Dynamic linking package; two-dimensional addressing scheme
Segmented/ demand paged	Segments not loaded on demand	Table handling overhead; memory needed for page and segment tables	Three-dimensional addressing scheme

Key Terms

address resolution: the process of changing the address of an instruction or data item to the address in main memory at which it is to be loaded or relocated.

associative memory: the name given to several registers, allocated to each active process, whose contents associate several of the process segments and page numbers with their main memory addresses.

cache memory: a small, fast memory used to hold selected data and to provide faster access than would otherwise be possible.

clock cycle: the elapsed time between two ticks of the computer's system clock.

clock page replacement policy: a variation of the LRU policy that removes from main memory the pages that show the least amount of activity during recent clock cycles.

demand paging: a memory allocation scheme that loads a program's page into memory at the time it is needed for processing.

displacement: in a paged or segmented memory allocation environment, the difference between a page's relative address and the actual machine language address. Also called offset.

FIFO anomaly: an unusual circumstance through which adding more page frames causes an increase in page interrupts when using a FIFO page replacement policy.

first-in first-out (FIFO) policy: a page replacement policy that removes from main memory the pages that were brought in first.

Job Table (JT): a table in main memory that contains two values for each active job—the size of the job and the memory location where its page map table is stored.

least recently used (LRU) policy: a page-replacement policy that removes from main memory the pages that show the least amount of recent activity.

locality of reference: behavior observed in many executing programs in which memory locations recently referenced, and those near them, are likely to be referenced in the near future.

Memory Map Table (MMT): a table in main memory that contains an entry for each page frame that contains the location and free/busy status for each one.

page: a fixed-size section of a user's job that corresponds in size to page frames in main memory.

page fault: a type of hardware interrupt caused by a reference to a page not residing in memory. The effect is to move a page out of main memory and into secondary storage so another page can be moved into memory.

page fault handler: the part of the Memory Manager that determines if there are empty page frames in memory so that the requested page can be immediately copied from secondary storage, or determines which page must be swapped out if all page frames are busy. Also known as a *page interrupt handler*.

page frame: an individual section of main memory of uniform size into which a single page may be loaded without causing external fragmentation.

Page Map Table (PMT): a table in main memory with the vital information for each page including the page number and its corresponding page frame memory address.

page replacement policy: an algorithm used by virtual memory systems to decide which page or segment to remove from main memory when a page frame is needed and memory is full.

page swapping: the process of moving a page out of main memory and into secondary storage so another page can be moved into memory in its place.

paged memory allocation: a memory allocation scheme based on the concept of dividing a user's job into sections of equal size to allow for noncontiguous program storage during execution.

reentrant code: code that can be used by two or more processes at the same time; each shares the same copy of the executable code but has separate data areas.

sector: a division in a magnetic disk's track, sometimes called a "block." The tracks are divided into sectors during the formatting process.

segment: a variable-size section of a user's job that contains a logical grouping of code.

Segment Map Table (SMT): a table in main memory with the vital information for each segment including the segment number and its corresponding memory address.

segmented/demand paged memory allocation: a memory allocation scheme based on the concept of dividing a user's job into logical groupings of code and loading them into memory as needed to minimize fragmentation.

segmented memory allocation: a memory allocation scheme based on the concept of dividing a user's job into logical groupings of code to allow for noncontiguous program storage during execution.

subroutine: also called a "subprogram," a segment of a program that can perform a specific function. Subroutines can reduce programming time when a specific function is required at more than one point in a program.

thrashing: a phenomenon in a virtual memory system where an excessive amount of page swapping back and forth between main memory and secondary storage results in higher overhead and little useful work.

virtual memory: a technique that allows programs to be executed even though they are not stored entirely in memory.

working set: a collection of pages to be kept in main memory for each active process in a virtual memory environment.

Interesting Searches

For more background on a few of the topics discussed in this chapter, begin a search with these terms.

- Virtual Memory
- Working Set
- Cache Memory
- Belady's FIFO anomaly
- Thrashing

Exercises

Research Topics

A. The size of virtual memory can sometimes be adjusted by a computer user to improve system performance. Using an operating system of your choice, discover if you can change the size of virtual memory, how to do so, and the minimum and maximum recommended sizes of virtual memory. Cite your operating system and the steps required to retrieve the desired information. (*Important*! *Do not change* the size of your virtual memory settings. This exercise is for research purposes only.)

B. On the Internet or using academic sources, research the design of multicore memory and identify the roles played by cache memory. Does the implementation of cache memory on multicore chips vary from one manufacturer to another? Explain your research process and cite your sources.

Exercises

1. Compare and contrast internal fragmentation and external fragmentation. Explain the circumstances where one might be preferred over the other.
2. Explain the function of the Page Map Table in the memory allocation schemes described in this chapter. Explain your answer with examples from the schemes that use a PMT.
3. If a program has 471 bytes and will be loaded into page frames of 100 bytes each, and the instruction to be used is at byte 132, answer the following questions:
   a. How many pages are needed to store the entire job?
   b. Compute the page number and the exact displacement for each of the byte addresses where the data is stored. (Remember that page numbering starts at zero).
4. Assume a program has 510 bytes and will be loaded into page frames of 256 bytes each, and the instruction to be used is at byte 377. Answer the following questions:
   a. How many pages are needed to store the entire job?
   b. Compute the page number and exact displacement for each of the byte addresses where the data is stored.
5. Given that main memory is composed of only three page frames for public use and that a seven-page program (with Pages a, b, c, d, e, f, g) that requests pages in the following order:

   a, c, a, b, a, d, a, c, b, d, e, f

   a. Using the FIFO page removal algorithm, indicate the movement of the pages into and out of the available page frames (called a page trace

analysis). Indicate each page fault with an asterisk (*). Then compute the failure and success ratios.

b. Increase the size of memory so it contains four page frames for public use. Using the same page requests as above and FIFO, do another page trace analysis and compute the failure and success ratios.

c. What general statement can you make from this example? Explain your answer.

6. Given that main memory is composed of only three page frames for public use and that a program requests pages in the following order:

a, c, b, d, a, c, e, a, c, b, d, e

a. Using the FIFO page removal algorithm, indicate the movement of the pages into and out of the available page frames (called a page trace analysis) indicating each page fault with an asterisk (*). Then compute the failure and success ratios.

b. Increase the size of memory so it contains four page frames for public use. Using the same page requests as above and FIFO, do another page trace analysis and compute the failure and success ratios.

c. What general statement can you make from this example? Explain your answer.

7. Given that main memory is composed of three page frames for public use and that a program requests pages in the following order:

a, b, a, b, f, d, f, c, g, f, g, b, d, e

a. Using the FIFO page removal algorithm, perform a page trace analysis and indicate page faults with asterisks (*). Then compute the failure and success ratios.

b. Using the LRU page removal algorithm, perform a page trace analysis and compute the failure and success ratios.

c. What conclusions do you draw from this comparison of FIFO and LRU performance? Could you make general statements from this example? Explain why or why not.

8. Let us define "most-recently-used" (MRU) as a page removal algorithm that removes from memory the most recently used page. Perform a page trace analysis using page requests from the previous exercise for a system with three available page frames, and again with six available page frames. Compute the failure and success ratios of each configuration, and explain why or why not MRU is a viable memory allocation system.

9. By examining the reference bits for the six pages shown in the table below, identify which of the eight pages was referenced most often as of the last time snapshot (Time 6). Which page was referenced least often? Explain your reasoning.

Page Number	Time 0	Time 1	Time 2	Time 3	Time 4	Time 5	Time 6
0	10000000	11000000	01100000	10110000	01011000	10101100	01010110
1	10000000	01000000	00100000	10010000	01001000	00100100	00010010
2	10000000	01000000	00100000	10010000	11001000	11100100	01110010
3	10000000	11000000	11100000	01110000	10111000	11011100	01101110
4	10000000	11000000	01100000	10110000	01011000	00101100	10010110
5	10000000	01000000	11100000	01110000	10111000	11011100	11101110
6	10000000	01000000	10100000	01010000	00101000	00010100	10001010
7	10000000	01000000	00100000	00010000	10001000	11000100	01100010

10. To implement LRU, each page uses a referenced bit. If we wanted to implement a least frequently used (LFU) page removal algorithm, in which the page that was used the least would be removed from memory, what would we need to add to the tables? What software modifications would have to be made to support this new algorithm?
11. Calculate the cache Hit Ratio using the formula presented in this chapter assuming that the total number of requests is 2056 and that 647 of those requests are found in the cache.
12. Calculate the cache Hit Ratio using the formula presented in this chapter assuming that the total number of requests is 78985 and that 4029 of those requests are found in the cache. Is this Hit Ration better or worse than the result of the previous exercise?
13. Given three subroutines of 550, 290, and 600 words each, if segmentation is used then the total memory needed is the sum of the three sizes (if all three routines are loaded). However, if paging is used, then some storage space is lost because subroutines rarely fill the last page completely, and that results in internal fragmentation. Determine the total amount of wasted memory due to internal fragmentation when the three subroutines are loaded into memory using each of the following page sizes:
    a. 100 words
    b. 600 words
    c. 700 words
    d. 900 words
14. Using a paged memory allocation system with a page size of 2,048 bytes and an identical page frame size, and assuming the incoming data file is 20,992, calculate how many pages will be created by the file. Calculate the size of any resulting fragmentation. Explain whether this situation will result in internal fragmentation, external fragmentation, or both.

Advanced Exercises

15. Describe the logic you would use to detect thrashing. Once thrashing was detected, explain the steps you would use so the operating system could stop it.

16 Given that main memory is composed of four page frames for public use, use the following table to answer all parts of this problem:

Page Frame	Time When Loaded	Time When Last Referenced	Referenced Bit	Modified Bit
0	9	307	0	1
1	17	362	1	0
2	10	294	0	0
3	160	369	1	1

a. The contents of which page frame would be swapped out by FIFO?

b. The contents of which page frame would be swapped out by LRU?

c. The contents of which page frame would be swapped out by MRU?

d. The contents of which page frame would be swapped out by LFU?

17. Explain how Page Frame 0 in the previous exercise can have a modified bit of 1 and a referenced bit of 0.

18. Given the following Segment Map Tables for two jobs:

SMT for Job 1

Segment Number	Memory Location
0	4096
1	6144
2	9216
3	2048
4	7168

SMT for Job 2

Segment number	Memory location
0	2048
1	6144
2	9216

a. Which segments, if any, are shared between the two jobs?

b. If the segment now located at 7168 is swapped out and later reloaded at 8192, and the segment now at 2048 is swapped out and reloaded at 1024, what would the new segment tables look like?

Programming Exercises

19. This problem studies the effect of changing page sizes in a demand paging system.

    The following sequence of requests for program words is taken from a 460-word program: 10, 11, 104, 170, 73, 309, 185, 245, 246, 434, 458, 364. Main memory can hold a total of 200 words for this program, and the page frame size will match the size of the pages into which the program has been divided.

    Calculate the page numbers according to the page size and divide by the page size to get the page number. The number of page frames in memory is the total number, 200, divided by the page size. For example, in problem (a) the page size is 100, which means that requests 10 and 11 are on Page 0, and requests 104 and 170 are on Page 1. The number of page frames is two.

    a. Find the success frequency for the request list using a FIFO replacement algorithm and a page size of 100 words (there are two page frames).
    b. Find the success frequency for the request list using a FIFO replacement algorithm and a page size of 20 words (10 pages, 0 through 9).
    c. Find the success frequency for the request list using a FIFO replacement algorithm and a page size of 200 words.
    d. What do your results indicate? Can you make any general statements about what happens when page sizes are halved or doubled?
    e. Are there any overriding advantages in using smaller pages? What are the offsetting factors? Remember that transferring 200 words of information takes less than twice as long as transferring 100 words because of the way secondary storage devices operate (the transfer rate is higher than the access [search/find] rate).
    f. Repeat (a) through (c) above, using a main memory of 400 words. The size of each page frame will again correspond to the size of the page.
    g. What happened when more memory was given to the program? Can you make some general statements about this occurrence? What changes might you expect to see if the request list was much longer, as it would be in real life?
    h. Could this request list happen during the execution of a real program? Explain.
    i. Would you expect the success rate of an actual program under similar conditions to be higher or lower than the one in this problem?

20. Given the following information for an assembly language program:

    Job size = 3126 bytes

    Page size = 1024 bytes

Instruction at memory location 532:	`Load 1, 2098`
Instruction at memory location 1156:	`Add 1, 2087`
Instruction at memory location 2086:	`Sub 1, 1052`
Data at memory location 1052:	`015672`
Data at memory location 2098:	`114321`
Data at memory location 2087:	`077435`

    a. How many pages are needed to store the entire job?
    b. Compute the page number and the displacement for each of the byte addresses where the data is stored.
    c. Determine whether the page number and displacements are legal for this job.
    d. Explain why the page number and/or displacements may not be legal for this job.
    e. Indicate what action the operating system might take when a page number or displacement is not legal.