Bharathidasan University

Centre for Distance Education

Re-Accredited with A+ Grade by NAAC

(60th Rank among the Indian Universities in NIRF-2019)

B.Sc Computer Science

Core course VIII

OPERATING SYSTEMS

Dr.T.N.Ravi

Assistant professor & Research Coordinator

PG & Research Department of Computer Science

Periyar E.V.R College(Autonomous)

Trichy-620023

CORE COURSE VIII

OPERATING SYSTEMS

Objective:

To provide the Fundamental Concepts in an Operating System.

Unit I :Introducing Operating Systems

Introduction - What Is an Operating System-Operating System Software -A Brief History of Machine Hardware -Types of Operating Systems -Brief History of Operating System Development-Object-Oriented Design

Unit II Memory Management

Early Systems: Single-User Contiguous Scheme -Fixed Partitions-Dynamic Partitions- Best-Fit versus First-Fit Allocation -Deallocation - Relocatable Dynamic Partitions. Virtual Memory: Paged Memory Allocation-Demand Paging-Page Replacement Policies and Concepts - Segmented Memory Allocation-Segmented/Demand Paged Memory Allocation - Virtual Memory-Cache Memory

Unit III Processor Management

Overview-About Multi-Core Technologies-Job Scheduling Versus Process Scheduling- Process Scheduler-Process Scheduling Policies-Process Scheduling Algorithms -A Word About Interrupts-Deadlock-Seven Cases of Deadlock -Conditions for Deadlock- Modeling Deadlock-Strategies for Handling Deadlocks –Starvation-
Concurrent Processes: What Is Parallel Processing-Evolution of Multiprocessors- Introduction to Multi-Core Processors-Typical Multiprocessing Configurations--Process Synchronization Software

Unit IV Device Management

Types of Devices-Sequential Access Storage Media-Direct Access Storage Devices- Magnetic Disk Drive Access Times- Components of the I/O Subsystem- Communication among Devices-Management of I/O Requests

Unit: V File Management

The File Manager -Interacting with the File Manager -File Organization - Physical Storage Allocation -Access Methods-Levels in a File Management System - Access Control Verification Module

Text Book:

1. Understanding Operating Systems, Ann McIver McHoes and Ida M. Flynn, Course Technology, Cengage Learning, 2011.

Reference Book:

1. OperatingSystems,AchyutGodbole and AtulKahate , McGraw Hill Publishing, 2010

CHAPTER – I

Operating System – An Introduction

Objective:

In this chapter the reader the following aspects are dealt:

- The basic role of Operating system
- The operating system software subsystem and their functions
- The different types of operating system

What is an Operating System?

An operating System is an interface between computer user and computer hardware. An operating system is a system software which performs all the basic tasks like file management, memory management handling input, output and controlling peripheral devices such as disk drives and printers. It controls who can use the system and how. In Short, it is the boss.

Operating System Software:

The fig1.1 shows in an abstract representation of an operating system and demonstrates how its main components work together. There are four essential manager of every operating system.

1. Memory Manger
2. Processor Manger
3. Device Manager
4. File Manager

A network was not always an integral part of operating systems. Now most operating systems routinely incorporate a **Network Manager.**

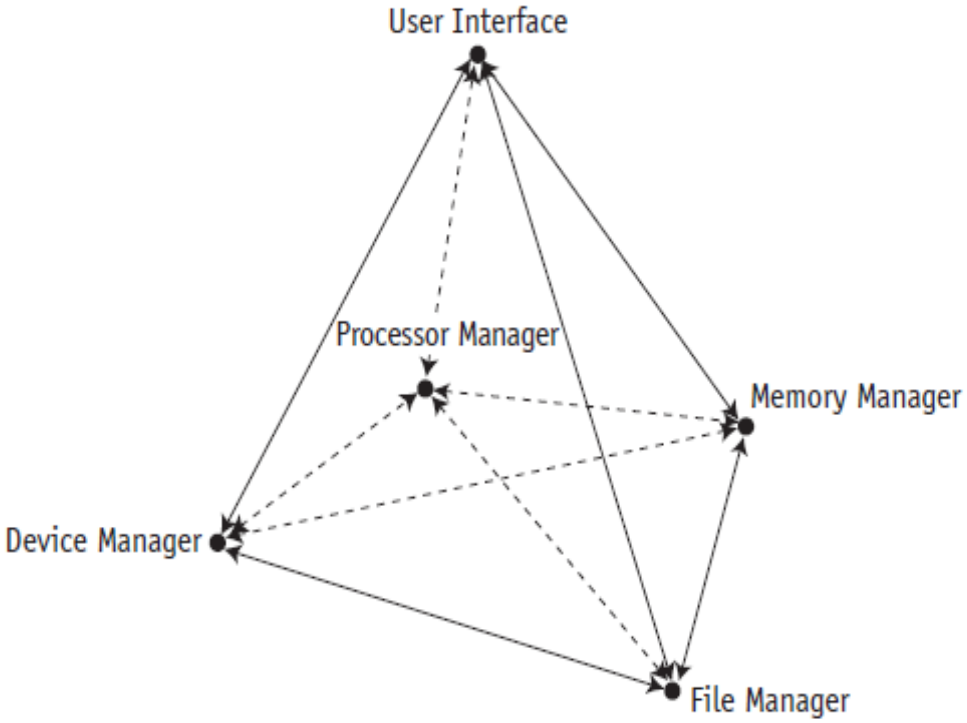


Figure 1.1 Operating System Software

The tasks performed by the subsystem manager are

1. Monitor its resources continuously
2. Enforce the policies that determine who gets what, when and how much
3. Allocate the resources when appropriate
4. Deallocate the resources when appropriate

Memory Management

Responsibility of memory manager is to preserve space in main memory occupied by the operating system itself. It can't allow any part of it be accidentally or intentionally altered.

Processor Management

A process is defined as an instance of execution of a program. The processor manager decides how to allocate the central processing unit (CPU) for a process. The process manager monitors the CPU whether it is executing a process or waiting for a READ or WRITES command to finish execution. The processor manger has two level of responsibilities.

i) To Handle the job

ii) To Handle the process with in the job

First part handled by the Job Scheduler. This is the high-level portion of the processor manger. This accepts or rejects the incoming jobs. The Second part handled by the Process Scheduler which is the low level portion of the process manger. It's responsibility is deciding which process gets the CPU and for how long.

Device management

The Device manager monitors every device, channel and control unit. Its job is to choose the most effective way to allocate all of the devices, printers, ports, disk drives based on a scheduling policy. The device manager allocates each resource when starting its operation after completion of a process deallocate the device and makes it available to the next process or job.

File Management

File manger keeps track of every file in the system. By using access policies it enforces restrictions on who has access to which files. Managing access control

is a key part of file management. Finally, the file manager allocates the resources and deallocates later.

Brief History of Machine Hardware:

The essential components of computer hardware are

- Memory
- I/O Devices
- Central Processing Unit (CPU)

Memory : (Random Access Memory, RAM) is where the data and instruction must reside to be processed.

I/O Devices : Includes all the peripheral units in the system such as printer, disk drives, CD/DVD drives, flash memory, keyboards and so on.

CPU: It is the brain with circuitry. This controls the operations of the entire computer system.

1970's Computers were classified by the capacity and price.

A **mainframe** was a large machine—in size and in internal memory capacity. Example : The IBM 360 model 30 consist of CPU had 5 feet high and 6 feet wide, had an internal memory of 64K (considered large at that time), and a price tag of $200,000.

The **minicomputer** was developed to meet the needs of smaller institutions, those with only a few dozen users. One of the early minicomputers was marketed

by Digital Equipment Corporation to satisfy the needs of large schools and small colleges that began offering computer science courses in the early 1970s. Minicomputers are smaller in size and memory capacity and cheaper than mainframes.

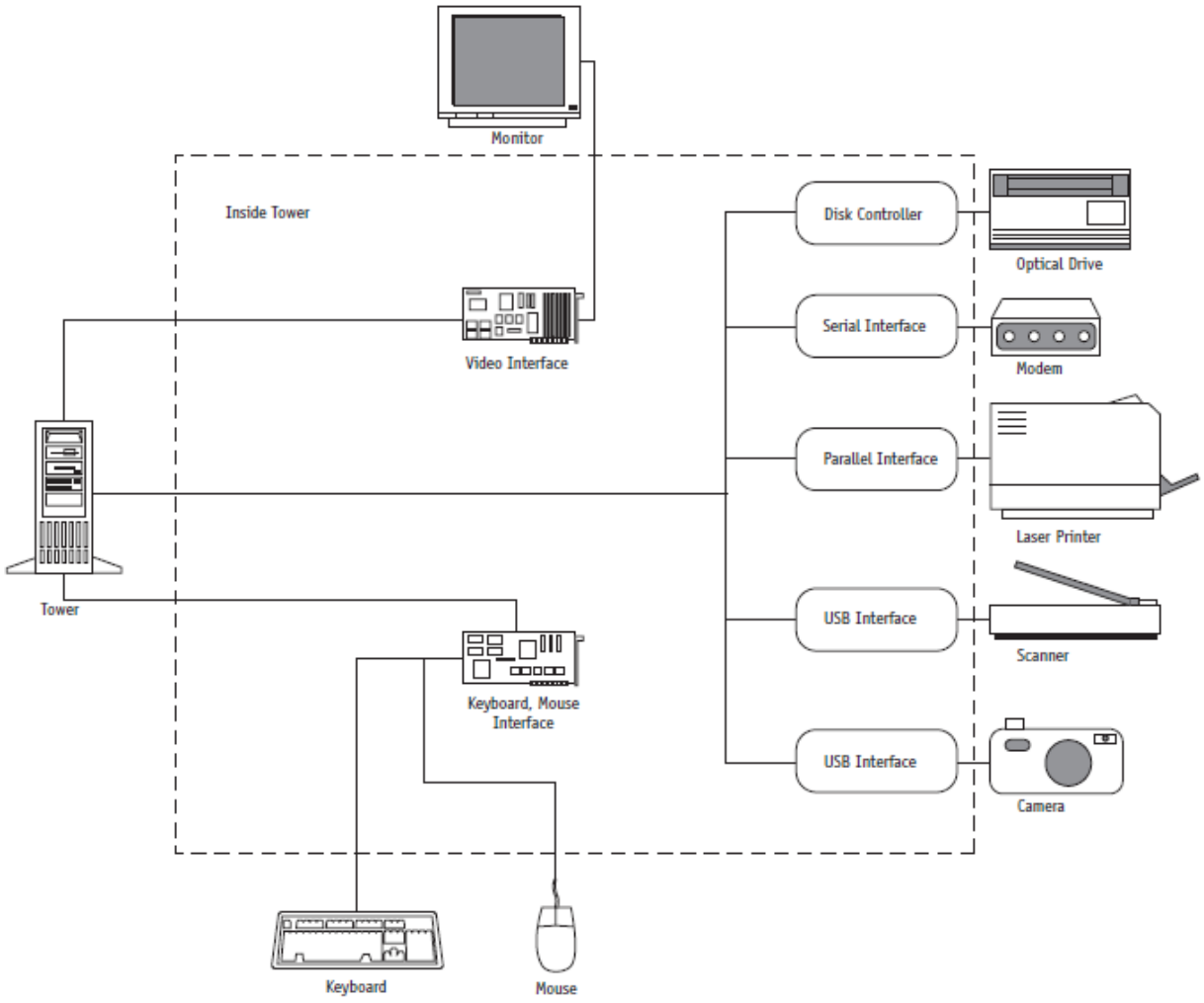


Figure 1.2 Hardware Architecture

The supercomputer was developed primarily for government applications needing massive and fast number-crunching ability to carry out military operations and weather forecasting. A Cray supercomputer is a typical example with six to thousands of processors performing up to 2.4 trillion floating point operations per second (2.4 Teraflops).

The **microcomputer** was developed to offer inexpensive computation capability to individual users. Early models featured a revolutionary amount of memory: 64K. Their physical size was smaller than the minicomputers. Microcomputers grew to accommodate software with larger capacity and greater speed.

Powerful microcomputers were developed for commercial, educational, and government enterprises are called **workstations**. It has very powerful CPUs, large amounts of main memory, and extremely high-resolution graphic displays to meet their needs.

Servers are powerful computers that provide specialized services to other computers on client/server networks.

Networking is an integral part of modern computer systems because it can connect Workstations, servers, and peripheral devices into integrated computing systems.

Networking capability has become a standard feature in many computing devices: Personal organizers, Personal Digital Assistants (PDAs), cell phones, and handheld Web browsers.

At one time, computers were classified by memory capacity; now they're distinguished by processor capacity.

Types of Operating System

Operating systems for computers fall into five categories distinguished by response time and how data is entered into the system.

1. Batch
2. Interactive
3. Real-Time
4. Hybrid
5. Embedded

Batch Operating System:

There is no interaction between user and the computer. The user submit a job by assembling into a deck of cards and running the entire deck of cards through a card reader as a group- a batch. The efficiency of the batch system is measured in throughput(The number of jobs completed in a given amount of time).

Interactive System :

Interactive operating system is one that allows these users to directly interact with the operating system. This would allow user to interact directly with the computer system via commands entered from a terminal.

Real-time System:

A Real-time operating system intended to serve real time applications. The system is subjected to real time (ie) responses should be guaranteed within a specified time constraints or system should meet the specified deadline. There are two types of real time system depending on the consequences of missing deadline.

Hard real-time system: If the predicted deadline is missed it causes total system failure.

Soft-Real time system: If the predicted deadline is missed the system suffers performance degradation.

Hybrid system: It is the combinations of batch and interactive operating system. They appear to be interactive because individual user can access the system and get fast response but such system actually accepts and runs batch program in the background when the interactive load is light.

Embedded System: Embedded systems are computers placed inside other products to add features and capabilities.eg. House Hold appliances, digital Music players.

Brief History of operating system development

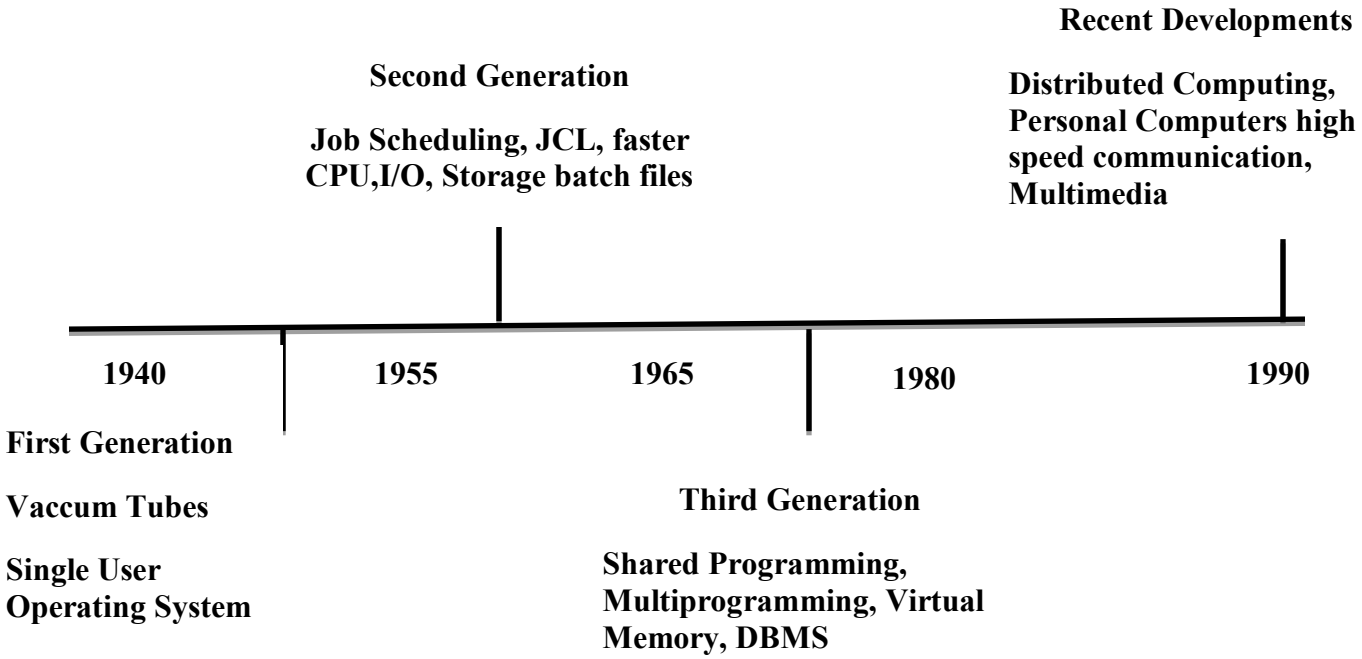


Figure 1.3 History of Operating System

1940's

- First generation computers based on vacuum tube technology
- No Standard operating system software
- A typical program included every instruction needed by the computer to perform the task requested.
- The machine was operated by the programmers from the main console.
- To run a program the programmers reserve machine for the estimated time. As a result the machine was poorly utilized.

1950's

- Second generation computers (1955-1965) were given much more importance of the cost effectiveness of the system. Two improvements were widely adopted.
- Computer operators: Humans hired to facilitate machine operations
- Job Scheduling: This is a productivity improvement scheme that group together programs with similar requirements.
- Job Control languages helped the operating system to co-ordinate and manage the system resources by identifying the users and their jobs and specifying the resources required to execute each job.

Factors to improve the performance of the CPU:

- The speed of I/O devices gradually increased.
- To use more storage space records were grouped into blocks before they were retrieved or stored.
- To reduce the discrepancy in between I/O and the CPU an interfaced called control unit was placed between them to act as buffer.
- Time Interrupts were developed to allow job sharing.

During the second generation programs were still assigned to the processor one at a time.

1960's

- Third generation computers designed towards better use of the system resources. Faster CPU but their speed caused problems.

- Multiprogramming concept was introduced to load many programs at one time and sharing the attention of single CPU.
- Program scheduling which was begun with second generation systems, continued at this time
- Few advances made in Process management and data management
- Total operating system was customized to suit user needs.

1970's

- During the late 1970s, Multiprogramming schemes to increase CPU use were limited by the physical capacity of the main memory, which was a limited resource and very expensive.
- A solution to this physical limitation was the development of virtual memory,
- A system with virtual memory would divide the programs into parts and keep them in secondary storage, bringing each part into memory only as it was needed.
- At this time, Database management software became a popular tool because it organized data in an integrated manner, minimized redundancy, and simplified updating and access of data.
- Query systems were introduced to retrieve specific pieces of the database.
- Application programs are introduced to perform standard operations.

1980's

- Development in the 1980s dramatically improved the cost/performance ratio of computer components.

- Programming functions were being carried out by the system's software
- **firmware**, used to indicate that a program is permanently held in read-only memory (ROM)
- **multiprocessing** (having more than one processor) concept allows to execute programs in parallel.
- 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.

1990s

- The overwhelming demand for Internet capability in the mid-1990s sparked the proliferation of networking capability.
- The World Wide Web, Web accessibility and e-mail became standard features of almost every operating system.
- Increased demand for tighter security to protect hardware and software.
- The decade also introduced a proliferation of multimedia applications demanding additional power, flexibility, and device compatibility for most operating systems.

2000s

- The new century emphasized the need for operating systems to offer improved flexibility, reliability, and speed.
- To meet the need for computers that could accommodate multiple operating systems running at the same time and sharing resources, the concept of virtual machines was developed and became commercially viable.
- **Virtualization** is the creation of partitions on a single server, with each partition supporting a different operating system.

- Processing speed has enjoyed a similar advancement with the development of multicore processors.

Object-Oriented Design

Object-Oriented design was the driving force behind this new organization. Which is a programming philosophy whereby programs consist of self-contained, reusable modules called objects, each of which supports a specific function, but which are categorized into classes of objects that share the same function. By working on objects, programmers can modify and customize pieces of an operating system without disrupting the integrity of the remainder of the system.

The first operating systems were designed as a comprehensive single unit, as shown in Figure 1.4 (a). They stored all required elements of the operating system in memory such as memory allocation, process scheduling, device allocation, and file management.

This type of architecture made it cumbersome and time consuming for programmers to add new components to the operating system, or to modify existing ones.

Most recently, the part of the operating system that resides in memory has been limited to a few essential functions, such as process scheduling and memory allocation, while all other functions, such as device allocation, are provided by special modules, which are treated as regular applications, as shown in Figure 1.4

(b). This approach makes it easier to add new components or modify existing ones.

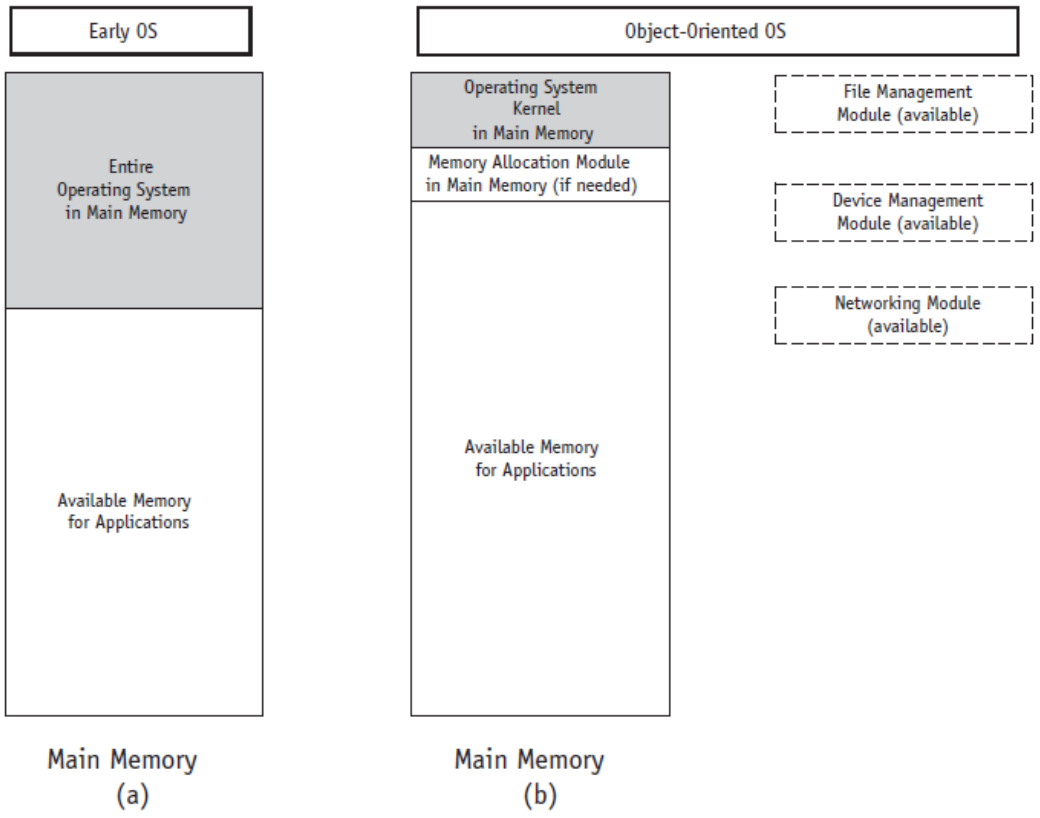


Figure 1.4: (a) Early operating systems, loaded in their entirety into main memory. (b) Object-oriented operating systems, load only the critical elements into main memory and call other objects as needed.

In addition, using a modular, object-oriented approach can can make software development groups more productive than was possible with procedural structured programming.

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; but like any complex 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 its up to the designer of the operating system to choose the policies that best match the system's environment.

Exercises

1. Name five current operating systems (not mentioned in this chapter) and the computers or configurations each operate.

2. Name the five key concepts about an operating system that you think a novice user needs to know and understand.

CHAPTER 2

Memory Management - Early Systems

Objectives

This chapter deals following concepts:

- The basic functionality of the three memory allocation schemes
- Best-fit memory allocation as well as first-fit memory allocation schemes
- How a memory list keeps track of available memory
- The importance of deallocation of memory in a dynamic partition system
- The importance of the bounds register in memory allocation schemes
- The role of compaction and how it improves memory allocation efficiency

Introduction

The management of **main memory** is critical. The *entire* system has been directly dependent on two things:

1) How much memory is available and
2) How it is optimized while jobs are being processed.

This chapter introduces the Memory Manager and four types of memory allocation schemes:

- Single-user Contiguous Scheme
- Fixed partitions,
- Dynamic partitions, and
- Relocatable dynamic partitions.

Single-User Contiguous Scheme

Each program to be processed was loaded in its entirety into memory and allocated as much contiguous space in memory as it needed. If the program was too large and didn't fit the available memory space, it couldn't be executed.

Example : ***The available memory is 200K. There are four jobs in the job list as per Single user contiguous scheme one program fit in memory at a time. The remainder of memory was unused.***

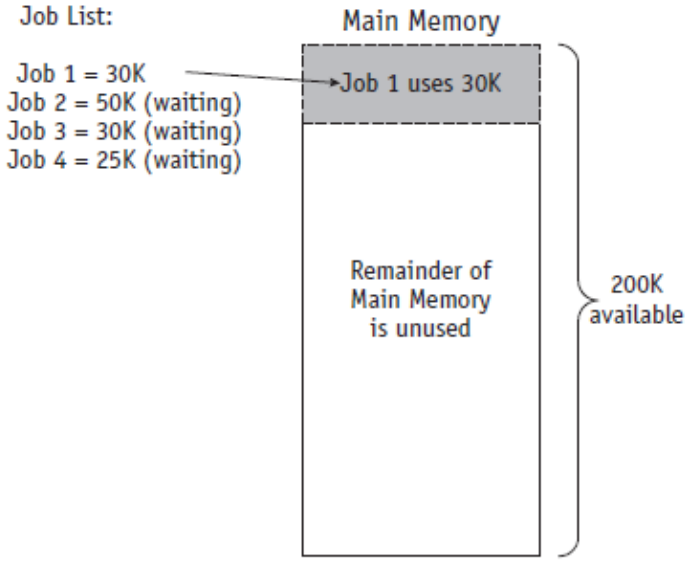


Figure 2.1 Single User Contiguous Scheme

Algorithm to Load a Job in a Single-User System

1 Store first memory location of program into base register (for memory protection)
2 Set program counter (it keeps track of memory space used by the program) equal to address of first memory location
3 Read first instruction of program
4 Increment program counter by number of bytes in instruction
5 Has the last instruction been reached?
    if yes, then stop loading program
    if no, then continue with step 6
6 Is program counter greater than memory size?
    if yes, then stop loading program
    if no, then continue with step 7
7 Load instruction in memory
8 Read next instruction of program
9 Go to step 4

Advantage:

It is simple to implement

Disadvantages:

- This type of memory allocation scheme doesn't support multiprogramming or networking.
- It can handle only one job at a time.

A new scheme was needed to manage memory, which used partitions to take advantage of the computer system's resources by overlapping independent operations.

Fixed Partitions

The first attempt to allow for multiprogramming used **fixed partitions** (also called **static partitions**) within the main memory.

Technique:

- In this technique, main memory is pre-divided into fixed size partitions.
- The size of each partition is fixed and cannot be changed.
- Each partition is allowed to store only one process.
- Once a partition was assigned to a job, no other job could be allowed to enter its boundaries, either accidentally or intentionally.

This algorithm checks whether the size of the job must be matched with the size of the partition to make sure it fits completely. Then, when a block of sufficient size is located, the status of the partition must be checked to see if it's available. Partition scheme allows several programs to be in memory at the same time.

Algorithm to Load a Job in a Fixed Partition

```
1 Determine job's requested memory size
2 If job_size › size of largest partition
    Then reject the job
        print appropriate message to operator
        go to step 1 to handle next job in line
    Else
      continue with step 3
3 Set counter to 1
4 Do while counter ‹= number of partitions in memory
  If job_size › memory_partition_size(counter)
  Then counter = counter + 1
  Else
    If memory_partition_size(counter) = "free"
      Then load job into memory_partition(counter)
        change memory_partition_status(counter) to "busy"
        go to step 1 to handle next job in line
    Else
      counter = counter + 1
    End do
5 No partition available at this time, put job in waiting queue
6 Go to step 1 to handle next job in line
```

In order to allocate memory spaces to jobs, the operating system's Memory Manager must keep a table, such as Table 2.1, which shows each memory partition size, its address, its access restrictions, and its current status (free or busy).

Table 2.1 ***A simplified fixed partition memory table with the free partition shaded.***

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

The Figure 2.2 illustrates the Main memory uses fixed partition allocation of Table 2.1. Job 3 must wait even though 70K of free space is available in partition1, where job1 only occupies 30K of 100k available. The jobs are allocated space on the basis of "first available partition of required size".

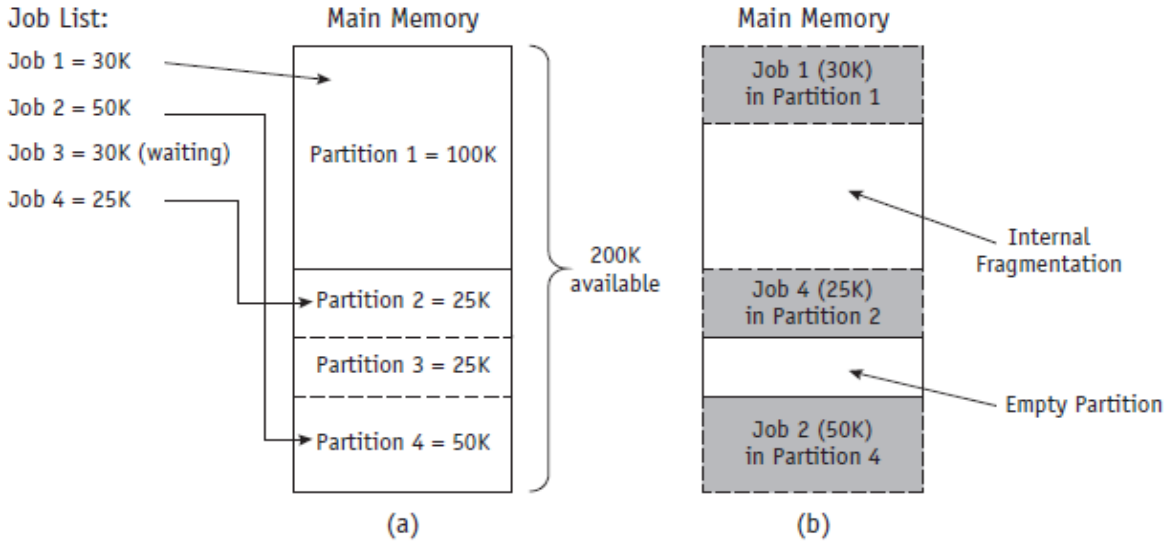


Figure 2.2 Main memory use during fixed partition allocation of Table 2.1. Job 3 must wait even though 70K of free space is available in Partition 1, where Job 1 only occupies 30K of the 100K available. The jobs are allocated space on the basis of "first available partition of required size."

Advantages:

- The fixed partition scheme works well if all of the jobs run on the system are of the same size

Disadvantages:

- It occurs when the space is left inside the partition after allocating the partition to a process.
- This space is called as internally fragmented space.
- This space cannot be allocated to any other process.
- This is because only static partitioning allows storing only one process in each partition.
- Internal Fragmentation occurs only in static partitioning.

Dynamic Partitioning

- Dynamic partitioning is a variable size partitioning scheme.
- It performs the allocation dynamically.
- When a process arrives, a partition of size equal to the size of process is created.
- Then, that partition is allocated to the process.

Figure 2.3 demonstrates the dynamic partition scheme fully utilizes the memory when the first job was loaded. The subsequent allocation of memory creates fragments of free memory between blocks of allocated memory. This problem is called **external fragmentation.**

In the last snapshot, (e) in Figure 2.3, there are three free partitions of 5K, 10K, and 20K—35K in all—enough to accommodate Job 8, which only requires 30K. However they are not contiguous and, because the jobs are loaded in a contiguous manner, this scheme forces Job 8 to wait.

Advantages:

- It does not suffer from internal fragmentation.
- Degree of multiprogramming is dynamic.
- There is no limitation on the size of processes.

Disadvantages:

- It suffers from external fragmentation.
- Allocation and deallocation of memory is complex.

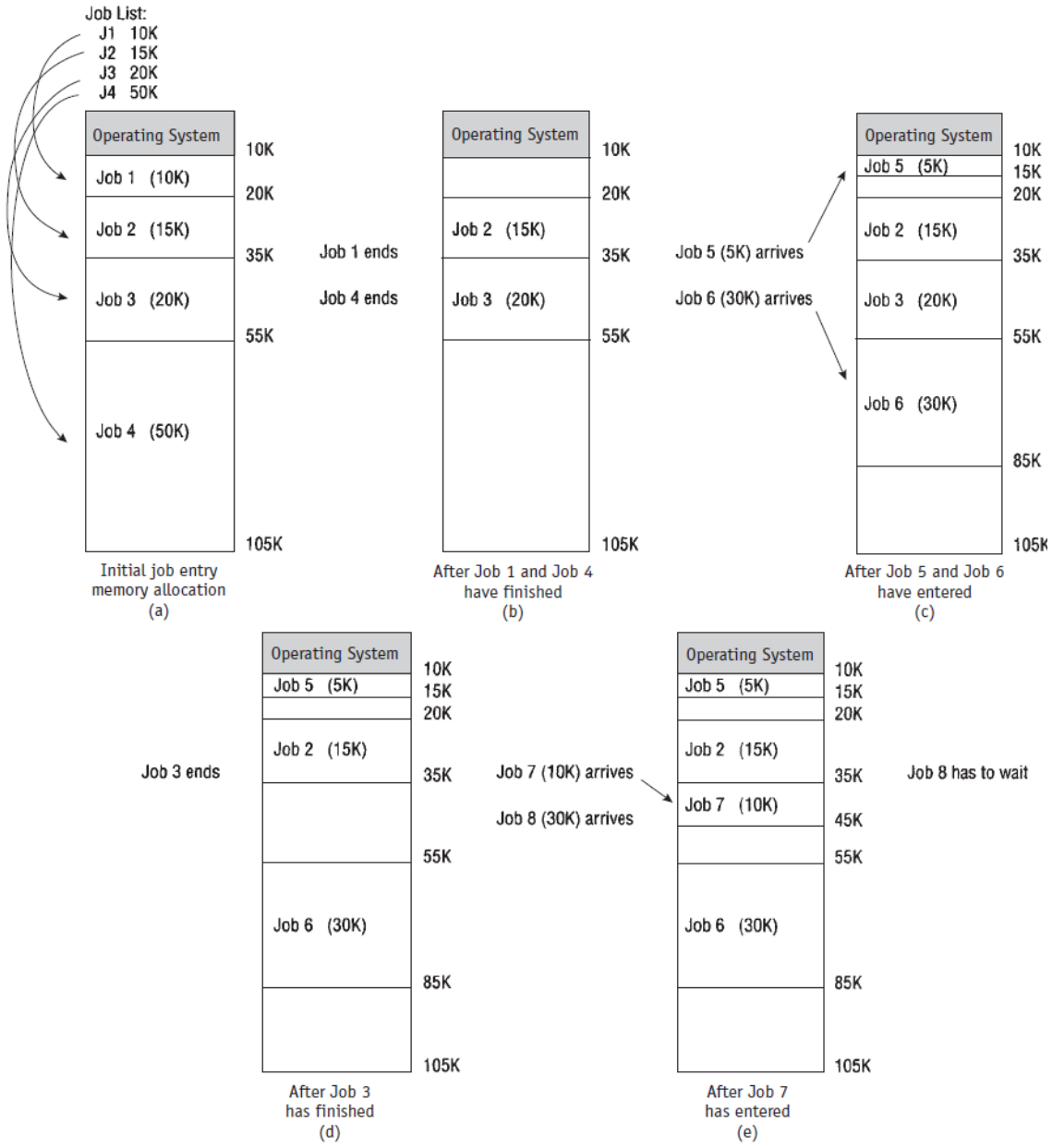


Figure 2.3 Main memory use during dynamic partition allocation. Five snapshots (a-e) of main memory as eight jobs are submitted for processing and allocated space on the basis of "first come, first served." Job 8 has to wait (e) even though there's enough free memory between partitions to accommodate it.

Best-Fit versus 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.

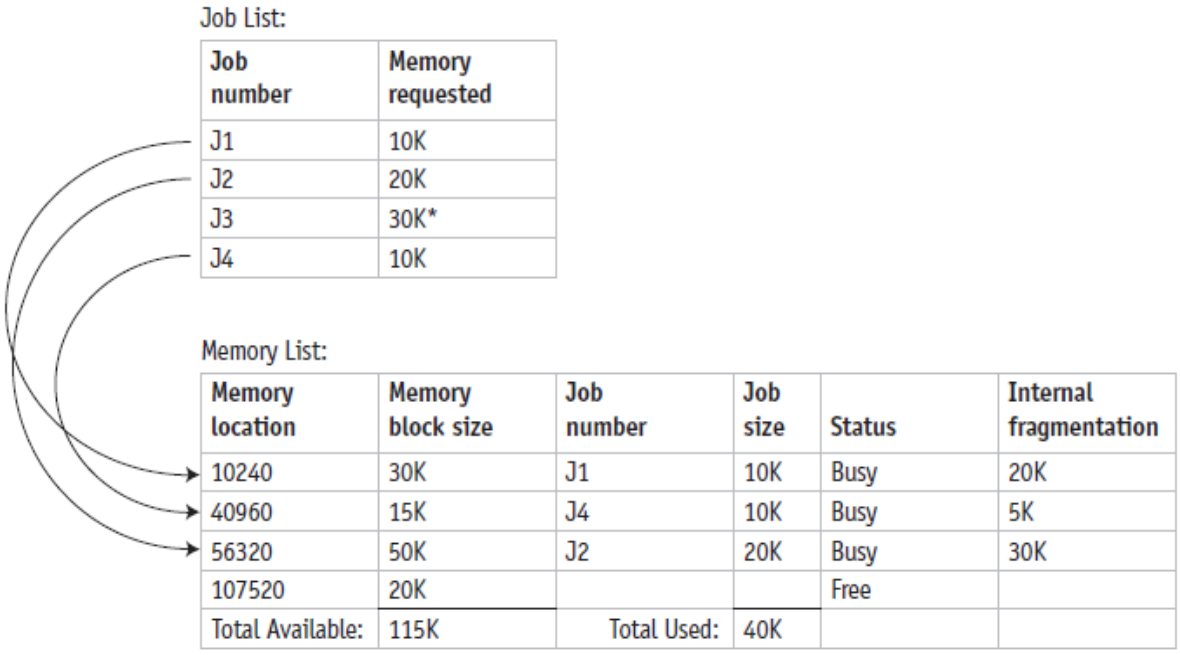
These partitions may be allocated on the basis of **first-fit memory allocation** (first partition fitting the requirements) or **best-fit memory allocation** (least wasted space, the smallest partition fitting the requirements). For both schemes, the Memory Manager organizes the memory lists of the free and used partitions (free/busy) either by size or by location.

The best-fit allocation method keeps the free/busy lists in order by size, smallest to largest. The first-fit method keeps the free/busy lists organized by memory locations, 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 in making the allocation.

Figure 2.4 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, if all of the fragments of memory were added together, there would be more than enough room to accommodate it. 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 2.5. In this particular case, a best-fit scheme would yield better memory utilization.

Figure 2.4 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 Job3. 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 and unused memory). Notice that the memory list is ordered according to memory location.



Job List:

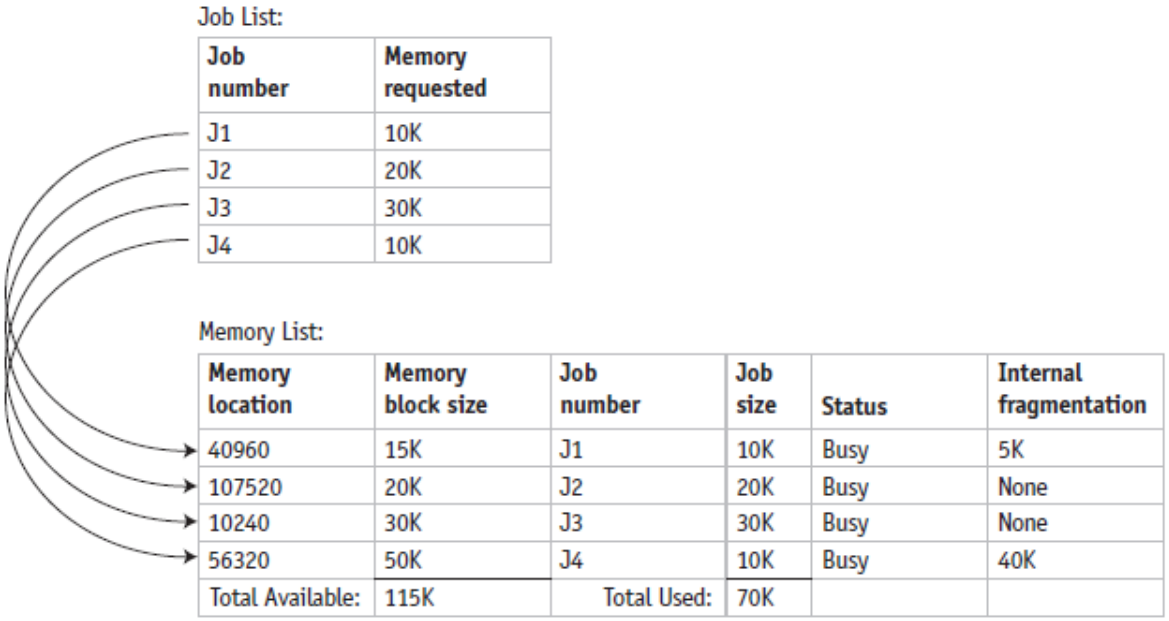
Job number	Memory requested
J1	10K
J2	20K
J3	30K*
J4	10K

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 2.4 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). N otice that the memory list is ordered according to memory location.

Figure 2.5 Best-Fit scheme. Job 1 is allocated to the closest fitting free partition, as are Job 2 and Job3. 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 ordered according to memory size. This scheme uses more efficiently but it's slower to implement.



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
40960	15K	J1	10K	Busy	5K
107520	20K	J2	20K	Busy	None
10240	30K	J3	30K	Busy	None
56320	50K	J4	10K	Busy	40K
Total Available:	115K	Total Used:	70K		

Figure 2.5 ***Bes t-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***

The algorithm for best-fit and first-fit are very different. The algorithm for best-fit is slightly more complex because the goal is to find the smallest memory block into which the job will fit:

Problem in Best-Fit

1. Entire table must be searched before the allocation can be made because the memory blocks are physically stored in sequence according to their location in memory

First-Fit Algorithm

```
1 Set counter to 1
2 Do while counter <= number of blocks in memory
    If job_size > memory_size(counter)
      Then counter = counter + 1
    Else
      load job into memory_size(counter)
      adjust free/busy memory lists
      go to step 4
  End do
3 Put job in waiting queue
4 Go fetch next job
```

Problem in Best-Fit

1. Entire table must be searched before the allocation can be made because the memory blocks are physically stored in sequence according to their location in memory .

2. 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.

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.

Best-Fit Algorithm

```
1 Initialize memory_block(0) = 99999
2 Compute initial_memory_waste = memory_block(0) – job_size
3 Initialize subscript = 0
4 Set counter to 1
5 Do while counter <= number of blocks in memory
    If job_size > memory_size(counter)
      Then counter = counter + 1
    Else
      memory_waste = memory_size(counter) – job_size
  If initial_memory_waste > memory_waste
      Then subscript = counter
      initial_memory_waste = memory_waste
      counter = counter + 1
  End do
6 If subscript = 0
  Then put job in waiting queue
  Else
      load job into memory_size(subscript)
      adjust free/busy memory lists
7 Go fetch next job
```

Deallocation

Until now, we've considered only the problem of how memory blocks are allocated, but eventually there comes a time when memory space must be released, or **deallocated**.

For a fixed partition system, the process is quite straightforward. When the job is completed, the Memory Manager resets the status of the memory block where the job was stored to "free." Any code—for example, binary values with 0 indicating free and 1indicating 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 because the algorithm 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 2.2 shows how deallocation occurs in a dynamic memory allocation system when the job to be deallocated is next to one free memory block.

Algorithm to Deallocate Memory Blocks

```
If job_location is adjacent to one or more free blocks
    Then
    If job_location is between two free blocks
       Then merge all three blocks into one block
       memory_size(counter-1) = memory_size(counter-1) + job_size
       + memory_size(counter+1)
       set status of memory_size(counter+1) to null entry
    Else
       merge both blocks into one
       memory_size(counter-1) = memory_size(counter-1) + job_size
Else
    search for null entry in free memory list
    enter job_size and beginning_address in the entry slot
    set its status to "free"
```

Table 2.2 This is the original free list before deallocation for Case 1. The asterisk indicates the free memory block that's adjacent to the soon-to-be-free memory block.

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 have been inserted here for clarity. The job size is 200 and its beginning location is 7600.*

After deallocation the free list looks like the one shown in Table 2.3

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).

Table 2.3 case 1: this is the free list after deallocation. The asterisk indicates the location where changes were made to the free memory block.

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

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 2.4.

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.

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 2.5. This negates the need to rearrange the list at the expense of memory.

Table 2.4 Case 2: This is the original free list before deallocation. The asterisks indicate the two free memory blocks that are adjacent to the soon-to-be-free memory block.

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 2.5 Case 2: The free list after a job has released

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 2.6 and Table 2.7.

Table 2.6 Case 3: Original free list before deallocation. The soon-to-be-free memory block 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

Table 2.7 Case 3: Busy memory list before deallocation. The job to be deallocated is of size 445 and begins at location 8805. The asterisk indicate the soon-to-be-free memory block.

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

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 2.8. This mechanism ensures that all blocks are entered in the lists according to the beginning address of their memory location from smallest to largest.

Table 2.8 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.

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

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 a null entry to free to indicate that a new block of memory is available, as shown in Table 2.9.

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 2.9. This negates the need to rearrange the list at the expense of memory.

Table 2.9 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.

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

Relocatable Dynamic Partitions

Both of the fixed and dynamic memory allocation schemes suffer from fragmentation. The solution to both problems was the development of **relocatable dynamic partitions**.

With this memory allocation scheme, the Memory Manager relocates programs together 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 garbage collection or

defragmentation, is performed by the operating system to reclaim fragmented sections of the memory space.

Compaction Steps:

Every program in memory must be relocated so the programs become contiguous Operating system must distinguish between addresses and data values Every address adjusted to account for the program's new location in memory data values left alone

The program in Figure 2.6 and Figure 2.7 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.

```
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 2.6: 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.

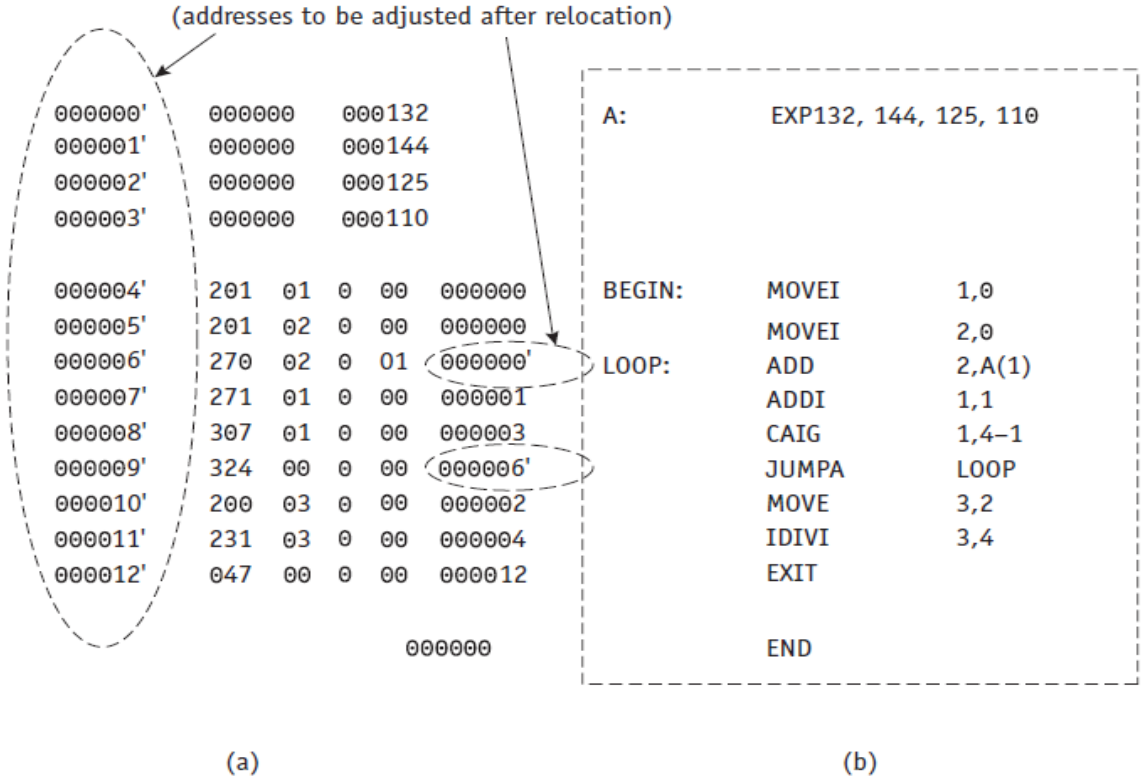


Figure 2.7 The original assembly language program after it has been processed by the assembler, shown on the right (a). To run the program, the assembler translates it into machine readable code (b) with all addresses marked by a special symbol (shown here as an apostrophe) to distinguish addresses from data values. All addresses (and no data values) must be adjusted after relocation.

Figure 2.8 illustrates what happens to a program in memory during compaction and relocation.

Compaction issues:

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.

What lists have to be updated?

Free list

- Must show the partition for the new block of free memory

Busy list

- Must show the new locations for all of the jobs already in process that were relocated

  Each job will have a new address

Exception: those already at the lowest memory locations

To answer the other two questions we must learn more about the hardware components of a computer Special-purpose registers used for relocation:

Bounds register

Stores highest location accessible by each program

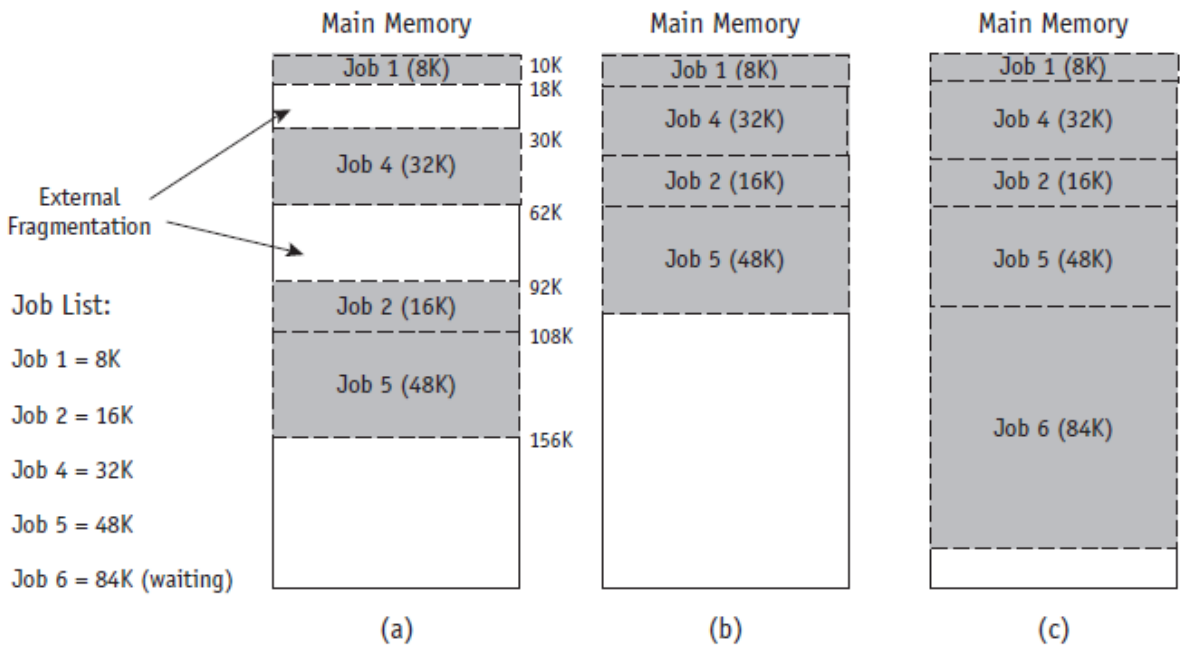


Figure 2.8 ***Three snapshots of memory before and after compaction with the operating system occupying the first 10K of memory. When Job 6 arrives requiring 84K, the initial memory layout in (a) shows external fragmentation totaling 96K of space. Immediately after compaction (b), external fragmentation has been eliminated, making room for Job 6 which, after loading, is shown in (c).***

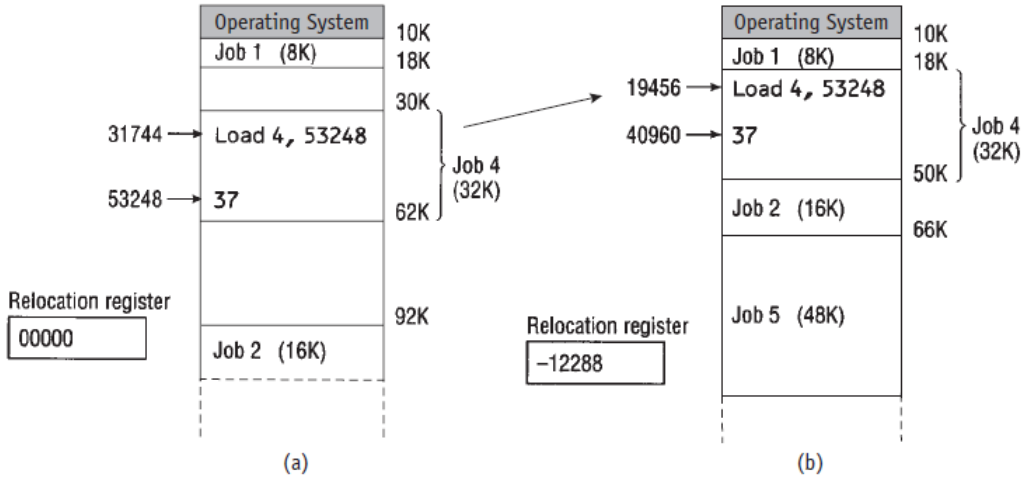
Relocation register

- Contains the value that must be added to each address referenced in the program Must be able to access the correct memory addresses after relocation
- If the program is not relocated, "zero" value stored in the program's relocation register

Figure 2.9 illustrates what happens during relocation by using the relocation register (all values are shown in decimal form).

In effect, by compacting and relocating,

- Memory Manager optimizes the use of memory
  - Improves throughput
- Options for timing of compaction:
  - When a certain percentage of memory is busy
  - When there are jobs waiting to get in
  - After a prescribed amount of time has elapsed
- Compaction entails more overhead



Contents of relocation register and close-up of Job 4 memory area (a) before relocation and (b) after relocation and compaction.

Goal: optimize 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. They have three things in common: They all require that the entire program (1) be loaded into memory, (2) be stored contiguously, and (3) remain in memory until the job is completed.

Consequently, each inputs severe restrictions on the size of the jobs because they can only be as large as the biggest partitions in memory. These schemes were sufficient for the first three generations of computers, which processed jobs in batch mode.

Turnaround time was measured in hours, or sometimes days, but that was a period when users expected such delays between the submission of their jobs and pick up of output.

Exercise

1. Given the following information

Job list:

Job Number	Memory Requested	Memory Block	Memory Block Size
Job 1	690 K	Block 1	900 K (low-order memory)
Job 2	275 K	Block 2	910 K
Job 3	760 K	Block 3	300 K (high-order memory)

a. Use the best-fit algorithm to indicate which memory blocks are allocated toeach 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.

2. Given the following information:

Job list:

Job Number	Memory Requested	Memory Block	Memory Block Size
Job 1	275 K	Block 1	900 K (low-order memory)
Job 2	920 K	Block 2	910 K
Job 3	690 K	Block 3	300 K (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.

CHAPTER 3

MEMORY MANAGEMENT: VIRTUAL MEMORY

In this chapter we'll follow the evolution of virtual memory with four memory allocation schemes that first remove the restriction of storing the programs contiguously, and then eliminate the requirement that the entire program reside in memory during its execution. These schemes are

- Paged
- Demand paging,
- Segmented,
- Segmented/demand paged allocation,

which form the foundation for our current virtual memory methods.

Paged Memory Allocation

Paged memory allocation is based on the concept of dividing each incoming job into **pages** of equal size. That will be loaded into memory locations called page frames. Operating systems choose a page size that is the same as the memory block size and that is also the same size as the sections of the disk on which the job is stored.

The sections of a disk are called **sectors** (or sometimes blocks), and the sections of main memory are called **page frames.** The scheme works quite efficiently when the pages, sectors, and page frames are all the same size. The

exact size determined by the disk's sector size. Therefore, one sector will hold one page of job instructions and fit into one page frame of memory.

Before executing a program, the 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 them

The primary advantage of storing programs in noncontiguous locations 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 (and no internal fragmentation in most pages).

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 and complexity of the operating system software, which increases overhead.

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. Notice in Figure 3.1 that the last page (Page 3) is not fully utilized because the job is less than 400 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 till a problem (but only in the last page of a job).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 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.

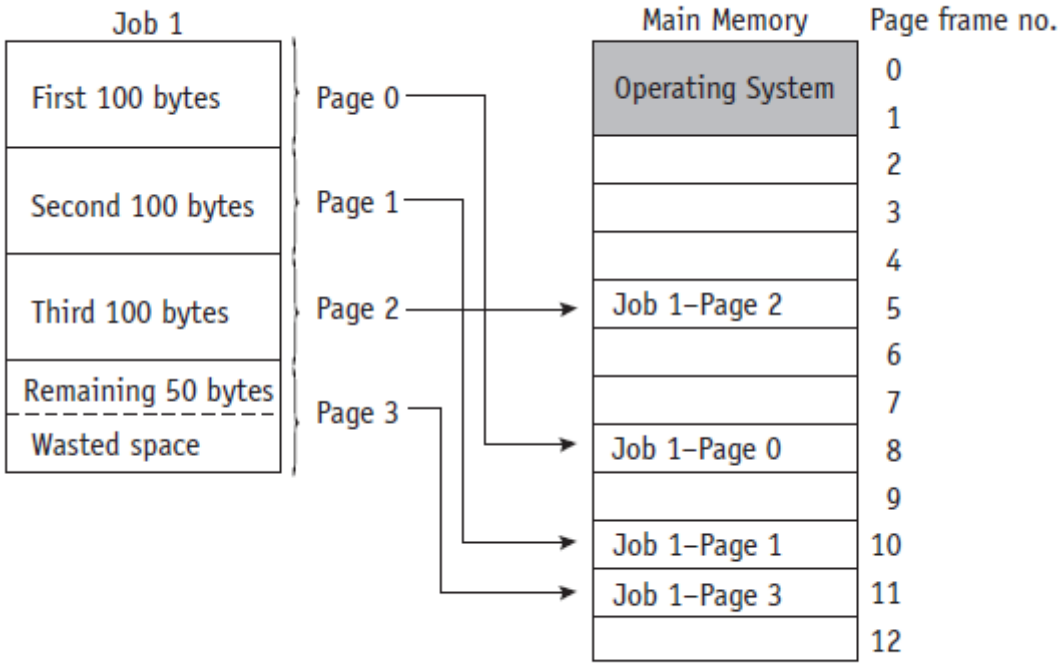


Figure 3.1 ***Programs that are too long to fit on a single page are split into equal-sized pages that can be stored in free page frames. In this example, each page frame can hold 100 bytes. Job 1 is 350 bytes long and is divided among four page frames, leaving internal fragmentation in the last page frame. (The Page Map Table for this job is shown later in Table 3.2.)***

There are essentially three tables that perform this function: the Job Table, Page Map Table, and Memory Map Table.

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).

Page Map Table (PMT), which contains the page number and its corresponding page frame memory address. Actually, the PMT includes only one entry per page.

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

Table 3.1 This section of the Job Table (a) initially has three entries, one for each job in progress. When the second job ends

(b), its entry in the table is released and it is replaced

(c) by information about the next job that is to be processed.

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)

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.

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 349.

The **displacement**, or **offset**, of a byte is the factor used to locate that byte within its page frame. It is a relative factor.

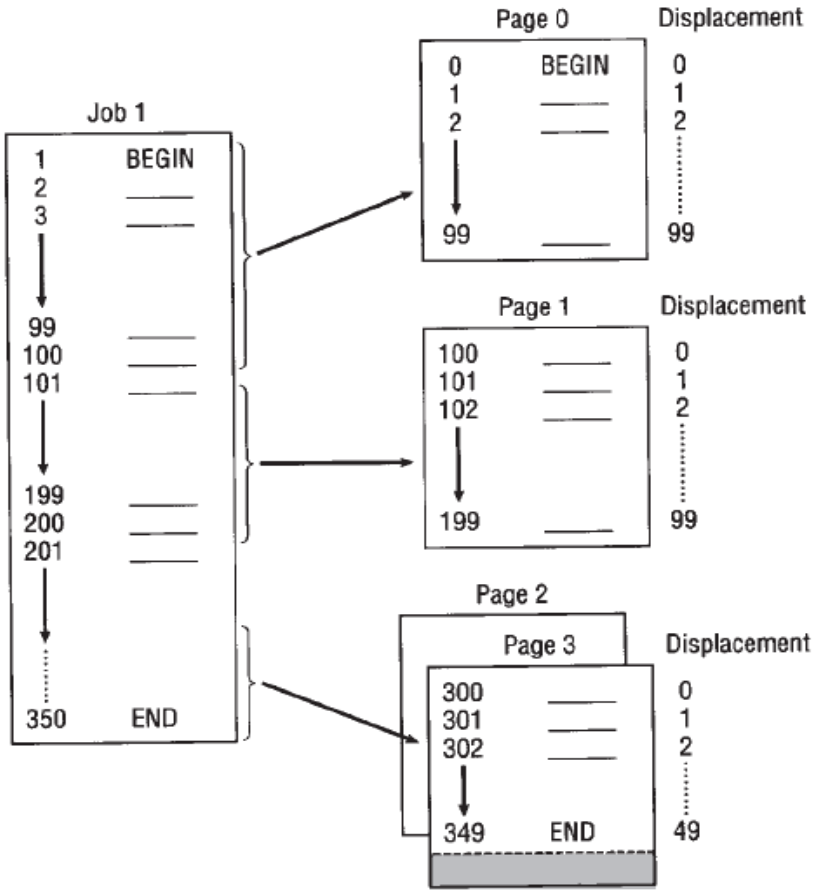
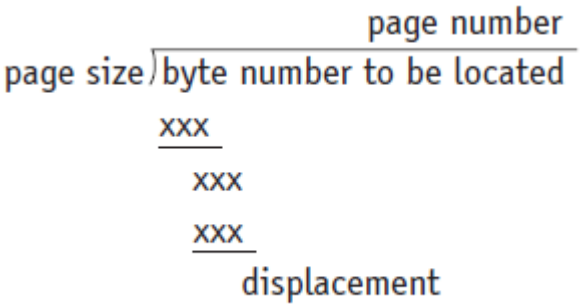


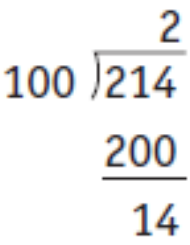
Figure 3.2 ***Job 1 is 350 bytes long and is divided into four pages of 100 lines each.***

The operating system uses an algorithm to calculate the page and displacement; it is a simple arithmetic calculation. To find the address of a given program

instruction, the byte number is divided by the page size, keeping the remainder as an integer.



For example, if we use 100 bytes as the page size, the page number and the displacement (the location within that page) of byte 214 can be calculated using long division like this: The quotient (2) is the page number, and the remainder (14) is the displacement. So the byte is located on Page 2, 15 lines (Line 14) from the top of the page.



STEP 1: Do the arithmetic computation just described to determine the page number and displacement of the requested byte.

• Page number = the integer quotient from the division of the job space address by the page size

• Displacement = the remainder from the page number division

For example, if we use 100 bytes as the page size, the page number and the displacement of byte 214

In this example, the computation 214 ÷ 100 shows that the page number is 2 and the displacement is 14.

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.

Table 3.2 Page Map Table for Job 1 in Figure 3.1.

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

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).

ADDR_PAGE_FRAME = PAGE_FRAME_NUM * PAGE_SIZE

ADDR_PAGE_FRAME = 5(100)

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_ADDR_IN_MEM = ADDR_PAGE_FRAME + DISPL

INSTR_ADDR_IN_MEM = 500 + 14

The result of this maneuver tells us exactly where byte 14 is located in main memory.

Advantage : Jobs to be allocated in noncontiguous memory locations so that memory is used more efficiently and more jobs can fit in the main memory (which is synonymous).

Disadvantages: address resolution causes increased overhead. Internal fragmentation is still a problem, although 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 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.

Demand Paging

Demand paging introduced the concept 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, 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.

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

• User-written error handling modules are processed only when a specific error is detected during execution. Many modules are mutually exclusive. Certain program options are either mutually exclusive or not always accessible. This is easiest to visualize in menu-driven programs. Many tables are assigned a large fixed amount of address space even though only a fraction of the table is actually used.

Figure 3.3When 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.

Most important innovations of demand paging was that it made virtual memory feasible. The key to the successful implementation of this scheme is Use of a high-speed direct access storage device (such as hard drives or flash memory) that can work directly with the CPU. That is vital because pages must be passed quickly from secondary storage to main memory and back again.

How and when the pages are passed (also called swapped) 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 Table, and the Memory Map Table) to implement the algorithm. These tables are basically the same as for paged memory allocation but with the addition of three new fields.

Page Memory Table : The first field tells the system where to find each page. If it is already in memory, the system will be spared the time required to bring it from secondary storage.

The second field, noting if the page has been modified, 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, already there, is correct.

The third field, which indicates any recent activity, 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.

For example, in Figure 3.4 the number of total job pages is 15, and the number of total available page frames is 12. (The operating system occupies the first four of the 16 page frames in main memory.)

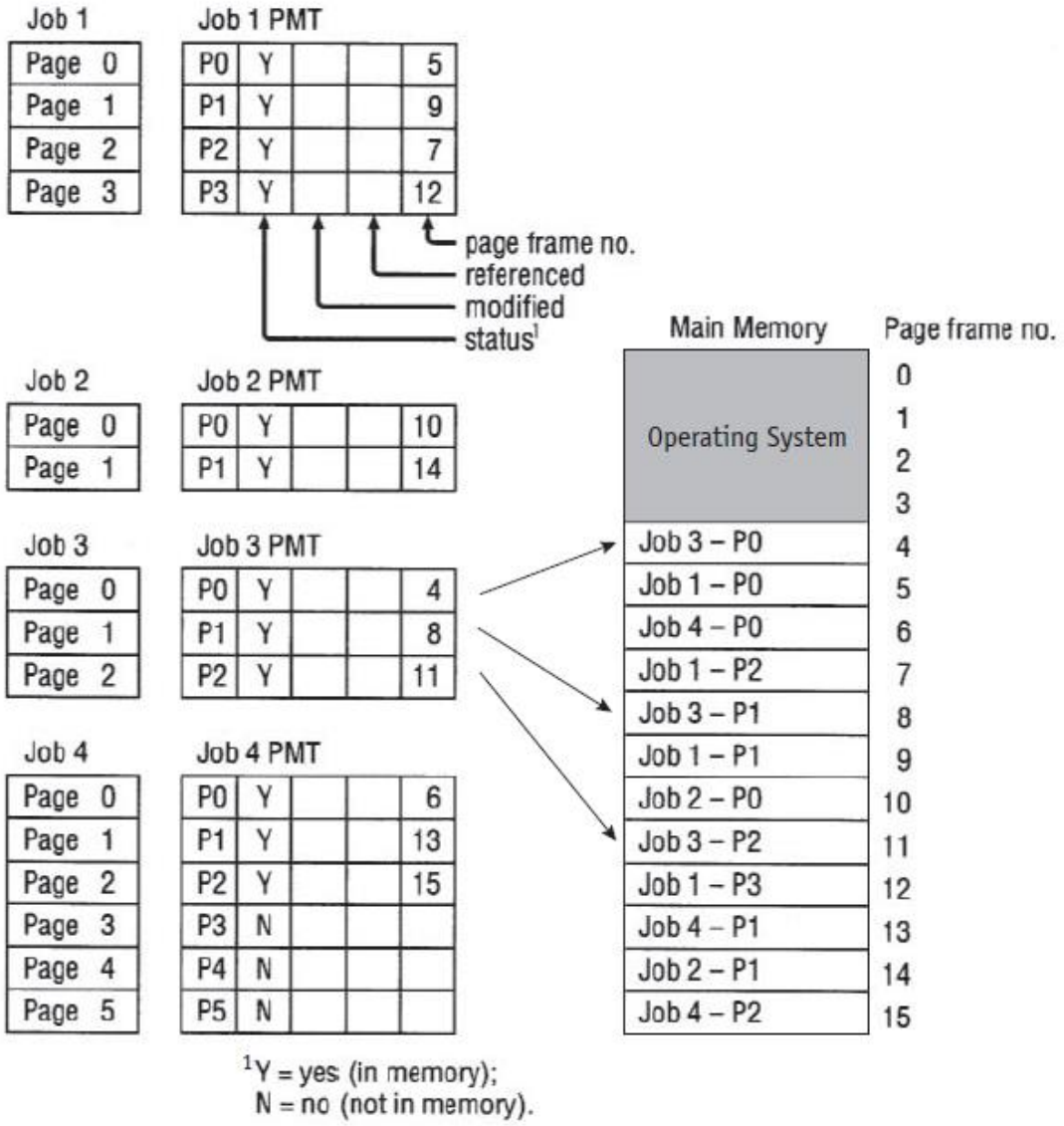


Figure 3.4 Demand paging requires that the Page Map Table for each job keep track of each page as it is loaded or removed from main memory. Each PMT tracks the status of the page, whether it has been modified, whether it has been recently referenced, and the page frame number for each page currently in main memory. (Note: For this illustration, the Page Map Tables have been simplified. See Table 3.3 for more detail.)

Assuming the processing status illustrated in Figure 3.4, what happens when Job 4 requests that Page 3 be brought into memory if there are no empty page frames available?

To move in a new page, a resident page 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 empty page frame.

Swapping process requires close interaction between Hardware components, software algorithms and policy schemes. The hardware components generate the address of the required page, find the page number, and determine whether it is already in memory.

When the requested page is in secondary storage but not in memory,it is consider as a page fault which can be carried over by operating system software. The section of the operating system that resolves these problems is called the **page fault handler**.

It determines whether there are empty page frames in memory so the requested page can be immediately copied from secondary storage. If all page frames are busy, the page fault handler must decide which page will be swapped out. (This decision is directly dependent on the predefined policy for page removal.) Then the swap is made.

Before continuing, 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) and the Memory Map Table. Finally, the instruction that was interrupted is resumed and processing continues.

Page Fault Handler Algorithm

1. *If there is no free page frame Then*
   select page to be swapped out using page removal algorithm
   update job's Page Map Table
   If content of page had been changed then
   write page to disk
   End if
   End if
2. *Use page number from step 3 from the Hardware Instruction Processing Algorithm to get disk address where the requested page is stored (the File Manager, to be discussed in Chapter 8, uses the page number to get the disk address)*
3. *Read page into memory*
4. *Update job's Page Map Table*
5. *Update Memory Map Table*
6. *Restart interrupted instruction*

When there is an excessive amount of **page swapping** between main memory and secondary storage, the operation becomes inefficient. This phenomenon is called **thrashing.**

We can demonstrate this with a simple example. Suppose the beginning of a loop falls at the bottom of a page and is completed at the top of the next page, as in the C program in Figure 3.5. this example would generates100 page faults (and swaps).

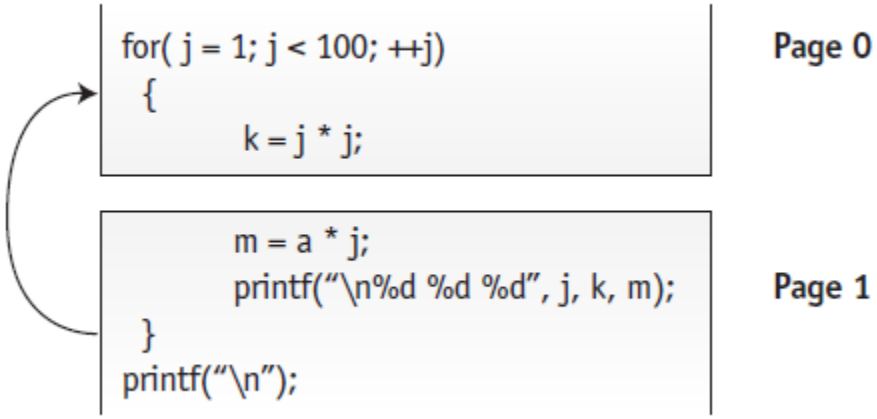


Figure 3.5 An example of demand paging that causes a page swap each time the loop is executed and results in thrashing. If only a single page frame is available, this program will have one page fault each time the loop is executed.

In such extreme cases, the 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

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.

Two of the most well-known are first-in first-out and least recently used. The **first-in first-out (FIFO) policy** is based on the theory 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.

First-In First-Out

The first-in first-out (FIFO) page replacement policy will remove the pages that have been in memory the longest. The process of swapping pages is illustrated in Figure 3.6.

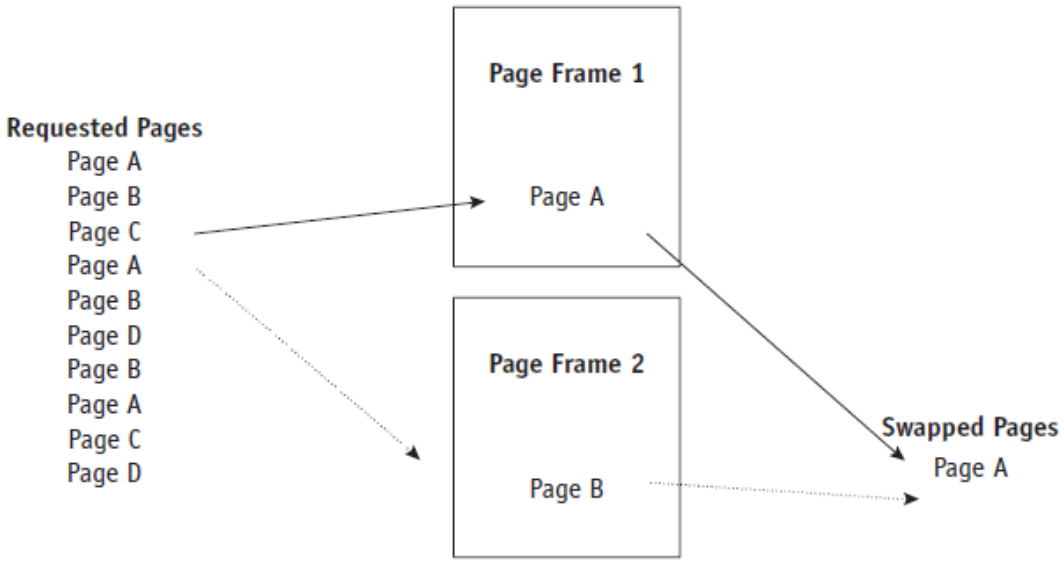


Figure 3.6 The FIFO policy in action with only two page frames available. When the program calls for Page C,Page A must be moved out of the first page frame to make room for it, as shown by the solid lines. When Page A is needed again, it will replace Page B in the second page frame, as shown by the dotted lines. The entire sequence is shown in Figure 3.7.

Figure 3.7 shows how the FIFO algorithm works by following a job with four pages (A, B, C, D) as it is processed by a system with only two available page frames. Figure 3.8 displays how each page is swapped into and out of memory and marks each interrupt with an asterisk.

We then count the number of page interrupts and compute the failure rate and the success rate. The job to be processed needs its pages 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 existing one to be swapped out to secondary storage. A page interrupt, which we identify with an asterisk (*), is generated when a new page needs to be loaded into memory, whether a page is swapped out or not.

The efficiency of this configuration is dismal—there are 9 page interrupts out of 11 page requests due to the limited number of page frames available and the need for many new pages.

To calculate the failure rate, we divide the number of interrupts by the number of page requests. 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.

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. The page order can't be changed by the system, although the size of main memory can be changed; but buying more memory may not always be the best solution.

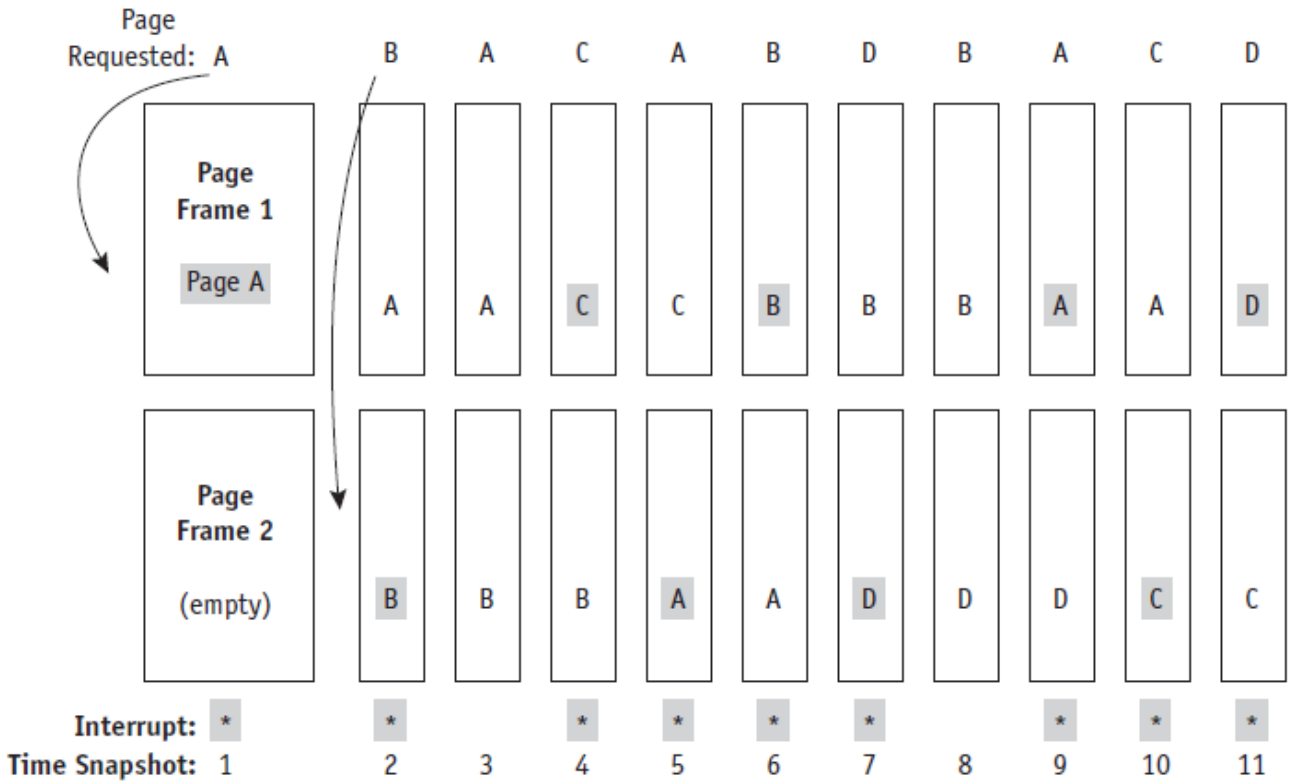


Figure 3.7 Using a FIFO policy, this page trace analysis shows how each page requested is swapped into the two available page frames. When the program is ready to be processed, all four pages are in secondary storage. When the program calls a page that isn't already in memory, a page interrupt is issued, as shown by the gray boxes and asterisks. This program resulted in nine page interrupts.

Least Recently Used

The least recently used (LRU) page replacement policy swaps out the pages that show the least amount of recent activity, figuring that these pages are the least likely to be used again in the immediate future.

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 3.8. To implement this policy, a queue of the requests is kept in FIFO order, a time stamp of when the job entered

the system is saved, or a mark in the job's PMT is made periodically.

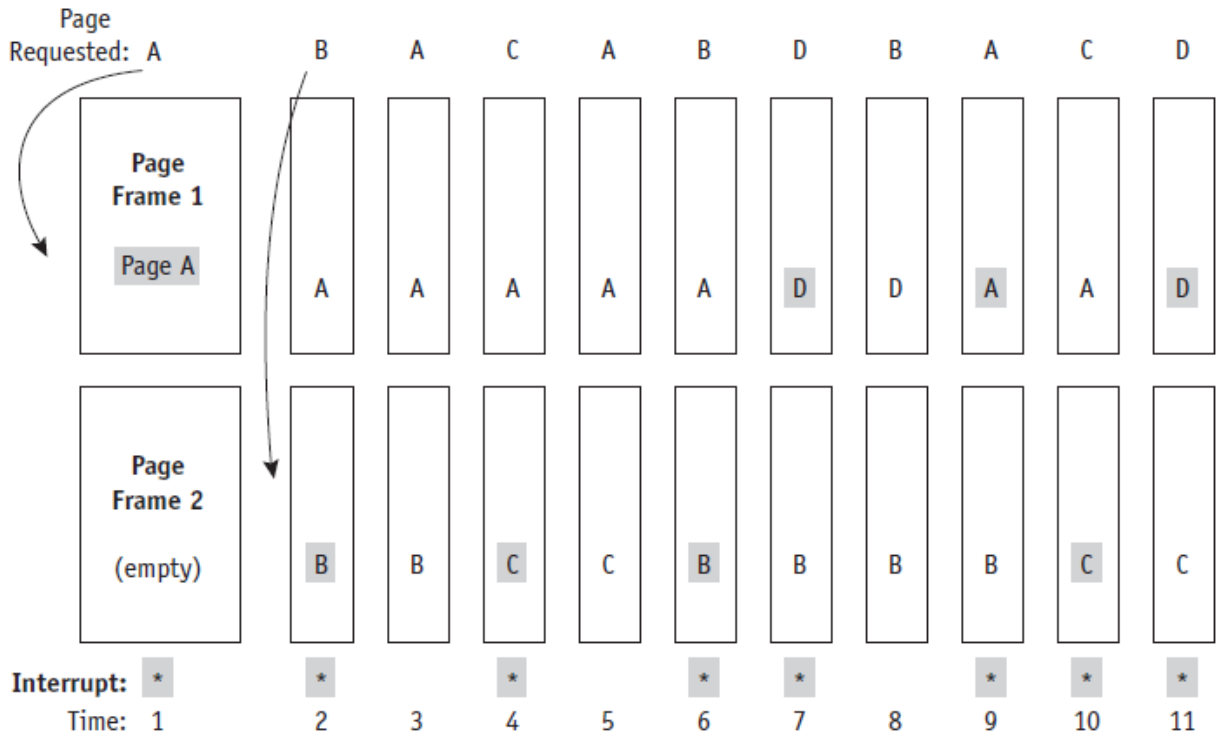


Figure 3.8Memory management using an LRU page removal policy for the program shown in Figure 3.7. Throughout the program, 11 page requests are issued, but they cause only 8 page interrupts.

Here, there are 8 page interrupts out of 11 page requests, so the failure rate is 8/11, or 73 percent. LRU is a stack algorithm removal policy, which means that an increase in memory will never cause an increase in the number of page interrupts.

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.** 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 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 3.9 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 3.9 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 space that has a reference bit of zero, the space previously occupied by page 210.

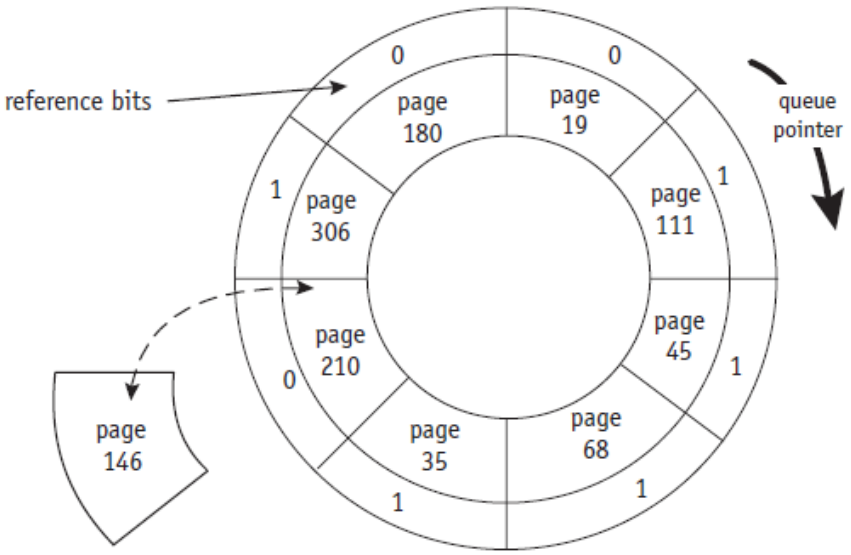


Figure 3.9 A circular queue, which contains the page number and its reference bit. The pointer seeks the next candidate for removal and replaces page 210 with a new page, 146.

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, as shown in 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 ticks would have a reference byte of 00000000, while one that has been referenced once every time tick 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 3.10 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. Other page removal algorithms, MRU (most recently used) and LFU (least frequently used).

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 3.10 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.

Segmented memory allocation

In a segmented memory allocation, each job is divided into several **segments** of different sizes, one for each module that contains pieces that perform related functions. Segmented memory allocation was designed to reduce page faults that resulted from having a segment's loop split over two or more pages. Eg. **subroutine**

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 3.11 and 3.12 show the same job, Job 1, composed of a main program and two subroutines, together with its Segment Map Table and actual main memory allocation.

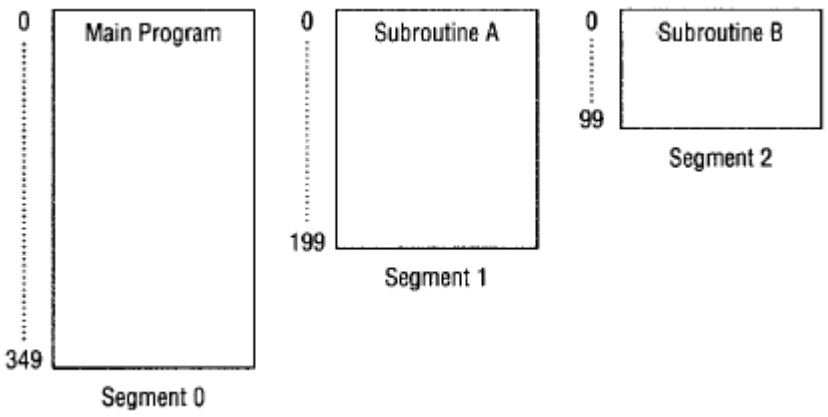


Figure 3.11 Segmented memory allocation. Job 1 includes a main program, Subroutine A, and Subroutine B. It is one job divided into three segments.

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).

The instructions within each segment are ordered sequentially, but the segments don't need to be stored contiguously in memory. We only need to know

where each segment is stored. The contents of the segments themselves are contiguous in this scheme. 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.

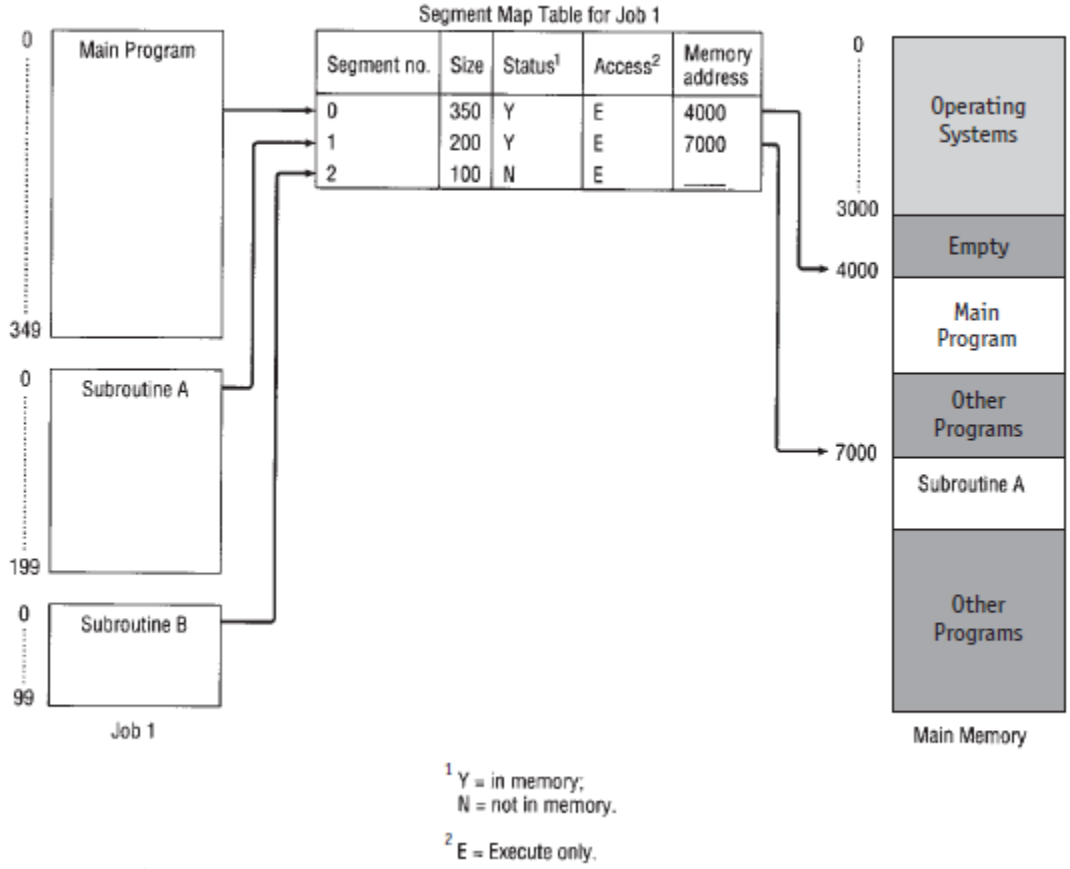


Figure 3.12 The Segment Map Table tracks each segment for Job 1.

In Figure 3.13, Segment 1 includes all of Subroutine A so the system finds the beginning address of Segment 1, address 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). Its code could look like this:

ACTUAL_MEM_LOC = BEGIN_MEM_LOC + DISPLACEMENT

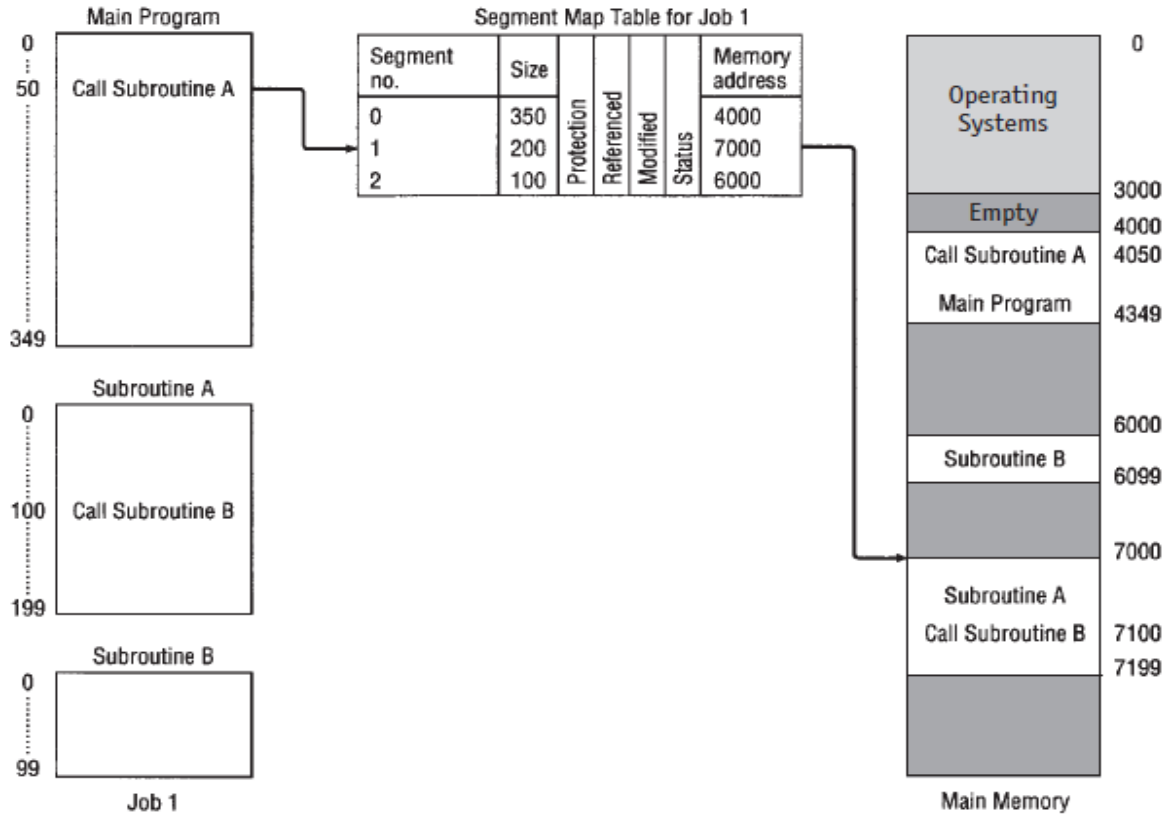


Figure 3.13 During execution, the main program calls Subroutine A, which triggers the SMT to look up its location in memory.

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.

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

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

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.

This allocation scheme follows noncontiguous unit but subdivides it into pages of equal size, 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 3.14, requires four tables:

- The Job Table lists every job in process (one for the whole system).
- The Segment Map Table lists details about each segment (one for each job).
- The Page Map Table lists details about every page (one for each segment).
- The Memory Map Table monitors the allocation of the page frames in main memory(one for the whole system).The SMT actually includes additional information regarding protection (such as the authority to read, write, execute,

and delete parts of the file), as well as which users have access to that segment (user only, group only, or everyone—some systems call these access categories owner, group, and world, respectively). In addition, the PMT includes the status, modified, and referenced bits.

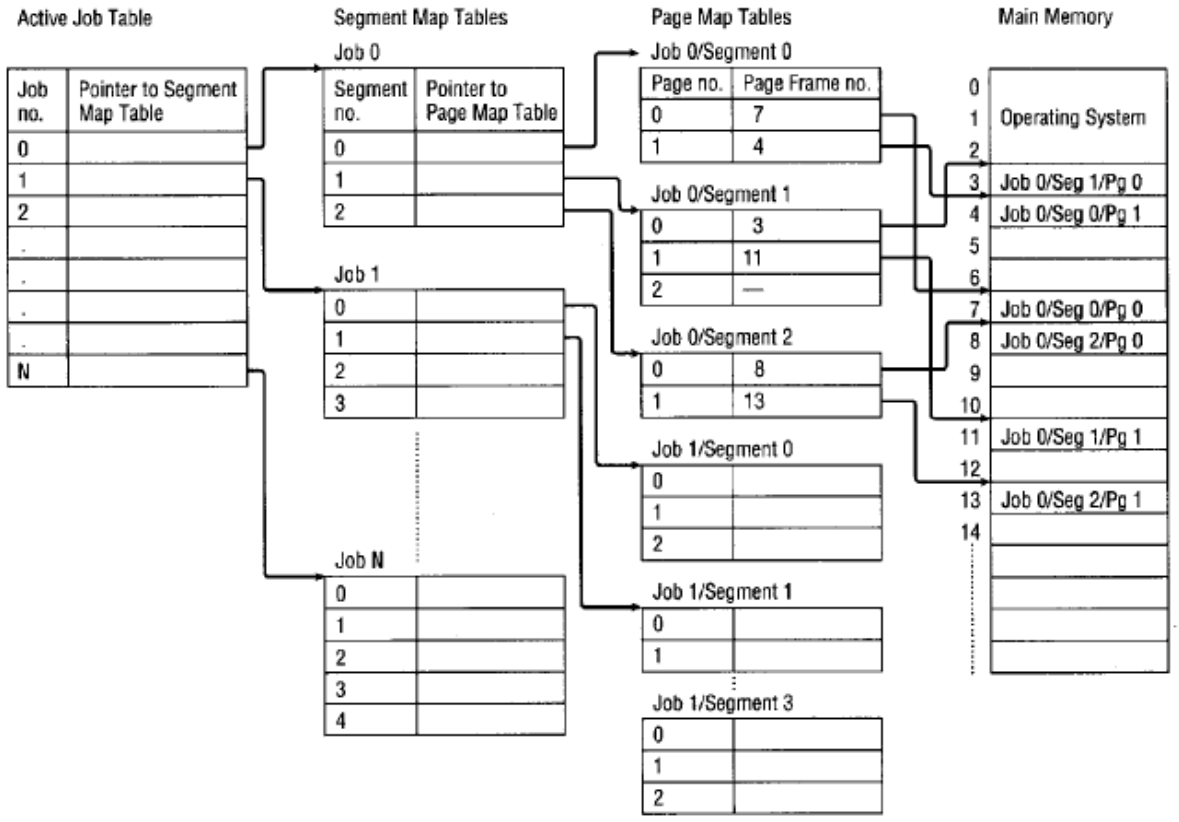


Figure 3.14 How the Job Table, Segment Map Table, Page Map Table, and main memory interact in a segment/paging scheme.

To access a location in memory, the system must locate the address, which is composed of three entries: segment number, page number within that segment, and displacement within that page.

The major disadvantages of this memory allocation scheme are the overhead required for the extra tables and the time required to reference the segment table

and the page table. To minimize the number of references, many systems use 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. Their task is to associate several segment and page numbers belonging to the job being processed with their main memory addresses. To appreciate the role of associative memory, it is important to understand the procedure: associative memory, which stores the information related to the most-recently-used pages.

Then when a page request is issued, two searches begin—one through the segment and page 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 stopped (or eliminated) 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 SMTs 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 and 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 is used to hold the information on this requested page.

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. In some systems the searches do not run in parallel, but the search of the SMT and PMT follows the search of the associative registers.

Virtual Memory

Virtual memory is a technique that allows programs to be executed even though they are not stored entirely in memory. Virtual memory gives a way to swap pages in/out of memory. Although the swapping of overlays between main memory and secondary storage was done by the system, the tedious task of dividing the program into sections was done by the programmer. It was the concept of overlays that suggested paging and segmentation and led to virtual memory, which was then implemented through demand paging and segmentation schemes. These schemes are compared in Table 3.3.

Table 3.3 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 PMT	Requires SMT

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 is needed that is not already in memory, 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 management has several advantages:

- A job's size is no longer restricted to the size of main memory (or the free space 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 many users in a time-sharing environment.
- It eliminates external fragmentation and minimizes internal fragmentation by combining segmentation and paging (internal fragmentation occurs in the program).
- 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 is small in size (compared to main memory), it can use faster, more expensive memory chips and can be five to ten times faster than main memory and match the speed of the CPU. Therefore, when frequently used data or instructions are stored in cache memory, memory access time can be cut down significantly and the CPU can execute instructions faster, thus raising the overall performance of the computer system.

Computer systems automatically store 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 3.15.

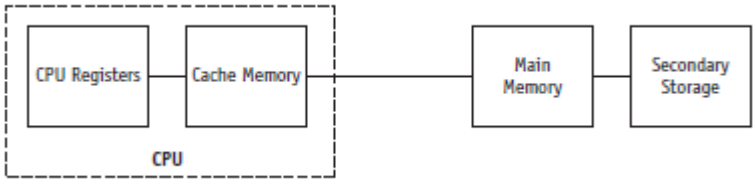


Figure 3.15 The path used by modern computers to connect the main memory and the CPU via cache memory.

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. 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.

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

- *Cache size*
- *Block size*
- *Block replacement algorithm*
- *Rewrite policy.*

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 (h), is used to determine the performance of cache memory and represents the percentage of total memory requests that are found in the cache:

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

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.

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

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 using the following formula:

AvgMemAccessTime = AvgCacheAccessTime + (1 – h) * AvgMainMemAccTime

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

Conclusion

The Memory Manager has the task of allocating memory to each job to be executed, and reclaiming it when execution is completed. Each scheme we discussed in Chapters 2 and 3 was designed to address a different set of pressing problems; but, as we have seen, when some problems were solved, others were created. Table 3.7 shows how memory allocation schemes compare.

Scheme	Problem Solved	Problem Created	Changes in Software
Single-user contiguous		Job size limited to physical memory size; CPU often idle	None
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	None
Relocatable dynamic partitions	Internal fragmentation	Compaction overhead; Job size limited to physical memory size	Compaction algorithm
Paged	Need for compaction	Memory needed for tables; Job size limited to physical memory size; Internal fragmentation returns	Algorithms to handle Page Map Tables
Demand paged	Job size no longer limited to memory size; More efficient memory use; Allows large-scale multiprogramming and time-sharing	Larger number of tables; Possibility of thrashing; Overhead required by page interrupts; Necessary paging hardware	Page replacement algorithm; Search algorithm 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	Large virtual memory; Segment loaded on demand	Table handling overhead; Memory needed for page and segment tables	Three-dimensional addressing scheme

Exercise

1. Given that main memory is composed of three page frames for public use and that a seven-page program (with pages a, b, c, d, e, f, g) requests pages in the following order:

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

   a. Using the FIFO page removal algorithm, do a page trace analysis indicating page faults with asterisks (*). 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. Did the result correspond with your intuition? Explain.
2. Given that main memory is composed of three page frames for public use and that a program requests pages in the following order:

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

   a. Using the FIFO page removal algorithm, perform a page trace analysis indicating 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. Which is better?
   c. Why do you think it is better? Can you make general statements from this example? Why or why not?

CHAPTER 4

PROCESSOR MANAGEMENT

Objectives

- The difference between job scheduling and process scheduling, and how they relate
- The advantages and disadvantages of process scheduling algorithms that are preemptive versus non preemptive
- The goals of process scheduling policies in single-core CPUs
- Six different process scheduling algorithms
- The role of internal interrupts and the tasks performed by the interrupt handler

In a multiprogramming environment, the OS decides which process gets the processor when and for how much time. This function is called process scheduling

OVERVIEW

In a simple system, one with a single user and one processor, the process is busy only when it is executing the user's jobs. However, when there are many users, or when there are multiple processes to be run by a single CPU, the processor must be allocated to each job in a fair and efficient manner.

Before we begin, let's clearly define some terms.

Program is a group of instructions to carry out a specified task. In the operating system, a program or job is a unit of work that has been submitted by the user.

A **Process** is an active entity that requires a set of resources, including a processor and special registers, to perform its function. A process, also called a task, is a single instance of a program in execution.

Thread is a portion of a process that can run independently.

The **processor**, also known as the CPU (for central processing unit), is the part of the machine that performs the calculations and executes the programs.

Multiprogramming requires that the processor be allocated to each job or to each process for a period of time and deallocated at an appropriate moment.

About Multi-Core Technologies

A dual-core, quad-core, or other multi-core processor is a computer processor integrated circuit with two or more separate processing units, called cores, each of which reads and executes program instructions, as if the computer had several processors

Multi-core 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 increased current leakage and the amount of heat generated by the chip.

One solution was to replace a single large processor with two half-sized processors, or four quarter-sized processors. For the Processor Manager, multiple cores are more complex to manage than a single core.

Job Scheduling Versus Process Scheduling

The Processor Manager is a composite of two sub managers:

- Job Scheduler
- Process Scheduler.

The job scheduling is the mechanism to select which process has to be brought into the ready queue. The scheduling of jobs is actually handled on two levels by most operating systems.

The Process scheduling is the mechanism to select which process has to be executed next and allocates the CPU to that process.The job scheduling is known as the long-term scheduling while the CPU scheduling is known as the short-term scheduling

Process Scheduler

The Process Scheduler determines which jobs will get the CPU, when, and for how long. It also decides when processing should be interrupted, determines which queues the job should be moved to during its execution, and recognizes when a job should be terminated.

The Process Scheduler is the low-level scheduler that assigns the CPU to the processes of those jobs placed on the READY queue by the Job Scheduler.

To schedule the CPU, the Process Scheduler takes advantage of a common trait among most computer programs: they alternate between CPU cycles and I/O cycles.

For example, I/O-bound jobs have shorter CPU cycles and long I/O cycles, whereas CPU-bound jobs have long CPU cycles and shorter I/O cycles.

```
{
printf("\nEnter the first integer: ");
scanf("%d", &a);
printf("\nEnter the second integer: ");
scanf("%d", &b);                                  } I/O cycle

c = a+b
d = (a*b)-c
e = a-b                                           } CPU cycle
f = d/e

printf("\n a+b= %d", c);
printf("\n (a*b)-c = %d", d);
printf("\n a-b = %d", e);                          } I/O cycle
printf("\n d/e = %d", f);
}
```

The total effect of all CPU cycles, from both I/O-bound and CPU-bound jobs, approximates a Poisson distribution curve as shown in Figure 4.1.

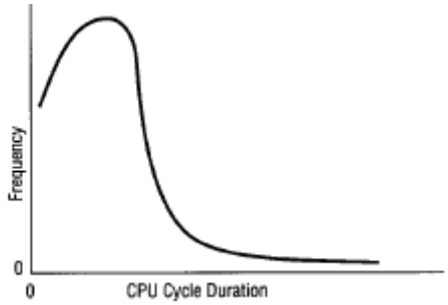


Figure 4.1 Distribution of CPU Cycle time

In some cases, especially when the system is overloaded, the middle-level scheduler removes active jobs from memory to reduce the degree of multiprogramming, which allows jobs to be completed faster.

The jobs that are swapped out and eventually swapped back in are managed by the middle-level scheduler. In a single-user environment, there's no distinction made

between job and process scheduling because only one job is active in the system at any given time. So the CPU and all other resources are dedicated to that job.

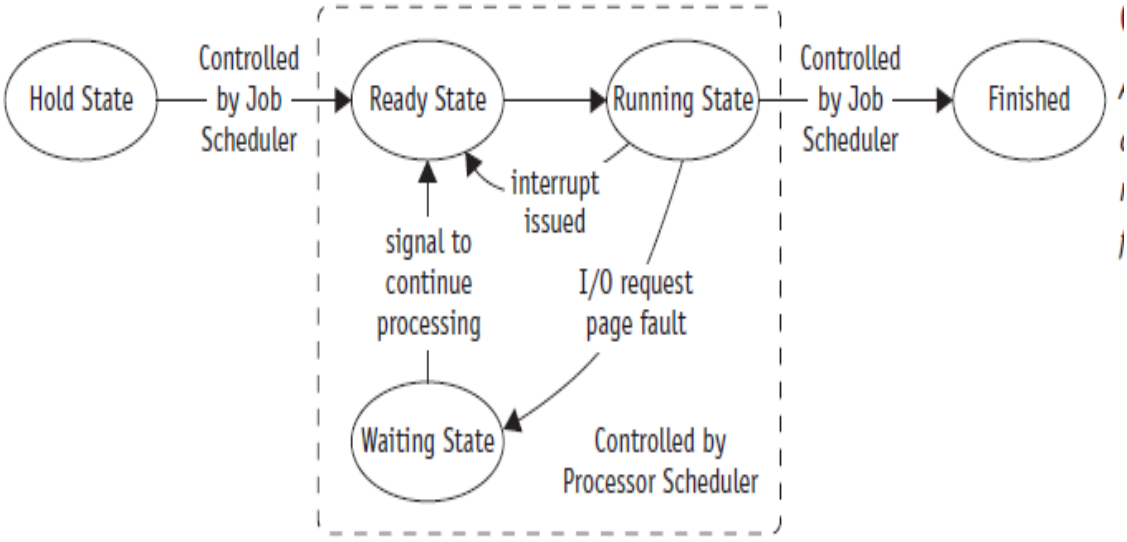


Figure 4.2 A typical job changes status as it moves through the system from HOLD to FINISHED

Job and Process Status

There are five states: HOLD, READY, RUNNING, WAITING and TO FINISH. These are called the job status or the process status.

When the job is accepted by the system, it's put on HOLD and placed in a queue. The job spooler (or disk controller) creates a table with the characteristics of each job in the queue like an estimate of CPU time, priority, special I/O devices required, and maximum memory required.

This table is used by the Job Scheduler to decide which job is to be run next from HOLD, the job moves to READY when it's ready to run but is waiting for the

CPU or in some system it is directly placed to READY. RUNNING means that the job is being processed.

WAITING means that the job can't continue until a specific resource is allocated or an I/O operation has finished upon completion, the job is FINISHED and returned to the user.

The transition from one job or process status to another is initiated by either the Job Scheduler or the Process Scheduler:

- The transition from HOLD to READY is initiated by the Job Scheduler
- The transition from READY to RUNNING, RUNNING back to READY, RUNNING to WAITING and WAITING to READY are handled by the Process Scheduler
- Eventually, the transition from RUNNING to FINISHED is initiated by the Process Scheduler or the Job Scheduler either when
    1. The job is successfully completed and it ends execution or
    2. The operating system indicates that an error has occurred and the job is being terminated prematurely.

Process Control Blocks

Each process in the system is represented by a data structure called a Process Control Block (PCB). The PCB contains the basic information about the job, including what it is, where it's going, how much of its processing has been completed, where it's stored, and how much it has spent in using resources.

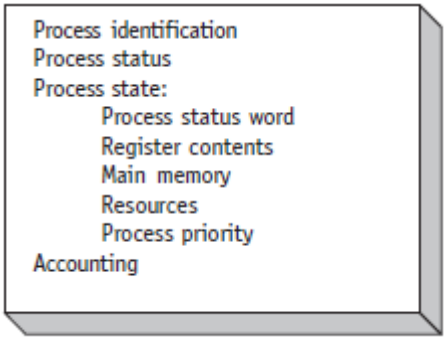


Figure 4.3 Contents of each job's Process Control Block

Process Identification

Each job is uniquely identified by the user's identification and a pointer connecting it to its descriptor

Process Status indicates the current status of the job—HOLD, READY, RUNNING, or WAITING —and the resources responsible for that status.

Process state contains all of the information needed to indicate the current state of the job such as:

- **Process Status Word -** The current instruction counter and register contents when the job isn't running but is either on HOLD or is READY or WAITING. If the job is RUNNING, this information is left undefined.
- **Register Contents**—The contents of the register if the job has been interrupted and is waiting to resume processing.
- **Main Memory—**proper information, including the address where the job is stored and, in the case of virtual memory, the mapping between virtual and physical memory locations.

Accounting

This contains information used mainly for billing purposes and performance measurement. It indicates what kind of resources the job used and for how long.

Typical charges include:

- Amount of CPU time used from beginning to end of its execution.
- Total time the job was in the system until it exited.
- Main storage occupancy—how long the job stayed in memory until it finished execution. This is usually a combination of time and space used
- Secondary storage used during execution.
- System programs used, such as compilers, editors, or utilities.
- Number and type of I/O operations, including I/O transmission time that includes utilization of channels, control units, and devices.
- Time spent waiting for I/O completion.
- Number of input records read and number of output records written.
- **Resources**—information about all resources allocated to this job. Each resource has an identification field listing its type and a field describing details of its allocation, such as the sector address on a disk.
- **Process Priority**—used by systems using a priority scheduling algorithm to select which job will be run next.

PCBs and Queuing

A job's PCB is created when the Job Scheduler accepts the job and is updated as the job progresses from the beginning to the end of its execution. Queues use PCBs to track jobs. The PCB contains all of the data about the job needed by the operating system to manage the processing of the job.

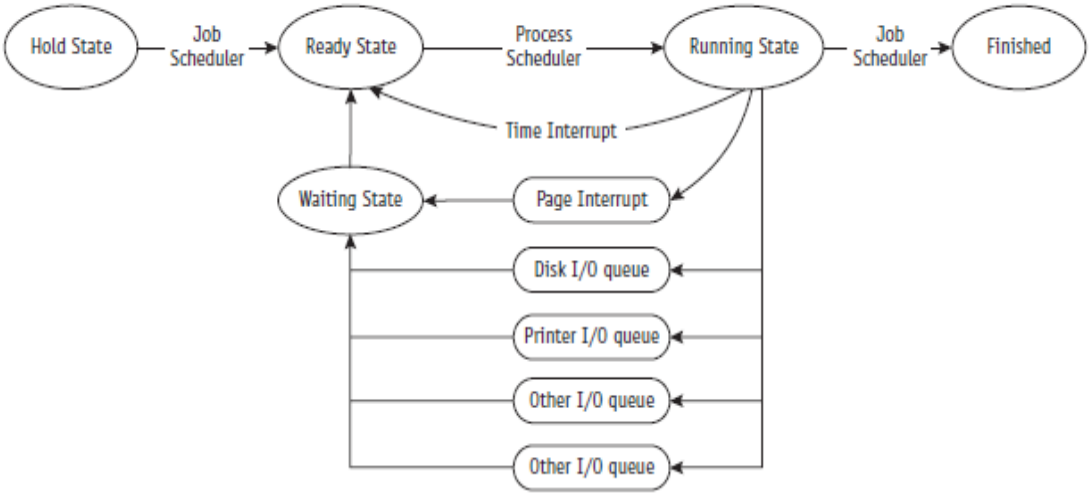


Figure 4.4 Queuing paths from HOLD to FINISHED. The Job and Processor Schedulers release the resources when the job leaves the RUNNING state

The PCBs for every ready job are linked on the READY queue, and all of the PCBs for jobs just entering the system are linked on the HOLD queue. The jobs that are WAITING, however, are linked together by "reason for waiting," so the PCBs for the jobs in this category are linked into several queues.

Process Scheduling Policies

In a multiprogramming environment, there are usually more jobs to be executed. Before the operating system can schedule them, it needs to resolve three limitations of the system:

1. There are a finite number of resources (such as disk drives, printers, and tape drives);
2. Some resources, once they're allocated, can't be shared with another job (e.g., printers);
3. Some resources require operator intervention.

Criteria come to decide a good scheduling policy:

- Maximize throughput. Run as many jobs as possible in a given amount of time.
- Minimize response time. Quickly turn around interactive requests.
- Minimize turnaround time. Move entire jobs in and out of the system quickly. This could be done by running all batch jobs first .
- Minimize waiting time. Move jobs out of the READY queue as quickly as possible.This could only be done by reducing the number of users allowed on the system so the CPU would be available immediately whenever a job entered the READY queue.
- Maximize CPU efficiency. Keep the CPU busy 100 percent of the time. This could be done by running only CPU-bound jobs (and not I/O-bound jobs).
- Ensure fairness for all jobs. Give everyone an equal amount of CPU and I/O time.

If the system favors one type of user then it hurts another or doesn't efficiently use its resources. The final decision rests with the system designer, who must determine which criteria are most important for that specific system

For example, "maximize CPU utilization while minimizing response time and balancing the use of all system components through a mix of I/O-bound and CPU-bound jobs." So select the scheduling policy that most closely satisfies your criteria.

Although the Job Scheduler selects jobs to ensure that the READY and I/O queues remain balanced, that job may claim the CPU for a very long time before issuing an I/O request. This extensive use of the CPU will increase the READY queue and empty out the I/O queues.

To solve this problem, the Process Scheduler often uses a timing mechanism and periodically interrupts running processes when a predetermined slice of time has expired.

When that happens, the scheduler suspends all activity on the job currently running and reschedules it into the READY queue; it will be continued later. The CPU is now allocated to another job that runs until one of three things happens:

1. The timer goes off
2. The job issues an I/O command
3. The job is finished.

Then the job moves to the READY queue, the WAIT queue, or the FINISHED queue, respectively. Preemptive scheduling is used when a process

switches from running state to ready state or from waiting state to ready state; it is widely used in time-sharing environments.

In case of non-preemptive scheduling does not interrupt a process running CPU in middle of the execution, it remains in the RUNNING state.

Process Scheduling Algorithms

The Process Scheduler relies on a process scheduling algorithm, based on a specific policy, to allocate the CPU and move jobs through the system. Early operating systems used non preemptive policies designed to move batch jobs through the system as efficiently as possible.

Most current systems, with their emphasis on interactive use and response time, use an algorithm that takes care of the immediate requests of interactive users.

Here are six process scheduling algorithms that have been used extensively.

First-Come, First-Served

First-come, first-served (FCFS) is a non preemptive scheduling algorithm that handles jobs according to their arrival time: the earlier they arrive, the sooner they're served. It's a very simple algorithm to implement because it uses a FIFO queue.

This algorithm is fine for most batch systems, but it is unacceptable for interactive systems because interactive users expect quick response times.

With FCFS, as a new job enters the system its PCB is linked to the end of the READY queue and it is removed from the front of the queue when the processor becomes available.

Strictly FCFS systems there are no WAIT queues, although there may be systems in which control (context) is switched on a natural wait (I/O request) and then the job resumes on I/O completion.

The following examples presume a strictly FCFS environment (no multiprogramming). Turnaround time is unpredictable with the FCFS policy; consider the following three jobs:

1. Job A has a CPU cycle of 15 milliseconds.
2. Job B has a CPU cycle of 2 milliseconds.
3. Job C has a CPU cycle of 1 millisecond.

For each job, the CPU cycle contains both the actual CPU usage and the I/O requests. That is, it is the total run time.

If all three jobs arrive almost simultaneously, we can calculate that the turnaround time for Job A is 15, for Job B is 17, and for Job C is 18. So the average turnaround time is:

$$(15 + 17 + 18)/3 = 16.67$$

However, if the jobs arrived in a different order, say C, B, A, then the results using the same FCFS algorithm would be Job A is 18, Job B is 3, and for Job C is 1

The average turnaround time is:(18 + 3 + 1)/3= 7.3.That's quite an improvement over the first sequence.

If one job controls the system, system performance depends on the scheduling policy and whether the job is CPU-bound or I/O-bound.

While a job with a long CPU cycle, the other jobs in the system are waiting for processing and joining the READY queue to wait. Then the READY list grew while the I/O queues would be empty, or stable, and the I/O devices would sit idle.

On the other hand, if the job is processing a lengthy I/O cycle, the I/O queues quickly overflow and the CPU could be sitting idle.

This situation is eventually resolved when the I/O-bound job finishes its I/O cycle, the queues start moving again, and the system can recover from the bottleneck.

FCFS is a less attractive algorithm because the turnaround time is variable than one that would serve the shortest job first, as the next scheduling algorithm.

Shortest Job Next

- This is also known as shortest job first, or SJF
- This is a non-preemptive, preemptive scheduling algorithm.
- Best approach to minimize waiting time.
- Easy to implement in Batch systems where required CPU time is known in advance.
- Impossible to implement in interactive systems where required CPU time is not known.
- The processor should know in advance how much time process will take

For example, here are four batch jobs, all in the READY queue, for which the CPU cycle, or run time, is estimated as follows:

Job:	A	B	C	D
CPU cycle:	5	2	6	4

The SJN algorithm would review the four jobs and schedule them for processing in this order: B, D, A, C. The timeline is shown in Figure.

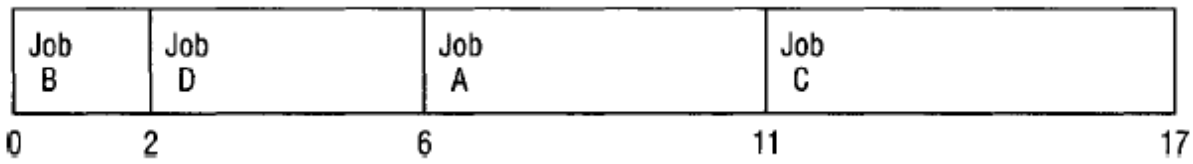


Figure 4.5 Timeline for job sequence B,D,A,C using SJN Algorithm

The average turnaround time is:

$$\frac{2 + 6 + 11 + 17}{4} = 9.0$$

Job B finishes in its given time (2)

Job D finishes in its given time plus the time it waited for B to run (4 + 2)

Job A finishes in its given time plus D's time plus B's time (5 + 4 + 2)

Job C finishes in its given time plus that of the previous three (6 + 5 + 4 + 2).

So the average time:

$$\frac{(2) + (4 + 2) + (5 + 4 + 2) + (6 + 5 + 4 + 2)}{4} = 9.0$$

Because the time for the first job appears in the equation four times, it has four times the effect on the average time than does the length of the fourth job, which appears only once.

Therefore, if the first job requires the shortest computation time, like other jobs ordered from shortest to longest, then the result will be the smallest possible average.

The formula for the average is as follows $t1(n) + t2(n - 1) + t3(n - 2) + \ldots + tn(n(1))$ n where n is the number of jobs in the queue and $tj(j = 1, 2, 3,\ldots,n)$ is the length of the CPU cycle for each of the jobs.

However, the SJN algorithm is optimal only when all of the jobs are available at the same time and the CPU estimates are available and accurate.

Priority Scheduling

- Priority scheduling is a non-preemptive algorithm and one of the most common scheduling algorithms in batch systems.
- Each process is assigned a priority. Process with highest priority is to be executed first and so on.
- Processes with same priority are executed on first come first served basis.
- Priority can be decided based on memory requirements, time requirements or any other resource requirement.

Priorities can also be determined by the Processor Manager based on characteristics intrinsic to the jobs such as:

- **Memory requirements :** Jobs requiring large amounts of memory could be allocated lower priorities than those requesting small amounts of memory, or vice versa.

- **Number and type of peripheral devices**: Jobs requiring many peripheral devices would be allocated lower priorities than those requesting fewer devices.
- **Total CPU time**: Jobs having a long CPU cycle, or estimated run time, would be given lower priorities than those having a brief estimated run time.
- **Amount of time already spent in the system** : This is the total amount of elapsed time since the job was accepted for processing.

There are also preemptive priority schemes.

Shortest Remaining Time

- Shortest remaining time (SRT) is the preemptive version of the SJN algorithm.
- The processor is allocated to the job closest to completion but it can be preempted by a newer ready job with shorter time to completion.
- Impossible to implement in interactive systems where required CPU time is not known.
- It is often used in batch environments where short jobs need to give preference

It is often used in batch environments, when it is desirable to give preference to short jobs. Operating system has to frequently monitor the CPU time for all the jobs in the READY queue and must perform context switching for the jobs being swapped (switched) at preemption time.

The example in Figure 4.6 shows how the SRT algorithm works with four jobs that arrived in quick succession (one CPU cycle apart).

Arrival time:	0	1	2	3
Job:	A	B	C	D
CPU cycle:	6	3	1	4

In this case, the turnaround time is the completion time of each job minus its arrival time:

Job:	A	B	C	D
Turnaround:	14	4	1	6

So the average turnaround time is:

$$\frac{14 + 4 + 1 + 6}{4} = 6.25$$

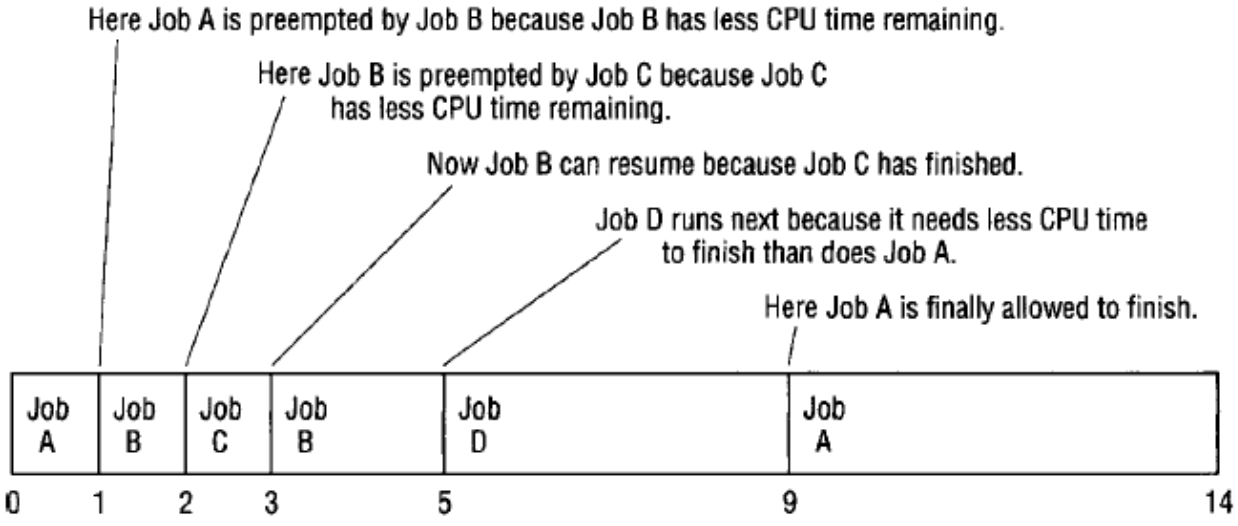


Figure 4.6 Timeline for Job sequence A,B,C,D using preemptive SRT.

SRT at 6.25 is faster than SJN where the average turnaround time is 7.75 for the same example. However, we neglected to include the time required by the SRT algorithm to do the context switching.

Context switching is required by all preemptive algorithms. When Job A is preempted, all of its processing information must be saved in its PCB for later, when Job A's execution is to be continued, and the contents of Job A's PCB are loaded into the appropriate registers so it can start running again; this is a context switch.

So although SRT appears to be faster, in a real operating environment its advantages are diminished by the time spent in context switching. A precise comparison of SRT and SJN would have to include the time required to do the context switching.

Round Robin

- Round Robin is the preemptive process scheduling algorithm.
- Each process is provided a fix time to execute, it is called a quantum.
- Once a process is executed for a given time period, it is preempted and other process executes for a given time period.
- Context switching is used to save states of preempted processes

Scheduling process:

- Jobs are placed in the READY queue using a first-come, first-served scheme
- The Process Scheduler selects the first job from the front of the queue
- Sets the timer to the time quantum, and allocates the CPU to this job.
- If processing isn't finished when time expires, the job is preempted

- Put at the end of the READY queue and its information is saved in its PCB.

In the event that the job's CPU cycle is shorter than the time quantum, one of two actions will take place:

1. If this is the job's last CPU cycle and the job is finished, then all resources allocated to it are released and the completed job is returned to the user;
2. If the CPU cycle has been interrupted by an I/O request, then information about the job is saved in its PCB and it is linked at the end of the appropriate I/O queue.

Later, when the I/O request has been satisfied, it is returned to the end of the READY queue to await allocation of the CPU.

Arrival time:	0	1	2	3
Job:	A	B	C	D
CPU cycle:	8	4	9	5

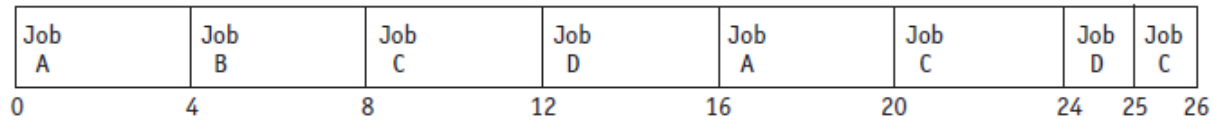


Figure 4.7 Timeline for job sequence A, B, C, D using the preemptive round robin algorithm with time slices of 4 ms.

The turnaround time is the completion time minus the arrival time:

Job:	A	B	C	D
Turnaround:	20	7	24	22

So the average turnaround time is:

$$\frac{20 + 7 + 24 + 22}{4} = 18.25$$

Job A was preempted once because it needed 8 milliseconds to complete its CPU cycle, Job B terminated in one time quantum. Job C was preempted twice because it needed 9 milliseconds to complete its CPU cycle Job D was preempted once because it needed 5 milliseconds.

In their last execution or swap into memory, both Jobs D and C used the CPU for only 1 millisecond and terminated before their last time quantum expired, releasing the CPU sooner.

The efficiency of round robin depends on the size of the time quantum in relation to the average CPU cycle. If the quantum is too large then the algorithm reduces to the FCFS scheme.

If the quantum is too small, then the amount of context switching slows down the execution of the jobs and the amount of overhead is dramatically increased, The amount of context switching increases as the time quantum decreases in size.

The time quantum size depends on the system.

- If it's an interactive environment, the system is expected to respond quickly to its users, especially when they make simple requests.
- If it's a batch system, response time is not a factor (turnaround is) and overhead becomes very important.

Here are two general rules of thumb for selecting the proper time quantum:

1. It should be long enough to allow 80 percent of the CPU cycles to run to completion,
2. It should be at least 100 times longer than the time required to perform one context switch. .

Multiple-Level Queues

Multiple-level queues are not an independent scheduling algorithm. They make use of other existing algorithms to group and schedule jobs with common characteristics.

- Multiple queues are maintained for processes with common characteristics.
- Each queue can have its own scheduling algorithms.
- Priorities are assigned to each queue.

For example, CPU-bound jobs can be scheduled in one queue and all I/O-bound jobs in another queue. The Process Scheduler then alternately selects jobs from each queue and assigns them to the CPU based on the algorithm assigned to the queue.

The scheduling policy is based on some predetermined scheme that allocates special treatment to the jobs in each queue. Within each queue, the jobs are served in FCFS fashion.

Multiple-level queues raise some interesting questions:

- Is the processor allocated to the jobs in the first queue until it is empty before moving to the next queue, or does it travel from queue to queue until the last job on the last queue has been served and then go back to serve the first job on the first queue, or something in between?

- Is this fair to those who have earned, or paid for, a higher priority?
- Is it fair to those in a low-priority queue?
- If the processor is allocated to the jobs on the first queue and it never empties out, when will the jobs in the last queues be served?
- Can the jobs in the last queues get "time off for good behavior" and eventually move to better queues?

The answers depend on the policy used by the system to service the queues. There are four primary methods to the movement:

- No Movement Between Queue
- Movement Between Queues
- Variable Time Quantum Per Queue
- Aging

not allowing movement between queues, moving jobs from queue to queue, moving jobs from queue to queue and increasing the time quantum's for lower queues, and giving special treatment to jobs that have been in the system for a long time (aging).

Case 1: No Movement Between Queues

No movement between queues is a very simple policy that rewards those who have high-priority jobs. The processor is allocated to the jobs in the high-priority queue in FCFS fashion and jobs in low-priority queues allowed only when the high priority queues are empty.

Case 2: Movement Between Queues

High-priority jobs are treated like all the others once they are in the system. When a time quantum interrupt occurs, the job is preempted and moved to the end of the next lower queue. A job may also have its priority increased;

This policy is fairest in a system in which the jobs are handled according to their computing cycle characteristics: CPU-bound or I/O-bound.

Case 3: Variable Time Quantum Per Queue

Variable time quantum per queue is a variation of the movement between queue policy, and it allows for faster turnaround of CPU-bound jobs.

In this scheme, each of the queues is given a time quantum twice as long as the previous queue. The highest queue might have a time quantum of 100 milliseconds. So the second-highest queue would have a time quantum of 200 milliseconds, the third would have 400 milliseconds, and so on. If there are enough queues, the lowest one might have a relatively long time quantum of 3 seconds or more.

If a job doesn't finish its CPU cycle in the first time quantum, it is moved to the end of the next lower-level queue.

Case 4: Aging

Aging is used to ensure that jobs in the lower-level queues will eventually complete their execution. The operating system keeps track of each job's waiting time and when a job gets too old—the system moves the job to the next highest queue, and so on until it reaches the top queue.

As you might expect, indefinite postponement means that a job's execution is delayed for an undefined amount of time because it is repeatedly preempted so other jobs can be processed. Eventually the situation could lead to the old job's starvation.

Indefinite postponement is a major problem when allocating resources .

A Word About Interrupts

The Memory Manager issued page interrupts to accommodate job requests. This chapter introduces another type of interrupt that occurs when the time quantum expires and the processor is deallocated from the running job and allocated to another one.

There are other interrupts that are caused by events internal to the process. I/O interrupts are issued when a READ or WRITE command is issued.

Internal interrupts, or synchronous interrupts, also occur as a direct result of the arithmetic operation or job instruction currently being processed.

Illegal arithmetic operations, such as the following, can generate interrupts:

- Attempts to divide by zero
- Floating-point operations generating an overflow or underflow
- Fixed-point addition or subtraction that causes an arithmetic overflow

Illegal job instructions, such as the following, can also generate interrupts:

- Attempts to access protected or nonexistent storage locations
- Attempts to use an undefined operation code
- Operating on invalid data

- Attempts to make system changes, such as trying to change the size of the time quantum

The control program that handles the interruption sequence of events is called the interrupt handler. When the operating system detects a non-recoverable error, the interrupt handler typically follows this sequence:

1. The type of interrupt is described and stored.
2. The state of the interrupted process is saved, including the value of the program counter, the mode specification, and the contents of all registers.
3. The interrupt is processed: The error message and state of the interrupted process are sent to the user; program execution is halted; any resources allocated to the job are released; and the job exits the system.
4. The processor resumes normal operation.

If we're dealing with internal interrupts only, which are non-recoverable, the job is terminated in Step 3. However, when the interrupt handler is working with an I/O interrupt, time quantum, or other recoverable interrupt, Step 3 simply halts the job and moves it to the appropriate I/O device queue, or READY queue (on time out).

Later, when the I/O request is finished, the job is returned to the READY queue. If it was a time out (quantum interrupt), the job (or process) is already on the READY queue.

Conclusion

The Processor Manager must allocate the CPU among all the system's users. In this chapter we've made the distinction between job scheduling, the selection of incoming jobs based on their characteristics, and process scheduling, the instant-by-instant allocation of the CPU. We've also described how interrupts are generated and resolved by the interrupt handler.

Each scheduling algorithm presented in this chapter has unique characteristics, objectives, and applications. A system designer can choose the best policy and algorithm only after carefully evaluating their strengths and weaknesses. Table 4.1 shows how the algorithms presented in this chapter compare.

Algorithm	Policy Type	Best for	Disadvantages	Advantages
FCFS	Nonpreemptive	Batch	Unpredictable turnaround times	Easy to implement
SJN	Nonpreemptive	Batch	Indefinite postponement of some jobs	Minimizes average waiting time
Priority scheduling	Nonpreemptive	Batch	Indefinite postponement of some jobs	Ensures fast completion of important jobs
SRT	Preemptive	Batch	Overhead incurred by context switching	Ensures fast completion of short jobs
Round robin	Preemptive	Interactive	Requires selection of good time quantum	Provides reasonable response times to interactive users; provides fair CPU allocation
Multiple-level queues	Preemptive/ Nonpreemptive	Batch/ interactive	Overhead incurred by monitoring of queues	Flexible scheme; counteracts indefinite postponement with aging or other queue movement; gives fair treatment to CPU-bound jobs by incrementing time quantums on lower-priority queues or other queue movement

(table 4.1)

Comparison of the scheduling algorithms discussed in this chapter.

Exercise

1. Given the following information

Job	Arrival Time	CPU Cycle
A	0	10
B	2	12
C	3	3
D	6	1
E	9	15

   Draw a timeline for each of the following scheduling algorithms. (It may be helpful to first compute a start and finish time for each job.)

   a. FCFS
   b. SJN
   c. SRT

   Round robin (using a time quantum of 5, ignore context switching and natural wait)

2. Using the same information from Exercise 1, calculate which jobs will have arrived ready for processing by the time the first job is finished or interrupted using each of the following scheduling algorithms.

   a. FCFS
   b. SJN
   c. SRT
   d. Round robin (using a time quantum of 5, ignore context switching and natural wait)

3. Using the same information given for Exercise 1, compute the waiting time and turnaround time for every job for each of the following scheduling algorithms (ignoring context switching overhead).

   a. FCFS
   b. SJN
   c. SRT d. Round robin (using a time quantum of 2)

CHAPTER 5

Process Management

Objectives

- Several causes of system deadlock and live lock
- The difference between preventing and avoiding deadlocks
- How to detect and recover from deadlocks
- The concept of process starvation and how to detect and recover from it
- The concept of a race and how to prevent it
- The difference between deadlock, starvation, and race

Introduction to Deadlock

Every process needs some resources to complete its execution. However, the resource is granted in a sequential order.

1. The process requests for some resource.
2. OS grant the resource if it is available otherwise let the process waits.
3. The process uses it and release on the completion.

A Deadlock is a situation where each of the computer process waits for a resource which is being assigned to some another process. In this situation, none of the process gets executed since the resource it needs, is held by some other process which is also waiting for some other resource to be released.

As shown in Figure 5.1, there's no simple and immediate solution to a deadlock; no one can move forward until someone moves out of the way, but no one can move out of the way until either someone advances or the rear of a line

moves back. Obviously it requires outside intervention to remove one of the four vehicles from an intersection or to make a line move back. Only then can the deadlock be resolved.

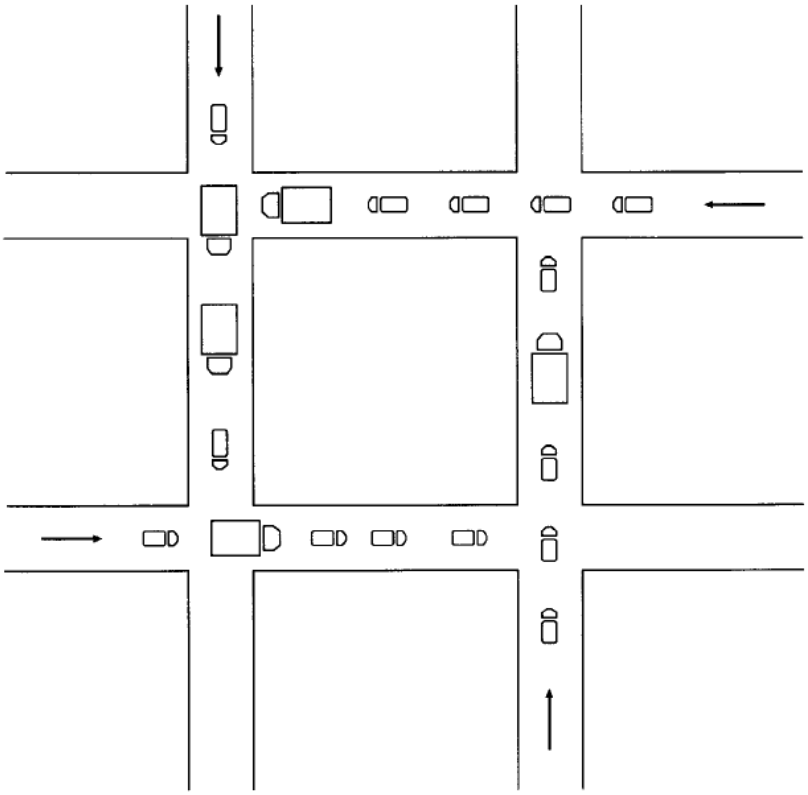


Figure 5.1 *A classic case of traffic deadlock on four one-way streets. This is "gridlock," where no vehicles can move forward to clear the traffic jam.*

Seven Cases of Deadlock

A deadlock usually occurs when non-sharable, non-preemptable resources. They can also occur on sharable resources that are locked, such as disks and databases. Directed graphs visually represent the system's resources and processes, and show how they are deadlocked. Using a series of squares (for resources) and circles (for processes), and connectors with arrows (for requests), directed graphs can be manipulated to understand how deadlocks occur.

Case 1: Deadlocks on File Requests

If jobs are allowed to request and hold files for the duration of their execution, a Deadlock can occur as the simplified directed graph shown in Figure 5.2 graphically illustrates.

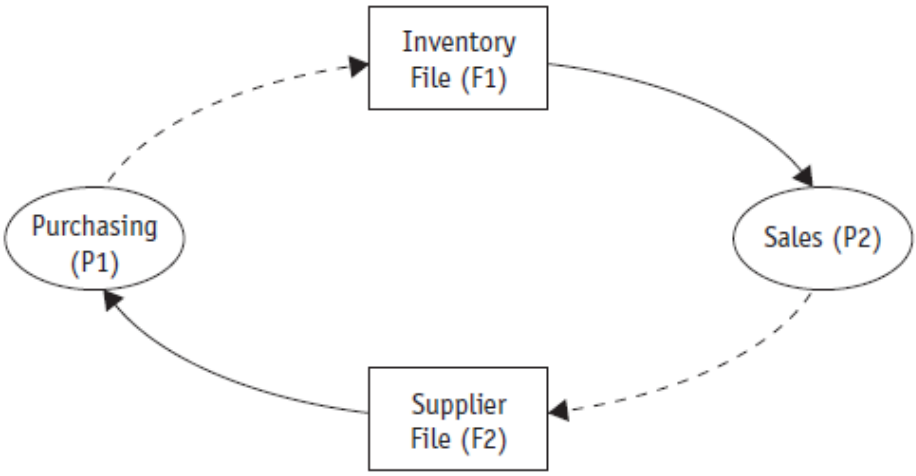


Figure 5.2 : *Case 1. These two processes, shown as circles, are each waiting for a resource, shown as rectangles, that has already been allocated to the other process, thus creating a deadlock.*

For example, consider the company with two application programs, purchasing (P1) and sales (P2), which are active at the same time. Both need to access two files, inventory (F1) and suppliers (F2), to read and write transactions. One day the system deadlocks when the following sequence of events takes place:

1. Purchasing (P1) accesses the supplier file (F2) to place an order for more lumber.
2. Sales (P2) accesses the inventory file (F1) to reserve the parts that will be required to build the home ordered that day.
3. Purchasing (P1) doesn't release the supplier file (F2) but requests the inventory file (F1) to verify the quantity of lumber on hand before placing its order for more, but P1 is blocked because F1 is being held by P2.
4. Meanwhile, sales (P2) doesn't release the inventory file (F1) but requests thesupplier file (F2) to check the schedule of a subcontractor. At this point, P2 is also blocked because F2 is being held by P1.

Any other programs that require F1 or F2 will be put on hold as long as this situation continues. This deadlock will remain until one of the two programs is closed or forcibly removed and its file is released. Only then can the other program continue and the system returns to normal.

Case 2: Deadlocks in Databases

A deadlock can also occur if two processes access and lock records in a database. **Locking** is a technique used to guarantee the integrity of the data through

which the user locks out all other users while working with the database. Locking can be done at three different levels:

- The entire database can be locked for the duration of the request
- A subsection of the database can be locked or
- Only the individual record can be locked until the process is completed.

Locking the entire database prevents a deadlock from occurring but it restricts access to the database to one user at a time and, in a multiuser environment, response times are significantly slowed; this is normally an unacceptable solution.

When the locking is performed on only one part of the database, access time is improved but the possibility of a deadlock is increased because different processes sometimes need to work with several parts of the database at the same time.

Here's a system that locks each record when it is accessed until the process is Completed. There are two processes (P1 and P2), each of which needs to update two records (R1 and R2), and the following sequence leads to a deadlock:

1. P1 accesses R1 and locks it.
2. P2 accesses R2 and locks it.
3. P1 requests R2, which is locked by P2.
4. P2 requests R1, which is locked by P1.

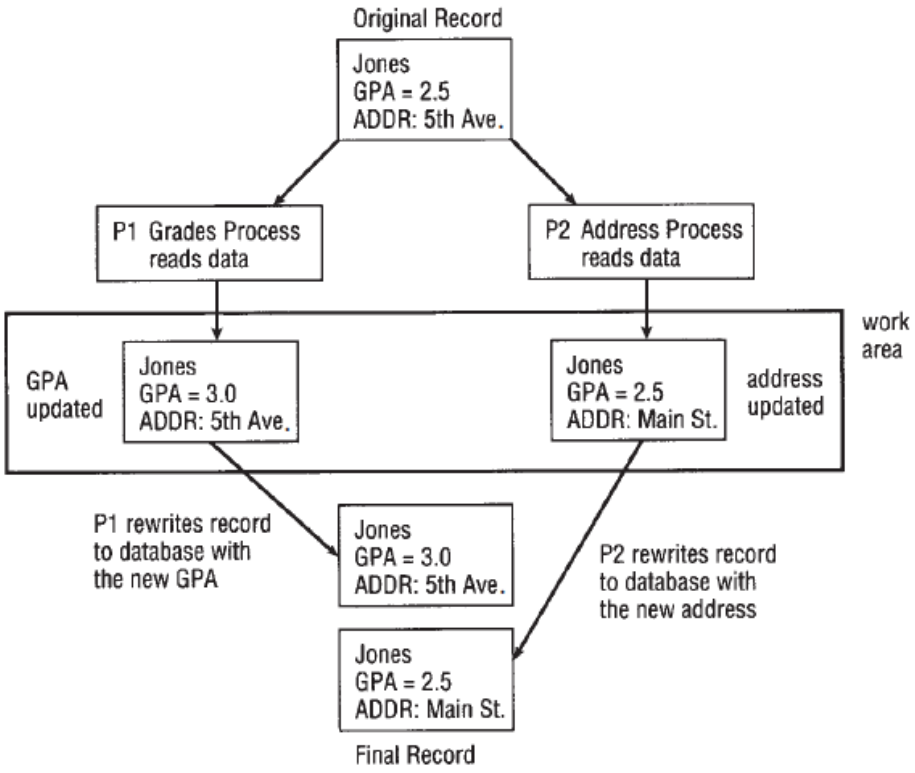


Figure 5.3 *Case 2. P1 finishes first and wins the race but its version of the record will soon be overwritten by P2.Regardless of which process wins the race, the final version of the data will be incorrect.*

Let's say you are a student of a university that maintains most of its files on a database that can be accessed by several different programs, including one for grades and another listing home addresses. You've just moved so you send the university a change of address form at the end of the fall term, shortly after grades are submitted. And one fateful day, both programs race to access your record in the database:

1. The grades process (P1) is the first to access your record (R1), and it copies the record to its work area.

2. The address process (P2) accesses your record (R1) and copies it to its work area P1 changes your student record (R1) by entering your grades for the fall term and calculating your new grade average.
3. P2 changes your record (R1) by updating the address field.
4. P1 finishes its work first and rewrites its version of your record back to the database. Your grades have been updated, but your address hasn't.
5. P2 finishes and rewrites its updated record back to the database. Your address has been changed, but your grades haven't. According to the database, you didn't attend school this term.

If we reverse the order and say that P2 won the race, your grades will be updated but not your address. Depending on your success in the classroom, you might prefer one mishap over the other; but from the operating system's point of view, either alternative is unacceptable because incorrect data is allowed to corrupt the database.

The system can't allow the integrity of the database to depend on a random sequence of events.

Case 3: Deadlocks in Dedicated Device Allocation

The use of a group of dedicated devices, such as a cluster of DVD read/write drives, can also deadlock the system.

Let's say two users from the local board of education are each running a program

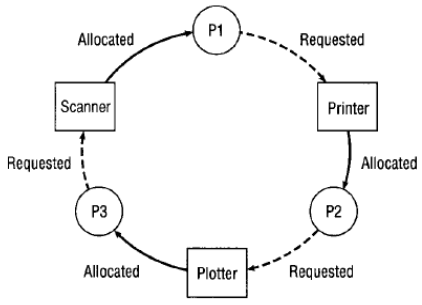
(P1 and P2), and both programs will eventually need two DVD drivers to copy files from one disc to another. The system is small, however, and when the two programs are begun, only two DVD-R drives are available and they're allocated on an "as requested" basis. Soon the following sequence transpires:

1. P1 requests drive 1 and gets it.
2. P2 requests drive 2 and gets it.
3. P1 requests drive 2 but is blocked.
4. P2 requests drive 1 but is blocked.

Neither job can continue, because each is waiting for the other to finish and release its drive—an event that will never occur. A similar series of events could deadlock any group of dedicated devices.

Case 4: Deadlocks in Multiple Device Allocation

Deadlocks aren't restricted to processes contending for the same type of device; they can happen when several processes request, and hold on to, several dedicated devices while other processes act in a similar manner as shown in Figure 5.4.



Consider the case with three programs (P1, P2, and P3) and three dedicated devices: scanner, printer, and plotter. The following sequence of events will result in deadlock:

1. P1 requests and gets the scanner.
2. P2 requests and gets the printer.
3. P3 requests and gets the plotter.
4. P1 requests the printer but is blocked.
5. P2 requests the plotter but is blocked.
6. P3 requests the scanner but is blocked.

As in the earlier examples, none of the jobs can continue because each is waiting for a resource being held by another.

Case 5: Deadlocks in Spooling

- Most systems have transformed dedicated devices such as a printer into a sharable device by installing a high-speed device, a disk, between it and the CPU.
- Disk accepts output from several users and acts as a temporary storage area for all output until printer is ready to accept it (spooling).
- If printer needs all of a job's output before it will begin printing, but spooling system fills available disk space with only partially completed output, then a deadlock can occur.

Case 6: Deadlocks in a Network

A network that's congested or has filled a large percentage of its I/O buffer space can become deadlocked if it doesn't have protocols to control the flow of messages through the network as shown in Figure 5.5.

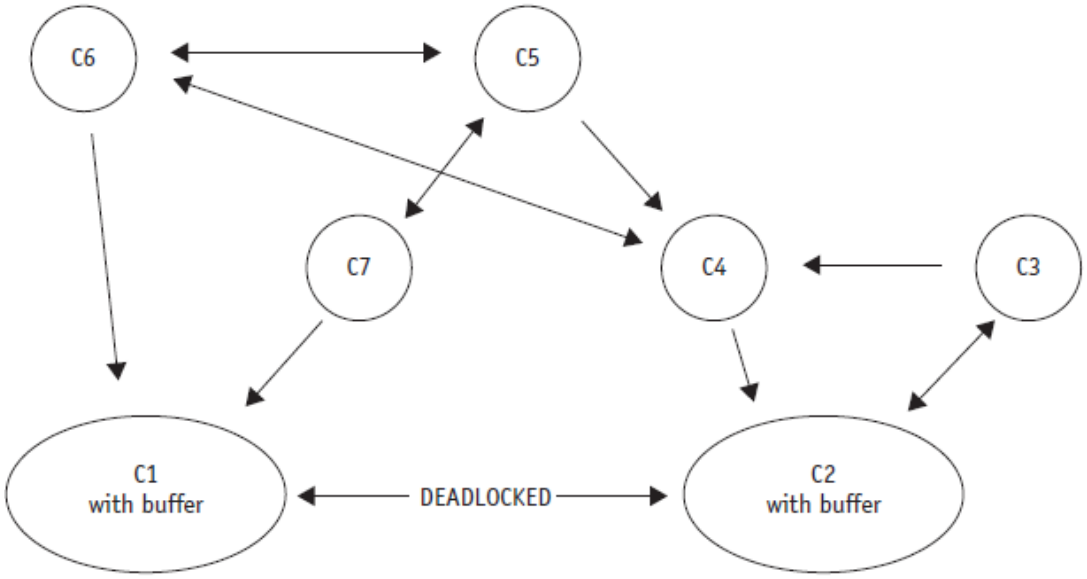


Figure 5.5 : *Case 6, deadlocked network flow. Notice that only two nodes, C1 and C2, have buffers. Each circle represents a node and each line represents a communication path. The arrows indicate the direction of data flow.*

For example, a medium-sized word-processing center has seven computers on a network, each on different nodes. The direction of the arrows in Figure 5.5 indicates the flow of messages.

Messages received by C1 from C6 and C7 and destined for C2 are buffered in an output queue. Messages received by C2 from C3 and C4 and destined for C1

are buffered in an output queue. As the traffic increases, the length of each output queue increases until all of the available buffer space is filled. At this point C1 can't accept any more messages (from C2 or any other computer) because there's no more buffer space available to store them. For the same reason, C2 can't accept any messages from C1 or any other computer, not even a request to send. The communication path between C1 and C2 becomes deadlocked; and because C1 can't send messages to any other computer except C2 and can only receive messages from C6 and C7, those routes also become deadlocked. C1 can't send word to C2 about the problem and so the deadlock can't be resolved without outside intervention.

Case 6: Deadlocks in Disk Sharing

- Disks are designed to be shared, so it's not uncommon for 2 processes access different areas of same disk.
- Without controls to regulate use of disk drive, competing processes could send conflicting commands and deadlock the system.

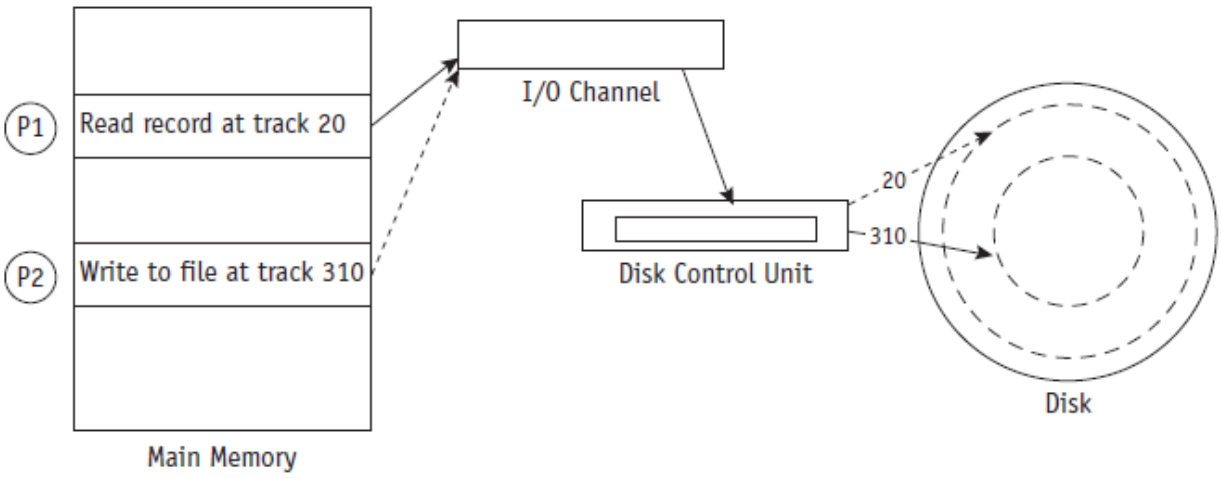


Figure 5.6 *Case 7. Two processes are each waiting for an I/O request to be filled: one at track 20 and one at track 310. But by the time the read/write arm reaches one track, a competing command for the other track has been issued, so neither command is satisfied and livelock occurs.*

For example, at an insurance company the system performs many daily transactions. One day the following series of events ties up the system:

1. Customer Service (P1) wishes to show a payment so it issues a command to read the balance, which is stored on track 20 of a disk.
2. While the control unit is moving the arm to track 20, P1 is put on hold and the I/O channel is free to process the next I/O request.
3. While the arm is moving into position, Accounts Payable (P2) gains control of the I/O channel and issues a command to write someone else's payment to a record stored on track 310. If the command is not "locked out," P2 will be put on hold while the control unit moves the arm to track 310.
4. Because P2 is "on hold" while the arm is moving, the channel can be captured again by P1, which reconfirms its command to "read from track 20."
5. Because the last command from P2 had forced the arm mechanism to track 310, the disk control unit begins to reposition the arm to track 20 to satisfy P1. The I/O channel would be released because P1 is once again put on hold, so it could be captured by P2, which issues a WRITE command only to discover that the arm mechanism needs to be repositioned.

As a result, the arm is in a constant state of motion, moving back and forth between tracks 20 and 310 as it responds to the two competing commands, but satisfies neither.

Conditions for Deadlock

In each of these seven cases, the deadlock involved the interaction of several processes and resources, but each deadlock was preceded by the simultaneous occurrence of four conditions that the operating system (or other systems) could have recognized:

- Mutual exclusion
- Resource holding,
- No preemption
- circular wait
-

To remove deadlock from our system, we need to avoid any one of the above four conditions of deadlock.

Mutual exclusion: A resource can only be held by one process at a time. If another process is also demanding the same resource then it has to wait for the allocation of that resource. For example, a printer can't print documents of two users at the same time.

Hold and Wait: Hold and wait arises when a process holds some resources and is waiting for some other resources that are being held by some other waiting process.

No Preemption: This is a technique in which a process can't forcefully take the resource of other processes. But if we found some resource due to which, deadlock

is happening in the system, then we can forcefully preempt that resource from the process that is holding that resource.

In fact, if the four conditions can be prevented from ever occurring at the same time, deadlocks can be prevented. Although this concept is obvious, it isn't easy to implement.

Modeling Deadlocks

Holt showed how the four conditions can be modeled using directed graphs. These graphs use two kinds of symbols: processes represented by circles and resources represented by squares. A solid arrow from a resource to a process, shown in Figure 5.7(a), means that the process is holding that resource. A dashed line with an arrow from a process to a resource, shown in Figure 5.7(b), means that the process is waiting for that resource. The direction of the arrow indicates the flow. If there's a cycle in the graph then there's a deadlock involving the processes and the resources in the cycle.

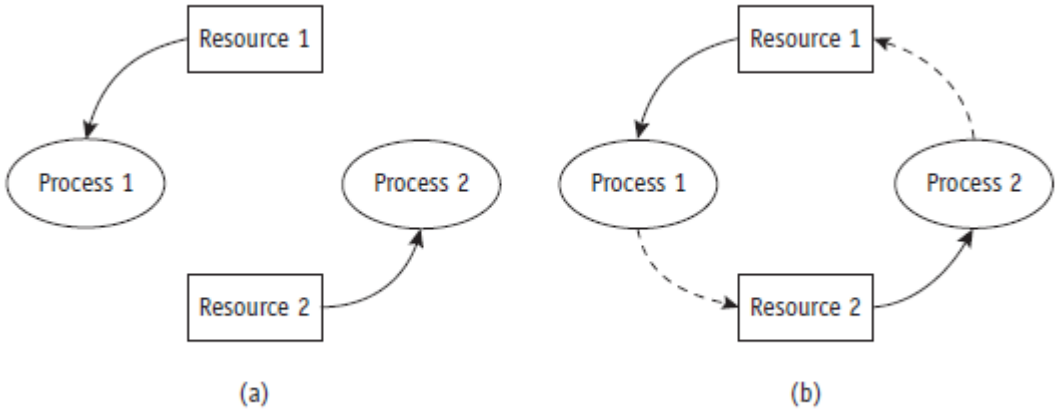


Figure 5.7 In (a), Resource 1 is being held by Process 1 and Resource 2 is held by Process 2 in a system that is not deadlocked. In (b), Process 1 requests Resource 2 but doesn't release Resource 1, and Process 2 does the same — creating a deadlock. (If one process released its resource, the deadlock would be resolved.)

The following system has three processes—P1, P2, P3—and three resources—R1, R2, R3—each of a different type: printer, disk drive, and plotter. Because there is no specified order in which the requests are handled, we'll look at three different possible scenarios using graphs to help us detect any deadlocks.

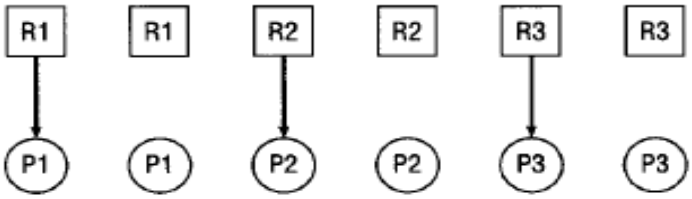
Scenario 1

The first scenario's sequence of events is shown in Table 5.1. The directed graph is shown in Figure 5.8.

Table 5.1 First scenario's sequence of events is shown in the directed graph in Figure 5.8.

Event	Action
1	P1 requests and is allocated the printer R1.
2	P1 releases the printer R1.
3	P2 requests and is allocated the disk drive R2.
4	P2 releases the disk R2.
5	P3 requests and is allocated the plotter R3.
6	P3 releases the plotter R3.

Table 5.1 First scenario's sequence of events is shown in the directed graph in Figure 5.8. Notice in the directed graph that there are no cycles. Therefore, we can safely conclude that a deadlock can't occur even if each process requests every resource if the resources are released before the next process requests them.



Scenario 2

Now, consider a second scenario's sequence of events shown in Table 5.2.

Table 5.2 The second scenario's sequence of events is shown in the two directed graphs shown in Figure 5.9.

Event	Action
1	P1 requests and is allocated R1.
2	P2 requests and is allocated R2.
3	P3 requests and is allocated R3.
4	P1 requests R2.
5	P2 requests R3.
6	P3 requests R1.

The progression of the directed graph is shown in Figure 5.9. A deadlock occurs because every process is waiting for a resource that is being held by another process, but none will be released without intervention.

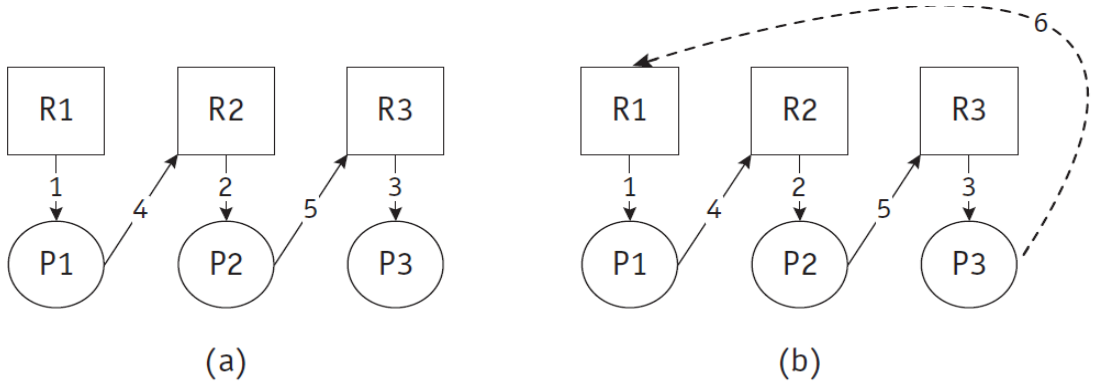


Figure 5.9 Second scenario. The system (a) becomes deadlocked (b) when P3 requests R1. Notice the circular wait.

Scenario 3

The third scenario is shown in Table 5.3. As shown in Figure 5.10, the resources are released before deadlock can occur.

Table 5.3 The third scenario's sequence of events is shown in the directed graph in Figure 5.10.

Event	Action
1	P1 requests and is allocated R1.
2	P1 requests and is allocated R2.
3	P2 requests R1.
4	P3 requests and is allocated R3.
5	P1 releases R1, which is allocated to P2.
6	P3 requests R2.
7	P1 releases R2, which is allocated to P3.

Table 5.3 The third scenario's sequence of events is shown in the directed graph in Figure 5.

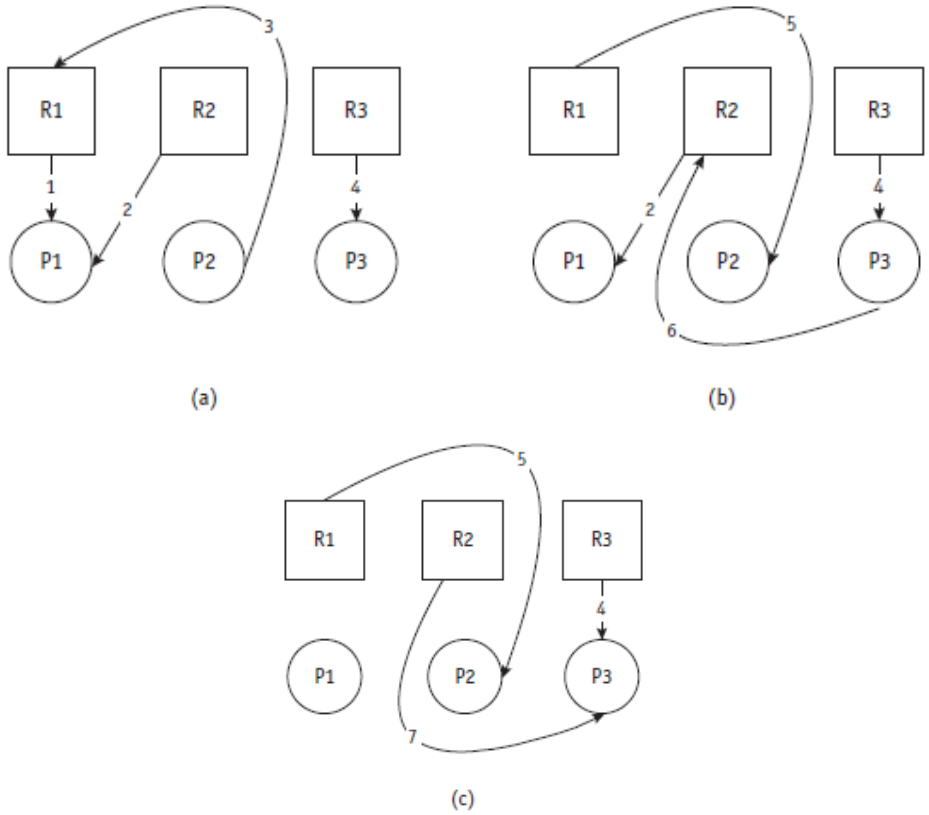


Figure 5.10 The third scenario. After event 4, the directed graph looks like (a) and P2 is blocked because P1 is holding on to R1. However, event 5 breaks the deadlock and the graph soon looks like (b).Again there is a blocked process, P3, which must wait for the release of R2 in event 7 when the graph looks like (c).

Strategies for Handling Deadlocks

In general, operating systems use one of three strategies to deal with deadlocks:

- Prevent one of the four conditions from occurring (prevention).
- Avoid the deadlock if it becomes probable (avoidance).
- Detect the deadlock when it occurs and recover from it gracefully (detection).

Prevention

To prevent a deadlock, the operating system must eliminate one of the four necessary conditions, a task complicated by the fact that the same condition can't be eliminated from every resource.

Mutual exclusion is necessary in any computer system because some resources such as memory, CPU, and dedicated devices must be exclusively allocated to one user at a time.

Resource holding, where a job holds on to one resource while waiting for another one that's not yet available, could be sidestepped by forcing each job to request, at creation time, every resource it will need to run to completion.

No preemption could be bypassed by allowing the operating system to deallocate resources from jobs.

Circular wait can be bypassed if the operating system prevents the formation of a circle.

This scheme of "hierarchical ordering" removes the possibility of a circular wait and therefore guarantees the removal of deadlocks.

Avoidance

Even if the operating system can't remove one of the conditions for deadlock, it can avoid one if the system knows ahead of time the sequence of requests

associated with each of the active processes. As was illustrated in the graphs presented in Figure 5.7 through Figure 5.11, there exists at least one allocation of resources sequence that will allow jobs to continue without becoming deadlocked.

One such algorithm was proposed by Dijkstra in 1965 to regulate resource allocation to avoid deadlocks. The Banker's Algorithm is based on a bank with a fixed amount of capital that operates on the following principles:

- No customer will be granted a loan exceeding the bank's total capital.
- All customers will be given a maximum credit limit when opening an account.
- No customer will be allowed to borrow over the limit.
- The sum of all loans won't exceed the bank's total capital.

For our example, the bank has a total capital fund of $10,000 and has three customers, C1, C2, and C3, who have maximum credit limits of $4,000, $5,000, and $8,000, respectively. Table 5.4 illustrates the state of affairs of the bank after some loans have been granted to C2 and C3. This is called a safe state because the bank still has enough money left to satisfy the maximum requests of C1, C2, or C3.

Table 5.4 The bank started with $10,000 and has remaining capital of $4,000 after these loans. Therefore, it's in a "safe state."

Customer	Loan Amount	Maximum Credit	Remaining Credit
C1	0	4,000	4,000
C2	2,000	5,000	3,000
C3	4,000	8,000	4,000
Total loaned: $6,000			
Total capital fund: $10,000			

A few weeks later after more loans have been made, and some have been repaid, the bank is in the unsafe state represented in Table 5.5.

Table 5.5. The bank only has remaining capital of $1,000 after these loans and therefore is in an "unsafe state."

Customer	Loan Amount	Maximum Credit	Remaining Credit
C1	2,000	4,000	2,000
C2	3,000	5,000	2,000
C3	4,000	8,000	4,000
Total loaned: $9,000			
Total capital fund: $10,000			

This is an unsafe state because with only $1,000 left, the bank can't satisfy anyone's maximum request; and if the bank lent the $1,000 to anyone, then it would be deadlocked (it can't make a loan).

In this example the system has 10 devices. Table 5.6 shows our system in a safe state and Table 5.7 depicts the same system in an unsafe state. As before, a safe state is one in which at least one job can finish because there are enough available resources to satisfy its maximum needs. Then, using the resources released by the finished job, the maximum needs of another job can be filled and that job can be finished, and so on until all jobs are done.

Table 5.6 Resource assignments after initial allocations. A safe state: Six devices are allocated and four units are still available.

Job No.	Devices Allocated	Maximum Required	Remaining Needs
1	0	4	4
2	2	5	3
3	4	8	4
Total number of devices allocated: 6			
Total number of devices in system: 10			

Table 5.7 Resource assignments after later allocations. An unsafe state: Only one unit is available but every job requires at least two to complete its execution

Job No.	Devices Allocated	Maximum Required	Remaining Needs
1	2	4	2
2	3	5	2
3	4	8	4
Total number of devices allocated: 9			
Total number of devices in system: 10			

The operating system must be sure never to satisfy a request that moves it from a safe state to an unsafe one. Therefore, as users' requests are satisfied, the operating system must identify the job with the smallest number of remaining resources and make sure that the number of available resources is always equal to, or greater than, the number needed for this job to run to completion. Requests that would place the safe state in jeopardy must be blocked by the operating system until they can be safely accommodated

Detection

The directed graphs showed the existence of a circular wait indicated a deadlock, so it's reasonable to conclude that deadlocks can be detected by building directed resource graphs.

The detection algorithm can be explained by using directed resource graphs. And "reducing" them. Begin with a system that is in use, as shown in Figure 5.12(a). The steps to reduce a graph are these:

Find a process that is currently using a resource and not waiting for one. This process can be removed from the graph (by disconnecting the link tying the resource to the process, such as P3 in Figure 5.12(b)), and the resource can be returned to the "available list." This is possible because the process would eventually finish and return the resource.

Find a process that's waiting only for resource classes that aren't fully allocated (such as P2 in Figure 5.12(c)). This process isn't contributing to deadlock since it would eventually get the resource it's waiting for, finish its work, and return the resource to the "available list" as shown in Figure 5.12(c)."

Go back to step 1 and continue with steps 1 and 2 until all lines connecting resources to processes have been removed, eventually reaching the stage shown in Figure 5.12(d). If there are any lines left, this indicates that the request of the process in question can't be satisfied and that a deadlock exists.

Figure 5.12 illustrates a system in which three processes—P1, P2, and P3—and three resources—R1, R2, and R3—aren't deadlocked.

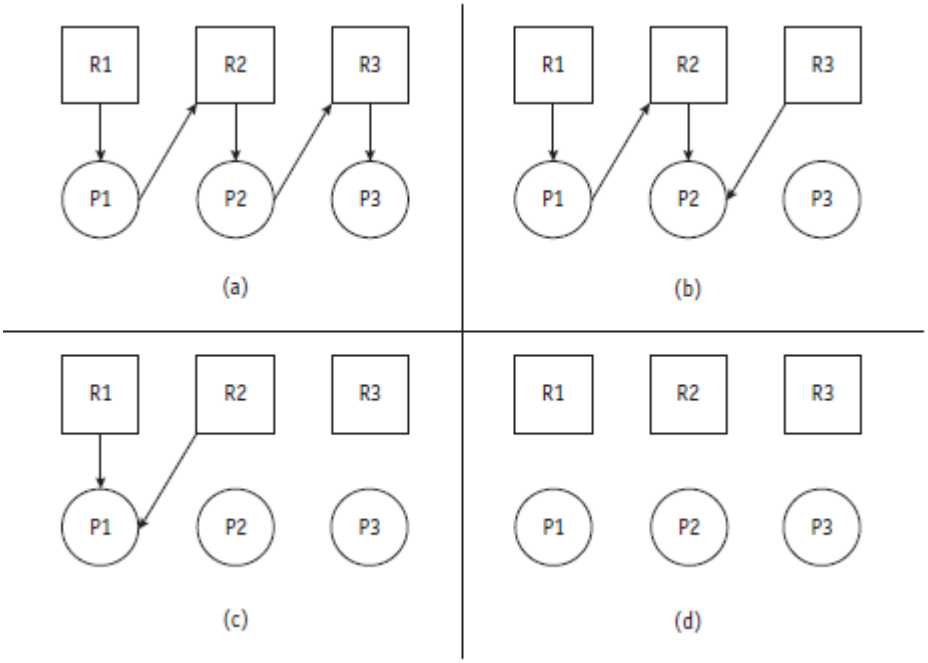


Figure 5.12 shows the stages of a graph reduction from (a), the original state. In (b), the link between P3 and R3 can be removed because P3 isn't waiting for any other resources to finish, so R3 is released and allocated to P2 (step 1). In (c), the links between P2 and R3 and between P2 and R2 can be removed because P2 has all of its requested resources and can run to completion—and then R2 can be allocated to P1. Finally, in (d), the links between P1 and R2 and between P1 and R1 can be removed because P1 has all of its requested resources and can finish successfully. Therefore, the graph is completely resolved.

However, Figure 5.13 shows a very similar situation that is deadlocked because of a key difference: P2 is linked to R1.

The deadlocked system in Figure 5.13 can't be reduced. In (a), the link between P3 and R3 can be removed because P3 isn't waiting for any other resource, so R3 is released and allocated to P2. But in (b), P2 has only two of the three resources it needs to finish and it is waiting for R1. But R1 can't be released

by P1 because P1 is waiting for R2, which is held by P2; moreover, P1 can't finish because it is waiting for P2 to finish (and release R2), and P2 can't finish because it's waiting for R1. This is a circular wait.

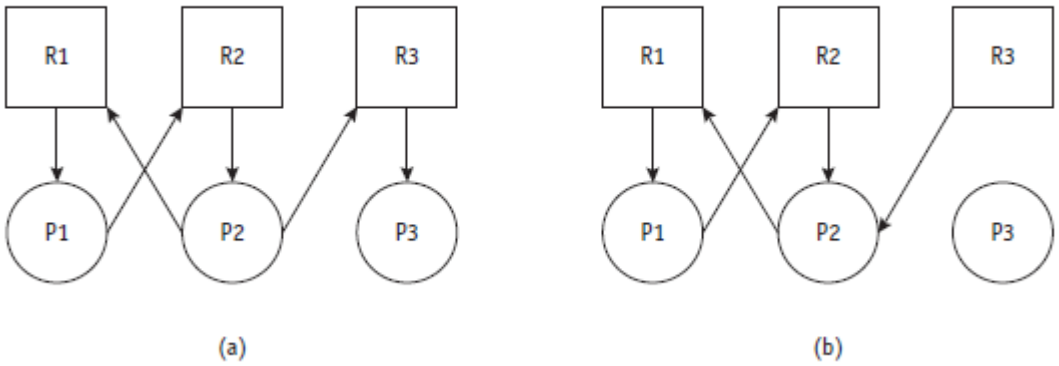


Figure 5.13 Even after this graph (a) is reduced as much as possible (by removing the request from P3), it is still deadlocked (b).

Recovery

Once a deadlock has been detected, the system returned to normal as quickly as possible.

The first simplest recovery method, and the most drastic, is to terminate every job that's active in the system and restart them from the beginning.

The second method is to terminate only the jobs involved in the deadlock and ask their users to resubmit them.

The third method is to identify which jobs are involved in the deadlock and terminate them one at a time, checking to see if the deadlock is eliminated after

each removal, until the deadlock has been resolved. Once the system is freed, the remaining jobs are allowed to complete their processing and later the halted jobs are started again from the beginning.

The fourth method can be put into effect only if the job keeps a record, a snapshot, of its progress so it can be interrupted and then continued without starting again from the beginning of its execution.

Fifth method in our list, selects a non deadlocked job, preempts the resources it's holding, and allocates them to a deadlocked process so it can resume execution, thus breaking the deadlock.

The sixth method stops new jobs from entering the system, which allows the non deadlocked jobs to run to completion so they'll release their resources.

Starvation

Starvation is the result of conservative allocation of resources where a single job is prevented from execution because it's kept waiting for resources that never become available. To illustrate this, the case of the dining philosophers problem was introduced by Dijkstra.

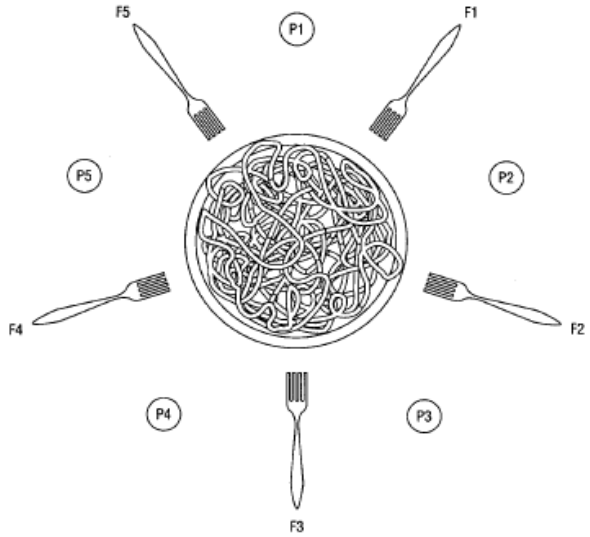


Figure 5.14 The dining philosophers' table, before the meal begins.

Five philosophers are sitting at a round table, each deep in thought, and in the center lies a bowl of spaghetti that is accessible to everyone. There are forks on the table—one between each philosopher, as illustrated in Figure 5.14. Local custom dictates that each philosopher must use two forks, the forks on either side of the plate, to eat the spaghetti, but there are only five forks—not the 10 it would require for all five thinkers to eat at once—and that's unfortunate for Philosopher 2.

When they sit down to dinner, Philosopher 1 (P1) is the first to take the two forks (F1 and F5) on either side of the plate and begins to eat. Inspired by his colleague,Philosopher 3 (P3) does likewise, using F2 and F3. Now Philosopher 2 (P2) decides to begin the meal but is unable to start because no forks are available: F1 has been allocated to P1, and F2 has been allocated to P3, and the only remaining fork can be used only by P4 or P5. So (P2) must wait. Soon, P3 finishes eating, puts down his two forks, and resumes his pondering. Should the fork beside him (F2), that's now free, be allocated to the hungry philosopher (P2)?

Although it's tempting, such a move would be a bad precedent because if the philosophers are allowed to tie up resources with only the hope that the other required resource will become available, the dinner could easily slip into an unsafe state; it would be only a matter of time before each philosopher held a single fork—and nobody could eat.

So the resources are allocated to the philosophers only when both forks are available at the same time. The status of the "system" is illustrated in Figure 5.15. P4 and P5 are quietly thinking and P1 is still eating when P3 (who should be full) decides to eat some more; and because the resources are free, he is able to take F2 and F3 once again. Soon thereafter, P1 finishes and releases F1 and F5, but P2 is still not able to eat because F2 is now allocated. This scenario could continue forever; and as long as P1 and P3 alternate their use of the available resources, P2 must wait. P1 and P3 can eat any time they wish while P2 starves—only inches from nourishment.

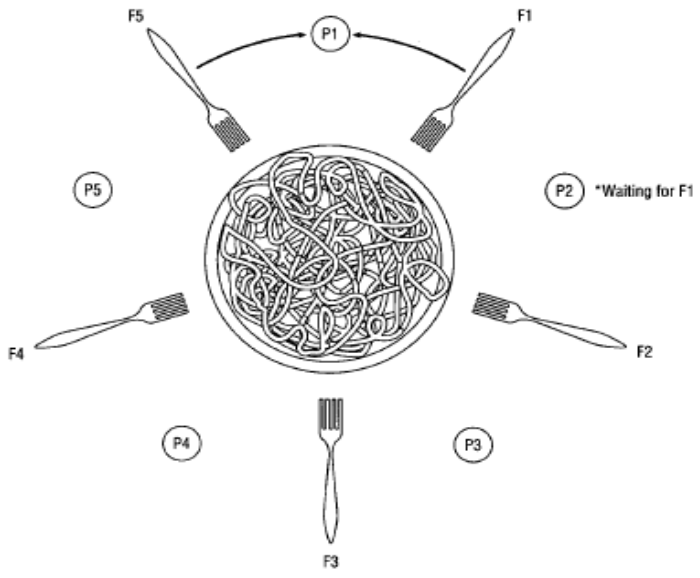


Figure 5.15 Each philosopher must have both forks to begin eating, the one on the right and the one on the left. Unless the resources,the forks, are allocated fairly, some philosophers may starve.

In a computer environment, the resources are like forks and the competing processes are like dining philosophers. If the resource manager doesn't watch for starving processes and jobs, and plan for their eventual completion, they could remain in the system forever waiting for the right combination of resources.

To address this problem, an algorithm designed to detect starving jobs can be implemented, which tracks how long each job has been waiting for resources Once starvation has been detected, the system can block new jobs until the starving jobs have been satisfied. This algorithm must be monitored closely: If monitoring is done too often, then new jobs will be blocked too frequently and throughput will be diminished. If it's not done often enough, then starving jobs will remain in the system for an unacceptably long period of time.

Conclusion

Every operating system must dynamically allocate a limited number of resources while avoiding the two extremes of deadlock and starvation. In this chapter we discussed several methods of dealing with livelocks and deadlocks: prevention, avoidance, and detection and recovery. Deadlocks can be prevented by not allowing the four conditions of a deadlock to occur in the system at the same time. By eliminating at least one of the four conditions (mutual exclusion, resource holding, no preemption, and circular wait), the system can be kept deadlock-free. As we've seen, the disadvantage of a preventive policy is that each of these conditions is vital to different parts of the system at least some of the time, so prevention algorithms are complex and to routinely execute them involves high overhead.

Deadlocks can be avoided by clearly identifying safe states and unsafe states and requiring the system to keep enough resources in reserve to guarantee that all jobs active in the system can run to completion. The disadvantage of an avoidance policy is that the system's resources aren't allocated to their fullest potential.

If a system doesn't support prevention or avoidance, then it must be prepared to detect and recover from the deadlocks that occur. Unfortunately, this option usually relies on the selection of at least one "victim"—a job that must be terminated before it finishes execution and restarted from the beginning.

Exercise:

1. Consider an archival system with 13 dedicated devices. All jobs currently running on this system require a maximum of five drives to complete but they each run for long periods of time with just four drives and request the fifth one only at the very end of the run. Assume that the job stream is endless.
    a. Suppose your operating system supports a very conservative device allocation policy so that no job will be started unless all the required drives have been allocated to it for the entire duration of its run.
        1. What is the maximum number of jobs that can be active at once? Explain your answer.
        2. What are the minimum and maximum number of tape drives that may be idle as a result of this policy? Explain your answer.
    b. Suppose your operating system supports the Banker's Algorithm.
        3. What is the maximum number of jobs that can be in progress at once? Explain your answer.
        4. What are the minimum and maximum number of drives that may be idle as a result of this policy? Explain your answer.

For the three systems described below in 3 -4, given that all of the devices are of the same type, and using the definitions presented in the discussion of the Banker's Algorithm, answer these questions:

a. Determine the remaining needs for each job in each system.

b. Determine whether each system is safe or unsafe.

c. If the system is in a safe state, list the sequence of requests and releases that will make it possible for all processes to run to completion.

d. If the system is inn unsafe state, show how it's possible for deadlock to occur.

3. System A has 12 devices; only one is available

Job No.	Devices Allocated	Maximum Required	Remaining Needs
1	5	6	
2	4	7	
3	2	6	
4	0	2	

4. System B has 14 devices and only two are available

Job No.	Devices Allocated	Maximum Required	Remaining Needs
1	5	8	
2	3	9	
3	4	8	

5.

5. Consider the directed resource graph shown in Figure 5.17 and answer the following questions:

a. Are there any blocked processes?

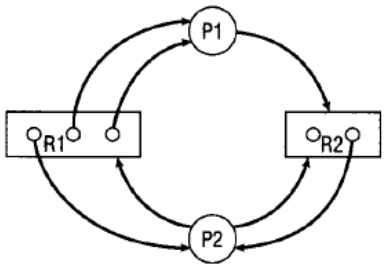
b. Is this system deadlocked?

c. What is the resulting graph after reduction by P1?

d. What is the resulting graph after reduction by P2?

e. If Both P1 and P2 have requested R2, answer these questions:

1. What is the status of the system if the request by P2 is granted before that of P1?
2. What is the status of the system if the request by P1 is granted before that of P2?



Directed resource graph

6. Consider the directed resource graph and answer the following questions:

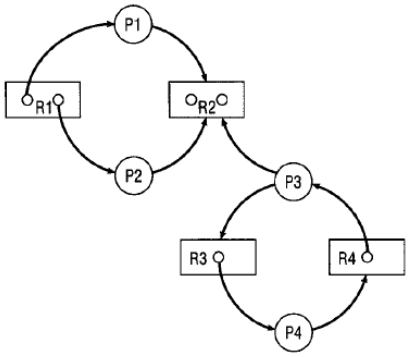
a. Identify all of the deadlocked processes.

b. Can the directed graph be reduced, partially or totally?

c. Can the deadlock be resolved without selecting a victim?

d. Which requests by the three processes for resources from R2 would you satisfy to minimize the number of processes involved in the deadlock?

e. Conversely, which requests by the three processes for resources from R2 would you satisfy to maximize the number of processes involved in deadlock?



Directed resource graph

CHAPTER 6

Concurrent Processes

Overview

In this chapter we look at another common situation, *multiprocessing* systems, which have several processors working together in several distinctly different configurations. Multiprocessing systems include single computers with multiple cores as well as linked computing systems with only one processor each to share processing among them.

What Is Parallel Processing?

Parallel processing, one form of **multiprocessing**, is a situation in which two or more processors operate in unison. That means two or more CPUs are executing instructions simultaneously. In multiprocessing systems, the Processor Manager has to coordinate the activity of each processor, as well as synchronize cooperative interaction among the CPUs.

Benefits to parallel processing systems:

- Increased reliability: The reliability stems from the availability of more than one CPU, If one processor fails, then the others can continue to operate and absorb the load.
- Faster processing. : The increased processing speed is often achieved because sometimes instructions can

be processed in parallel, two or more at a time, in one of several ways. Synchronization is the key to the success of multiprocessing system.

Evolution of Multiprocessors

Multiprocessing can take place at several different levels, each of which requires a different frequency of synchronization, as shown in Table 6.2. Notice that at the job level, multiprocessing is fairly begin. It's as if each job is running on its own workstation with shared system resources. On the other hand, when multiprocessing takes place at the thread level, a high degree of synchronization is required to disassemble each process, perform the thread's instructions, and then correctly reassemble the process.

Table 6.2 Levels of parallelism and the required synchronization among processors.

Parallelism Level	Process Assignments	Synchronization Required
Job Level	Each job has its own processor and all processes and threads are run by that same processor.	No explicit synchronization required.
Process Level	Unrelated processes, regardless of job, are assigned to any available processor.	Moderate amount of synchronization required to track processes.
Thread Level	Threads are assigned to available processors.	High degree of synchronization required, often requiring explicit instructions from the programmer.

Introduction to Multi-Core Processors

Multi-core processors have several processors on a single chip. As processors became smaller in size (as predicted by Moore's Law) and faster in processing speed, CPU designers began to use nanometer-sized transistors but the space between transistors became ever closer. When transistors are placed extremely close together, electrons have the ability to spontaneously tunnel, at random, from one transistor to another, causing a tiny but measurable amount of current to leak.

One solution was to create a single chip (one piece of silicon) with two "processor cores" in the same amount of space. With this arrangement, two sets of calculations can take place at the same time.

The two cores on the chip generate less heat than a single core of the same size and tunneling is reduced; however, the two cores each run more slowly than the single core chip. Therefore, to get improved performance from a dual core chip, the software has to be structured to take advantage of the double calculation capability of the new chip design.

Building on their success with two-core chips, designers have created multi-core processors with predictions, as of this writing, that 80 or more cores will be placed on a single chip.

Typical Multiprocessing Configurations

Three typical Multiprocessing configurations are:

1. Master/Slave
2. Loosely Coupled
3. Symmetric.

Master/Slave Configuration

The **master/slave** configuration is an asymmetric multiprocessing system. Think of it as a single-processor system with additional slave processors, each of which is managed by the primary master processor as shown in Figure 6.1.

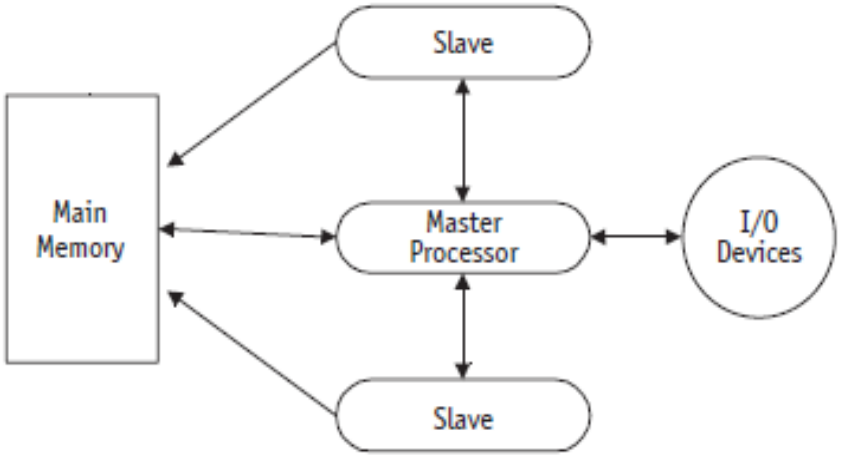


Figure 6.1 In a master/slave multiprocessing configuration, slave processors can access main memory directly but they must send all I/O requests through the master processor

The master processor is responsible for managing the entire system—all files, devices, memory, and processors. Therefore, it maintains the status of all processes in the system, performs storage management activities, schedules the work for the other processors, and executes all control programs.

Advantage :

The configuration is simple.

Disadvantages:

- Its reliability is no higher than for a single-processor system because if the master processor fails, the entire system fails.
- It can lead to poor use of resources because if a slave processor should become free while the master processor is busy, the slave must wait until the master becomes free and can assign more work to it.
- It increases the number of interrupts because all slave processors must interrupt the master processor every time they need operating system intervention, such as for I/O requests. This creates long queues at the master processor level when there are many processors and many interrupts.

Loosely Coupled Configuration

The **loosely coupled configuration** features several complete computer systems, each with its own memory, I/O devices, CPU, and operating system, as shown in Figure 6.2.

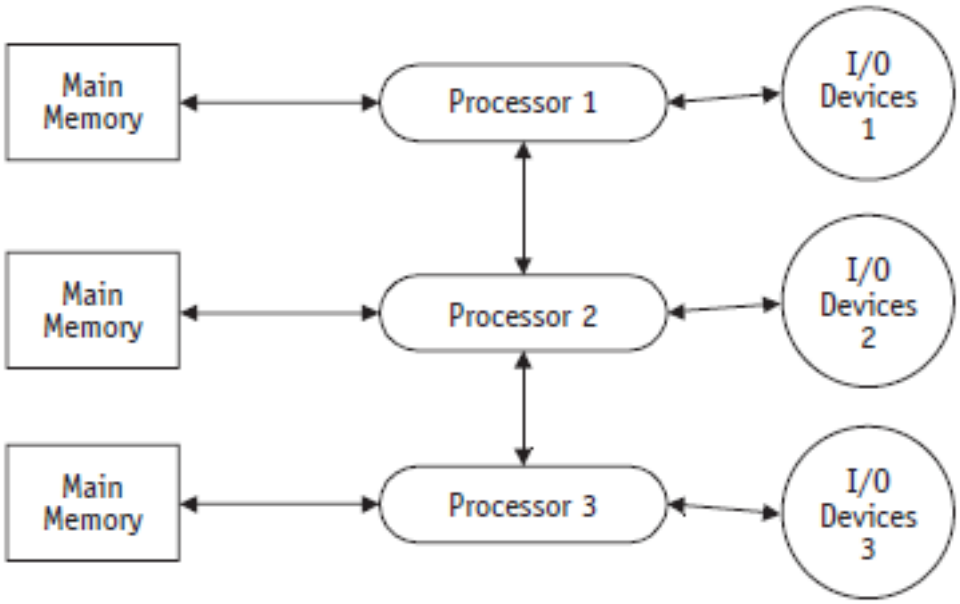


Figure 6.2 In a loosely coupled multiprocessingconfiguration, eachprocessor has its own dedicated resources

This configuration is called loosely coupled because each processor controls its own resources—its own files, access to memory, and its own I/O devices—and that means that each processor maintains its own commands and I/O management tables.

Symmetric Configuration

The **symmetric configuration** (also called tightly coupled) has four advantages over loosely coupled configuration:

- It's more reliable.
- It uses resources effectively.
- It can balance loads well.
- It can degrade gracefully in the event of a failure.

However, it is the most difficult configuration to implement because the processes must be well synchronized to avoid the problems of races and deadlocks,In a symmetric configuration (as depicted in Figure 6.3), processor scheduling is decentralized.

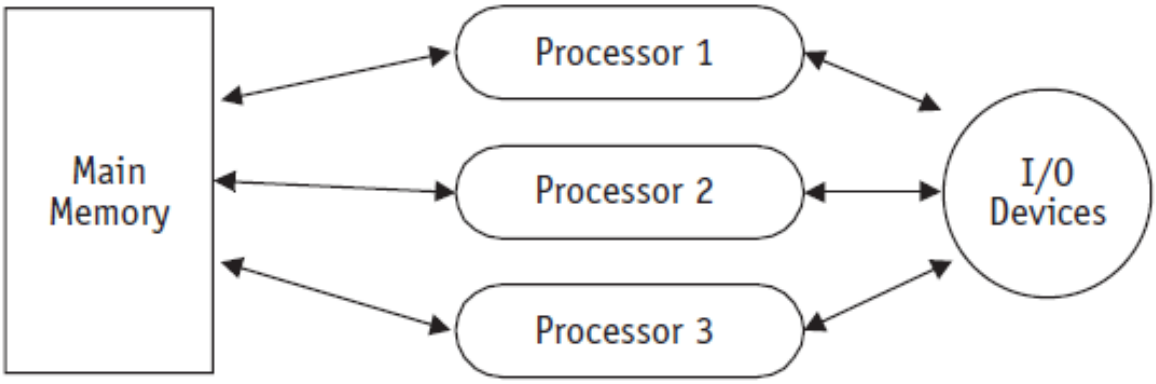


Figure 6.3 A symmetric multiprocessing configuration with homogeneous processors. Processes must be carefully synchronized to avoid deadlocks and starvation

A single copy of the operating system and a global table listing each process and its status is stored in a common area of memory so every processor has access to it. Each processor uses the same scheduling algorithm to select which process it will run next.

Whenever a process is interrupted, whether because of an I/O request or another type of interrupt, its processor updates the corresponding entry in the process list and finds another process to run. There are more conflicts as several processors try to access the same resource at the same time.

There is a need for algorithms to resolve conflicts between processors—that's called **process synchronization**.

Process Synchronization Software

Several synchronization mechanisms are available to provide cooperation and communication among processes. The common element in all synchronization schemes is to allow a process to finish work on a critical part of the program before other processes have access to it.

This is applicable both to multiprocessors and to two or more processes in a single-processor (time-shared) processing system. It is called a **critical region** because it is a critical section and its execution must be handled as a unit. As we've seen, the processes within a critical region can't be interleaved without threatening the integrity of the operation.

Synchronization is sometimes implemented as a lock-and-key arrangement , Several locking mechanisms have been developed, including

- Test-and-set
- WAIT andSIGNAL
- Semaphores.

Test-and-Set

Test-and-set is a single, indivisible machine instruction known simply as **TS.** In a single machine cycle it tests to see if the key is available and, if it is, sets it to unavailable. The actual key is a single bit in a storage location that can contain a

0 (if it's free) or a 1 (if busy). We can consider TS to be a function subprogram that has one parameter (the storage location) and returns one value (the condition code: busy/free), with the exception that it takes only one machine cycle.

Therefore, a process (Process 1) would test the condition code using the TS instruction before entering a critical region. If no other process was in this critical region, then Process 1 would be allowed to proceed and the condition code would be changed from 0 to 1. Later, when Process 1 exits the critical region, the condition code is reset to 0 so another process can enter. On the other hand, if Process 1 finds a busy condition code, then it's placed in a waiting loop where it continues to test the condition code and waits until it's free.

Although it's a simple procedure to implement, and it works well for a small number of processes, test-and-set has two major drawbacks.

First, when many processes are waiting to enter a critical region, starvation could occur because the processes gain access in an arbitrary fashion.

A second drawback is that the waiting processes remain in unproductive, resource-consuming wait loops, requiring context switching. This is known as **busy waiting**—which not only consumes valuable processor time but also relies on the competing processes to test the key, something that is best handled by the operating system or the hardware.

WAIT and SIGNAL

WAIT and SIGNAL is a modification of test-and-set that's designed to remove busy waiting. Two new operations, which are mutually exclusive and become part of the process scheduler's set of operations, are WAIT and SIGNAL. WAIT is activated when the process encounters a busy condition code. WAIT sets the process's process control block (PCB) to the blocked state and links it to the queue of processes waiting to enter this particular critical region. The Process Scheduler then selects another process for execution. SIGNAL is activated when a process exits the critical region and the condition code is set to "free." It checks the queue of processes waiting to enter this critical region and selects one, setting it to the READY state.

Eventually the Process Scheduler will choose this process for running. The addition of the operations WAIT and SIGNAL frees the processes from the busy waiting dilemma and returns control to the operating system, which can then run other jobs while the waiting processes are idle (WAIT).

Semaphores

A **semaphore** is a non-negative integer variable that's used as a binary signal. In an operating system, a semaphore performs a similar function: It signals if and when a resource is free and can be used by a process. Dijkstra (1965) introduced two operations to overcome the process synchronization problem we've discussed. Dijkstra called them P and V, and that's how they're known today. The

P for (to test) and the V for (to increment). The P and V operations do just that: They test and increment.

If we let *s* be a semaphore variable, then the V operation on *s* is simply to increment *s* by 1. The action can be stated as:

V(*s*): *s*: = *s* + 1

This in turn necessitates a fetch, increment, and store sequence. The operation P on *s* is to test the value of *s* and, if it's not 0, to decrement it by 1.
The action can be stated as:

P(*s*): If *s* > 0 then *s*: = *s* – 1

This involves a test, fetch, decrement, and store sequence.

Again this sequence must be performed as an indivisible action in a single machine cycle or be arranged so that the process cannot take action until the operation (test or increment) is finished.

As shown in Table 6.3, P3 is placed in the WAIT state (for the semaphore) on State 4. As also shown in Table 6.3, for States 6 and 8, when a process exits the critical region, the value of *s* is reset to 1 indicating that the critical region is free. This, in turn, triggers the awakening of one of the blocked processes, its entry into the critical region, and the resetting of *s* to 0. In State 7, P1 and P2 are not trying to do processing in that critical region and P4 is still blocked.

Table 6.3 The sequence of states for four processes calling test and increment (P and V) operations on the binary semaphore s.
(Note: The value of the semaphore before the operation is shown on the line preceding the operation. The current value is on the same line.)

State Number	Actions Calling Process	Operation	Running in Critical Region	Results Blocked on *s*	Value of *s*
0					1
1	P1	test(*s*)	P1		0
2	P1	increment(*s*)			1
3	P2	test(*s*)	P2		0
4	P3	test(*s*)	P2	P3	0
5	P4	test(*s*)	P2	P3, P4	0
6	P2	increment(*s*)	P3	P4	0
7			P3	P4	0
8	P3	increment(*s*)	P4		0
9	P4	increment(*s*)			1

After State 5 of Table 6.3, the longest waiting process, P3, was the one selected to enter the critical region, but that isn't necessarily the case unless the system is using a first-in, first-out selection policy Table 6.3, test and increment operations on semaphore *s* enforce the concept of mutual exclusion, which is necessary to avoid having two operations attempt to execute at the same time.

The name traditionally given to this semaphore in the literature is **mutex** and it stands for MUTual EXclusion. So the operations become:

test(mutex): if mutex > 0 then mutex: = mutex – 1

increment(mutex): mutex: = mutex + 1

Several processes trying to access the same shared critical region. The procedure can generalize to semaphores having values greater than 0 and 1.

Conclusion

Multiprocessing can occur in several configurations: in a single-processor system where interacting processes obtain control of the processor at different times, or in systems with multiple processors, where the work of each processor communicates and cooperates with the others and is synchronized by the Processor Manager.

Three multiprocessing systems are described in this chapter: master/slave, loosely coupled, and symmetric. Each can be configured in a variety of ways.
The success of any multiprocessing system depends on the ability of the system to synchronize its processes with the system's other resources.

The concept of mutual exclusion helps keep the processes with the allocated resources from becoming deadlocked. Mutual exclusion is maintained with a series of techniques, including test-and-set, WAIT and SIGNAL, and semaphores: test (P), increment (V), and mutex.

Exercise

1. Compare the processors' access to printers and other I/O devices for the master/slave and the symmetric multiprocessing configurations. Give a real-life example where the master/slave configuration might be preferred.
2. Compare the processors' access to main memory for the loosely coupled configuration and the symmetric multiprocessing configurations. Give areal-life example where the symmetric configuration might be preferred.
3. Describe the programmer's role when implementing explicit parallelism.
4. Describe the programmer's role when implementing implicit parallelism.
5. What steps does a well-designed multiprocessing system follow when it detects that a processor is failing? What is the central goal of most multiprocessing systems?
6. Give an example from real life of busy waiting.

CHAPTER 7

DEVICE MANAGEMENT

Despite the multitude of devices that appear (and disappear) swift rate of change in device technology, the Device Manager must manage every peripheral device of the system.

Device management involves four basic functions:

- Monitoring the status of each device, such as storage drives, printers, and other peripheral devices
- Enforcing policies to determine which process will get a device and for how long
- Allocating the devices
- Deallocating them at two levels—at the process (or task) level when an I/O command has been executed and the device is temporarily released, and then at the job level when the job is finished and the device is permanently released

Types of Devices

The system's peripheral devices generally fall into one of three categories: dedicated, shared, and virtual.

Dedicated devices

- Assigned to only one job at a time; they serve that job for the entire time it's active or until it releases them.
- Some devices, such as tape drives, printers, and plotters, demand this kind of allocation scheme
- The disadvantage of dedicated devices is that they must be allocated to a single user for the duration of a job's execution, which can be quite inefficient, especially when the device isn't used 100 percent of the time. And some devices can be shared or virtual.

Shared devices

- The device can be assigned to several processes.
- The device can be shared by several processes at the same time by interleaving their requests
- This interleaving must be carefully controlled by the Device Manager.

All conflicts—such as when Process A and Process B each need to read from the same disk—must be resolved based on predetermined policies to decide which request will be handled first.

Virtual devices

- A combination of the first two
- They're dedicated devices that have been transformed into shared devices.

- For example, printers (which are dedicated devices) are converted into sharable devices through a spooling program that reroutes all print requests to a disk.

For example, the universal serial bus (USB) controller is a virtual device that acts as an interface between the operating system, device drivers, and applications and the devices that are attached via the USB host. One USB host (assisted by USB hubs) can accommodate up to 127 different devices, including flash memory, cameras, scanners, musical keyboards, etc.

Each device is identified by the USB host controller with a unique identification number, which allows many devices to exchange data with the computer using the same USB connection.

Regardless of the specific attributes of the device, the most important differences among them are speed and degree of sharability.

Storage media are divided into two groups: **sequential access media,** which store records sequentially, one after the other; and **direct access storage devices** (DASD), which can store either sequential or direct access files. .

Sequential Access Storage Media

Magnetic tape was developed for routine secondary storage in early computer systems and features records that are stored serially, one after the other. The length of these records is usually determined by the application program and each record can be identified by its position on the tape.

To access a single record, the tape must be mounted and fast-forwarded from its beginning until the desired position is located. This can be a time-consuming process.

Characteristics of a Magnetic tape:

- A typical tape is 2400 feet long.
- Data is recorded on eight of the nine parallel tracks.
- Nine-track magnetic tape with three characters recorded using odd parity.
- A 1/2-inch wide reel of tape, can store thousands of characters, or bytes, per inch.
- The number of characters that can be recorded per inch is 600 bytes (bpi).
- Can store 10 records on one inch of tape.

If the records are stored individually, each record would need to be separated by a space to indicate its starting and ending places.

If the records are stored in blocks, then the entire block is preceded by a space and followed by a space, but the individual records are stored sequentially within the block.

Magnetic tape moves under the read/write head only when there's a need to access a record; Records would be written in the same way.

The tape needs time and space to stop, so a gap is inserted between each record. This inter record gap (IRG) is about 1⁄2 inch long regardless of the sizes of the records it separates.

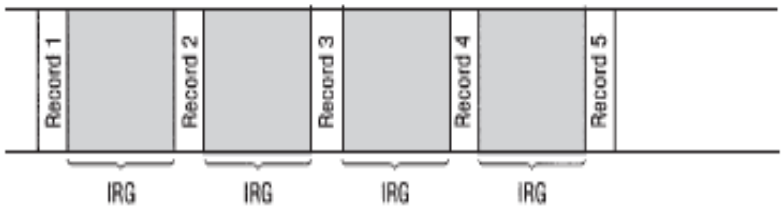


Figure 7.1 IRGs in magnetic Tape

Therefore, if 10 records are stored individually, there will be nine 1⁄2-inch(totally 4.5 inch) IRGs between each record. The transfer rate of the tape can be measured(in bpi) by:

Transfer rate (ips) = Density * Transport speed

Blocking has two distinct advantages:

- Fewer I/O operations are needed because a single READ command can move an entire block.
- Less tape is wasted because the size of the physical record exceeds the size of the gap.

The two disadvantages of blocking seem mild by comparison

- Overhead and software routines are needed for blocking, deblocking, and recordkeeping
- Buffer space may be wasted if you need only one logical record but must read an entire block to get it.

Table 7.1 Access times for 2400-foot magnetic tape with a tape transport speed of 200 ips

Benchmarks	Access Time
Maximum access	2.5 minutes
Average access	1.25 minutes
Sequential access	3 milliseconds

Direct Access Storage Devices

Direct access storage devices (DASDs) are devices that can directly read or write to a specific place.

DASDs can be grouped into three categories:

- Magnetic disks
- Optical discs
- Flash memory.

a) Fixed-Head Magnetic Disk Storage

A fixed-head magnetic disk looks like a large CD or DVD covered with magnetic film that has been formatted, usually on both sides, into concentric circles. Each circle is a track. Data is recorded serially on each track by the fixed read/write head positioned over it.

A fixed-head disk is very fast—faster than the movable-head disks. Its major disadvantages are : its high cost and its reduced storage space

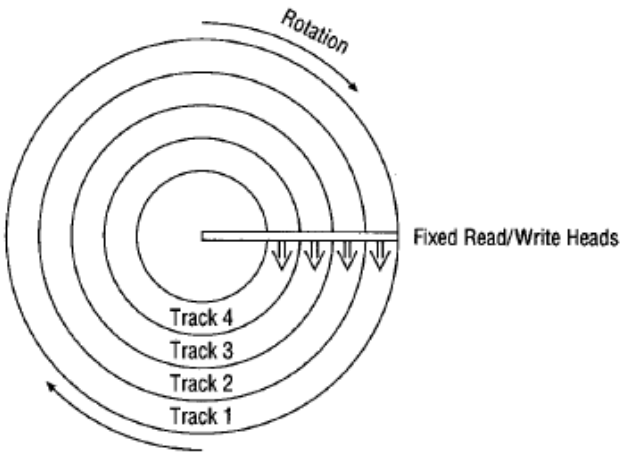


Figure 7.2 A fixed-head disk with our read/write heads, one per track

b) Movable-Head Magnetic Disk Storage

Movable-head magnetic disks, such as computer hard drives, have one read/write head that floats over each surface of each disk. Disks can be a single platter, or part of a disk pack, which is a stack of magnetic platters.

A typical disk pack—several platters stacked on a common central spindle, separated by enough space . Each platter (except those at the top and bottom of the stack) has two surfaces for recording, and each surface is formatted with a specific number of concentric tracks where the data is recorded.

The number of tracks varies, but typically there are a thousand or more on a high capacity hard disk. Each track on each surface is numbered: Track 0 identifies the outermost concentric circle on each surface; the highest-numbered track is in the center.

The arm moves two read/write heads between each pair of surfaces: one for the surface above it and one for the surface below.

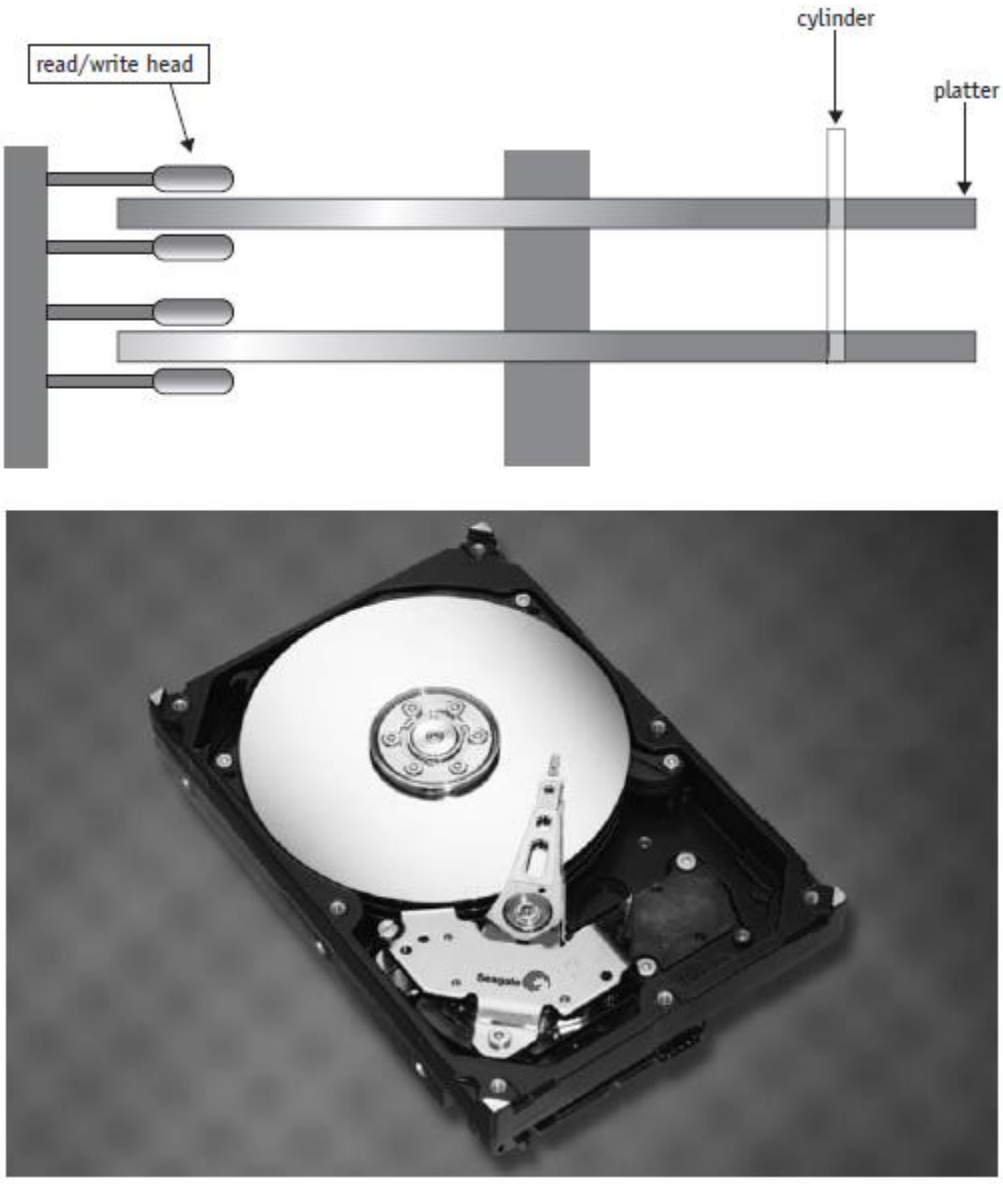


Figure 7.3 Disk pack and A typical hard drive from a PC showing the arm that floats over the surface of the disk

To access any given record, the system needs three things:

- Its cylinder number, so the arm can move the read/write heads to it;
- Its surface number, so the proper read/write head is activated;

- Its sector number, so the read/write head knows the instant when it should begin reading or writing.

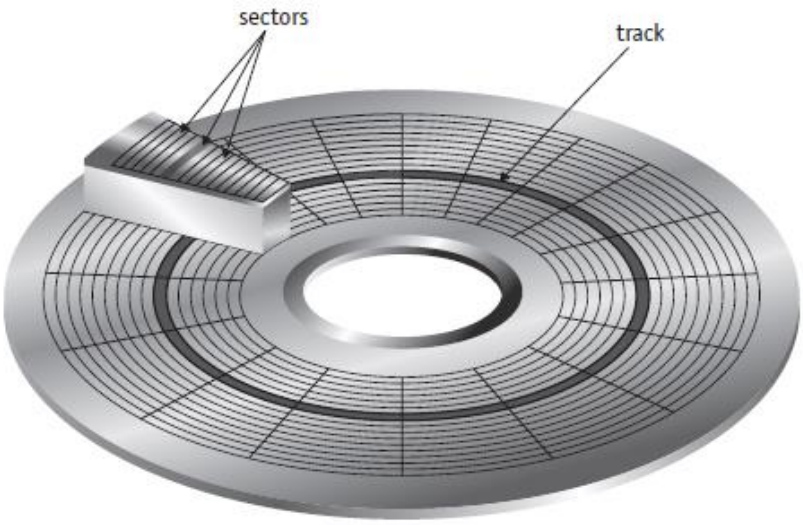


Figure 7.4 Tracks and sectors in Magnetic Disk

Disc Storage

The optical disc storage uses laser technology. Among the many differences between an optical disc and a magnetic disk is the design of the disc track and sectors.

A magnetic disk, which consists of concentric tracks of sectors, spins at a constant speed—this is called constant angular velocity (CAV). Because the sectors at the outside of the disk spin faster past the read/write head than the inner sectors, outside sectors are much larger than sectors located near the center of the disk. This format wastes storage space but maximizes the speed with which data can be retrieved.

On the other hand, **an optical disc** consists of a single spiraling track of same-sized sectors running from the center to the rim of the disc. This single track also has sectors, but all sectors are the same size regardless of their locations on the disc. This design allows many more sectors, and much more data, to fit on an optical disc compared to a magnetic disk of the same size. The disc drive adjusts the speed of the disc's spin to compensate for the sector's location on the disc—this is called constant linear velocity (CLV).

Two of the most important measures of optical disc drive performance are : data transfer rate and average access time.

- The data transfer rate is measured in megabytes per second and refers to the speed at which massive amounts of data can be read from the disc.
- Access time, which indicates the average time required to move the head to a specific place on the disc, is expressed in milliseconds (ms). The fastest units have the smallest average access time, which is the most important factor when searching for information randomly, such as in a database.

A fast data-transfer rate is most important for sequential disc access, whereas fast access time is crucial when retrieving data.

A third important feature of optical disc drives is cache size. Although it's not a speed measurement, cache size has a substantial impact on perceived performance.

There are several types of optical-disc systems, depending on the medium and the capacity of the discs: CDs, DVDs, and Blu-ray

To put data on an optical disc, a high-intensity laser beam burns indentations on the disc that are called pits. These pits, which represent 0s, contrast with the unburned flat areas, called lands, which represent 1s. The first sectors are located

in the center of the disc and the laser moves outward reading each sector in turn. If a disc has multiple layers, the laser's course is reversed to read the second layer with the arm moving from the outer edge to the inner.

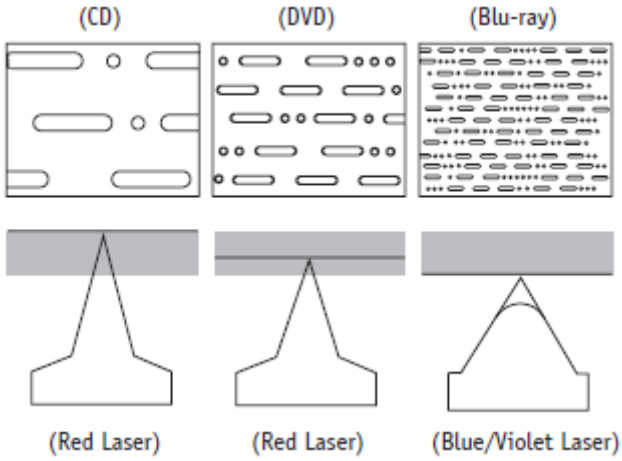


Figure 7.5 Red and Blue laser to write on CD,DVD and Blue-ray

CD and DVD Technology

In the CD or DVD player, data is read back by focusing a low-powered red laser on it. Light striking a land is reflected into a photodetector which converts the light intensity into digital 1s. Light striking a pit is scattered and absorbed which can be converted into digital 0s by the photodetector.

Recordable CD and DVD disc drives require more expensive disc controllers than the read-only disc players because they need to incorporate write mechanisms specific to each medium.

For example, a CD consists of several layers, including a gold reflective layer and a dye layer, which is used to record the data.

The write head uses a high-powered laser beam to record data. A permanent mark is made on the dye when the energy from the laser beam is absorbed into it and it cannot be erased after it is recorded.

Similarly, **recordable and rewritable CDs** (CD-RWs) use a process called phase change technology to write, change, and erase data. The disc's recording layer uses an alloy of silver, indium, antimony, and tellurium.

The recording layer has two different phase states: amorphous and crystalline. In the amorphous state, light is not reflected as well as in the crystalline state. To record data, a laser beam heats up the disc and changes the state from crystalline to amorphous.

When data is read by means of a low-power laser beam, the amorphous state scatters the light that does not get picked up by the read head. This is interpreted as a 0.

On the other hand, when the light hits the crystalline areas, light is reflected back to the read head and is interpreted as a 1. To erase data, the CD-RW drive uses a low-energy beam to heat up the pits just enough to loosen the alloy and return it to its original crystalline state.

DVDs use the same design and are the same size and shape as CDs, they can store much more data. A dual-layer, single-sided DVD can hold the equivalent of 13 CDs; Its red laser has a shorter wavelength than the CD's red laser.

Compression technology in the high capacity DVD, such as MPEG video compression, a single-sided, double-layer DVD can hold 8.6GB.

Blu-ray Disc Technology

A Blu-ray disc is the same physical size as a DVD or CD but the laser technology used to read and write data is quite different. The pits on a Blu-ray disc are much smaller and the tracks are wound much tighter than they are on a DVD or CD. The Blu-ray's blue-violet laser (405nm) has a shorter wavelength than the CD/DVD's red laser (650nm).

This allows data to be packed more tightly and stored in less space.In addition, the blue-violet laser can write on a much thinner layer on the disc.

Allowing multiple layers to be written on top of each other

- vastly increasing the amount of data that can be stored on a single disc.
- Each Blu-ray disc can hold much more data (50GB)
- Reading speed is also much faster with the fastest Blu-ray (432 Mbps)

Blu-ray discs are available in several formats:

- Read-only (BD-ROM)
- Recordable (BD-R)
- Rewritable (BD-RE).

Flash Memory Storage

Flash memory is a type of electrically erasable programmable read-only memory (EEPROM). It's a nonvolatile removable medium stores data securely even when it's removed from its power source. Flash memory was primarily used to store startup (boot up) information for computers also used to store data for cell phones, mobile devices, music players, cameras, and more.

Flash memory uses a phenomenon (known as Fowler-Nordheim tunneling) to send electrons through a floating gate transistor where they remain even after power is turned off. Flash memory allows users to store data. It is sold in a variety

Flash memory gets its name from the technique used to erase its data. To reset all values, a strong electrical field, called a flash, is applied to the entire card.

Magnetic Disk Drive Access Times

There can be as many as three factors that contribute to the time required to access a file:

- Seek time,
- Search time
- Transfer time.

Seek time is the time required to position the read/write head on the proper track. Seek time doesn't apply to devices with fixed read/write heads because each track has its own read/write head.

Search time, also known as rotational delay, is the time it takes to rotate the disk until the requested record is moved under the read/write head.

Transfer time is the fastest of the three; that's time taken to transfer the data from secondary storage to main memory.

Fixed-Head Drives

Fixed-head disk drives are fast. The total amount of time required to access data depends on the rotational speed, which varies from device to device theTotal access time is the (sum of search time plus transfer time)

search time (rotational delay)
+ transfer time (data transfer)
access time

Because the disk rotates continuously, there are three basic positions for the requested record in relation to the position of the read/write head

- Best possible situation - the record is next to the read/write head ; this gives a rotational delay of zero.
- The average situation - the record is directly opposite the read/write head; this gives a rotational delay of t/2 where t (time) is one full rotation.
- The worst situation - the record has just rotated past the read/write head; this gives a rotational delay of t because it will take one full rotation

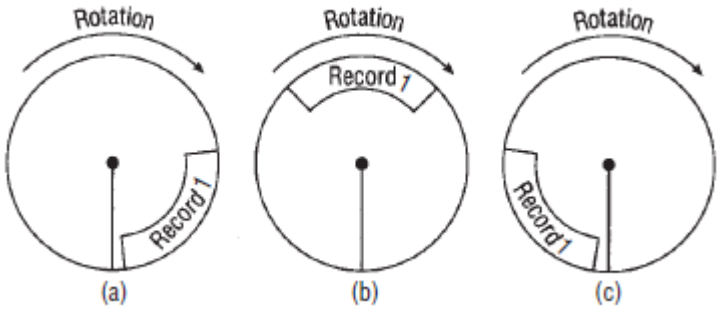


Figure 7.6 Best,Average and Worst situation of access time

If one complete revolution takes 16.8 ms, then the average rotational delay is 8.4 ms. The data-transfer time varies from device to device, but a typical value is 0.00094 ms per byte.

Benchmarks Access Time:

Maximum access 16.8 ms + 0.00094 ms/byte

Average access 8.4 ms + 0.00094 ms/byte

Sequential access Depends on the length of the record; generally less than 1 ms. If we were to read 10 records individually, we would multiply the access time for a single record by 10:

Access time = 8.4 + 0.094 = 8.494 ms

for one record **total access time = 10(8.4 + 0.094) = 84.940 ms** for 10 records

Table 5.2 Access times for a fixed-head disk drive at 16.8ms/revolution

Benchmarks	Access Time
Maximum access	50 ms + 16.8 ms + 0.00094 ms/byte
Average access	25 ms + 8.4 ms + 0.00094 ms/byte
Sequential access	Depends on the length of the record, generally less than 1 ms

On the other hand, to read one block of 10 records we would make a single access:

Access time = 8.4 + (0.094 * 10) = 8.4 + 0.94 = 9.34 ms for 10 records in one block

Movable-Head Devices

Movable-head disk drives add a third time element to the computation of access time. Seek time is the time required to move the arm into position over the proper track.

Access time :

seek time (arm movement)
search time (rotational delay)
\+ transfer time (data transfer)
access time

Seek time is the longest. We'll examine several seek strategies in a moment. The maximum seek time, which is the maximum time required to move the arm, is typically 50ms.

Table 5.3 Typical access time movable head drives

Benchmarks	Access Time
Maximum access	50 ms + 16.8 ms + 0.00094 ms/byte
Average access	25 ms + 8.4 ms + 0.00094 ms/byte
Sequential access	Depends on the length of the record, generally less than 1 ms

Benchmarks Access Time :

Maximum access 50 ms + 16.8 ms + 0.00094 ms/byte

Average access 25 ms + 8.4 ms + 0.00094 ms/byte

Sequential access Depends on the length of the record, generally less than 1 ms

Access time = 25 + 8.4 + 0.094 = 33.494 ms for one record

Total access time = 10 * 33.494 = 334.94 ms for 10 records

(about 1⁄3 of a second)

Components of the I/O Subsystem

The channel in I/O Subsystem keeps up the I/O requests from the CPU and pass them down the line to the appropriate control unit. The control units play a part of the mechanics.

I/O channels are programmable units placed between the CPU and the control units. Their job is to synchronize the fast speed of the CPU with the slow speed of the I/O device and they make it possible to overlap I/O operations with processor operations so the CPU and I/O can process concurrently. Channels use I/O channel programs..

The channel sends one signal for each function, and the I/O control unit interprets the signal, which might say “go to the top of the page” if the device is a printer or “rewind” if the device is a tape drive.

Although a control unit is sometimes part of the device, in most systems a single control unit is attached to several similar devices, so we distinguish between the control unit and the device.

Some systems also have a disk controller, or disk drive interface used to link the disk drives with the system bus. Disk drive interfaces control the transfer of information between the disk drives and the rest of the computer system.

At the start of an I/O command, the information passed from the CPU to the channel is this:

- I/O command (READ, WRITE, REWIND, etc.)
- Channel number
- Address of the physical record to be transferred (from or to secondary storage)
- Starting address of a memory buffer from which or into which the record is to be transferred

Because the channels are as fast as the CPU they work with, each channel can direct several control units by interleaving commands.

One channel and up to eight control units, each of which communicates with up to eight I/O devices. Channels are often shared because they're the most expensive items in the entire I/O subsystem.

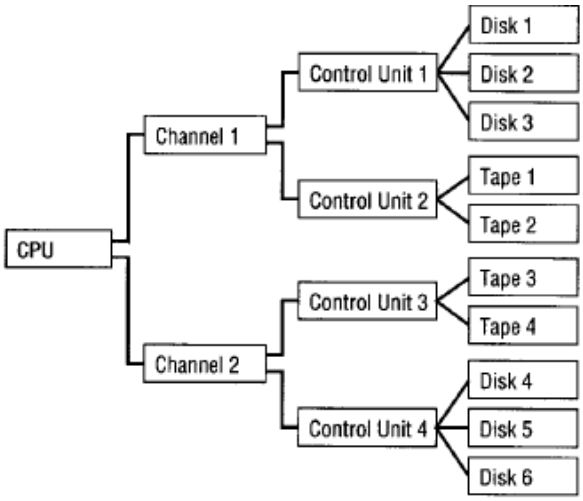


Figure 7.7 A Typical I/O Subsystem Configuration

Additional flexibility can be built into the system by connecting more than one channel to a control unit or by connecting more than one control unit to a single device.

These multiple paths increase the reliability of the I/O subsystem by keeping communication lines open even if a component malfunctions.

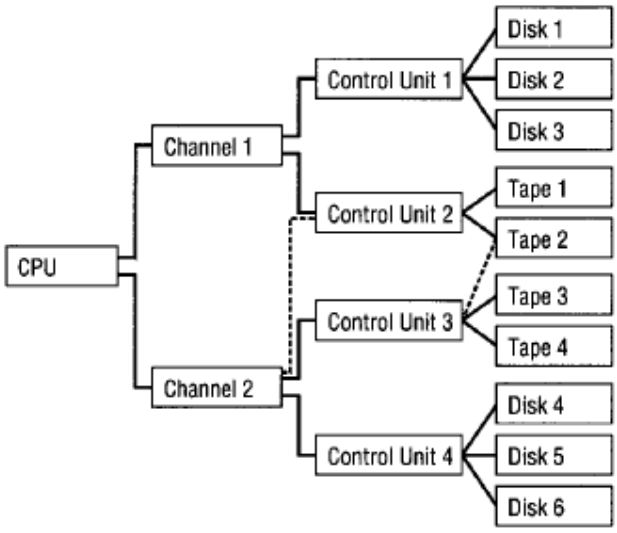


Figure 5.8 I/O Subsystem Configuration with multiple paths

Communication Among Devices

The Device Manager relies on several auxiliary features to keep running efficiently under the demanding conditions of a busy computer system, and there are three problems that must be resolved:

- It needs to know which components are busy and which are free.
- It must be able to accommodate the requests that come in during heavy I/O traffic.
- It must accommodate the disparity of speeds between the CPU and the I/O devices.

The first is solved by structuring the interaction between units.

1. The success of the operation depends on the system's ability to know when a device has completed an operation.
2. This is done with a hardware flag that must be tested by the CPU. This flag is made up of three bits and resides in the Channel Status Word (CSW).
3. It is in a predefined location in main memory and contains information indicating the status of the channel.
4. Each bit represents one of the components of the I/O subsystem, one each for the channel, control unit, and device. Each bit is changed from 0 to 1 to indicate that the unit has changed from free to busy.
5. Each component has access to the flag, which can be tested before proceeding with the next I/O operation to ensure that the entire path is free and vice versa.

There are two common ways to perform this test—**polling and using interrupts.**

Polling uses a special machine instruction to test the flag. For example, the CPU periodically tests the channel status bit (in the CSW). If the channel is still busy, the CPU performs some other processing task until the test shows that the channel is free.

The major disadvantage with this scheme is determining how often the flag should be polled. If polling is done too frequently, the CPU wastes time testing the flag to find out that the channel is still busy. On the other hand, if polling is done too rarely, the channel could sit idle for long periods of time.

Interrupts are a more efficient way to test the flag. A hardware mechanism does the test as part of every machine instruction executed by the CPU.

If the channel is busy, the flag is set so that execution of the current sequence of instructions is automatically interrupted and control is transferred to

the interrupt handler, which is part of the operating system and resides in a predefined location in memory.

The interrupt handler must find out which unit sent the signal, analyze its status, restart it when appropriate with the next operation, and finally return control to the interrupted process.

Some sophisticated systems are equipped with hardware that can distinguish between several types of interrupts. These interrupts are ordered by priority, and each one can transfer control to a corresponding location in memory. The memory locations are ranked in order according to the same priorities.

Direct memory access (DMA) is an I/O technique that allows a control unit to directly access main memory. This means that once reading or writing has begun, the remainder of the data can be transferred to and from memory without CPU intervention.

However, it is possible that the DMA control unit and the CPU compete for the system bus if they happen to need it at the same time.

To activate this process, the CPU sends enough information—such as

- The type of operation (read or write),
- The unit number of the I/O device needed,
- The location in memory where data is to be read from or written to
- The amount of data (bytes or words) to be transferred to the DMA control unit to initiate the transfer of data;
- The CPU then can go on to another task while the control unit completes the transfer independently.

The DMA controller sends an interrupt to the CPU to indicate that the operation is completed. This mode of data transfer is used for high-speed devices such as disks.

Without DMA, the CPU is responsible for the physical movement of data between main memory and the device—a time-consuming task that results in significant overhead and decreased CPU utilization.

Buffers are used extensively to better synchronize the movement of data between the relatively slow I/O devices and the very fast CPU.

Buffers are temporary storage areas residing in three convenient locations throughout the system:

- Main memory
- Channels
- Control units.

They're used to store data read from an input device before it's needed by the processor and to store data before send to an output device. A typical use of buffers occurs when blocked records are either read from, or written to, an I/O device.

To minimize the idle time for devices and, even more important, to maximize their throughput, the technique of double buffering is used, as shown in Figure 7.9.

In this system, two buffers are present in main memory, channels, and control units. The objective is to have a record ready to be transferred to or from

memory at any time to avoid any possible delay. Thus, while one record is being processed by the CPU, another can be read or written by the channel.

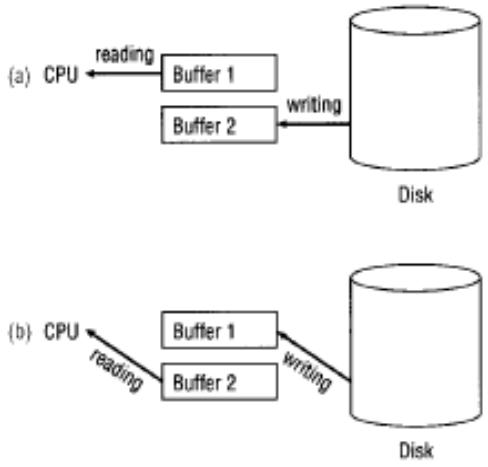


Figure 7.9 Example of double buffering (a) the CPU is reading from Buffer 1 as Buffer 2 is being filled.(b) Once Buffer 2 is filled, it can be read quickly by the CPU while Buffer 1 is being filled again

When using blocked records, after receiving the command to "READ last logical record," the channel can start reading the next physical record. When the first READ command is received, two records are transferred from the device to immediately fill both buffers.

While the data from one buffer has been processed, the second buffer is ready. As the second is being read, the first buffer is being filled with data from a third record, and so on.

Management of I/O Requests

The Device Manager divides the I/O request task into three parts with each one handled by a specific software component of the I/O subsystem.

- The I/O traffic controller watches the status of all devices, control units, and channels.
- The I/O scheduler implements the policies that determine the allocation, and access of the devices, control units, and channels.
- The I/O device handler performs the actual transfer of data and processes the device interrupts.

The traffic controller has three main tasks:

1. It must determine if there's at least one path available;
2. If there's more than one path available, it must determine which to select;
3. If the paths are all busy, it must determine when one will become available.

To do all this, the traffic controller maintains a database containing the status and connections for each unit in the I/O subsystem, grouped into Channel Control Blocks, Control Unit Control Blocks, and Device Control Blocks, as shown in Table 5.4.

Table 5.4 Information of Each Control Block

Channel Control Block	Control Unit Control Block	Device Control Block
• Channel identification	• Control unit identification	• Device identification
• Status	• Status	• Status
• List of control units connected to it	• List of channels connected to it	• List of control units connected to it
• List of processes waiting for it	• List of devices connected to it	• List of processes waiting for it
	• List of processes waiting for it	

To choose a free path to satisfy an I/O request, the traffic controller traces backward from the control block of the requested device through the control units to the channels.

If a path is not available, the process linked to the queues for control blocks of the requested device, control unit, and channel. Later, when a path becomes available, the traffic controller quickly selects the first PCB from the queue for that path.

The **I/O scheduler** performs the same job as the Process Scheduler, it allocates the devices, control units, and channels. Under heavy loads, when the number of requests is greater than the number of available paths, the I/O scheduler must decide which request to satisfy first.

Some systems allow the I/O scheduler to give preference to I/O requests from high-priority programs. The I/O scheduler must synchronize its work with the traffic controller to make sure that a path is available to satisfy the selected I/O requests.

The **I/O device handler** processes the I/O interrupts, handles error conditions, and provides detailed scheduling algorithms, which are extremely device dependent. Each type of I/O device has its own device handler algorithm.

Device Handler Seek Strategies

A seek strategy for the I/O device handler is the predetermined policy that the device handler uses to allocate access to the device among the many processes that may be waiting for it. It determines the order in which the processes get the device. The goal is to keep the seek time to a minimum.

The most commonly used seek strategies

- First-Come First-Served (FCFS)
- Shortest Seek Time First (SSTF)
- SCAN and its variations LOOK, N-Step SCAN, C-SCAN, and C-LOOK.

Every scheduling algorithm should do the following:

- Minimize arm movement
- Minimize mean response time
- Minimize the variance in response time

First-come, first-served (FCFS)

- It is the simplest device-scheduling algorithm;
- It is easy to program and essentially fair to users.
- However, on average, it doesn't meet any of the three goals of a seek strategy.

To illustrate, consider a single-sided disk with one recordable surface where the tracks are numbered from 0 to 49. It takes 1 ms to travel from one track to the next adjacent one.

For Example, take a disk with tracks numbered from 0 to 49 it is currently retrieving data from Track 15, the following list of requests has arrived: Tracks 4, 40, 11, 35, 7, and 14.

The path of the read/write head looks is shown in Figure 7.10.

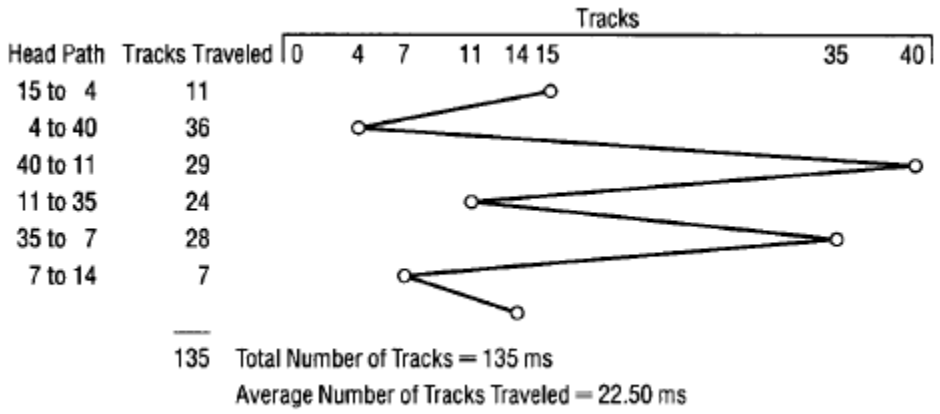


Figure 7.10 Disk arm movements in FCFS

It takes a long time, 135 ms, to satisfy the entire series of requests, search time and data transfer. FCFS has an obvious disadvantage of extreme arm movement: from 15 to 4, up to 40, back to 11, up to 35, back to 7, and, finally, up to 14.

Remember, seek time is the most time-consuming of the three functions performed here, so any algorithm that can minimize it is preferable to FCFS.

Shortest seek time first (SSTF)

The shortest jobs are processed first and longer jobs are made to wait. With SSTF, the request with the track closest to the one being served is the next to be satisfied, thus minimizing overall seek time.

Figure 7.11 shows what happens to the same track requests that took 135 ms to service using FCFS;

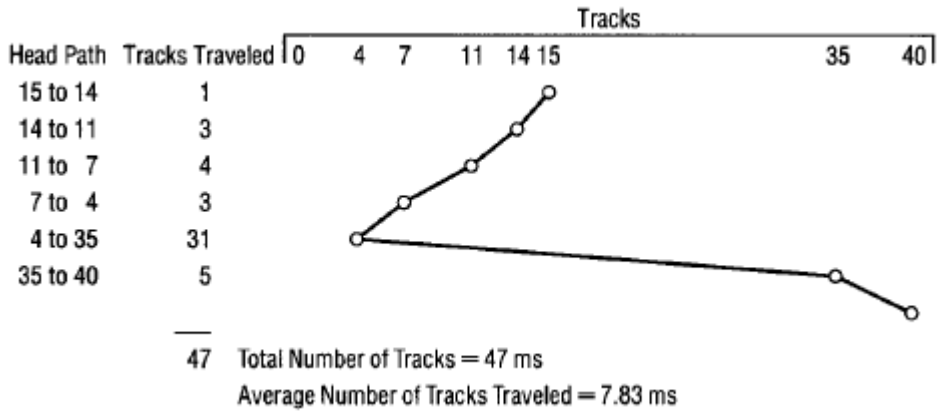


Figure 7.11 Disk arm movement in SSTF

It took 47 ms to satisfy all requests—which is about one third of the time required by FCFS. That's a substantial improvement. But SSTF has its disadvantages. Remember that the Shortest Job Next (SJN) process scheduling algorithm had a tendency to favor the short jobs.. The same holds true for SSTF.

With SSTF, the system notes that Track 13 is closer to the arm's present position than the older request for Track 7, so Track 13 is handled first. The next closest is Track 16, so off it goes—moving farther and farther away from Tracks 7 and 1.

In fact, during periods of heavy loads, the arm stays in the center of the disk, where it can satisfy the majority of requests easily and it ignores those on the outer edges of the disk.

SCAN

The algorithm moves the arm from the outer to the inner track, servicing every request in its path. When it reaches the innermost track, it reverses direction and moves toward the outer tracks, again servicing every request in its path.

The most common variation of SCAN is LOOK, sometimes known as the elevator algorithm, in which the arm doesn't necessarily go to either edge. It moves to the edge only if there are requests there.

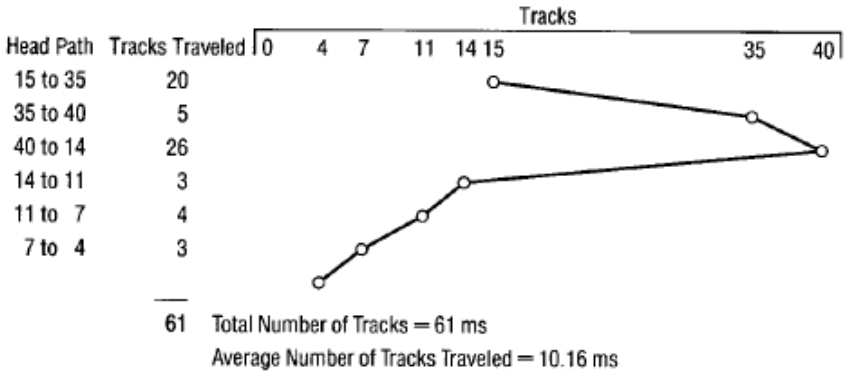


Figure 7.12 The LOOK algorithm makes the arm move systematically from the first requested track at one edge of the disk to the last requested track at the other edge. In this example, all track requests are on the wait queue.

It took 61 ms to satisfy all requests, 14 ms more than with SSTF. But as requests arrive, each is incorporated in its proper place in the queue and serviced when the arm reaches that track.

This eliminates a great deal of arm movement and saves time in the end. In fact, SCAN meets all three goals for seek strategies. Variations of SCAN, in addition to LOOK, are N-Step SCAN, C-SCAN, and C-LOOK.

N-Step SCAN holds all new requests until the arm starts on its way back. Any requests that arrive while the arm is in motion are grouped for the arm's next sweep.

With **C-SCAN** (an abbreviation for Circular SCAN), the arm picks up requests on its path during the inward sweep. When the innermost track has been

reached, it immediately returns to the outermost track and starts servicing requests that arrived during its last inward sweep.

However, there are many requests at the far end of the disk and these have been waiting the longest. Therefore, C-SCAN is designed to provide a more uniform wait time.

C-LOOK is an optimization of C-SCAN, just as LOOK is an optimization of SCAN. In this algorithm, the sweep inward stops at the last high-numbered track request, so the arm doesn't move the last track. In addition, the arm doesn't necessarily return to the end track; it returns only to the lowest-numbered track that's requested.

Which strategy is best? It's up to the system designer to select the best algorithm for each environment.

- FCFS works well with light loads; but as soon as the load grows, service time becomes unacceptably long.
- SSTF is quite popular and intuitively appealing. It works well with moderate loads but has the problem of localization under heavy loads.
- SCAN works well with light to moderate loads and eliminates the problem of indefinite postponement. SCAN is similar to SSTF in throughput and means service times.
- C-SCAN works well with moderate to heavy loads and has a very small variance in service times.

The best scheduling algorithm for a specific computing system may be a combination of more than one scheme. For instance, it might be a combination of

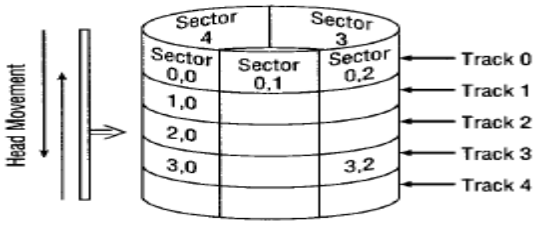
two schemes: SCAN or LOOK during light to moderate loads, and C-SCAN or C-LOOK during heavy load times.

Search Strategies: Rotational Ordering

To optimize search times by ordering the requests once the read/write heads have been positioned. This search strategy is called rotational ordering.

To help illustrate the Rotational ordering, let's consider a virtual cylinder with a movable read/write head.

For this example, we'll assume that the cylinder has only five tracks, numbered 0 through 4, and that each track contains five sectors, numbered 0 through 4. We'll take the requests in the order in which they arrive.



Request List	
Track	Sector
0	1
1	4
1	3
2	0
2	3
2	4
3	2
3	0

Figure 7.13 This movable-head cylinder takes 5 ms to move the read/write head from one track to the next. The read/write head is initially positioned at Track 0, Sector 0. It takes 5 ms to rotate the cylinder from Sector 0 to Sector 4 and 1 ms to transfer one sector from the cylinder to main memory.

Each request is satisfied as it comes in. If the requests are ordered within each track so that the first sector requested on the second track is the next number higher than the one just served, rotational delay will be minimized

To properly implement this algorithm, the device controller must provide rotational sensing so the device driver can see which sector is currently under the

read/write head. Under heavy I/O loads, this kind of ordering can significantly increase throughput, especially if the device has fixed read/write heads rather than movable heads.

Conclusion

The Device Manager's job is to manage every system device as effectively as possible despite the unique characteristics of Balancing the demand for these devices is a complex task that's divided among several devices. The devices have varying speeds and degrees of sharability; some can handle direct access and some only sequential access. For magnetic media, they can have one or many read/write heads, and the heads can be in a fixed position for optimum speed or able to move across the surface for optimum storage space.

For optical media, the Device Manager tracks storage locations and adjusts the disc's speed accordingly so data is recorded and retrieved correctly. For flash memory, the Device Manager tracks every USB device and assures that data is sent and received correctly. hardware components: channels, control units, and the devices themselves. The success of the I/O subsystem depends on the communications that link these parts.

Exercise

1. Name three examples of secondary storage media other than hard disks.

2. Describe how primary storage differs from secondary storage.

3. Explain the differences between buffering and blocking.

4. Given the following characteristics for a disk pack with 10 platters yielding

18 recordable surfaces:

Rotational speed = 10 ms

Transfer rate = 0.1 ms/track

Density per track = 19,000 bytes

Number of records to be stored = 200,000 records

Size of each record = 160 bytes

Block size = 10 logical records

Number of tracks per surface = 500

Find the following:

a. Number of blocks per track
b. Waste per track
c. Number of tracks required to store the entire file
d. Total waste to store the entire file
e. Time to write all of the blocks (Use rotational speed; ignore the time it takesto move to the next track.)
f. Time to write all of the records if they're not blocked. (Use rotational speed; ignore the time it takes to move to the next track.)
g. Optimal blocking factor to minimize waste
h. What would be the answer to (e) if the time it takes to move to the next track were 5 ms?
i. What would be the answer to (f) if the time it takes to move to the next track were 5 ms?

5. Given that it takes 1 ms to travel from one track to the next, and that the arm is originally positioned at Track 15 moving toward the low-numbered tracks, and you are using the LOOK scheduling policy, compute how long it will take to satisfy the following requests—4, 40, 35, 11, 14, and 7. All requests are present in the wait queue. (Ignore rotational time and transfer time; just consider seek time.)

CHAPTER 8

FILE MANAGEMENT

This chapter shows how files are organized logically, how they're stored physically, how they're accessed, and who is allowed to access them. It also specifies the interaction between the File Manager and the Device Manager.

The efficiency of the File Manager depends on Organization of files are organized (sequential, direct, or indexed sequential) Storage method of files (contiguously, non contiguously, or indexed) File's structure of records are structured (fixed-length or variable length).

The File Manager

The File Manager (also called the file management system) is the software responsible for creating, deleting, modifying, and controlling access to files—as well as for managing the resources used by the files. These functions are performed in collaboration with the Device Manager.

Responsibilities of the File Manager

To carry out its responsibilities, it must perform these four tasks:

1. Keep track of where each file is stored.
2. Use a policy that will determine where and how the files will be stored, making sure to efficiently use the available storage space and provide efficient access to the files.
3. Allocate each file when a user has been cleared for access to it, then record its use.

4. Deallocate the file when the file is to be returned to storage, and communicate its availability to others who may be waiting for it.

Definitions

A **field** is a group of related bytes that can be identified by the user with a name, type, and size. A record is a group of related fields.

A **file** is a group of related records that contains information to be used by specific application programs to generate reports.

A **database** is groups of related files that are interconnected at various levels to give users flexibility of access to the data stored. If the user's database requires a specific structure, the File Manager must be able to support it.

Program files contain instructions and data files contain data; but as far as storage is concerned, the File Manager treats them exactly the same way.

Directories are special files with listings of filenames and their attributes.

Interacting with the File Manager

The user communicates with the File Manager, which responds to specific commands like OPEN, DELETE, RENAME, and COPY.

The first time a user gives the command to save a file, it's actually the CREATE command. In other operating systems, the OPEN NEW command within a program indicates to the File Manager that a file must be created.

To access a file, the user doesn't need to know Its exact physical location on the disk pack (the cylinder, surface, and sector), The medium in which it's stored

(archival tape, magnetic disk, optical disc, or flash storage), The network specifics.

File access is a complex process. Each logical command is broken down into a sequence of signals that trigger the step-by-step actions performed by the device and supervise the progress of the operation by testing the device's status.

For example, when a user's program issues a command to read a record from a disk the READ instruction has to be decomposed into the following steps:

1. Move the read/write heads to the cylinder or track where the record is to be found
2. Wait for the rotational delay until the sector containing the desired record passes under the read/write head.
3. Activate the appropriate read/write head and read the record.
4. Transfer the record to the main memory.
5. Set a flag to indicate that the device is free to satisfy another request.

While all of this is going on, the system must check for possible error conditions. The File Manager does all of this, Without the File Manager, every program would need to include instructions to operate all of the different types of devices and every model within each type.

Typical Volume Configuration

Normally the active files for a computer system reside on secondary storage units. Some devices accommodate removable storage units—such as CDs, DVDs, floppy disks, USB devices, and other removable media. Storage units, such as hard disks and disk packs are non removable.

Each storage unit, whether it's removable or not, is considered a volume, and each volume can contain several files, so they're called "multi file volumes."

However, some files are extremely large and are contained in several volumes; not surprisingly, these are called "multi volume files." Each volume in the system is given a name. The File Manager writes this name and other descriptive information

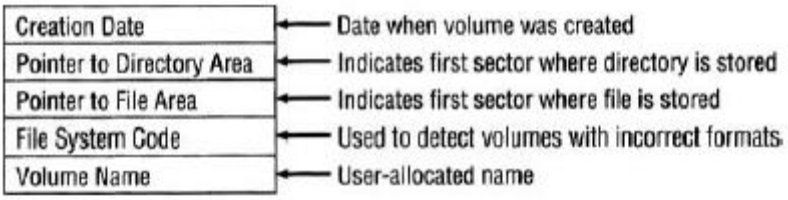


Figure 8.1 The volume descriptor, which is stored at the beginning of each volume,includes this vital information about the storage unit.

Figure 8.1 The volume descriptor, which is stored at the beginning of each volume. The master file directory (MFD) is stored immediately after the volume descriptor and lists the names and characteristics of every file contained in that volume. The remainder of the volume is used for file storage.

The first operating systems supported only a single directory per volume. This directory was created by the File Manager and contained the names of files, usually organized in alphabetical, spatial, or chronological order.

Although it was simple to implement and maintain, this scheme had some major disadvantages:

- It would take a long time to search for an individual file, especially if the MFD was organized in an arbitrary order.

- If the user had more than 256 small files stored in the volume, the directory space would fill up before the disk storage space filled up.
- Users couldn't create subdirectories to group the files that were related.
- Multiple users couldn't safeguard their files from other users because the entire directory was freely made available to every user in the group on request.

Each program in the entire directory needed a unique name, even those directories serving many users, so only one person using that directory could have a program named Program1.

Introducing Subdirectories

File Managers create an MFD for each volume that can contain entries for both files and subdirectories. A subdirectory is created when a user opens an account to access the computer system.

Although this user directory is treated as a file, its entry in the MFD is flagged to indicate to the File Manager

Today's File Managers encourage users to create their own subdirectories, so related files can be grouped together. Many computer users and some operating systems call these subdirectories "folders." This structure is an extension of the previous two-level directory organization, and it's implemented as an upside-down tree, as shown in Figure 5.2.

Tree structures allow the system to efficiently search individual directories because there are fewer entries in each directory. However, the path to the

requested file may lead through several directories. When the user wants to access a specific file, the filename is sent to the File Manager.

The File Manager first searches the MFD for the user's directory. It then searches the user's directory and any subdirectories for the requested file. Each file entry in every directory contains information describing the file; It's called the file descriptor. Information typically included in a file descriptor includes the following:

- Filename—within a single directory, filenames must be unique; in some operating systems, the filenames are case sensitive
- File type—the organization and usage that are dependent on the system
- File size—although it could be computed from other information, the size is kept here for convenience
- File location—identification of the first physical block (or all blocks) where the file is stored
- Date and time of creation
- Owner
- Protection information—access restrictions, based on who is allowed to access the file and what type of access is allowed
- Record size—its fixed size or its maximum size, depending on the type of record

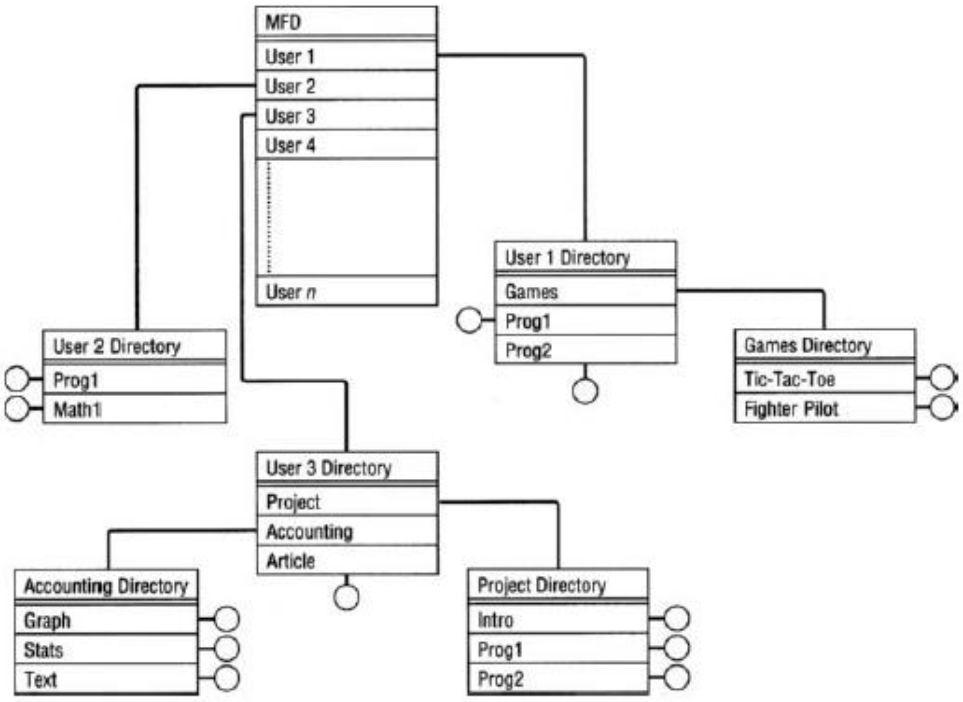


Figure 8.2 File Directory Tree Structure

File-Naming Conventions

A file's name can be much longer than it appears. Depending on the File Manager, it can have from two to many components like relative filename and an extension.

The term "complete filename" can be used to identify the file's absolute filename (that's the long name that includes all path information), and "relative filename" to indicate the name without path information..

Generally, the relative filename can vary in length from one to many characters and can include letters of the alphabet, as well as digits.

Most operating systems allow names with dozens of characters including spaces, hyphens, underlines, and certain other keyboard characters.

Some operating systems require an extension that's appended to the relative filename. It's usually two or three characters long and is separated from the relative name by a period, to identify the type of file or its contents.

To open a file with an unrecognized extension, Windows asks the user to choose an application to associate with that type of file.

Some extensions (such as EXE, BAT, COB, and FOR) are restricted by certain operating systems. There may be other components required for a file's complete name. Using a Windows operating system and a personal computer with three disk drives, the file's complete name is composed of its relative name and extension, preceded by the drive label and directory name:

C:\IMFST\FLYNN\INVENTORY_COST.DOC

This indicates to the system that the file INVENTORY_COST.DOC requires a word processing application program, and it can be found in the directory; IMFST; subdirectory FLYNN in the volume residing on drive C.

Why don't users see the complete file name when accessing a file?

First, the File Manager selects a directory for the user where all operations requested by that user start from this "home" or "base" directory.

Second, from this home directory, the user selects a subdirectory, which is called a current directory or working directory.

Thereafter, the files are presumed to be located in this current directory. Whenever a file is accessed, the user types in the relative name, and the File Manager adds the proper prefix.

As long as users refer to files in the working directory, they can access their files without entering the complete name.

The concept of a current directory is based on the underlying hierarchy of a tree structure and allows programmers to retrieve a file by typing only its relative filename…

INVENTORY_COST.DOC

…and not its complete filename:

C:\IMFST\FLYNN\INVENTORY_COST.DOC

File Organization

File organization is the arrangement of records within a file because all files are composed of records. When a user gives a command to modify the contents of a file, it's actually a command to access records within the file.

Record Format

All files are composed of records. Within each file, the records are all presumed to have the same format: they can be of fixed length or of variable length. And these records, regardless of their format, can be blocked or not blocked.

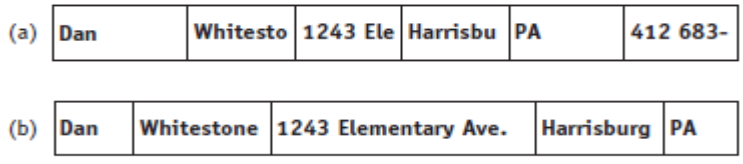


Figure 8.3 When data is stored in fixed length fields (a), data that extends beyond the fixed size is truncated. (b), the size expands to fit the contents, but it takes longer to access it.

Fixed-length records are the most common because they're the easiest to access directly and ideal for data files. The critical aspect of fixed-length records is the size of the record. If it's too small—smaller than the number of characters to be stored in the record—the leftover characters are truncated. But if the record size is too large— larger than the number of characters to be stored—storage space is wasted.

Variable-length records don't leave empty storage space and don't truncate any characters but are difficult to access directly because it's hard to calculate exactly where the record is located.

The amount of space that's actually used to store the supplementary information varies from system to system and conforms to the physical limitations of the storage medium.

Physical File Organization

The physical organization of a file is the way in which the records are arranged and the characteristics of the medium used to store it.

On magnetic disks (hard drives), files can be organized in one of several ways: sequential, direct, or indexed sequential. To select the best of these file organizations, the programmer or analyst usually considers these practical characteristics:

Volatility of the data—the frequency with which additions and deletions are made

Activity of the file—the percentage of records processed during a given run

Size of the file

Response time—the amount of time the user is willing to wait before the requested operation is completed

Sequential record organization is the easiest to implement because records are stored and retrieved serially, one after the other. To find a specific record, the file is searched from its beginning until the requested record is found.

To speed the process.

A key field from the record selected and then sorts the records by that field.
When a user requests a specific record, the system searches only the key field of each record in the file.
The search is ended when either an exact match is found or the key field for the requested is not found.

A direct record organization uses direct access files, which, of course, can be implemented only on direct access storage devices. In these files any record can be accessed in any order. So it's also known as "random organization," and its files are called "random access files."

Records are identified by their relative addresses—known as logical addresses are computed when the records are stored and then again when the records are retrieved.

In this method the user identifies a field and designates it as the key field because it uniquely identifies each record. The program used to store the data follows a set of instructions, called a hashing algorithm, that transforms each key into a number: the record's logical address.

This is given to the File Manager, which takes the necessary steps to translate the logical address into a physical address. The same procedure is used to retrieve a record.

The problem with hashing algorithms is that several records with unique keys (such as customer numbers) may generate the same logical address—and then there's a collision. Records that collide are stored in an overflow area that was set aside when the file was created.

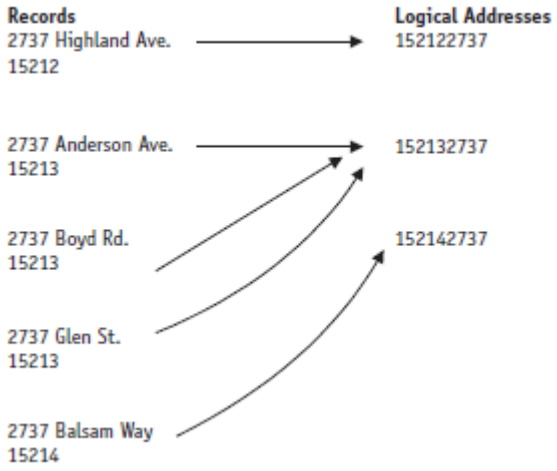


Figure 8.4 Hashing algorithm

Indexed sequential record organization combines the best of sequential and direct access. It's created and maintained through an Indexed Sequential Access Method (ISAM) application, which removes the burden of handling overflows and preserves record order from the shoulders of the programmer.

This type of organization doesn't create collisions because it doesn't use the result of the hashing algorithm to generate a record's address.

Instead, it uses this information to generate an index file through which the records are retrieved. This organization divides an ordered sequential file into blocks of equal size. Their size is determined by the File Manager. Each entry in the index file contains the highest record key and the physical location of the data block, and the records with smaller keys, are stored.

For most dynamic files, indexed sequential is the choice for organization because it allows both direct access to a few requested records and sequential access to many.

Physical Storage Allocation

The File Manager must work with files not just as whole units but also as logical units or records. Records within a file must have the same format, but they can vary in length.

In turn, records are subdivided into fields. In most cases, their structure is managed by application programs and not the operating system. An exception is made for those systems that are heavily oriented to database applications, where the File Manager handles field structure.

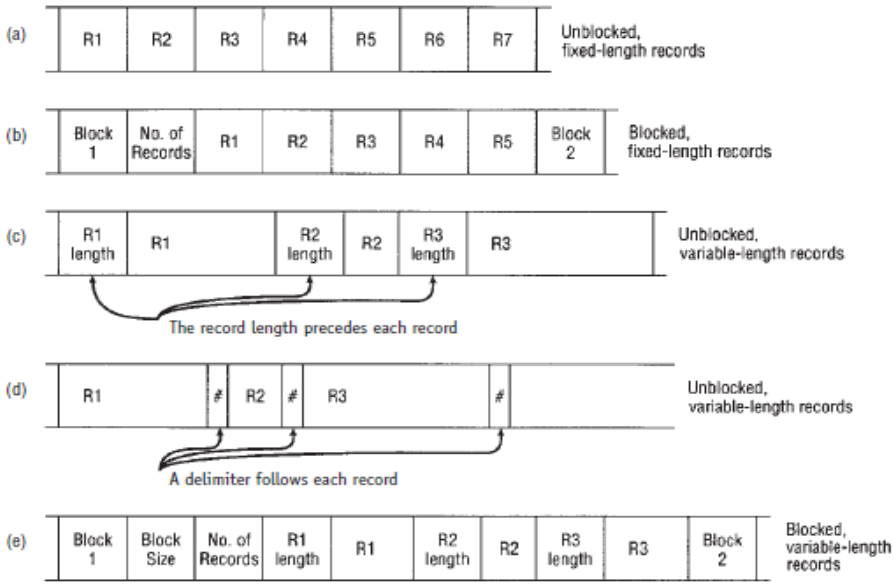


Figure 8.5 Every record in a file must have the same format but can be of different sizes, as shown in these five examples of the most common record formats. The supplementary information in (b),(c),(d) and (e) is provided by the file manager, when the record is stored

Contiguous Storage

When records use contiguous storage, they're stored one after the other. This was the scheme used in early operating systems.

Advantages

- It's very simple to implement and manage.
- Ease of direct access because every part of the file is stored in the same compact area.

Disadvantages:

- A file can't be expended unless there's empty space available immediately following it.
- Fragmentation, which can be overcome by compacting and rearranging files.

Free Space	File A Record 1	File A Record 2	File A Record 3	File A Record 4	File A Record 5	File B Record 1	File B Record 2	File B Record 3	File B Record 4	Free Space	File C Record 1

Figure 8.6 with contiguous file storage, File A can't be expanded without being rewritten to a larger storage area. File B can be expanded, by only one record replacing the free space preceding File C.The File Manager keeps track of the empty storage areas by treating them as files— they're entered in the directory but are flagged to differentiate them from real files. Usually the directory is kept in order by sector number, so adjacent empty areas can be combined into one large free space.

Non-contiguous Storage

Non-contiguous storage allocation allows files to use any storage space available on the disk and is linked together with pointers. The physical size of each extent is determined by the operating system and is usually 256—or another power of two—bytes.File extents are usually linked in one of two ways. Linking can take place at the storage level, where each extent points to the next one in the sequence.

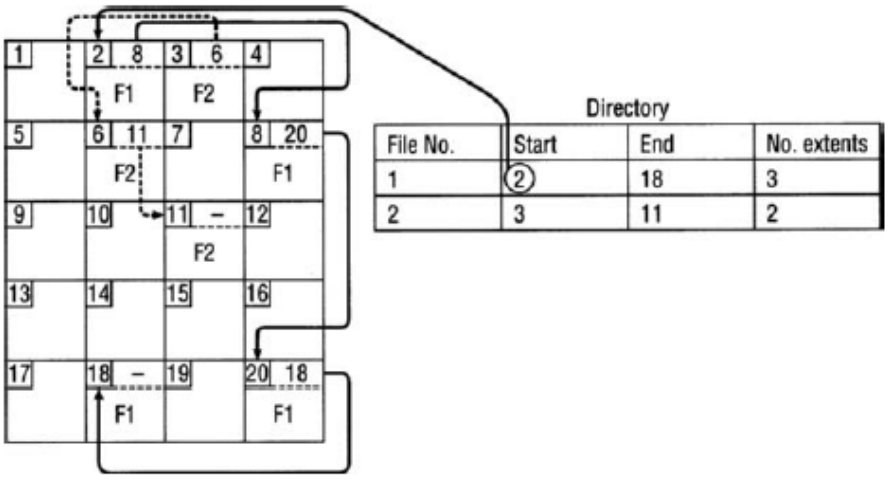


Figure 8.7 Non Contiguous file storage with link taking place at the storage file 1 starts in address 2.

The alternative is for the linking to take place at the directory level. Although both noncontiguous allocation schemes eliminate external storage fragmentation and the need for compaction, they don't support direct access because there's no easy way to determine the exact location of a specific record.

Files are usually declared to be either sequential or direct when they're created, so the File Manager can select the most efficient method of storage allocation: contiguous for direct files and noncontiguous for sequential.

Indexed Storage

Indexed storage allocation allows direct record access by bringing together the pointers linking every extent of that file into an index block.

Every file has its own index block, which consists of the addresses of each disk sector that makeup the file. The index lists each entry in the same order in which the sectors are linked

When a file is created, the pointers in the index block are all set to null. Then each sector is filled, the pointer is set to the appropriate sector address.

This scheme supports both sequential and direct access, but it doesn't necessarily improve the use of storage space because each file must have an index block

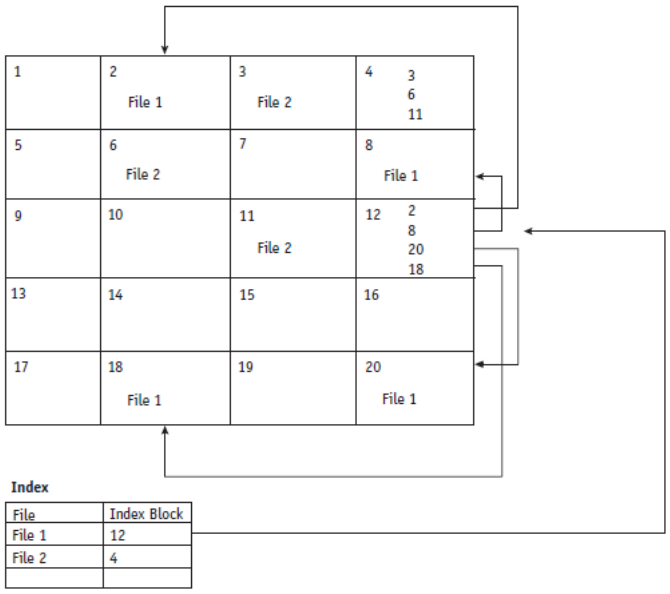


Figure8.8 Indexed storage allocation with a one-level index, allowing direct access to each record for the files

For larger files with more entries, several levels of indexes can be generated; in which case, to find a desired record, the File Manager accesses the first index (the highest level), which points to a second index (lower level), which points to an even lower-level index and eventually to the data record.

Access Methods

Access methods are dictated by a file's organization; the most flexibility is allowed with indexed sequential files and the least with sequential. A file that has been organized in sequential fashion can support only sequential access to its records, and these records can be of either fixed or variable length.

The File Manager uses the address of the last byte read to access the next sequential record. Therefore, the current byte address (CBA) must be updated every time a record is accessed, such as when the READ command is executed.

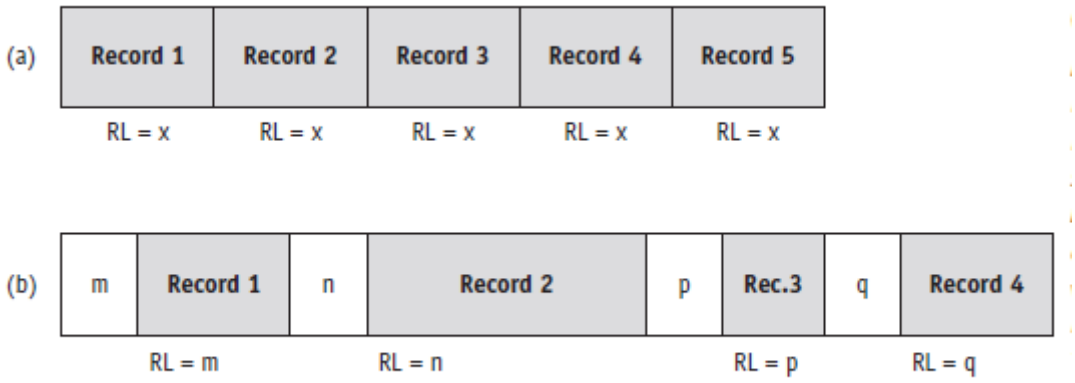


Figure8.9 Fixed versus Variable length records (a)Fixed length record (b) Variable length records

Sequential Access

For sequential access of fixed-length records, the CBA is updated simply by incrementing it by the record length (RL), which is a constant:

$$CBA = CBA + RL$$

For sequential access of variable-length records, the File Manager adds the length of the record (RL) plus the number of bytes used to hold the record length to the CBA.

$$CBA = CBA + N + RL$$

Direct Access

If a file is organized in direct fashion, In the case of direct access with fixed-length records, the CBA can be computed directly from the record length and the desired record number RN minus 1:

$$CBA = (RN - 1) * RL$$

For example, if we're looking for the beginning of the eleventh record and the fixed record length is 25 bytes, the CBA would be:

$$CBA=(11 – 1) * 25 = 250$$

However, if the file is organized for direct access with variable-length records, it's virtually impossible to access a record directly because the address of the desired record can't be easily computed. Therefore, to access a record, the File Manager must do a sequential search through the records.

An alternative is for the File Manager to keep a table of record numbers and their CBAs. Then, to fill a request, this table is searched for the exact storage location of the desired record, so the direct access reduces to a table lookup.

Indexed sequential file

Records in an indexed sequential file can be accessed either sequentially or directly either of the procedures to compute the CBA.The index file must be searched for the pointer to the block where the data is stored. Because the index file is smaller than the data file,

Levels in a File Management System

The efficient management of files can't be separated from the efficient management of the devices that house them.

The highest level module is called the "basic file system," and it passes information through the access control verification module to the logical file system, which, in turn, notifies the physical file system, which works with the Device Manager. Figure 8.14 shows the hierarchy.

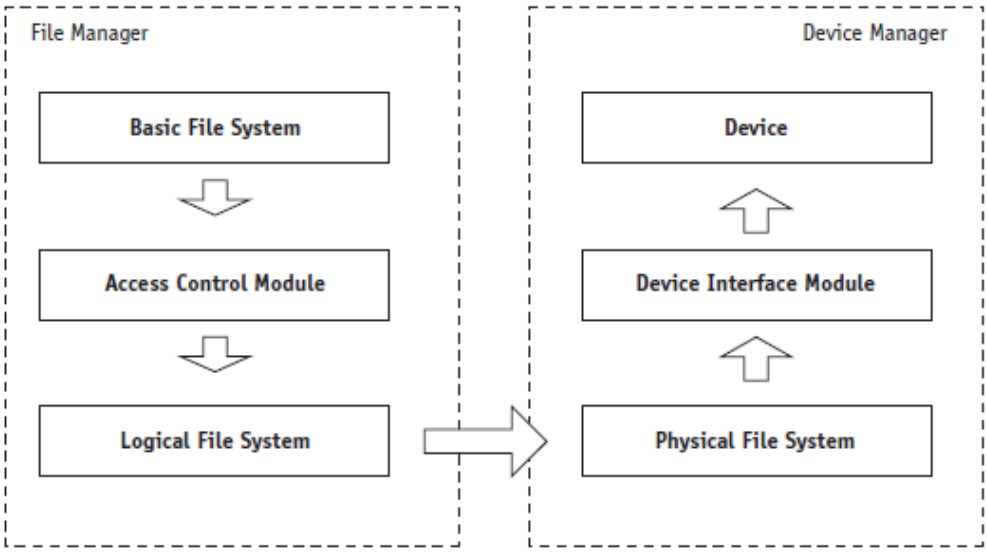


Figure 8.10 Typical modules of a file management system showing how information is passed from the File Manager to the Device Manager

Each level of the file management system is implemented using structured and modular programming techniques. The higher positioned modules pass information to the lower modules.It can perform the required service and continue the communication down the chain to the lowest module.Each of the modules can be further subdivided into more specific tasks

READ RECORD NUMBER 7 FROM FILE CLASSES INTO STUDENT

CLASSES- file name

STUDENT- data record previously defined within the program and occupying specific memory locations.

Table 8.1 A typical list of files stored in the directory called journal

Access Control	No. of Links	Group	Owner	No. of Bytes	Date	Time	Filename
drwxrwxr-x	2	journal	comp	12820	Jan 10	19:32	ArtWarehouse
drwxrwxr-x	2	journal	comp	12844	Dec 15	09:59	bus_transport
-rwxr-xr-x	1	journal	comp	2705221	Mar 6	11:38	CLASSES
-rwxr--r--	1	journal	comp	12556	Feb 20	18:08	PAYroll
-rwx------	1	journal	comp	8721	Jan 17	07:32	supplier

Because the file has already been opened, pertinent information about the file CLASSES has been brought into the operating system's active file table.

This information are record size, the address of its first physical record, its protection, and access control information. This information is used by the basic file system. Then activates the access control verification module to verify that this user is permitted to perform this operation with this file.

If access is allowed, information and control are passed along to the logical file system. If not, a message saying "access denied" is sent to the user.

Using the information passed down by the basic file system, the logical file system transforms the record number to its byte address using the familiar formula:

$$CBA = (RN - 1) * RL$$

This result, together with the address of the first physical record is passed down to the physical file system, which computes the location where the desired record physically resides.

If there's more than one record in that block, it computes the record's offset using these formulas:

block number = integers(byte address/physical block size)+ address of the first physical record

offset = remainder

This information is passed on to the device interface module.It transforms the block number to the actual cylinder/surface/record combination needed to retrieve the information from the secondary storage device. The device-scheduling algorithms come into play, as the information is placed in a buffer and control returns to the physical file system, which copies the information into the desired memory location.

Finally, when the operation is complete, the "all clear" message is passed on to all other modules.

Access Control Verification Module

The advantages of file sharing are numerous.

- Saving space,
- Synchronization of data updates
- Improves the efficiency of the system's resources
- There's a reduction of I/O operations.

The disadvantage of file sharing is that the integrity of each file must be safeguarded; that calls for control over who is allowed to access the file and what type of access is permitted.

There are five possible actions that can be performed on a file—the ability to READ only, WRITE only, EXECUTE only, DELETE only, or some combination of the four. Each file management system has its own method to control file access.

Access Control Matrix

The access control matrix is easy to implement, but because of its size it only works well for systems with a few files and a few users.

In the matrix, each column identifies a user and each row identifies a file. The intersection of the row and column contains the access rights for that user to that file, as Table 8.2 illustrates.

Table 8.2- Access control matrix

	User 1	User 2	User 3	User 4	User 5
File 1	RWED	R–E–	----	RWE–	--E–
File 2	----	R–E–	R–E–	--E–	----
File 3	----	RWED	----	--E–	----
File 4	R–E–	----	----	----	RWED
File 5	----	----	----	----	RWED

R = Read Access, W = Write Access, E = Execute Access, D = Delete Access, and a dash (-) = Access Not Allowed. Alice cannot read File 3 but can Read File 1 and File 4

Access Control Lists

The access control list is a modification of the access control matrix. Each file is entered in the list and contains the names of the users who are allowed to access it and the type of access each is permitted. To shorten the list, only those who may use the file are named , Some systems shorten the access control list even more by putting every user into a category: system, owner, group, and world.

- SYSTEM or ADMIN- have unlimited access to all files in the system.

- OWNER has absolute control over all files created in the owner's account.
- A OWNER may create a GROUP file so that all users of that group have access to that file
- WORLD is composed of all other users in the system

Table 8.3 Access Control List

File	Access
File 1	USER1 (RWED), USER2 (R-E-), USER4 (RWE-), USER5 (--E-), WORLD (----)
File 2	USER2 (R-E-), USER3 (R-E-), USER4 (--E-), WORLD (----)
File 3	USER2 (RWED), USER4 (--E-), WORLD (----)
File 4	USER1 (R-E-), USER5 (RWED), WORLD (----)
File 5	USER5 (RWED), WORLD (----)

Capability Lists

A capability list shows the access control information from a different perspective. It lists every user and the files to which each has access

Table 8.4 Capability List

User	Access
User 1	File 1 (RWED), File 4 (R-E-)
User 2	File 1 (R-E-), File 2 (R-E-), File 3 (RWED)
User 3	File 2 (R-E-)
User 4	File 1 (RWE-), File 2 (--E-), File 3 (--E-)
User 5	File 1 (--E-), File 4 (RWED), File 5 (RWED)

A capability list may be equated to specific concert tickets that are made available to only individuals whose names appear on the list. On the other hand, an

access control list can be equated to the reservation list in a restaurant that has limited seating, with each seat assigned to a certain individual.

Conclusion

The File Manager controls every file in the system and processes the user's commands (read, write, modify, create, delete, etc.) to interact with any file on the system. It also manages the access control procedures to maintain the integrity and security of the files under its control.

To achieve its goals, the File Manager must be able to accommodate a variety of file organizations, physical storage allocation schemes, record types, and access methods.

And, as we've seen, this requires increasingly complex file management software. In this chapter we discussed:

- Sequential, direct, and indexed sequential file organization
- Contiguous, noncontiguous, and indexed file storage allocation
- Fixed-length versus variable-length records
- Three methods of access control
- Data compression techniques

To get the most from a File Manager, it's important for users to realize the strengths and weaknesses of its segments—which access methods are allowed on which devices and with which record structures—and the advantages and disadvantages of each in overall efficiency.

Exercise

1. Give an example of the names of three files from your own computer that do not reside at the root or master directory. For each file, list both the relative filename and its complete filename.
2. In your own words, describe the purpose of the working directory and how it can speed or slow file access. In your opinion, should there be more than one working directory? Explain.
3. For each of the following entries in an access control matrix for User 2010, give the type of access allowed for each of the following files:
    a. File1 –E
    b.File2 RWED
    c. File3 R-E
    d.File4 R---
4. For each of the following entries in an access control list, give the type of access allowed for each user of File221 and describe who is included in the WORLD category:
    a. User2010 R-E
    b. User2014 –E
    c. User2017 RWED
    d. WORLD R---