III YEAR – V SEMESTER
COURSE CODE: 7BCE5C1
CORE COURSE-IX–OPERATING SYSTEM

Unit-I

Introduction to Operating Systems: Introduction, What is an Operating systems, Operating system components and goals, Operating systems architecture. Process Concepts: Introduction, Process States, Process Management, Interrupts, Interprocess Communication.

Unit II

Asynchronous Concurrent Execution: Introduction, Mutual Exclusion, Implementing Mutual Exclusion Primitives, Software solutions to the Mutual Exclusion Problem, Hardware solution to the Mutual Exclusion Problem, Semaphores. Concurrent Programming: Introduction,Monitors.

Unit III

Deadlock and Indefinite Postponement: Introduction, Examples of Deadlock, Related Problem Indefinite Postponement, Resource concepts, Four Necessary conditions for Deadlock, Deadlock solution, Deadlock Prevention, Deadlock Avoidance with Dijkstra's Banker's algorithm, Deadlock Detection, Deadlock Recovery. **Processor Scheduling:** Introduction, Scheduling levels, Preemptive Vs NonPreemptive Scheduling Priorities, Scheduling objective, Scheduling criteria, Scheduling algorithms.

Unit IV

Real Memory Organization and Management: Introduction, Memory organization, Memory Management, Memory Hierarchy, Memory Management Strategies, Contiguous Vs Non-Contiguous Memory allocation, Fixed Partition Multiprogrammimg, Variable Partition multiprogramming.

Virtual Memory Management: Introduction, Page Replacement, Page Replacement Strategies, Page Fault Frequency (PFF) Page replacement, Page Release, Page Size.

Unit V

Disk Performance Optimization: Introduction, Why Disk Scheduling is necessary, Disk Scheduling strategies, Rotational optimization.

File and Database Systems: Introduction, Data Hierarchy, Files, File Systems, File Organization, File Allocation, Free Space Management, File Access control.

Text Book:

1. Operating Systems, Deitel&DeitelChoffnes, Pearson education, Third edition, 2008. UNIT I: Chapter 1: 1.1, 1.2, 1.12, 1.13 &
   Chapter 3: 3.1, 3.2, 3.3, 3.4, 3.5
   UNIT II: Chapter 5: 5.1, 5.2, 5.3, 5.4(up to 5.4.2), 5.5, 5.6 &
   Chapter 6: 6.1, 6.2
   UNIT III: Chapter 7: 7.1, 7.2, 7.3, 7.4, 7.5, 7.6, 7.7, 7.8, 7.9, 7.10
   Chapter 8: 8.1, 8.2, 8 3, 8.4, 8.5, 8.6, 8.7
   UNIT IV: Chapter 9: 9.1, 9 2, 9.3, 9 4, 9.5, 9.6, 9.8, 9.9
   Chapter 11: 11.1, 11.5, 11.6, 11.8, 11.9, 11.10
   UNIT V: Chapter 12: 12.1, 12.4, 12.5, 12.6
   Chapter 13: 13.1, 13 2, 13.3, 13.4, 13.5, 13.6, 13.7, 13.8

DEPARTMENT OF COMPUTER SCIENCE

III YEAR – V SEMESTER SUBJECT CODE: 7BCE5C1

CORE COURSE-IX–OPERATING SYSTEM

Unit I

Introduction to Operating Systems: Introduction, What is an Operating systems, Operating system components and goals, Operating systems architecture. Process Concepts: Introduction, Process States, Process Management, Interrupts, Interprocess Communication.

Unit II

Asynchronous Concurrent Execution: Introduction, Mutual Exclusion, Implementing MutualExclusion Primitives, Software solutions to the Mutual Exclusion Problem, Hardware solution to the MutualExclusion Problem,SemaphoresConcurrent Programming:Introduction,Monitors.

Unit III

Deadlock and Indefinite Postponement: Introduction, Examples of Deadlock, Related Problem Indefinite Postponement, Resource concepts, Four Necessary conditions for Deadlock, Deadlock solution, Deadlock Prevention, Deadlock Avoidance with Dijkstra's Banker's algorithm, DeadlockDetection,DeadlockRecovery. **Processor Scheduling:** Introduction, Scheduling levels, Preemptive Vs NonPreemptive Scheduling Priorities, Scheduling objective, Scheduling criteria, Scheduling algorithms.

Unit IV

Real Memory Organization and Management: Introduction, Memory organization, Memory Management, Memory Hierarchy, Memory Management Strategies, Contiguous Vs Non-Contiguous Memory allocation, Fixed Partition Multiprogrammimg, Variable Partition multiprogramming.

Virtual Memory Management: Introduction, Page Replacement, Page Replacement Strategies, Page Fault Frequency (PFF) Page replacement, Page Release, Page Size.

Unit V

Disk Performance Optimization: Introduction, Why Disk Scheduling is necessary, Disk Scheduling strategies, Rotational optimization.

File and Database Systems: Introduction, Data Hierarchy, Files, File Systems, File Organization, File Allocation, Free Space Management, File Access control.

Text Book:

1. Operating Systems, Deitel&DeitelChoffnes, Pearson education, Third edition, 2008. UNIT I: Chapter 1: 1.1, 1.2, 1.12, 1.13 &
   Chapter 3: 3.1, 3.2, 3.3, 3.4, 3.5
   UNIT II: Chapter 5: 5.1, 5.2, 5.3, 5.4(up to 5.4.2), 5.5, 5.6 &

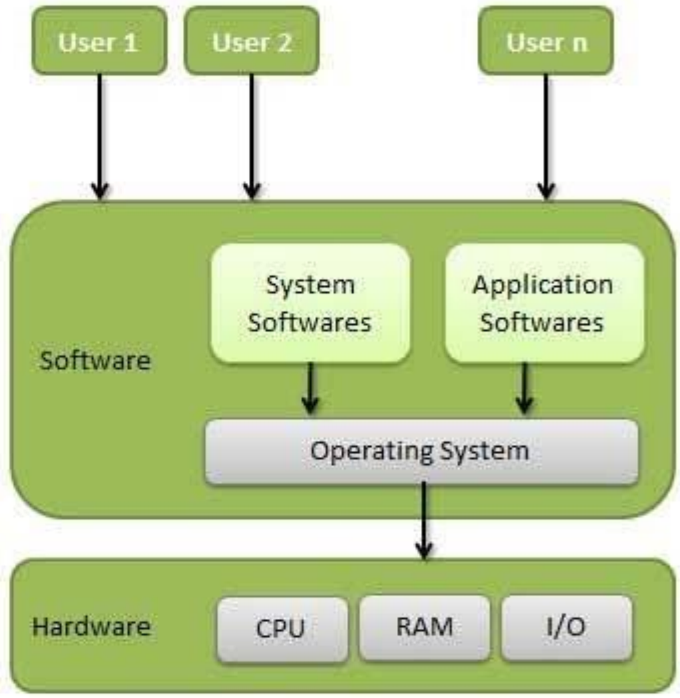
Chapter 6: 6.1, 6.2
UNIT III: Chapter 7: 7.1, 7.2, 7.3, 7.4, 7.5, 7.6, 7.7, 7.8, 7.9, 7.10
Chapter 8: 8.1, 8.2, 8 3, 8.4, 8.5, 8.6, 8.7
UNIT IV: Chapter 9: 9.1, 9 2, 9.3, 9 4, 9.5, 9.6, 9.8, 9.9
Chapter 11: 11.1, 11.5, 11.6, 11.8, 11.9, 11.10
UNIT V: Chapter 12: 12.1, 12.4, 12.5, 12.6
Chapter 13: 13.1, 13 2, 13.3, 13.4, 13.5, 13.6, 13.7, 13.8

UNIT-I
INTRODUCTIONS TO OPERATING SYSTEM

What is an Operating system?

An operating system is a program that acts as an interface between the user and the computer hardware and controls the execution of all kinds of programs.

- ❖ Operating system are primarily resource managers-they manage hardware, including processor, memory, input/output devices and communication devices.
- ❖ They manage applications and other software abstractions



Characteristics of Operating System

Here is a list of some of the most prominent characteristic features of Operating Systems −

- **Memory Management** − Keeps track of the primary memory, i.e. what part of it is in use by whom, what part is not in use, etc. and allocates the memory when a process or program requests it.
- **Processor Management** − Allocates the processor (CPU) to a process and deallocates the processor when it is no longer required.

- **Device Management** − Keeps track of all the devices. This is also called I/O controller that decides which process gets the device, when, and for how much time.
- **File Management** − Allocates and de-allocates the resources and decides who gets the resources.
- **Security** − Prevents unauthorized access to programs and data by means of passwords and other similar techniques.
- **Job Accounting** − Keeps track of time and resources used by various jobs and/or users.
- **Control Over System Performance** − Records delays between the request for a service and from the system.
- **Interaction with the Operators** − Interaction may take place via the console of the computer in the form of instructions. The Operating System acknowledges the same, does the corresponding action, and informs the operation by a display screen.
- **Error-detecting Aids** − Production of dumps, traces, error messages, and other debugging and error-detecting methods.
- **Coordination Between Other Software and Users** − Coordination and assignment of compilers, interpreters, assemblers, and other software to the various users of the computer systems.

Operating system components

□ process scheduler
□ memory manager
□ I/O manager
□ interprocess communication (IPC) manager
□ file system manager

Process scheduler

The **process scheduler,** which determines when and for how long a process executes on a processor.

Memory manager

The **memory manager,** which determines when and how memory is allocated to processes and what to do when main memory becomes full.

I/O manager

The **I/O manager,** which services input and output requests from and to hardware devices, respectively.

Interprocess communication (IPC) manager

The **interprocess communication (IPC) manager,** which allows processes to communicate with one another.

File system manager

The **file system manager,** which organizes named collections of data on storage devices and provides an interface for accessing data on those devices.

Operating System Goals

- Efficiency
- Robustness
- Scalability
- Extensibility
- Portability
- Security
- Interactivity
- Usability

Efficiency

An **efficient operating system** achieves high **throughput** and low average turnaround time.

Robustness

A **robust operating system** is fault tolerant and reliable.

(i.e., by minimizing loss of work and by preventing damage to the system's hardware).

Scalability

A **scalable operating system** is able to use resources as they are added.

Extensibility

An **extensible operating system** will adapt well to new technologies and provide capabilities to extend the operating system to perform tasks beyond its original design.

Portability

A **portable operating system** is designed such that it can operate on many hardware configurations.

Security

A **secure operating system** prevents users and software from accessing services and resources without authorization.

Interactivity

An **interactive operating system** allows applications to respond quickly to user actions, or events.

Usability

A **usable operating system** is one that has the potential to serve a significant user base.

Operating system architecture

Operating system architecture refers to the fundamental structures of a software system and the discipline of creating such structures and systems.

- □ Monolithic architecture
- □ Layered architecture
- □ Microkernel architecture
- □ Networked and Distributed Operating Systems

Monolithic architecture

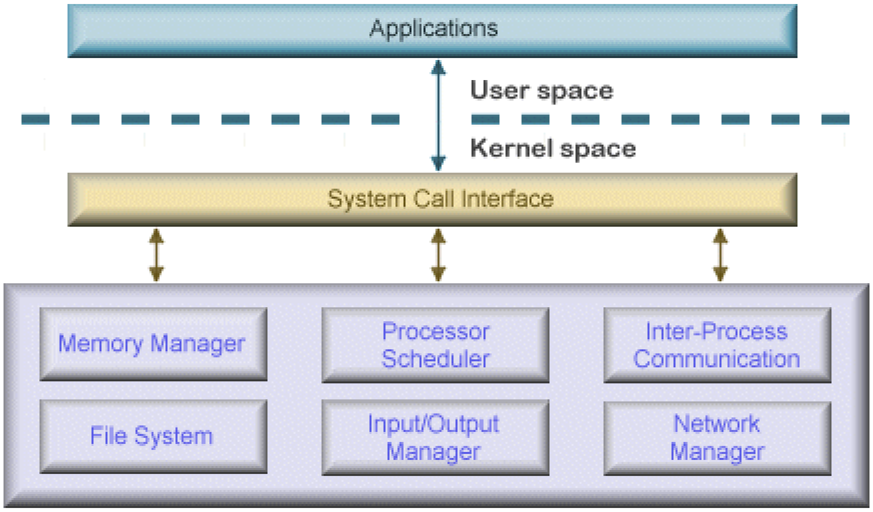
Kernel

It is the earliest and most common operating system architecture.

The **kernel** is a computer program at the core of a computer's operating system with complete control over everything in the system. It is the "portion of the operating system code that is always resident in memory". It facilitates interactions between hardware and software components.

Advantage: Highly efficient (Direct intercommunication between components)

Disadvantage: it is difficult to isolate the source of bugs and other errors.



A monolithic OS architecture

Kernel space

Kernel space is strictly reserved for running a privileged operating system kernel, kernel extensions, and most device drivers.

User space

User space refers to all code that runs outside the operating system's kernel. User space is the memory area where application software and some drivers execute.

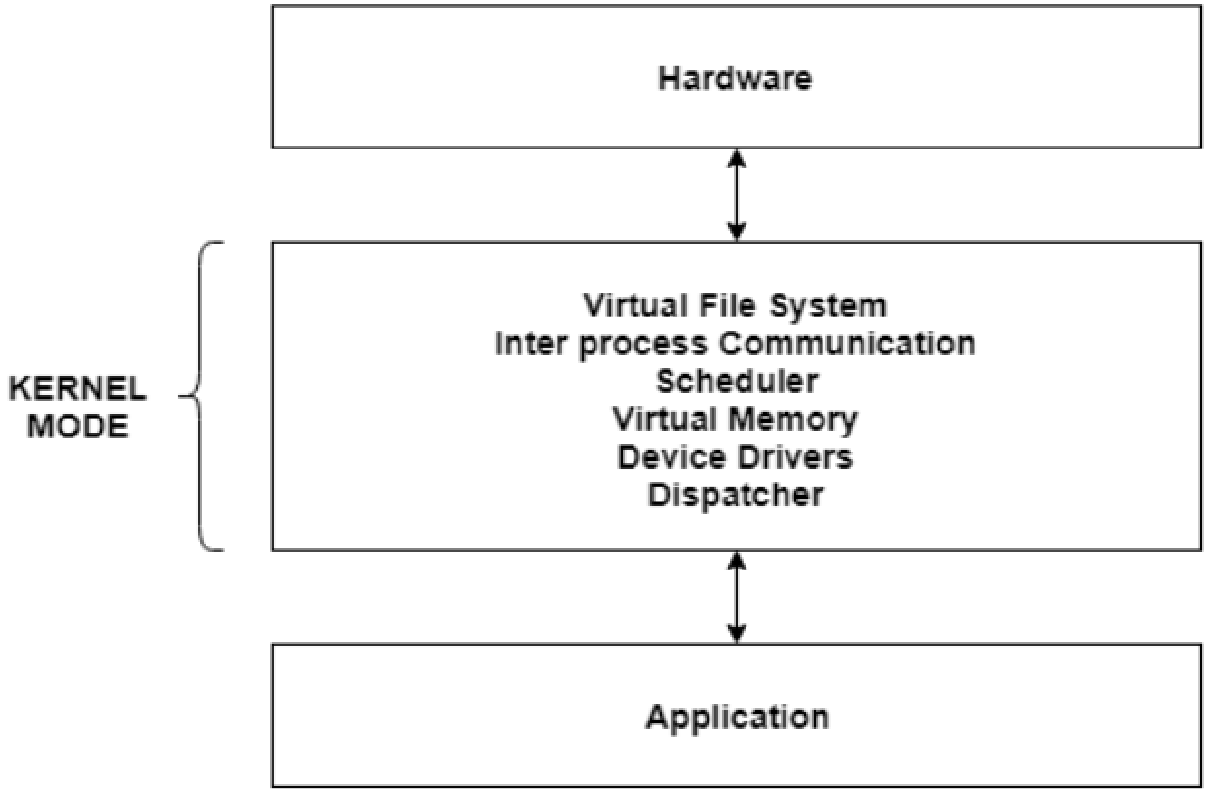
Monolithic Kernal

A **monolithic kernel** is an operating system architecture where the entire operating system is working in kernel space. The monolithic model differs from other operating system architectures in that it alone defines a high-level virtual interface over computer hardware.

Example

OS/360, VMS and Linux Virtual
Memory System

A diagram that demonstrates the architecture of a monolithic system is as follows −



Monolithic Kernel Operating System

Advantage

- Since there is less software involved it is faster.
- As it is one single piece of software it should be smaller both in source and compiled forms.
- Less code generally means fewer bugs which can translate to fewer security problems.

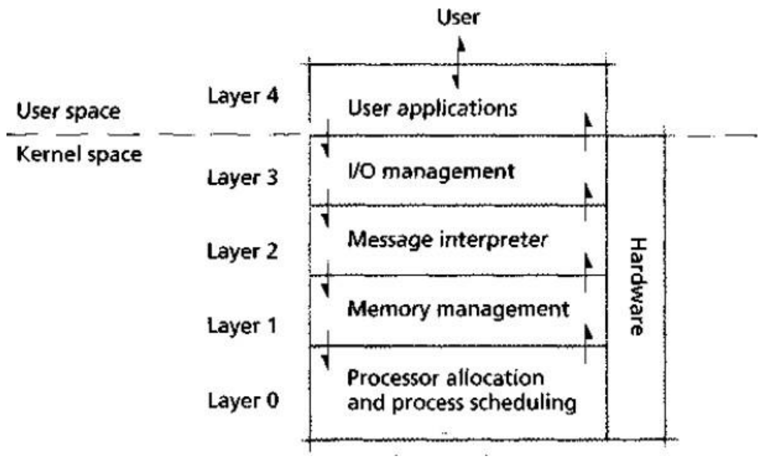
Disadvantage

- Kernels often become very large and difficult to maintain.
- A bug in a device driver might crash the entire system
- The fact that large kernels can become very difficult to maintain.

Layered architecture

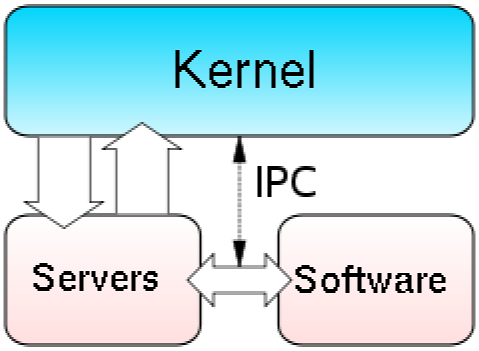
The layered approach to operating systems attempts to address this issue by grouping components that perform similar functions into layers. Each layer communicates exclusively with those immediately above and below it.

Lower level layers provide services to higher-level ones using an interface that hides their implementation.



Microkernel architecture

The functionality of the system is moved out of the traditional "kernel", into a set of "servers" that communicate through a "minimal" kernel, leaving as little as possible in "system space" and as much as possible in "user space".



Advantages:

- Easier to maintain
- Patches can be tested in a separate instance, and then swapped in to take over a production instance.
- Rapid development time and new software can be tested without having to reboot the kernel.

Disadvantages

- Larger running memory footprint
- More software for interfacing is required, there is a potential for

performance loss.

- Process management in general can be very complicated.

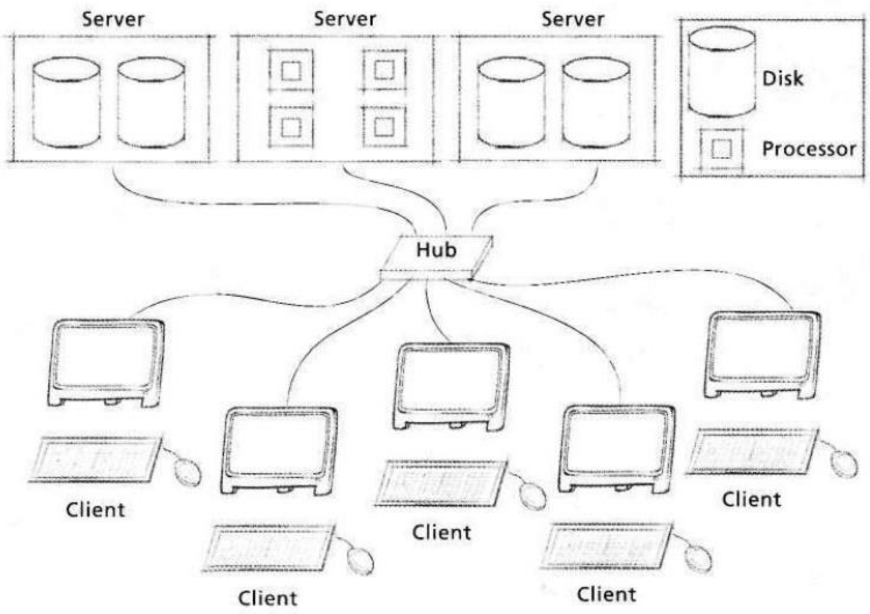
Differences between Microkernel and Monolithic Kernel

Some of the differences between microkernel and monolithic kernel are given as follows

- ✓ The microkernel is much smaller in size as compared to the monolithic kernel.
- ✓ The microkernel is easily extensible whereas this is quite complicated for the monolithic kernel.
- ✓ The execution of the microkernel is slower as compared to the monolithic kernel.
- ✓ Much more code is required to write a microkernel than the monolithic kernel.
- ✓ Examples of Microkernel are QNX, Symbian, L4 Linux etc. Monolithic Kernel examples are Linux, BSD etc.

Networked and Distributed Operating Systems

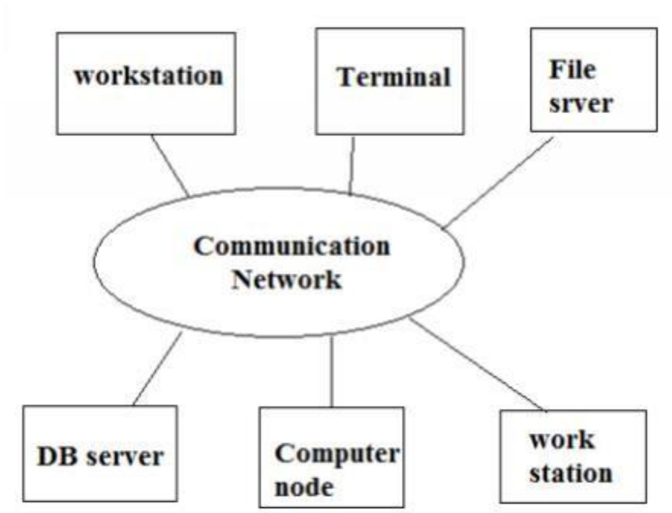
A network operating system enables its processes to access resources (e.g., files) that reside on other independent computers on a network. The structure of many networked and distributed operating systems is often based on the client/server model.



Distributed Operating Systems

A distributed operating system is a single operating system that manages resources on more than one computer system.

Distributed systems provide the illusion that multiple computers are a single powerful computer, so that a process can access all of the system's resources regardless of the process's location within the distributed system's network of computers.



Example
MIT's Chord operating system The
Amoeba operating system

Advantages of distributed system

1. The distributed system gives high performance than the single system. Because processing is combined and storage capacity is more. So it provides better services to the consumers.
2. It is more scalable as we can add the resources easily and increase the power of computing. Therefore sharing also becomes easy.
3. The important advantage of the distributed computing system is reliability. It is more reliable than a single system. If one machine from the system fails, the rest of the computers remain unaffected and the system can work as a whole.

Disadvantages of distributed system

1. If one or more network links fail or any server fails, then it is harmful to the other nodes which are connected and consuming the services from it.
2. It is easy to share the data in a distributed system. So, this can create a risk of security .
3. Distributed system supports the less software.

Process concepts

Process

A **process** can be thought of as a program in execution. A process will need certain resources—such as CPU time, memory, files, and I/O devices —to accomplish its task.

- Process memory is divided into four sections as shown in Figure 3.1 below:

- The text section comprises the compiled program code, read in from non-volatile storage when the program is launched.
- The data section stores global and static variables, allocated and initialized prior to executing main.
- The heap is used for dynamic memory allocation, and is managed via calls to new, delete, malloc, free, etc.
- The stack is used for local variables. Space on the stack is reserved for local variables when they are declared (at function entrance or elsewhere, depending on the language), and the space is freed up when the variables go out of scope. Note that the stack is also used for function return values, and the exact mechanisms of stack management may be language specific.
- Note that the stack and the heap start at opposite ends of the process's free space and grow towards each other. If they should ever meet, then either a stack overflow error will occur, or else a call to new or malloc will fail due to insufficient memory available.

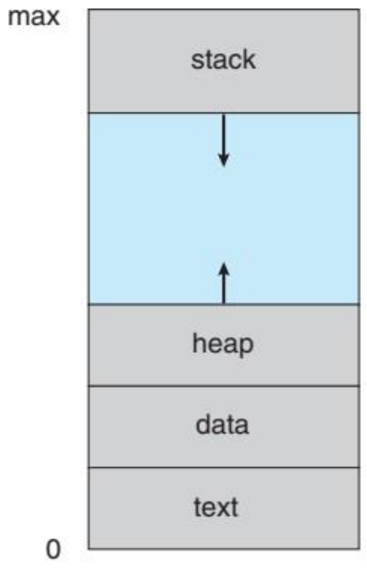


Figure 3.1 Process in memory.

Process state

As a process executes, it changes **state**. The state of a process is defined in part by the current activity of that process.

- **New**. The process is being created.
- **Running**. Instructions are being executed.
- **Waiting**. The process is waiting for some event to occur (such as an I/O completion or reception of a signal).
- **Ready**. The process is waiting to be assigned to a processor.
- **Terminated**. The process has finished execution.

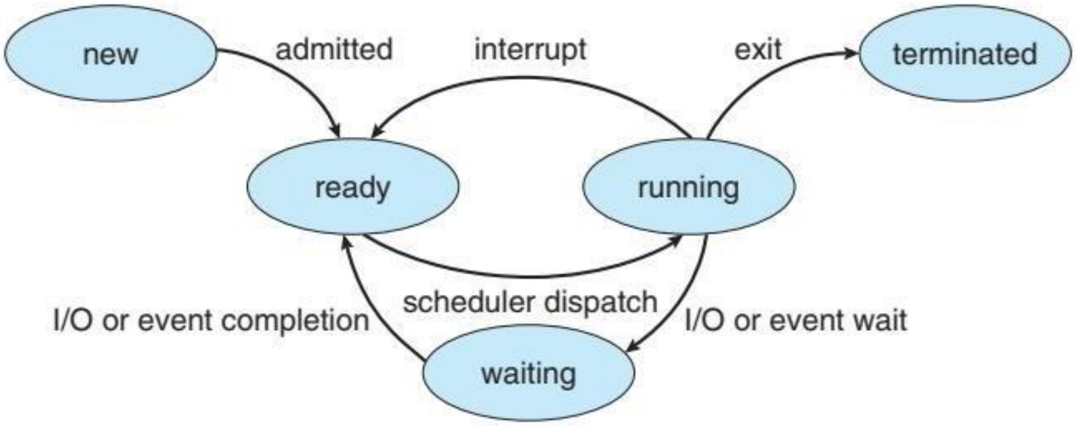


Figure 3.2 Diagram of process state.

Process Control Block

Each process is represented in the operating system by a **process control block (PCB)**—also called a **task control block**.

- **Process State -** Running, waiting, etc., as discussed above.
- **Process ID**, and parent process ID.
- **CPU registers and Program Counter -** These need to be saved and restored when swapping processes in and out of the CPU.
- **CPU-Scheduling information -** Such as priority information and pointers to scheduling queues.
- **Memory-Management information -** E.g. page tables or segment tables.
- **Accounting information -** user and kernel CPU time consumed, account numbers, limits, etc.
- **I/O Status information -** Devices allocated, open file tables, etc.

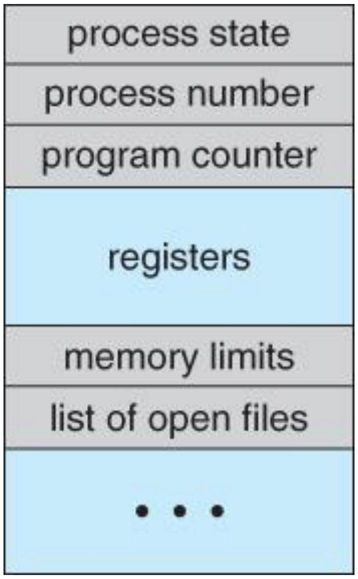


Figure 3.3 - Process control block (PCB)

Process Management (operations)

The processes in most systems can execute concurrently, and they may be created and deleted dynamically.

- Process creation
- Process termination

Process creation

During the course of execution, a process may create several new processes. As mentioned earlier, the creating process is called a parent process, and the new processes are called the children of that process. Each of these new processes may in turn create other processes, forming a **tree** of processes.

Most operating systems (including UNIX, Linux, and Windows) identify processes according to a unique **process identifier** (or **pid**), which is typically an integer number.

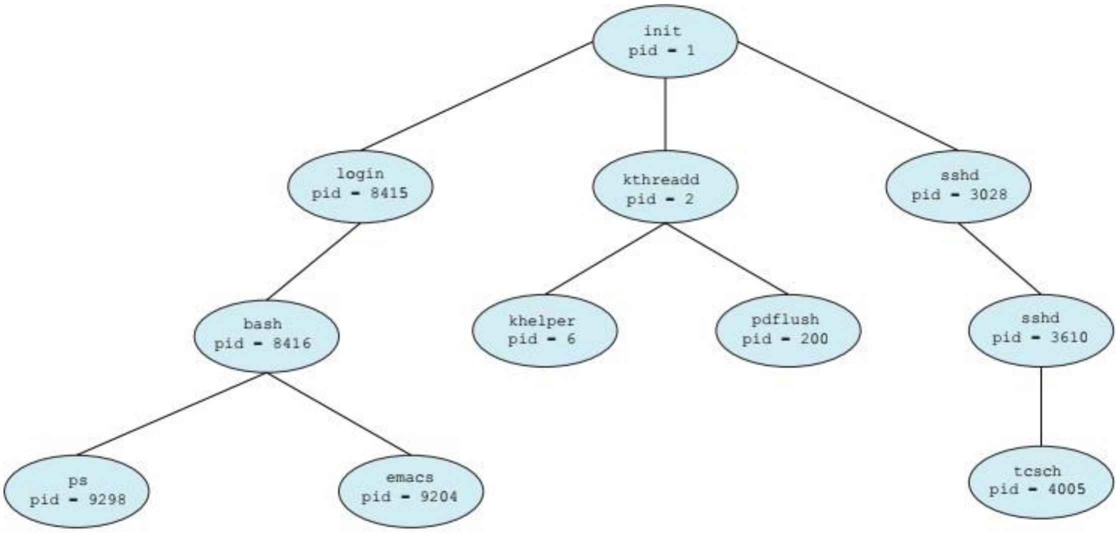


Figure 3.8 A tree of processes on a typical Linux system.

The init process (which always has a pid of 1) serves as the root parent process for all user processes.

We see two children of init—kthreadd and sshd. The kthreadd process is responsible for creating additional processes that perform tasks on behalf of the kernel (in this situation, khelper and pdflush).

The sshd process is responsible for managing clients that connect to the system by using ssh (which is short for ***secure shell***).
The login process is responsible for managing clients that directly log onto the system. In this example, a client has logged on and is using the bash shell, which has been assigned pid 8416.
On UNIX and Linux systems, we can obtain a listing of processes by using the ps command.
For example, the command

ps -el

will list complete information for all processes currently active in the system.

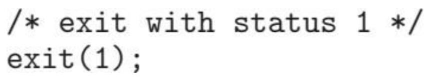
fork ()

A new process is created by the fork () system call.

Process termination

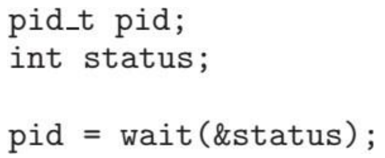
exit ()

A process terminates when it finishes executing its final statement and asks the operating system to delete it by using the exit () system call.
To illustrate process execution and termination, consider that, in Linux and UNIX systems, we can terminate a process by using the exit() system call, providing an exit status as a parameter:



```
/* exit with status 1 */
exit(1);
```

A parent process may wait for the termination of a child process by using the wait() system call. The wait() system call is passed a parameter that allows the parent to obtain the exit status of the child.



```
pid_t pid;
int status;

pid = wait(&status);
```

When a process terminates, its resources are deallocated by the operating system.

Interrupt

Hardware and Software Concepts, interrupts enable software to respond to signals from hardware.

Interrupt handler

The operating system may specify a set of instructions, called an **interrupt handler,** to be executed in response to each type of interrupt. This allows the operating system to gain control of the processor to manage system resources.

- Software interrupt (synchronous)
- Hardware interrupt (asynchronous)

Software interrupt

A software interrupt is requested by the processor itself upon executing particular instructions or when certain conditions are met. Every software interrupt signal is associated with a particular interrupt handler.

Software interrupts may also be unexpectedly triggered by program execution errors. These interrupts typically are called *traps* or *exceptions*.

For example, **a divide-by-zero exception**

Hardware interrupt

A hardware interrupt is a condition related to the state of the hardware that may be signaled by an external hardware device, e.g., an interrupt request (IRQ)

Hardware devices issue asynchronous interrupts to communicate a status change to the processor.

For example, the **keyboard** generates an interrupt when a user presses a key; the **mouse** generates an interrupt when it moves or when one of its buttons is pressed.

Interprocess communication

Interprocess communication is the mechanism provided by the operating system that allows processes to communicate with each other.

A process is ***independent*** if it cannot affect or be affected by the other processes executing in the system. Any process that does not share data with any other process is independent.

A process is ***cooperating*** if it can affect or be affected by the other processes executing in the system. Clearly, any process that shares data with other processes is a cooperating process.

Cooperating processes require an **interprocess communication (IPC)** mechanism that will allow them to exchange data and information.

There are two fundamental models of interprocess communication:

- ❖ **shared memory**
- ❖ **message passing**.

In the shared-memory model, a region of memory that is shared by cooperating processes is established. Processes can then exchange information by reading and writing data to the shared region.

In the message-passing model, communication takes place by means of messages exchanged between the cooperating processes. The two communications models are contrasted in Figure 3.12.

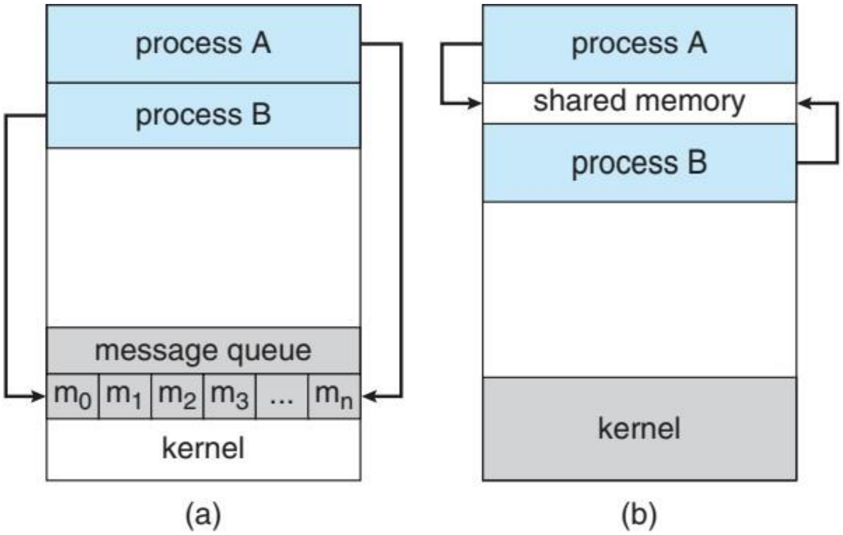


Figure 3.12 Communications models. (a) Message passing. (b) Shared memory.

Shared-Memory Systems

- In general the memory to be shared in a shared-memory system is initially within the address space of a particular process, which needs to make system calls in order to make that memory publicly available to one or more other processes.
- Other processes which wish to use the shared memory must then make their own system calls to attach the shared memory area onto their address space.
- Generally a few messages must be passed back and forth between the cooperating processes first in order to set up and coordinate the shared memory access.

```
#define BUFFER_SIZE 10

typedef struct {
    . . .
} item;
```

```
item buffer[ BUFFER_SIZE ];
int in = 0;
int out = 0;
```

Message-Passing Systems

- Message passing systems must support at a minimum system calls for "send message" and "receive message".
- A communication link must be established between the cooperating processes before messages can be sent.
- There are three key issues to be resolved in message passing systems as further explored in the next three subsections:
    - Direct or indirect communication (naming)
    - Synchronous or asynchronous communication
    - Automatic or explicit buffering.

- **direct communication** the sender must know the name of the receiver to which it wishes to send a message.
    - There is a one-to-one link between every sender-receiver pair.
    - For **symmetric** communication, the receiver must also know the specific name of the sender from which it wishes to receive messages.
      For **asymmetric** communications, this is not necessary.
- **Indirect communication** uses shared mailboxes, or ports.
    - Multiple processes can share the same mailbox or boxes.
    - Only one process can read any given message in a mailbox. Initially the process that creates the mailbox is the owner, and is the only one allowed to read mail in the mailbox, although this privilege may be transferred.
    - The OS must provide system calls to create and delete mailboxes, and to send and receive messages to/from mailboxes.

```
send(message)          receive(message)
```

UNIT II

Asynchronous Concurrent Execution

Introduction

- Concurrent execution
  - – More than one thread exists in system at once
  - – Can execute independently or in cooperation
  - – Asynchronous execution
    - Threads generally independent
    - Must occasionally communicate or synchronize
    - Complex and difficult to manage such interactions

Mutual exclusion

A mutual exclusion (mutex) is a program object that prevents simultaneous access to a shared resource. This concept is used in concurrent programming with a critical section, a piece of code in which processes or threads access a shared resource.

Implementing Mutual Exclusion Primitives

- ❖ enterMutualExclusion()
- ❖ exitMutualExclusion()

Implements the following properties:

1. The solution is implemented purely in software on a machine without specially designed mutual exclusion machine-language instructions. Each machine-language instruction is executed **indivisibly**—i.e., once started, it completes without interruption.
2. No assumption can be made about the relative speeds of asynchronous concurrent threads.
3. A thread that is executing instructions outside its critical section cannot prevent any other threads from entering their critical sections.
4. A thread must not be indefinitely postponed from entering its critical section.

Software solutions to the Mutual Exclusion Problem Some software solutions exist that use busy waiting to achieve mutual exclusion.

- ❖ Dekker's algorithm
- ❖ Peterson's algorithm
- ❖ Lamport's bakery algorithm

Dekker's algorithm

Dekker's algorithm is the first known correct solution to the mutual exclusion problem in concurrent programming. The solution is attributed to Dutch mathematician Th.J.Dekker by EdsgerW.Dijkstra in an unpublished paper on sequential process descriptions and his manuscript on cooperating sequential processes.

If two processes attempt to enter a critical section at the same time, the algorithm will allow only one process in, based on whose turn it is. If one process is already in the critical section, the other process will busy wait for the first process to exit. This is done by the use of two flags, wants_to_enter[0] and wants_to_enter[1], which indicate an intention to enter the critical section on the part of processes 0 and 1, respectively, and a variable turn that indicates who has priority between the two processes.

variables

```
                wants_to_enter : array of 2 booleans turn :
                integer
            wants_to_enter[0]    ←    false
            wants_to_enter[1] ← false turn ←
            0 // or 1
   p0:
        wants_to_enter[0] ← true while
        wants_to_enter[1] {
             if turn ≠ 0 { wants_to_enter[0] ← false
                  while turn ≠ 0 {
                     // busy wait
                  }
                  wants_to_enter[0] ← true
             }
        }
        // critical section
        ...
        turn ← 1 wants_to_enter[0] ←
        false
        // remainder section

   p1:
        wants_to_enter[1] ← true while
        wants_to_enter[0] {
             if turn ≠ 1 { wants_to_enter[1] ← false
                  while turn ≠ 1 {
                     // busy wait
                  }
                  wants_to_enter[1] ← true
             }
        }
          // critical section
        ...
        turn ← 0 wants_to_enter[1] ←
        false
        // remainder section
```

Peterson's algorithm

Peterson's algorithm (or **Peterson's solution**) is a concurrent programming algorithm for mutual exclusion that allows two or more processes to share a single-use resource without conflict, using only shared memory for communication. It was formulated by Gary L. Peterson in 1981.

The algorithm uses two variables, flag and turn. A flag[n] value of of true indicates that the process n wants to enter the critical section. Entrance to the critical section is granted for process P0 if P1 does not want to enter its critical section or if P1 has given priority to P0 by setting turn to 0.

```
  bool flag[2] = {false, false};
  int turn;

  P0:          flag[0] = true;
  P0_gate: turn = 1;
                        while (flag[1] == true && turn == 1)
{
                      // busy wait
}
// critical section
...
// end of critical section
flag[0] = false;

  P1:          flag[1] = true;
  P1_gate: turn = 0;
                        while (flag[0] == true && turn == 0)
{
                      // busy wait
}
// critical section
```

```
...
// end of critical section
flag[1] = false;
```

The algorithm satisfies the three essential criteria to solve the critical section problem, provided that changes to the variables turn, flag [0], and flag[1] propgate immediately and atomically. The while condition works even with preemption. The three criteria are mutual exclusion, progress, and bounded waiting.

Since turn can take on one of two values, it can be replaced by a single bit, meaning that the algorithm requires only three bits of memory.

Mutual exclusion

P0 and P1 can never be in the critical section at the same time: If P0 is in its critical section, then flag[0] is true. In addition, either flag[1] is false (meaning P1 has left its critical section), or turn is 0 (meaning P1 is just now trying to enter the critical section, but graciously waiting), or P1 is at label P1_gate (trying to enter its critical section, after setting flag[1] to true but before setting turn to 0 and busy waiting). So if both processes are in their critical sections then we conclude that the state must satisfy flag[0] and flag[1] and turn = 0 and turn = 1. No state can satisfy both turn = 0 and turn = 1, so there can be no state where both processes are in their critical sections.

Progress

Progress is defined as the following: if no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in making the decision as to which process will enter its critical section next. Note that for a process or thread, the remainder sections are parts of the code that are not related to the critical section. This selection cannot be postponed indefinitely. A process cannot immediately re-enter the critical section if the other process has set its flag to say that it would like to enter its critical section.

Bounded waiting

Bounded waiting, or bounded by pass means that the number of times a process is bypassed by another process after it has indicated its desire to enter the critical section is bounded by a function of the number of processes in the system. In Peterson's algorithm, a process will never wait longer than one turn for entrance to the critical section.

Lamport's bakery algorithm

Lamport's bakery algorithm is a computer algorithm devised by computer scientist Leslie Lamport, as part of his long study of the formal correctness of concurrent systems, which is intended to improve the safety in the usage of shared resources among multiple threads by means of mutual exclusion.

Lamport's bakery algorithm is one of many mutual exclusion algorithms designed to prevent concurrent threads entering critical sections of code concurrently to eliminate the risk of data corruption.

Algorithm

Analogy

Lamport envisioned a bakery with a numbering machine at its entrance so each customer is given a unique number. Numbers increase by one as customers enter the store. A global counter displays the number of the customer that is currently being served. All other customers must wait in a queue until the baker finishes serving the current customer and the next number is displayed. When the customer is done shopping and has disposed of his or her number, the clerk increments the number, allowing the next customer to be served. That customer must draw another number from the numbering machine in order to shop again.

Critical section

The critical section is that part of code that requires exclusive access to resources and may only be executed by one thread at a time. In the bakery analogy, it is when the customer trades with the baker that others must wait.

When a thread wants to enter the critical section, it has to check whether now is its turn to do so. It should check the number *n* of every other thread to make sure that it has the smallest one. In case another thread has the same number, the thread with the smallest *i* will enter the critical section first.

In pseudocode this comparison between threads *a* and *b* can be written in the form:

```
$(n_a, i_a) < (n_b, i_b)$ // $i_a$ - the customer number for thread $a$, $n_a$ - the thread number for thread $a$
```

which is equivalent to:

```
$(i_a < i_b)$ or $((i_a == i_b)$ and $(n_a < n_b))$
```

Once the thread ends its critical job, it gets rid of its number and enters the **non-critical section**.

Non-critical section

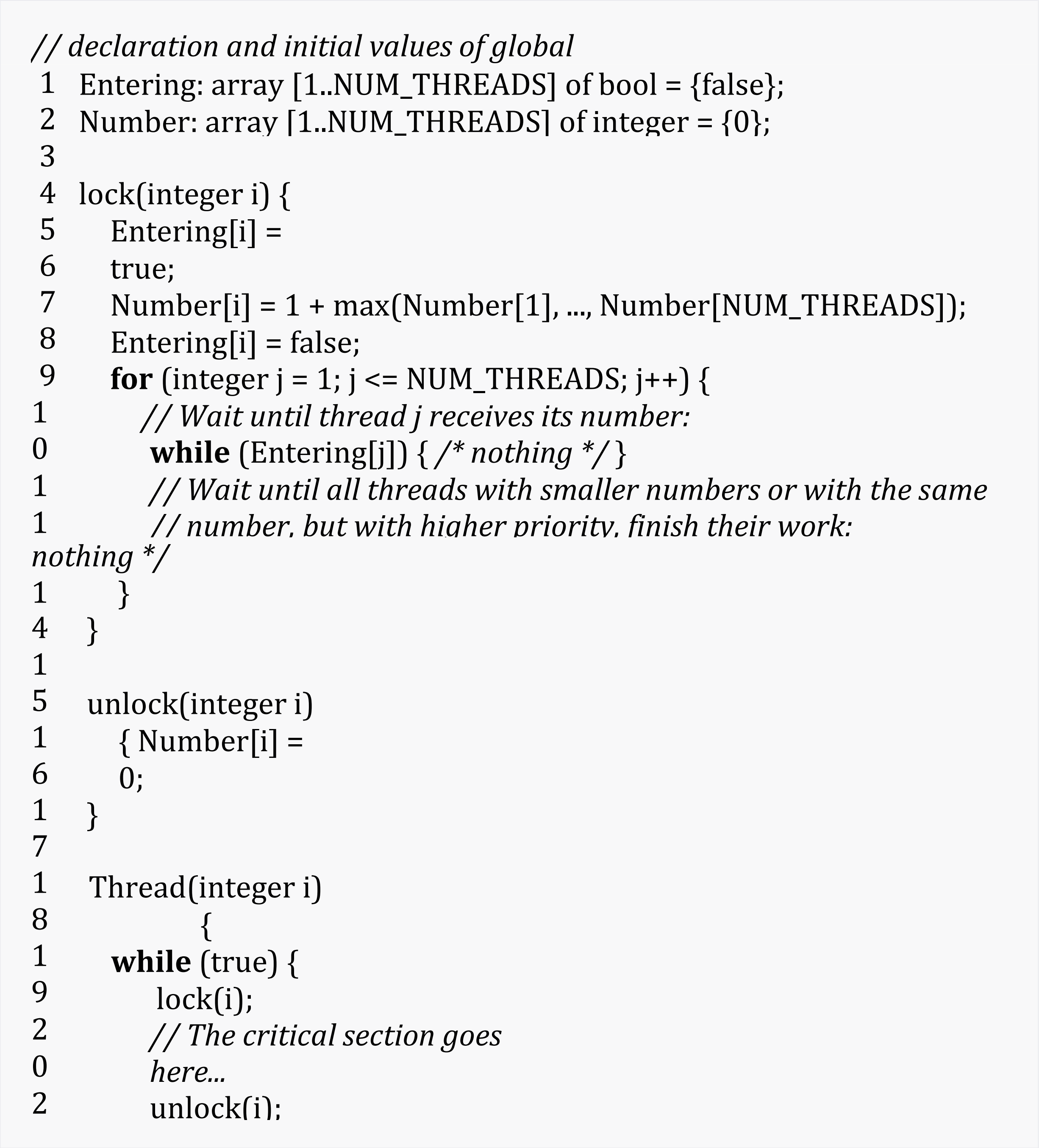
The non-critical section is the part of code that doesn't need exclusive access. It represents some thread-specific computation that doesn't interfere with other threads' resources and execution.

Implementation of the algorithm

Definitions

In Lamport's original paper, the *entering* variable is known as *choosing*, and the following conditions apply:

- Words choosing [i] and number [i] are in the memory of process i, and are initially zero.
- The range of values of number [i] is unbounded.
- A process may fail at any time. We assume that when it fails, it immediately goes to its noncritical section and halts.



```
// declaration and initial values of global
Entering: array [1..NUM_THREADS] of bool = {false};
Number: array [1..NUM_THREADS] of integer = {0};

lock(integer i) {
    Entering[i] =
    true;
    Number[i] = 1 + max(Number[1], ..., Number[NUM_THREADS]);
    Entering[i] = false;
    for (integer j = 1; j <= NUM_THREADS; j++) {
        // Wait until thread j receives its number:
         while (Entering[j]) { /* nothing */ }
         // Wait until all threads with smaller numbers or with the same
         // number, but with higher priority, finish their work:
nothing */
     }
 }

 unlock(integer i)
     { Number[i] =
     0;
 }

 Thread(integer i)
               {
    while (true) {
          lock(i);
         // The critical section goes
         here...
         unlock(i);
```

Hardware solution to the Mutual Exclusion Problem

This section presents several mechanisms provided in hardware to help solve the problem of mutual exclusion.

- *Disabling Interrupts*
- *Test-and-Set Instruction*
- *Swap Instruction*

Disabling Interrupts

On uniprocessor systems, the simplest solution to achieve mutual exclusion is to disable interrupts during a process's critical section. This will prevent any interrupt service routines from running (effectively preventing a process from being preempted). Although this solution is effective, it leads to many problems.

> An **interrupt handler**, also known as an **interrupt service routine** or **ISR**, is a special block of code associated with a specific interrupt condition.

If a critical section is long, then the system clock will drift every time a critical section is executed because the timer interrupt is no longer serviced, so tracking time is impossible during the critical section.

Also, if a process halts during its critical section, control will never be returned to another process, effectively halting the entire system. A more elegant method for achieving mutual exclusion is the busy-wait.

Busy-waiting, **busy-looping** or **spinning** is a technique in which a process repeatedly checks to see if a condition is true, such as whether keyboard input or a lock is available.

Test-and-set Instruction

Busy-waiting is effective for both uniprocessor and multiprocessor systems. The use of shared memory and an atomic test-and-set instruction provide the mutual exclusion. A process can test-and-set on a location in shared memory, and since the operation is atomic; only one process can set the flag at a time.

Any process that is unsuccessful in setting the flag can either go on to do other tasks and try again later, release the processor to another process and try again later, or continue to loop while checking the flag until it is successful in acquiring it. Preemption is still possible, so this method allows the system to continue to function—even if a process halts while holding the lock.

The **test-and-set** instruction is an instruction used to write 1 (set) to a memory location and return its old value as a single atomic (i.e., non- interruptible) operation. If multiple processes may access the same memory

location, and if a process is currently performing a test-and-set, no other process may begin another test-and-set until the first process's test-and-set is finished.
The **test-and-set** instruction enables a thread to perform this operation **atomically** (i.e., indivisibly). Such operations are also described as atomic **read-modify-write (RMW) memory operations.** Because the processor reads a value from memory, modifies its value in its registers and writes the modified value to memory without interruption.

A lock can be built using an atomic test-and-set instruction as follows:

```
function Lock(boolean *lock) {
    while (test_and_set(lock) == 1);
}
```

Swap instruction

It is common for programs to exchange (or swap) values stored in two different variables (consider, for example, the Quicksort algorithm).
Although the concept is simple, a successful exchange of values between two variables in most highlevel programming languages requires three instructions and the creation of a temporary variable:

```
temp = a;
a = b;
b = temp;
```

Because such swapping operations are performed regularly, many architectures support a swap instruction that enables a thread to exchange the values of the two variables atomically.

The instruction
swap (a, b)

First the instruction loads the value of b, which may be either true or false, into a temporary register. Then, the value of a is copied to b and the value of the temporary register is copied to a.

Semaphore

Semaphore is simply a variable that is non-negative and shared between threads. A semaphore is a signaling mechanism, and a thread that is waiting on a semaphore can be signaled by another thread.
It uses two atomic operations, 1)wait, and 2) signal for the process synchronization.
A semaphore is simply a variable. This variable is used to solve critical section problems and to achieve process synchronization in the multi processing environment.
The semaphore concept was invented by Dutch computer scientist Edsger Dijkstra in 1962

Characteristic of Semaphore

- It is a mechanism that can be used to provide synchronization of tasks.
- It is a low-level synchronization mechanism.
- Semaphore will always hold a non-negative integer value.
- Semaphore can be implemented using test operations and interrupts, which should be executed using file descriptors.

Types of Semaphores

The two common kinds of semaphores are

- Counting semaphores
    - Binary semaphores.

Counting semaphore

The value of counting semaphore at any point of time indicates the maximum number of processes that can enter in the critical section at the same time.
A process which wants to enter in the critical section first decrease the semaphore value by 1 and then check whether it gets negative or not.
If it gets negative then the process is pushed in the list of blocked processes (i.e. q) otherwise it gets enter in the critical section.
Counting semaphores are equipped with two operations, historically denoted as P and V. V means *verhogen* ("increase"). P means *proberen* ("to test" or "to try"). The canonical names V and P come from the initials of Dutch words.

A simple way to understand wait (P) and signal (V) operations is:

- **wait**: Decrements the value of semaphore variable by 1. If the new value of the semaphore variable is negative, the process executing wait is blocked (i.e., added to the semaphore's queue). Otherwise, the process continues execution, having used a unit of the resource.
The wait operation decrements the value of its argument S, if it is positive. If S is negative or zero, then no operation is performed.

```
wait(S)

{

  while (S<=0);



  S--;

}
```

- **signal**: Increments the value of semaphore variable by 1. After the increment, if the pre-increment value was negative (meaning there are processes waiting for a resource), it transfers a blocked process from the semaphore's waiting queue to the ready queue.

- The signal operation increments the value of its argument S.

```
signal(S)

{

  S++;

}
```

The counting semaphore concept can be extended with the ability to claim or return more than one "unit" from the semaphore, a technique implemented in Unix. The modified V and P operations are as follows, using square brackets to indicate atomic operations, i.e., operations which appear indivisible from the perspective of other processes:

```
function V(semaphore S, integer I): [$S \leftarrow S + I$]
function P(semaphore S, integer I):
    repeat:
        [if $S \geq I$:
        $S \leftarrow S - I$
            break]
```

Binary semaphore

Binary semaphores which are restricted to the values 0 and 1 (or locked/unlocked, unavailable/available) are called **binary semaphores** and are used to implement locks. The binary semaphores are quite similar to counting semaphores, but their value is restricted to 0 and 1. In this type of semaphore, the wait operation works only if semaphore = 1, and the signal operation succeeds when semaphore= 0.

Wait for Operation

This type of semaphore operation helps you to control the entry of a task into the critical section. However, If the value of wait is positive, then the value of the wait argument X is decremented. In the case of negative or zero value, no operation is executed. It is also called P(S) operation.

```
Copy CodeP(S)
{
    while (S<=0);
    S--;
}
```

Signal operation

This type of Semaphore operation is used to control the exit of a task from a critical section. It helps to increase the value of the argument by 1, which is denoted as V(S).

```
Copy CodeP(S)
{
    while (S>=0);
    S++;
}
```

Concurrent programming

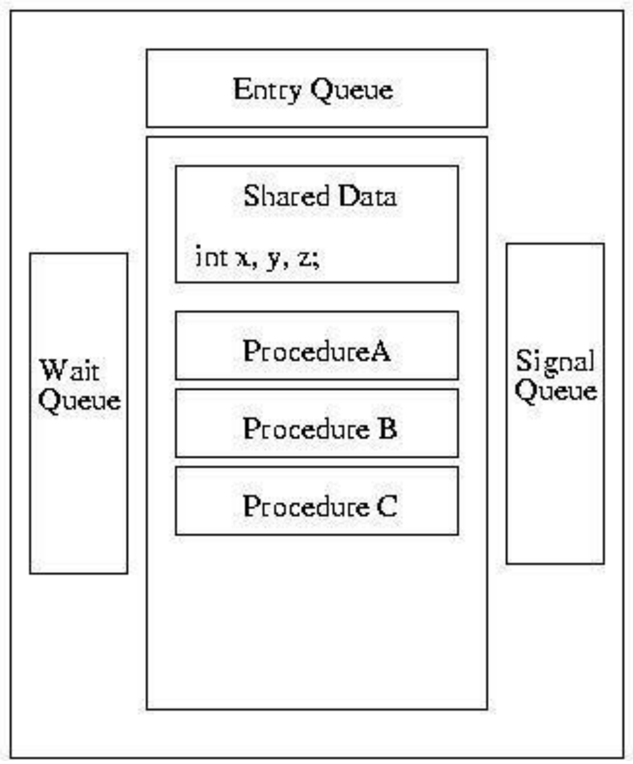
Concurrent programming is a technique in which two or more processes start, run in an *interleaved fashion* through **context switching** and complete in an overlapping time period by managing access to *shared resources.*

e.g. on a single core of CPU.

Monitor

A **monitor** is an object that contains both the data and procedures needed to perform allocation of a particular **serially reusable** shared resource or group of serially reusable shared resources.

Monitors were invented by **Per Brinch Hansen** and **C. A. R. Hoare**.



Condition Variables - Overview

The *condition variable* is a synchronization primitive that provides a queue for threads waiting for a resource. A thread tests to see if the resource is available. If it is available, it uses it. Otherwise it adds itself to the queue of threads waiting for the resource. When a thread has finished with a resource, it wakes up exactly one thread from the queue (or none, if the queue is empty). In the case of a sharable resource, a broadcast can be sent to wake up all sleeping threads.

Proper use of condition variables provides for safe access both to the queue and to test the resource even with concurrency. The implementation of condition variables involves several mutexes.

Condition variables support three operations:

- **wait** - add calling thread to the queue and put it to sleep
- **signal** - remove a thread form the queue and wake it up
- **broadcast** - remove and wake-up all threads on the queue

When using condition variables, an additional mutex must be used to protect the critical sections of code that test the lock or change the locks state.

Condition Variables - Typical Use

The following code illustrates a typical use of condition variables to acquire a resource.

Notes that both the mutex mx and the condition variable cv are passed into the wait function.

If you examine the implementation of wait below, you will find that the wait function atomically releases the mutex and puts the thread to sleep. After the thread is signalled and wakes up, it reacquires the resource. This is to prevent a *lost wake-up.* This situation is discussed in the section describing the implementation of condition variables.

spin_lock s;

```
GetLock (condition cv, mutex mx)
{
    mutex_acquire (mx);
    while (LOCKED)
        wait (c, mx);

    lock=LOCKED;
    mutex_release (mx);
}
ReleaseLock (condition cv, mutex mx)
{
    mutex_acquire (mx); lock
        = UNLOCKED;
        signal (cv);
    mutex_release (mx);
}
```

Condition Variables - Implementation

This is just one implementation of condition variables, others are possible.

Data Structure

The condition variable data structure contains a double-linked list to use as a queue. It also contains a semaphore to protect operations on this queue. This semaphore should be a spin-lock since it will only be held for very short periods of time.

```
struct condition {
          proc next;          /* doubly linked list
    implementation of */
          proc prev;          /* queue for blocked threads */
          mutex mx; /*protects queue */
                                        };
```

wait()

The wait() operation adds a thread to the list and then puts it to sleep. The mutex that protects the critical section in the calling function is passed as a parameter to wait(). This allows wait to atomically release the mutex and put the process to sleep.

If this operation is not atomic and a context switch occurs after the release_mutex (mx) and before the thread goes to sleep, it is possible that a process will signal before the process goes to sleep. When the waiting() process is restored to execution, it will enter the sleep queue, but the message to wake it up will be forever gone.

```
                    void wait (condition *cv, mutex *mx)
       {
          mutex_acquire(&c->listLock);                /* protect the queue
    */
                  enqueue (&c->next, &c->prev, thr_self()); /*
enqueue */
          mutex_release (&c->listLock); /* we're done with the list */
          /* The suspend and release_mutex() operation should be atomic */
                              release_mutex (mx));
          thr_suspend (self); /* Sleep 'til someone wakes us */
          mutex_acquire (mx); /* Woke up -- our turn, get resource lock */
                                      return;
       }
```

signal()

The signal() operation gets the next thread from the queue and wakes it up. If the queue is empty, it does nothing.

```
void signal (condition *c)
{
    thread_id tid;

    mutex_acquire (c->listlock); /* protect the queue */
        tid = dequeue(&c->next, &c->prev);
              mutex_release (listLock);

    if (tid>0) thr_continue (tid);

    return;
}
```

broadcast()

The broadcast operation wakes up every thread waiting for a particular resource. This generally makes sense only with sharable resources. Perhaps a writer just completed so all of the readers can be awakened.

```
void broadcast (condition *c)
{
thread_id tid;

mutex_acquire (c->listLock); /* protect the queue */
while (&c->next) /* queue is not empty */
{
            tid = dequeue(&c->next, &c->prev); /* wake
one */
            thr_continue (tid); /* Make it runnable */
}
mutex_release (c->listLock); /* done with the queue */
}
```

UNIT III

DEADLOCK AND INDEFINITE POSTPONEMENT

Introduction to Deadlock

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

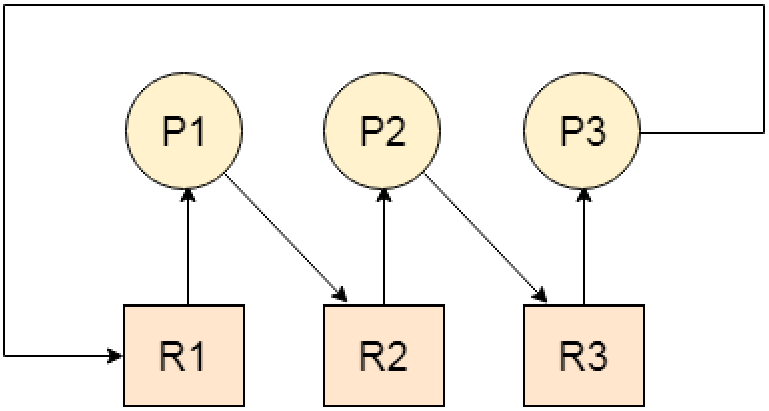
1. The process requests for some resource.
2. OS grant the resource if it is available otherwise let the process waits.
3. The process uses it and release on the completion.

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.

Let us assume that there are three processes P1, P2 and P3. There are three different resources R1, R2 and R3. R1 is assigned to P1, R2 is assigned to P2 and R3 is assigned to P3.

After some time, P1 demands for R1 which is being used by P2. P1 halts its execution since it can't complete without R2. P2 also demands for R3 which is being used by P3. P2 also stops its execution because it can't continue without R3. P3 also demands for R1 which is being used by P1 therefore P3 also stops its execution.

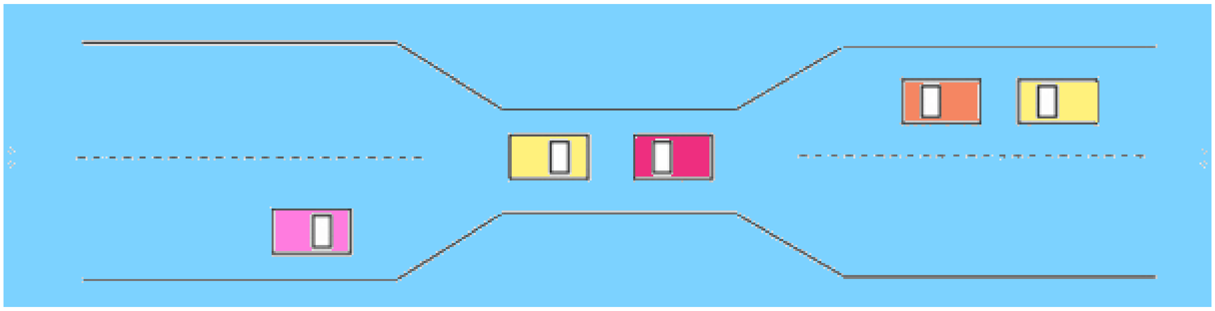
In this scenario, a cycle is being formed among the three processes. None of the process is progressing and they are all waiting. The computer becomes unresponsive since all the processes got blocked.



.

Example of Deadlock

- A real-world example would be traffic, which is going only in one direction.
- Here, a bridge is considered a resource.
- So, when Deadlock happens, it can be easily resolved if one car backs up (Preempt resources and rollback).
- Several cars may have to be backed up if a deadlock situation occurs.
- So starvation is possible.



Example of deadlock

Necessary conditions for Deadlocks

1. **Mutual Exclusion**

   A resource can only be shared in mutually exclusive manner. It implies, if two process cannot use the same resource at the same time.

2. **Hold and Wait**

   A process waits for some resources while holding another resource at the same time.

3. **No preemption**

   The process which once scheduled will be executed till the completion. No other process can be scheduled by the scheduler meanwhile.

4. **Circular Wait**

   All the processes must be waiting for the resources in a cyclic manner so that the last process is waiting for the resource which is being held by the first process.

DEADLOCK IN SPOOLING SYSTEMS

- Spooling systems are prone to deadlock.

Common solution

- Restrain input spoolers so that when the spooling file begins to reach some saturation threshold, the spoolers do not read in more print jobs.

Today's systems

- Printing begins before the job is completed so that a full spooling file can be emptied even while a job is still executing.

- Same concept has been applied to streaming audio and video.

EXAMPLE: DINING PHILOSOPHERS

Problem statement:

Five philosophers sit around a circular table. Each leads a simple life alternating between thinking and eating spaghetti. In front of each philosopher is a dish of spaghetti that is constantly replenished by a dedicated wait staff. There are exactly five forks on the table, one between each adjacent pair of philosophers. Eating spaghetti (in the most proper manner) requires that a philosopher use both adjacent forks (simultaneously). Develop a concurrent program free of deadlock and indefinite postponement that models the activities of the philosophers.

```
void typicalPhilosopher()
{
        while ( true )
        {
                think();
                eat();
        } // end while

} // end typicalPhilospher
```

Dining philosopher behavior

Constraints:

– To prevent philosophers from starving:

- Free of deadlock
- Free of indefinite postponement

– Enforce mutual exclusion

- Two philosophers cannot use the same fork at once

The problems of mutual exclusion, deadlock and indefinite postponement lie in the implementation of method eat.

```
void eat()
{
        pickUpLeftFork();
```

```
            pickUpRightFork();
            eatForSomeTime();
            putDownRightFork(
            );
            putDownLeftFork();
      } // eat
```

RELATED PROBLEM: INDEFINITE POSTPONEMENT

- A process may be delayed indefinitely while other processes receive the system's attention.
- This situation, called indefinite postponement, indefinite blocking, or starvation.
- Indefinite postponement may occur due to biases in a system's resource scheduling policies.
- Some systems prevent indefinite postponement by increasing a process's priority gradually as it waits for a resource —this technique is called **aging.**

RESOURCE CONCEPTS

- Resources that are **preemptible,** such as processors and main memory. Resources can be removed from a process without loss of work.
- Certain resources are **nonpreemptible;** they cannot be removed from the processes to which they are assigned until the processes voluntarily release them.
- For example, tape drives and optical scanners.

Reentrant code

– Cannot be changed while in use.

– May be shared by several processes simultaneously.

– Serially reusable code

– May be changed but is reinitialized each time it is used.

– May be used by only one process at a time.

FOUR NECESSARY CONDITIONS FOR DEADLOCK

1. Mutual exclusion condition

– Resource may be acquired exclusively by only one process at a time.

2. Wait-for condition (hold-and-wait condition)

– Process that has acquired an exclusive resource may hold that resource while the process waits to obtain other resources

3. No-preemption condition

– Once a process has obtained a resource, the system cannot remove it from the process's control until the process has finished using the resource.

4. Circular-wait condition

– Two or more processes are locked in a **"circular chain"** in which each process is waiting for one or more resources that the next process in the chain is holding.

DEADLOCK SOLUTIONS

There are four major areas of interest in deadlock research.

1. **Deadlock prevention**

   In deadlock prevention is to condition a system to remove any possibility of deadlocks occurring, but prevention methods can often result in poor resource utilization.

2. **Deadlock avoidance**

   Avoidance methods do not precondition the system to remove all possibility of deadlock.

3. **Deadlock detection**

   If a deadlock has occurred, and to identify the processes and resources that are involved.

4. **Deadlock recovery**

   Deadlock recovery methods are used to clear deadlocks from a system so that it may operate free them, and so that the deadlocked processes may complete their execution and free their resources.

Deadlock Prevention

If we simulate deadlock with a table which is standing on its four legs then we can also simulate four legs with the four conditions which when occurs simultaneously, cause the deadlock.

However, if we break one of the legs of the table then the table will fall definitely. The same happens with deadlock, if we can be able to violate one of the four necessary conditions and don't let them occur together then we can prevent the deadlock.

Let's see how we can prevent each of the conditions.

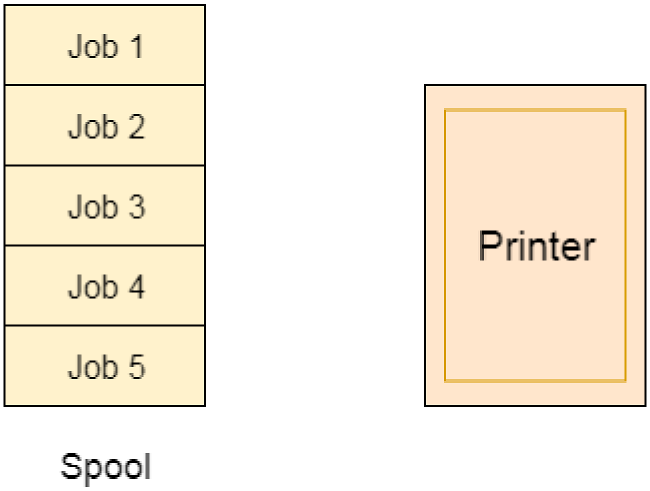
1. Mutual Exclusion

Mutual section from the resource point of view is the fact that a resource can never be used by more than one process simultaneously which is fair enough but that is the main reason behind the deadlock. If a resource could have been used by more than one process at the same time then the process would have never been waiting for any resource.

However, if we can be able to violate resources behaving in the mutually exclusive manner then the deadlock can be prevented.

Spooling

For a device like printer, spooling can work. There is a memory associated with the printer which stores jobs from each of the process into it. Later, Printer collects all the jobs and print each one of them according to FCFS. By using this mechanism, the process doesn't have to wait for the printer and it can continue whatever it was doing. Later, it collects the output when it is produced.



Although, Spooling can be an effective approach to violate mutual exclusion but it suffers from two kinds of problems.

1. This cannot be applied to every resource.
2. After some point of time, there may arise a race condition between the processes to get space in that spool.

We cannot force a resource to be used by more than one process at the same time since it will not be fair enough and some serious problems may arise in the performance. Therefore, we cannot violate mutual exclusion for a process practically.

2. Hold and Wait

Hold and wait condition lies when a process holds a resource and waiting for some other resource to complete its task. Deadlock occurs because there can be more than one process which are holding one resource and waiting for other in the cyclic order.

However, we have to find out some mechanism by which a process either doesn't hold any resource or doesn't wait. That means, a process must be assigned all the necessary resources before the execution starts. A process must not wait for any resource once the execution has been started.

!(Hold and wait) = !hold or !wait (negation of hold and wait is, either you don't hold or you don't wait)

This can be implemented practically if a process declares all the resources initially. However, this sounds very practical but can't be done in the computer system because a process can't determine necessary resources initially.

Process is the set of instructions which are executed by the CPU. Each of the instruction may demand multiple resources at the multiple times. The need cannot be fixed by the OS.

The problem with the approach is:

1. Practically not possible.
2. Possibility of getting starved will be increases due to the fact that some process may hold a resource for a very long time.

3. No Preemption

Deadlock arises due to the fact that a process can't be stopped once it starts. However, if we take the resource away from the process which is causing deadlock then we can prevent deadlock.

This is not a good approach at all since if we take a resource away which is being used by the process then all the work which it has done till now can become inconsistent.

Consider a printer is being used by any process. If we take the printer away from that process and assign it to some other process then all the data which has been printed can become inconsistent and ineffective and also the fact that the process can't start printing again from where it has left which causes performance inefficiency.

4. Circular Wait

To violate circular wait, we can assign a priority number to each of the resource. A process can't request for a lesser priority resource. This ensures that not a single process can request a resource which is being utilized by some other process and no cycle will be formed.

DEADLOCK AVOIDANCE WITH DIJKSTRA'S BANKER'S ALGORITHM

The Banker's Algorithm defines how a particular system can prevent deadlock by controlling how resources are distributed to users. The Banker's Algorithm prevents deadlock in operating systems that exhibit the following properties:

- The operating system shares a fixed number of resources, *t*, among a fixed number of processes, *n.*
- Each process specifies in advance the maximum number of resources that it requires to complete its work.
- The operating system accepts a process's request if that process's **maximum need** does not exceed the total number of resources available in the system, *t* (i.e., the process cannot request more than the total number of resources available in the system).
- Sometimes, a process may have to wait to obtain an additional resource, but the operating system guarantees a finite wait time.
- If the operating system is able to satisfy a process's maximum need for resources, then the process guarantees that the resource will be used and released to the operating system within a finite time.

The system is said to be in a safe **state** if the operating system can guarantee that all current processes can complete their work within a finite time. If not, then the system is said to be in an **unsafe state.**

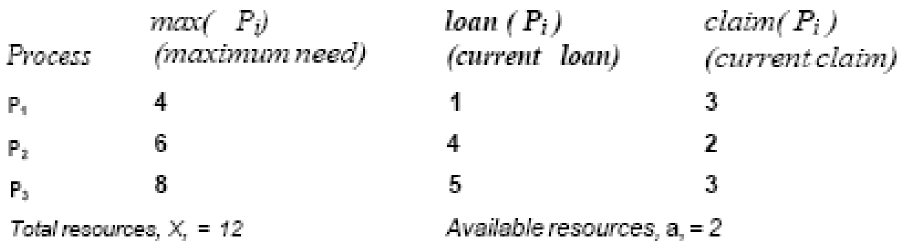
We also define four terms that describe the distribution of resources among processes.

- Let *max(Pi)* be the maximum number of resources that process P*i* requires during its execution. For example, if process P3 never requires more than two resources, then *max(P3) = 2.*
- Let *loan(Pi)* represent process P*i*'s current **loan** of a resource, where its loan is the number of resources the process has already obtained from the system. For example, if the system has allocated four resources to process P5, then *loan(P5)* = 4.
- Let *claim(Pi)* be the current **claim** of a process, where a process's claim is equal to its maximum need minus its current loan. For example, if process P7 has a maximum need of six resources and a current loan of four resources, then we have *claim(P7) = max(P7) - loan(P7)* = 6 - 4 = 2

- Let *a* be the number of resources still available for allocation. This is equivalent to the total number of resources *(t)* minus the sum of the loans to all the processes in the system, i.e.,

$$a = t - \sum_{i=1}^{n} loan(P_i)$$

EXAMPLE OF A SAFE STATE



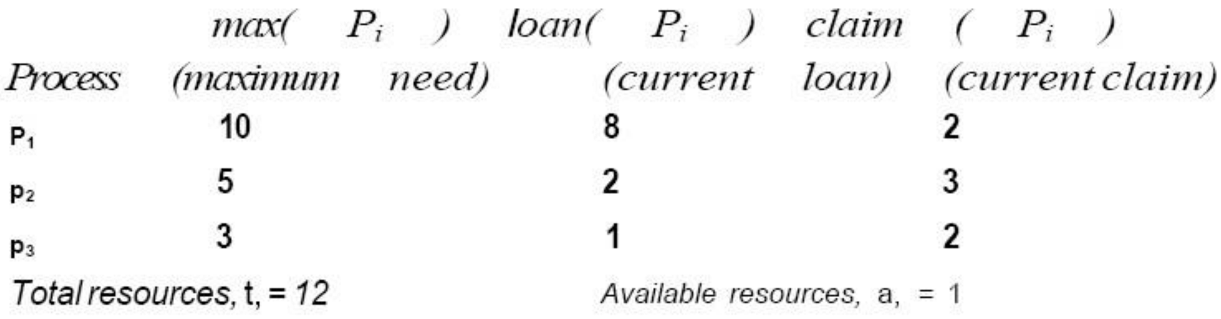
Process	$max(P_i)$ (maximum need)	$loan(P_i)$ (current loan)	$claim(P_i)$ (current claim)
P_1	4	1	3
P_2	6	4	2
P_3	8	5	3

Total resources, X, = 12 Available resources, a, = 2

Figure 7.6 | Safe State.

- This state is "safe" because process P2 currently has a loan of four resources and will eventually need a maximum of six, or two additional resources.
- The system has 12 resources, of which 10 are currently in use and two are available. If the system allocates these two available resources to P2, fulfilling P2's maximum need, then P2 can run to completion.
- After P2 finishes, it will release six resources, enabling the system to immediately fulfill the maximum needs of P1 (3) and P3 (3), enabling both of those processes to finish.

EXAMPLE OF AN UNSAFE STATE



Process	$max(P_i)$ (maximum need)	$loan(P_i)$ (current loan)	$claim(P_i)$ (current claim)
P_1	10	8	2
P_2	5	2	3
P_3	3	1	2

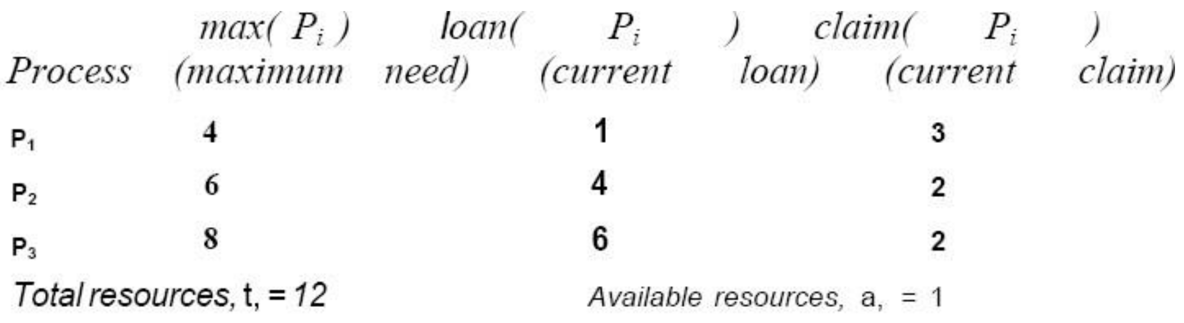
Total resources, t, = 12 Available resources, a, = 1

Figure 7.7 | Unsafe state.

- We sum the values of the third column and subtract from 12 to obtain a value of one for a.
- At this point, no matter which process requests the available resource, we cannot guarantee that all three processes will finish.
- In fact, suppose process P1 requests and is granted the last available resource.
- A three-way deadlock could occur if indeed each process needs to request at least one more resource before releasing any resources to the pool.
- It is important to note here that an unsafe state does not imply the existence of deadlock, nor even that deadlock will eventually occur.

EXAMPLE OF SAFE-STATE-TO-UNSAFE-STATE TRANSITION

- The current value of *a* is 2.
- Now suppose that process P3 requests an additional resource.
- If the system were to grant this request, then the new state would be as in Fig. 7.8.
- Now, the current value of *a* is 1, which is not enough to satisfy the current claim of any process, so the state is now unsafe.



Process	$max(P_i)$ (maximum need)	$loan(P_i)$ (current loan)	$claim(P_i)$ (current claim)
P_1	4	1	3
P_2	6	4	2
P_3	8	6	2

Total resources, t, = 12 Available resources, a, = 1

Figure 7.8 | *Safe-state-to-unsafe-state transition.*

BANKER'S ALGORITHM RESOURCE ALLOCATION

- The "mutual exclusion," "wait-for," and "no-preemption" conditions are allowed—processes are indeed allowed to hold resources while requesting and waiting for additional resources, and resources may not be preempted from a process holding those resources.

- The system may either grant or deny each request.
- If a request is denied, that process holds any allocated resources and waits for a finite time until that request is eventually granted.
- The system grants only requests that result in safe states.
- Resource requests that would result in unsafe states are repeatedly denied until they can eventually be satisfied.

WEAKNESSES IN THE BANKER'S ALGORITHM

The algorithm has a number of weaknesses.

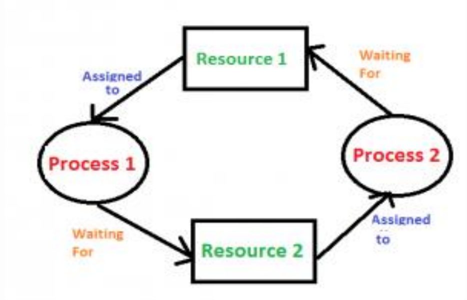
- It requires that there be a fixed number of resources to allocate.
- The algorithm requires that the population of processes remains fixed. In today's interactive and multiprogrammed systems the process population is constantly changing.
- The algorithm requires that the banker (i.e., the system) grant all requests within a "finite time."
- Similarly, the algorithm requires that clients (i.e., processes) repay all loans (i.e., return all resources) within a "finite time."
- The algorithm requires that processes state their maximum needs in advance. With resource allocation becoming increasingly dynamic, it is becoming more difficult to know a process's maximum needs.

For the reasons stated above, Dijkstra's Banker's Algorithm is not implemented in today's operating systems.

DEADLOCK DETECTION

❖ If resources have single instance

In this case for Deadlock detection we can run an algorithm to check for cycle in the Resource Allocation Graph. Presence of cycle in the graph is the sufficient condition for deadlock.



In the above diagram, resource 1 and resource 2 have single instances. There is a cycle R1 → P1 → R2 → P2. So, Deadlock is Confirmed.

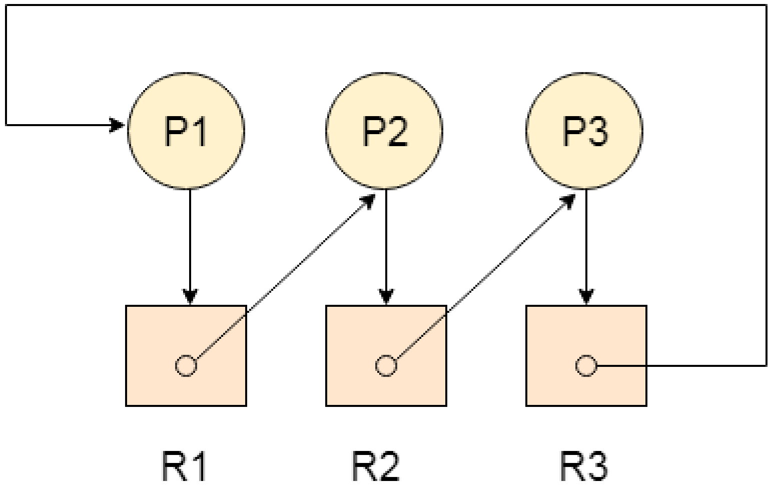
If there are multiple instances of resources:
Detection of the cycle is necessary but not sufficient condition for deadlock detection, in this case, the system may or may not be in deadlock varies according to different situations.

Resource Allocation Graph

If a cycle is being formed in a Resource allocation graph where all the resources have the single instance then the system is deadlocked.

In Case of Resource allocation graph with multi-instanced resource types, Cycle is a necessary condition of deadlock but not the sufficient condition.

The following example contains three processes P1, P2, P3 and three resources R2, R2, R3. All the resources are having single instances each.



If we analyze the graph then we can find out that there is a cycle formed in the graph since the system is satisfying all the four conditions of deadlock.

Allocation Matrix

Allocation matrix can be formed by using the Resource allocation graph of a system. In Allocation matrix, an entry will be made for each of the resource assigned. For Example, in the following matrix, en entry is being made in front of P1 and below R3 since R3 is assigned to P1.

Process	R1	R2	R3
P1	0	0	1
P2	1	0	0
P3	0	1	0

Request Matrix

In request matrix, an entry will be made for each of the resource requested. As in the following example, P1 needs R1 therefore an entry is being made in front of P1 and below R1.

Process	R1	R2	R3
P1	1	0	0
P2	0	1	0
P3	0	0	1

Avial = (0,0,0)

Neither we are having any resource available in the system nor a process going to release. Each of the process needs at least single resource to complete therefore they will continuously be holding each one of them.

We cannot fulfill the demand of at least one process using the available resources therefore the system is deadlocked as determined earlier when we detected a cycle in the graph

RESOURCE-ALLOCATION GRAPHS

- In Fig. 7.10(a), process P1 is requesting a resource of type R1. The arrow from P1 indicating that the resource request is under consideration.
- In Fig. 7.10(b), process P2 has been allocated a resource of type R2 (of which there are two).The arrow is drawn from the small circle within the large circle R2 to the square P2, to indicate that the system has allocated a specific resource.
- Figure 7.10(c) indicates a situation somewhat closer to a potential deadlock.
- Process P3 is requesting a resource of type R3, but the system has allocated the only R3

resource to process P4.

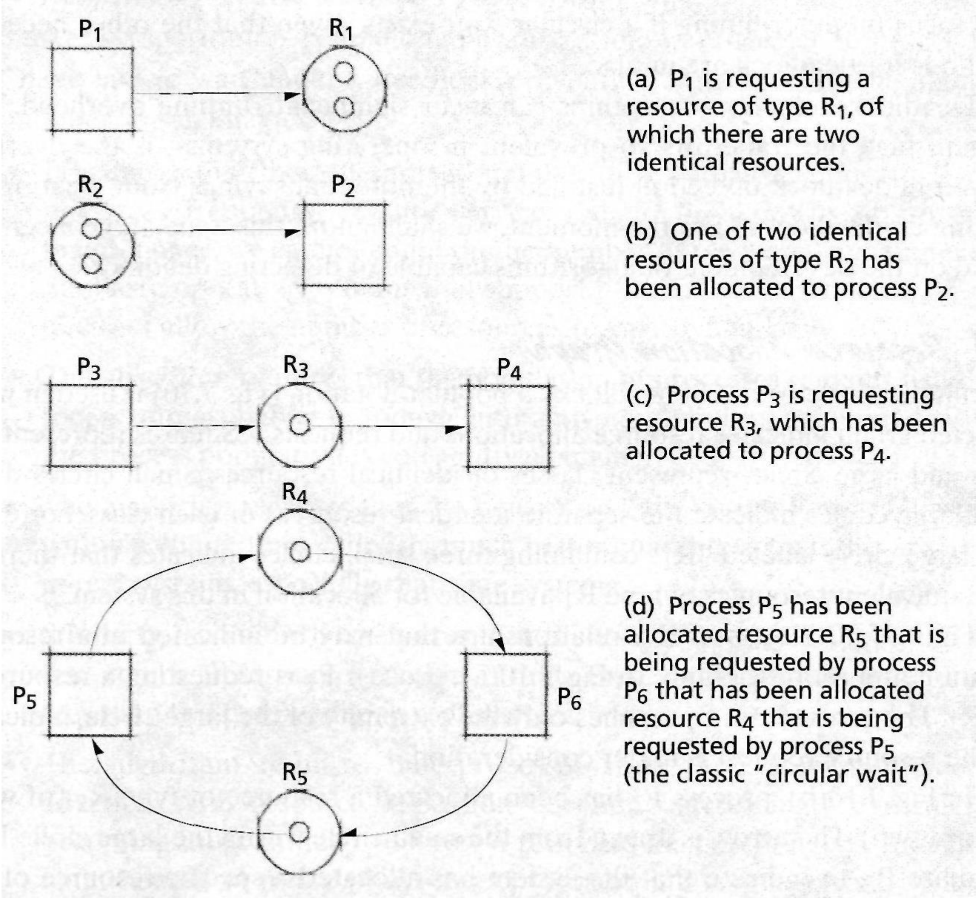

P1
R1
(a) P1 is requesting a resource of type R1, of which there are two identical resources.
R2
P2
(b) One of two identical resources of type R2 has been allocated to process P2.
P3
R3
P4
(c) Process P3 is requesting resource R3, which has been allocated to process P4.
R4
P5
P6
R5
(d) Process P5 has been allocated resource R5 that is being requested by process P6 that has been allocated resource R4 that is being requested by process P5 (the classic "circular wait").

Figure 7.10 | Resource-allocation and request graphs

- Figure 7.10(d) indicates a deadlocked system in which process P5 is requesting a resource of type R4, the onlyone of which the system has allocated to process P6.
- Process P6, is requesting a resource of type R5, the only one of which the system has allocated to process P5.
- This is an example of the "circular wait" necessary for a deadlocked system.

REDUCTION OF RESOURCE-ALLOCATION GRAPHS

One technique useful for detecting deadlocks involves graph reductions. If a process's resource requests may be granted, then we say that a graph may be **reduced** by that process. This reduction is equivalent to showing how the graph would look if the process was allowed to complete its execution and return its resources to the system.

If a graph can be reduced by all its processes, then there is no deadlock. If a graph cannot be reduced by all its processes, then the irreducible processes constitute the set of deadlocked processes in the graph.

Figure 7.11 shows a series of graph reductions demonstrating that a particular set of processes is not deadlocked.

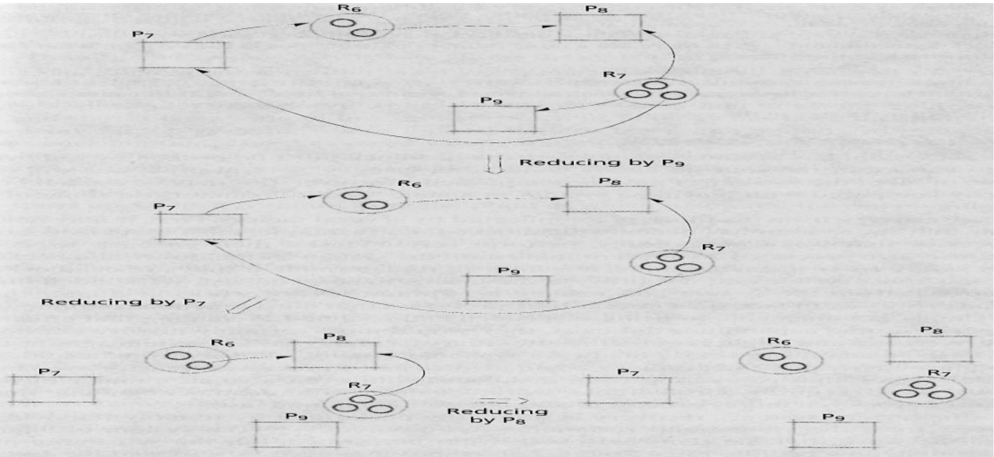


Figure 7.11 | *Graph reductions determining that no deadlock exists*

DEADLOCK RECOVERY

A traditional operating system such as Windows doesn't deal with deadlock recovery as it is time and space consuming process. Real-time operating systems use Deadlock recovery.

Recovery method

1. **Killing the process:** killing all the process involved in the deadlock. Killing process one by one. After killing each process check for deadlock again keep repeating the process till system recover from deadlock.
2. **Resource Preemption:** Resources are preempted from the processes involved in the deadlock, preempted resources are allocated to other processes so that there is a possibility of recovering the system from deadlock. In this case, the system goes into starvation.

PROCESSOR SCHEDULING

Definition

The process scheduling is the activity of the process manager that handles the removal of the running process from the CPU and the selection of another process on the basis of a particular strategy.

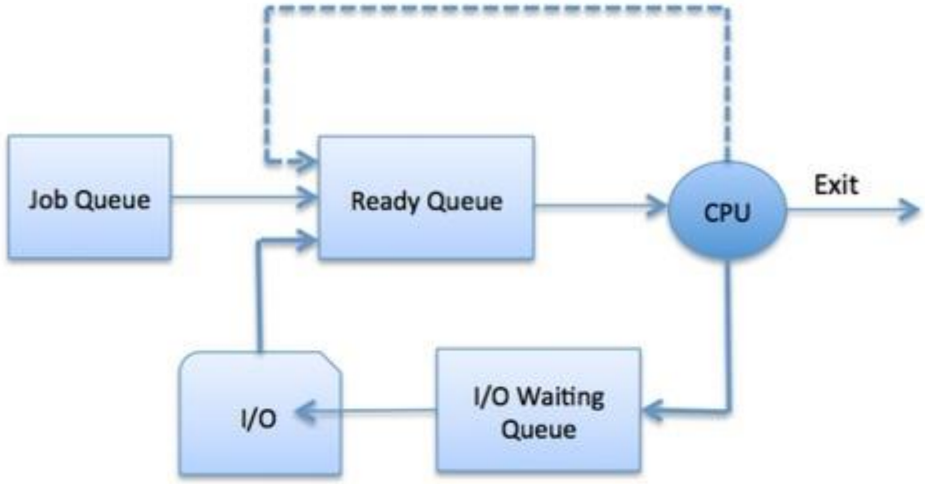
Process scheduling is an essential part of a Multiprogramming operating systems. Such operating systems allow more than one process to be loaded into the executable memory at a time and the loaded process shares the CPU using time multiplexing.

Process Scheduling Queues

The OS maintains all PCBs in Process Scheduling Queues. The OS maintains a separate queue for each of the process states and PCBs of all processes in the same execution state are placed in the same queue. When the state of a process is changed, its PCB is unlinked from its current queue and moved to its new state queue.

The Operating System maintains the following important process scheduling queues −

- **Job queue** − This queue keeps all the processes in the system.
- **Ready queue** − This queue keeps a set of all processes residing in main memory, ready and waiting to execute. A new process is always put in this queue.
- **Device queues** − The processes which are blocked due to unavailability of an I/O device constitute this queue.



The OS can use different policies to manage each queue (FIFO, Round Robin, Priority, etc.). The OS scheduler determines how to move processes between the ready and run queues which can only have one entry per processor core on the system; in the above diagram, it has been merged with the CPU.

Two-State Process Model

Two-state process model refers to running and non-running states which are described below −

S.N.	State & Description
1	**Running** When a new process is created, it enters into the system as in the running state.
2	**Not Running** Processes that are not running are kept in queue, waiting for their turn to execute. Each entry in the queue is a pointer to a particular process. Queue is implemented by using linked list. Use of dispatcher is as follows. When a process is interrupted, that process is transferred in the waiting queue. If the process has completed or aborted, the process is discarded. In either case, the dispatcher then selects a process from the queue to execute.

Schedulers

Schedulers are special system software which handle process scheduling in various ways. Their main task is to select the jobs to be submitted into the system and to decide which process to run. Schedulers are of three types −

- Long-Term Scheduler
- Short-Term Scheduler
- Medium-Term Scheduler

Long Term Scheduler

It is also called a **job scheduler**. A long-term scheduler determines which programs are admitted to the system for processing. It selects processes from the queue and loads them into memory for execution. Process loads into the memory for CPU scheduling.

The primary objective of the job scheduler is to provide a balanced mix of jobs, such as I/O bound and processor bound. It also controls the degree of multiprogramming. If the degree of multiprogramming is stable, then the average rate of process creation must be equal to the average departure rate of processes leaving the system.

On some systems, the long-term scheduler may not be available or minimal. Time-sharing operating systems have no long term scheduler. When a process changes the state from new to ready, then there is use of long-term scheduler.

Short Term Scheduler

It is also called as **CPU scheduler**. Its main objective is to increase system performance in accordance with the chosen set of criteria. It is the change of ready state to running state of the process. CPU scheduler selects a process among the processes that are ready to execute and allocates CPU to one of them.

Short-term schedulers, also known as dispatchers, make the decision of which process to execute next. Short-term schedulers are faster than long-term schedulers.

Medium Term Scheduler

Medium-term scheduling is a part of **swapping**. It removes the processes from the memory. It reduces the degree of multiprogramming. The medium-term scheduler is in-charge of handling the swapped out-processes.

A running process may become suspended if it makes an I/O request. A suspended processes cannot make any progress towards completion. In this condition, to remove the process from memory and make space for other processes, the suspended process is moved to the secondary storage. This process is called **swapping**, and the process is said to be swapped out or rolled out. Swapping may be necessary to improve the process mix.

Comparison among Scheduler

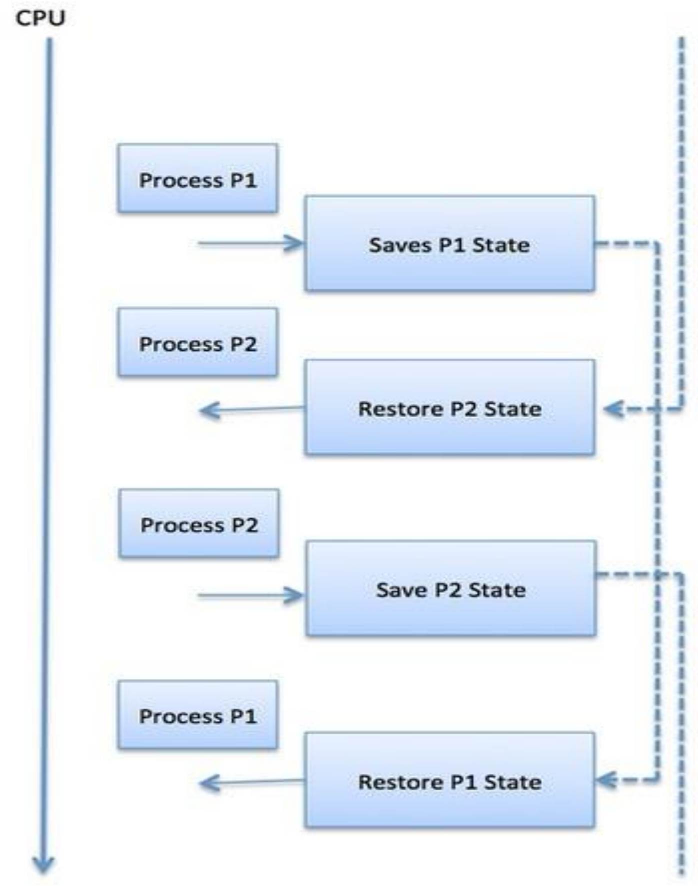
S.N.	Long-Term Scheduler	Short-Term Scheduler	Medium-Term Scheduler
1	It is a job scheduler	It is a CPU scheduler	It is a process swapping scheduler.
2	Speed is lesser than short term scheduler	Speed is fastest among other two	Speed is in between both short and long term scheduler.
3	It controls the degree of multiprogramming	It provides lesser control over degree of multiprogramming	It reduces the degree of multiprogramming.
4	It is almost absent or minimal	It is also minimal in time	It is a part of Time sharing

	in time sharing system	sharing system	systems.
5	It selects processes from pool and loads them into memory for execution	It selects those processes which are ready to execute	It can re-introduce the process into memory and execution can be continued.

Context Switch

A context switch is the mechanism to store and restore the state or context of a CPU in Process Control block so that a process execution can be resumed from the same point at a later time. Using this technique, a context switcher enables multiple processes to share a single CPU. Context switching is an essential part of a multitasking operating system features.

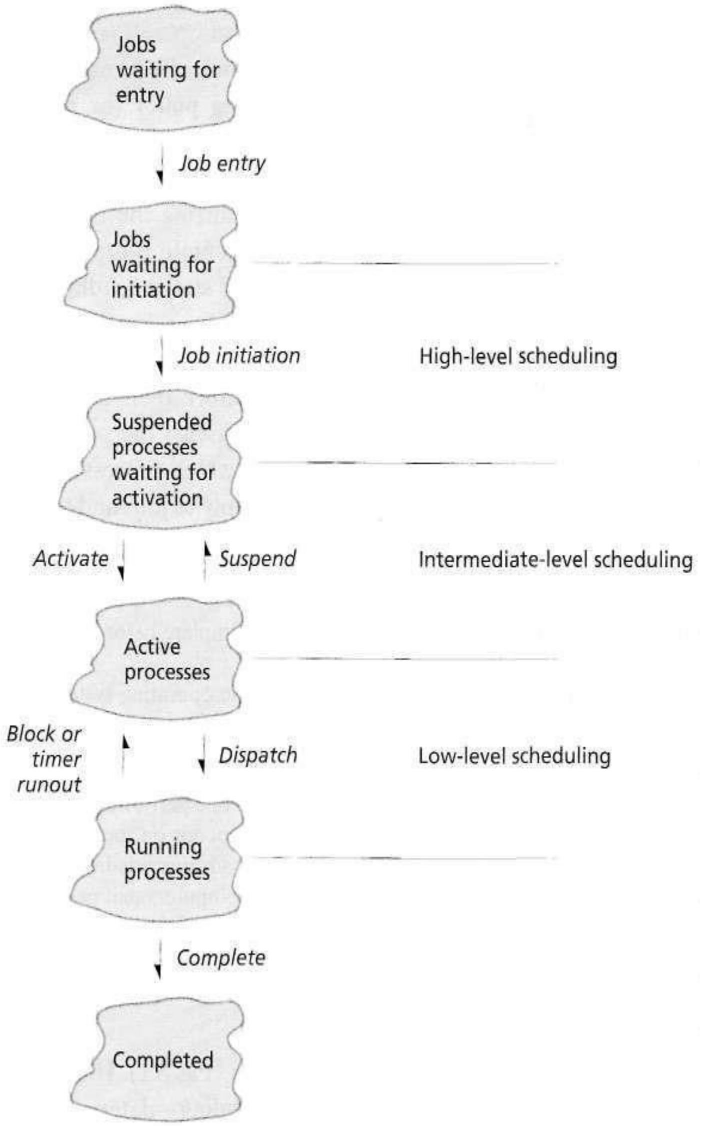
When the scheduler switches the CPU from executing one process to execute another, the state from the current running process is stored into the process control block. After this, the state for the process to run next is loaded from its own PCB and used to set the PC, registers, etc. At that point, the second process can start executing.



Context switches are computationally intensive since register and memory state must be saved and restored. To avoid the amount of context switching time, some hardware systems employ two or more sets of processor registers. When the process is switched, the following information is stored for later use.

- Program Counter
- Scheduling information
- Base and limit register value
- Currently used register
- Changed State
- I/O State information
- Accounting information

Scheduling Levels

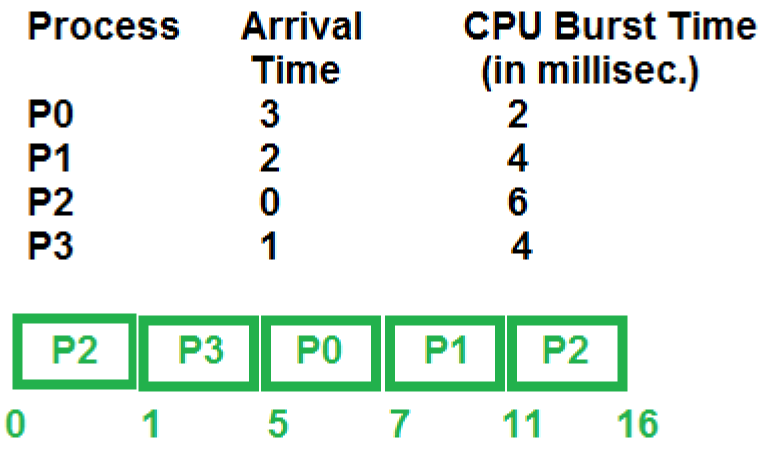


Preemptive and Non-Preemptive Scheduling

1. Preemptive Scheduling:

Preemptive scheduling is used when a process switches from running state to ready state or from waiting state to ready state. The resources (mainly CPU cycles) are allocated to the process for the limited amount of time and then is taken

away, and the process is again placed back in the ready queue if that process still has CPU burst time remaining. That process stays in ready queue till it gets next chance to execute.
Algorithms based on preemptive scheduling are: Round Robin (RR),Shortest Remaining Time First (SRTF), Priority (preemptive version), etc.

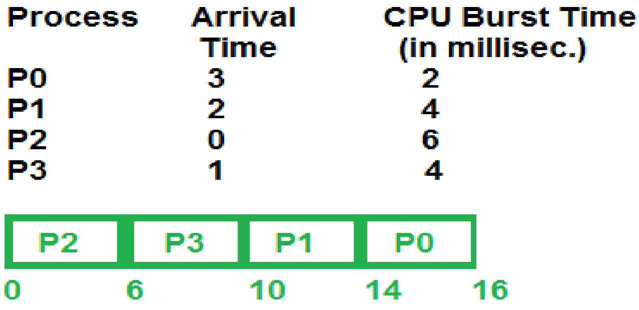


Process	Arrival Time	CPU Burst Time (in millisec.)
P0	3	2
P1	2	4
P2	0	6
P3	1	4

Preemptive Scheduling

2. Non-Preemptive Scheduling:

Non-preemptive Scheduling is used when a process terminates, or a process switches from running to waiting state. In this scheduling, once the resources (CPU cycles) is allocated to a process, the process holds the CPU till it gets terminated or it reaches a waiting state. In case of non-preemptive scheduling does not interrupt a process running CPU in middle of the execution. Instead, it waits till the process complete its CPU burst time and then it can allocate the CPU to another process.
Algorithms based on non-preemptive scheduling are: Shortest Job First (SJF basically non preemptive) and Priority (non preemptive version), etc.



Process	Arrival Time	CPU Burst Time (in millisec.)
P0	3	2
P1	2	4
P2	0	6
P3	1	4

Non-Preemtive Scheduling

Key Differences Between Preemptive and Non-Preemptive Scheduling:

1.In preemptive scheduling the CPU is allocated to the processes for the limited time whereas in Non-preemptive scheduling, the CPU is allocated to the process till it terminates or switches to waiting state.

2. The executing process in preemptive scheduling is interrupted in the middle of execution when higher priority one comes whereas, the executing process in non-preemptive scheduling is not interrupted in the

middle of execution and wait till its execution.

3. In Preemptive Scheduling, there is the overhead of switching the process from ready state to running state, vise-verse, and maintaining the ready queue. Whereas in case of non-preemptive scheduling has no overhead of switching the process from running state to ready state.

4. In preemptive scheduling, if a high priority process frequently arrives in the ready queue then the process with low priority has to wait for a long, and it may have to starve. On the other hands, in the non-preemptive scheduling, if CPU is allocated to the process having larger burst time then the processes with small burst time may have to starve.

5. Preemptive scheduling attain flexible by allowing the critical processes to access CPU as they arrive into the ready queue, no matter what process is executing currently. Non-preemptive scheduling is called rigid as even if a critical process enters the ready queue the process running CPU is not disturbed.

6. The Preemptive Scheduling has to maintain the integrity of shared data that's why it is cost associative as it which is not the case with Non-preemptive Scheduling.

PRIORITIES

Priority scheduling is a method of scheduling processes based on priority. In this method, the scheduler chooses the tasks to work as per the priority, which is different from other types of scheduling, for example, a simple round robin.

Priority scheduling involves priority assignment to every process, and processes with higher priorities are carried out first, whereas tasks with equal priorities are carried out on a first-come-first-served (FCFS) or round robin basis. An example of a general-priority-scheduling algorithm is the shortest-job-first (SJF) algorithm.

Priority Scheduling

Priorities can be either dynamic or static.

Static priorities are allocated during creation, whereas dynamic priorities are assigned depending on the behavior of the processes while in the system. To illustrate, the scheduler could favor input/output (I/O) intensive tasks, which lets expensive requests to be issued as soon as possible.

Priorities may be defined internally or externally. Internally defined priorities make use of some measurable quantity to calculate the priority of a given process. In contrast, external priorities are defined using criteria beyond the operating system (OS), which can include the significance of the process, the type as well as the sum of resources being utilized for computer use, user preference, commerce and other factors like politics, etc.

Priority scheduling can be either of the following:

- Preemptive: This type of scheduling may preempt the central processing unit (CPU) in the case the priority of the freshly arrived process being greater than those of the existing processes.
- Non-preemptive: This type of scheduling algorithm simply places the new process at the top of the ready queue.

Indefinite blocking, otherwise called starvation, is one of the major issues concerning priority scheduling algorithms. It is a state where a process is ready to be executed, but faces a long wait in getting assigned to the CPU.

It is often possible that a priority scheduling algorithm can make a low-priority process wait indefinitely. For example, in an intensely loaded system, if there are a number of higher priority processes, the low-priority processes may never get the CPU for execution.

A remedy to starvation is aging, which is a technique used to gradually increase the priority of those processes that wait for long periods in the system.

SCHEDULING OBJECTIVES

Depending on the system, the user and designer might expect the scheduler to:

1. *Maximize throughput. A* scheduling discipline should attempt to service the maximum number of processes per unit time.
2. *Maximize the number of interactive processes receiving "acceptable" response times.*
3. *Maximize resource utilization.* The scheduling mechanisms should keep the resources of the system busy.
4. *Avoid indefinite postponement. A* process should not experience an unbounded wait time before or while receiving service.
5. *Enforce priorities.* If the system assigns priorities to processes, the scheduling mechanism should favor the higher-priority processes.
6. *Minimize overhead.* Interestingly, this is not generally considered to be one of the most important objectives. Overhead often results in wasted resources. Overhead can greatly improve overall system performance.
7. *Ensure predictability.* By minimizing the statistical variance in process response times, a system can guarantee that processes will receive predictable service levels (see the Operating Systems Thinking feature, Predictability).

Despite the differences in goals among systems, many scheduling disciplines exhibit similar properties:

8. **Fairness.** A scheduling discipline is fair if all similar processes are treated the same, and no process can suffer indefinite postponement due to scheduling issues (see the Operating Systems Thinking feature, Fairness).
9. **Predictability.** A given process always should run in about the same amount of time under similar system loads.
10. **Scalability.** System performance should degrade gracefully (i.e., it should not immediately collapse) under heavy loads.

Scheduling Criteria:

Many criteria have been suggested for comparison of CPU scheduling algorithms.

CPU utilization: we have to keep the CPU as busy as possible. It may range from 0 to 100%. In a real system it should range from 40 – 90 % for lightly and heavily loaded system.

Throughput: It is the measure of work in terms of number of process completed per unit time. For long process this rate may be 1 process per hour, for short transaction, throughput may be 10 process per second.

Turnaround Time: It is the sum of time periods spent in waiting to get into memory, waiting in ready queue, execution on the CPU and doing I/O. The interval form the time of submission of a process to the time of completion is the turnaround time. Waiting time plus the service time.

Turnaround time= Time of completion of job - Time of submission of job. (waiting time + service time or burst time)

Waiting time: its the sum of periods waiting in the ready queue.

Response time: in interactive system the turnaround time is not the best criteria. Response time is the amount of time it takes to start responding, not the time taken to output that response. Types of

Scheduling: Scheduling algorithms can be divided into two categories with respect to how they deal with clock interrupts. 1.Preemptive Scheduling

2.Non preemptive Scheduling

Nonpreemptive scheduling algorithm picks a process to run and then just lets it run until it blocks (either on I/O or waiting for another process) or until it voluntarily releases the CPU. Even it runs for hours, it will not be forceably suspended. In effect no scheduling decisions are made during clock interrupts.

In contrasts, a preemptive scheduling algorithm picks a process and lets it run for a maximum of some fixed time. If it is still running at the end of the time interval, it is suspended and the scheduler picks another process to run (if one is available). Doing peemptive scheduling requires having a clock interrupt occur at the end of time interval to give control of the CPU back to the scheduler. If no clock is available, nonpreemptive scheduling is only the option.

CPU scheduling decisions may take place under the following four circumstances:

1. When a process switches from the running state to the waiting state (for example , I/O requests, or invocation of wait for the termination of one of the child process).

2. When a process switches from the running state to the ready state (for example, when an interrupt occurs).

3. When a process switches from the waiting state to the ready state(for example completion of I/O)

4. When a process terminates.

SCHEDULING ALGORITHMS

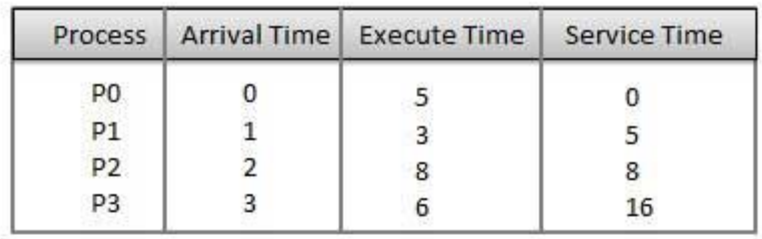
A Process Scheduler schedules different processes to be assigned to the CPU based on particular scheduling algorithms. There are six popular process scheduling algorithms which we are going to discuss in this chapter −

- First-Come, First-Served (FCFS) Scheduling
- Shortest-Job-Next (SJN) Scheduling
- Priority Scheduling
- Shortest Remaining Time
- Round Robin(RR) Scheduling
- Multiple-Level Queues Scheduling

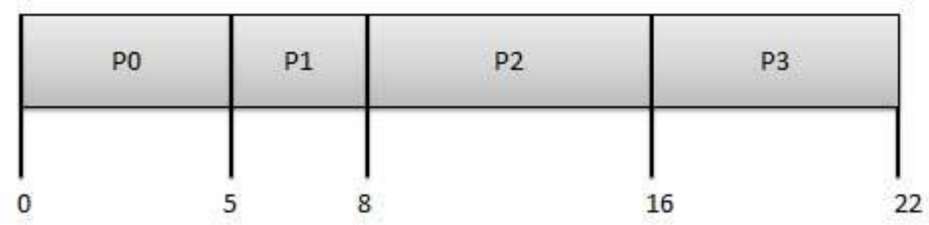
These algorithms are either **non-preemptive or preemptive**. Non-preemptive algorithms are designed so that once a process enters the running state, it cannot be preempted until it completes its allotted time, whereas the preemptive scheduling is based on priority where a scheduler may preempt a low priority running process anytime when a high priority process enters into a ready state.

First Come First Serve (FCFS)

- Jobs are executed on first come, first serve basis.
- It is a non-preemptive, pre-emptive scheduling algorithm.
- Easy to understand and implement.
- Its implementation is based on FIFO queue.
- Poor in performance as average wait time is high.



Process	Arrival Time	Execute Time	Service Time
P0	0	5	0
P1	1	3	5
P2	2	8	8
P3	3	6	16



Wait time of each process is as follows −

Process	**Wait Time : Service Time - Arrival Time**
P0	0 - 0 = 0
P1	5 - 1 = 4

P2	8 - 2 = 6
P3	16 - 3 = 13

Average Wait Time: (0+4+6+13) / 4 = 5.75

Shortest Job Next (SJN)

- This is also known as **shortest job first**, or SJF
- This is a non-preemptive, pre-emptive 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 processer should know in advance how much time process will take.

Given: Table of processes, and their Arrival time, Execution time

Process	Arrival Time	Execution Time	Service Time
P0	0	5	0
P1	1	3	5
P2	2	8	14
P3	3	6	8

Waiting time of each process is as follows −

Process	Waiting Time
P0	0 - 0 = 0
P1	5 - 1 = 4
P2	14 - 2 = 12
P3	8 - 3 = 5

Average Wait Time: (0 + 4 + 12 + 5)/4 = 21 / 4 = 5.25

Priority Based 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.

Given: Table of processes, and their Arrival time, Execution time, and priority. Here we are considering 1 is the lowest priority.

Process	Arrival Time	Execution Time	Priority	Service Time
P0	0	5	1	0
P1	1	3	2	11
P2	2	8	1	14
P3	3	6	3	5

Waiting time of each process is as follows −

Process	Waiting Time
P0	0 - 0 = 0
P1	11 - 1 = 10
P2	14 - 2 = 12
P3	5 - 3 = 2

Average Wait Time: (0 + 10 + 12 + 2)/4 = 24 / 4 = 6

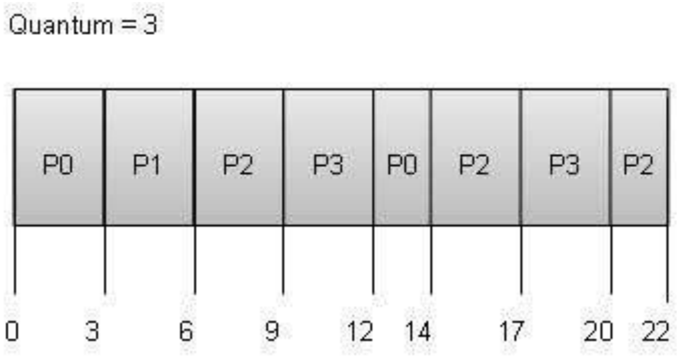
Shortest Remaining Time

- 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.

Round Robin Scheduling

- 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.



Wait time of each process is as follows −

Process	Wait Time : Service Time - Arrival Time
P0	(0 - 0) + (12 - 3) = 9
P1	(3 - 1) = 2
P2	(6 - 2) + (14 - 9) + (20 - 17) = 12
P3	(9 - 3) + (17 - 12) = 11

Average Wait Time: (9+2+12+11) / 4 = 8.5

Multiple-Level Queues Scheduling

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.

UNIT IV

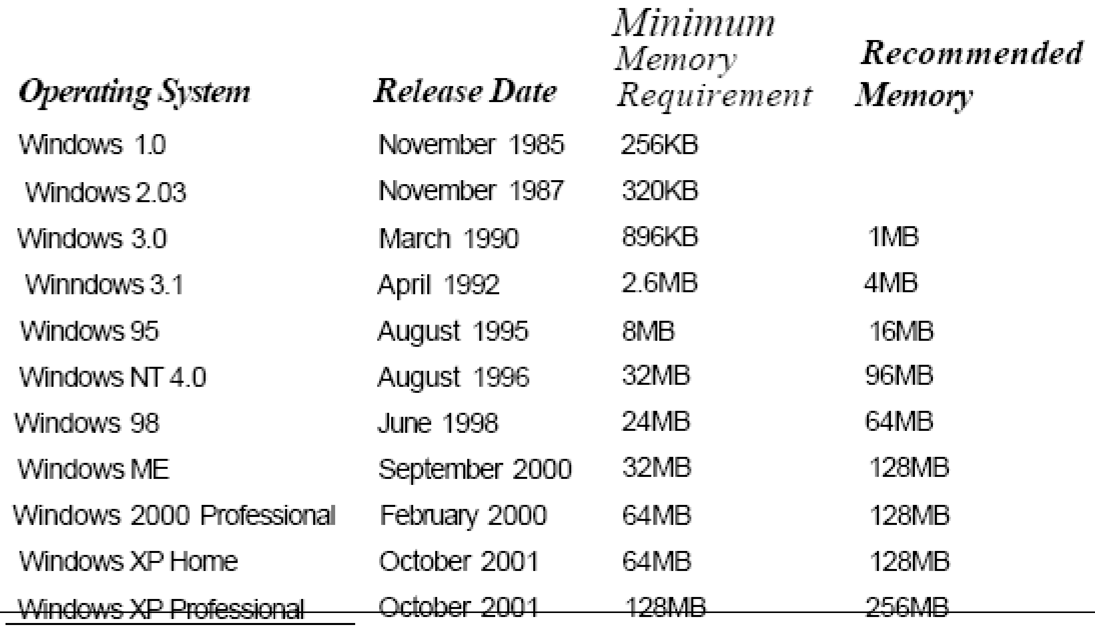
REAL MEMORY ORGANIZATION AND MANAGEMENT

INTRODUCTION

- The organization and management of the real memory (also called main memory, physical memory or primary memory) of a computer system has been a major influence on operating systems design.
- Secondary storage—most commonly disk and tape - provides massive, inexpensive capacity for the abundance of programs and data that must be kept readily available for processing. Main memory requires careful management.

MEMORY ORGANIZATION

- Main memory is still relatively expensive compared to secondary storage.
- Also, today's operating systems and applications require ever more substantial quantities (Fig. 9.1).
- For example, Microsoft recommends 256MB of main memory to efficiently run Windows XP Professional.



Operating System	*Release Date*	*Minimum Memory Requirement*	*Recommended Memory*
Windows 1.0	November 1985	256KB	
Windows 2.03	November 1987	320KB	
Windows 3.0	March 1990	896KB	1MB
Winndows 3.1	April 1992	2.6MB	4MB
Windows 95	August 1995	8MB	16MB
Windows NT 4.0	August 1996	32MB	96MB
Windows 98	June 1998	24MB	64MB
Windows ME	September 2000	32MB	128MB
Windows 2000 Professional	February 2000	64MB	128MB
Windows XP Home	October 2001	64MB	128MB
Windows XP Professional	October 2001	128MB	256MB

Memory management is the functionality of an operating system which handles or manages primary memory and moves processes back and forth between main memory and disk during execution. Memory management keeps track of each and every memory location, regardless of either it is allocated to some process or it is free. It checks how much memory is to be allocated to processes. It decides which process will get memory at what time. It tracks whenever some memory gets freed or unallocated and correspondingly it updates the status.

Process Address Space

The process address space is the set of logical addresses that a process references in its code. For example, when 32-bit addressing is in use, addresses can range from 0 to 0x7fffffff; that is, 2^31 possible numbers, for a total theoretical size of 2 gigabytes.

The operating system takes care of mapping the logical addresses to physical addresses at the time of memory allocation to the program. There are three types of addresses used in a program before and after memory is allocated −

S.N.	Memory Addresses & Description
1	**Symbolic addresses** The addresses used in a source code. The variable names, constants, and instruction labels are the basic elements of the symbolic address space.
2	**Relative addresses** At the time of compilation, a compiler converts symbolic addresses into relative addresses.
3	**Physical addresses** The loader generates these addresses at the time when a program is loaded into main memory.

Virtual and physical addresses are the same in compile-time and load-time address-binding schemes. Virtual and physical addresses differ in execution-time address-binding scheme.

The set of all logical addresses generated by a program is referred to as a **logical address space**. The set of all physical addresses corresponding to these logical addresses is referred to as a **physical address space.**

The runtime mapping from virtual to physical address is done by the memory management unit (MMU) which is a hardware device. MMU uses following mechanism to convert virtual address to physical address.

- The value in the base register is added to every address generated by a user process, which is treated as offset at the time it is sent to memory. For example, if the base register value is 10000, then an attempt by the user to use address location 100 will be dynamically reallocated to location 10100.
- The user program deals with virtual addresses; it never sees the real physical addresses.

Static vs Dynamic Loading

The choice between Static or Dynamic Loading is to be made at the time of computer program being developed. If you have to load your program statically, then at the time of compilation, the complete programs will be compiled and linked without leaving any external program or module dependency. The linker combines the object program with other necessary object modules into an absolute program, which also includes logical addresses.

If you are writing a Dynamically loaded program, then your compiler will compile the program and for all the modules which you want to include dynamically, only references will be provided and rest of the work will be done at the time of execution.

At the time of loading, with **static loading**, the absolute program (and data) is loaded into memory in order for execution to start.

If you are using **dynamic loading**, dynamic routines of the library are stored on a disk in relocatable form and are loaded into memory only when they are needed by the program.

Static vs Dynamic Linking

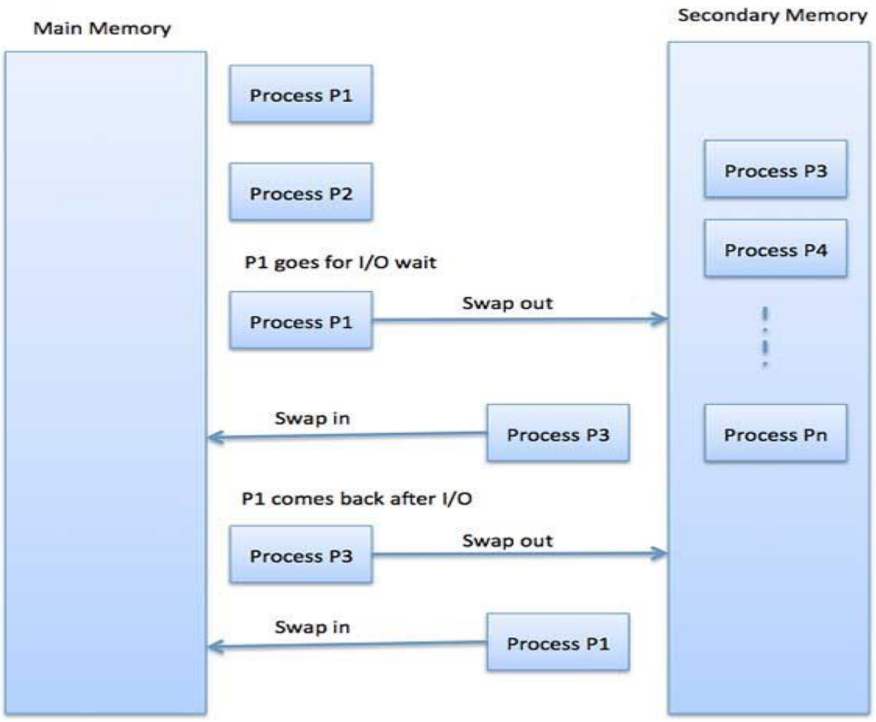
As explained above, when static linking is used, the linker combines all other modules needed by a program into a single executable program to avoid any runtime dependency.

When dynamic linking is used, it is not required to link the actual module or library with the program, rather a reference to the dynamic module is provided at the time of compilation and linking. Dynamic Link Libraries (DLL) in Windows and Shared Objects in Unix are good examples of dynamic libraries.

Swapping

Swapping is a mechanism in which a process can be swapped temporarily out of main memory (or move) to secondary storage (disk) and make that memory available to other processes. At some later time, the system swaps back the process from the secondary storage to main memory.

Though performance is usually affected by swapping process but it helps in running multiple and big processes in parallel and that's the reason Swapping is also known as a technique for memory compaction.



The total time taken by swapping process includes the time it takes to move the entire process to a secondary disk and then to copy the process back to memory, as well as the time the process takes to regain main memory.

Let us assume that the user process is of size 2048KB and on a standard hard disk where swapping will take place has a data transfer rate around 1 MB per second. The actual transfer of the 1000K process to or from memory will take

2048KB / 1024KB per second
= 2 seconds
= 2000 milliseconds

Now considering in and out time, it will take complete 4000 milliseconds plus other overhead where the process competes to regain main memory.

Memory Allocation

Main memory usually has two partitions −

- **Low Memory** − Operating system resides in this memory.
- **High Memory** − User processes are held in high memory.

Operating system uses the following memory allocation mechanism.

S.N.	Memory Allocation & Description
1	**Single-partition allocation** In this type of allocation, relocation-register scheme is used to protect user processes from each other, and from changing operating-system code and data. Relocation register contains value of smallest physical address whereas limit register contains range of logical addresses. Each logical address must be less than the limit register.
2	**Multiple-partition allocation** In this type of allocation, main memory is divided into a number of fixed-sized partitions where each partition should contain only one process. When a partition is free, a process is selected from the input queue and is loaded into the free partition. When the process terminates, the partition becomes available for another process.

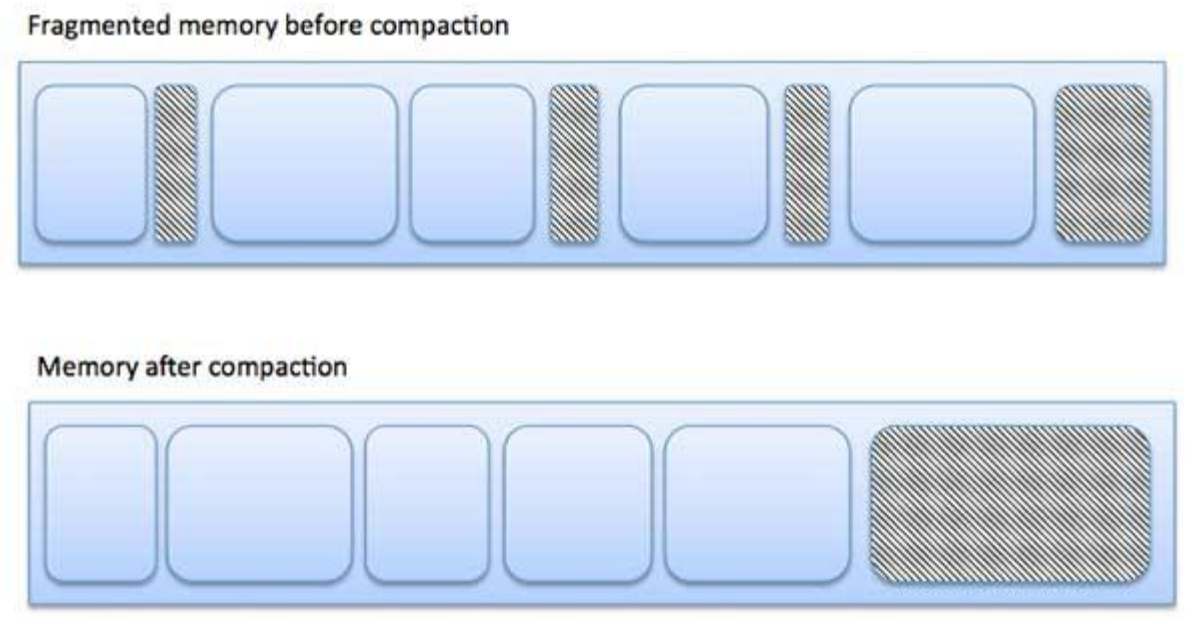
Fragmentation

As processes are loaded and removed from memory, the free memory space is broken into little pieces. It happens after sometimes that processes cannot be allocated to memory blocks considering their small size and memory blocks remains unused. This problem is known as Fragmentation.

Fragmentation is of two types −

S.N.	Fragmentation & Description
1	**External fragmentation** Total memory space is enough to satisfy a request or to reside a process in it, but it is not contiguous, so it cannot be used.
2	**Internal fragmentation** Memory block assigned to process is bigger. Some portion of memory is left unused, as it cannot be used by another process.

The following diagram shows how fragmentation can cause waste of memory and a compaction technique can be used to create more free memory out of fragmented memory −



Fragmented memory before compaction

Memory after compaction

External fragmentation can be reduced by compaction or shuffle memory contents to place all free memory together in one large block. To make compaction feasible, relocation should be dynamic.

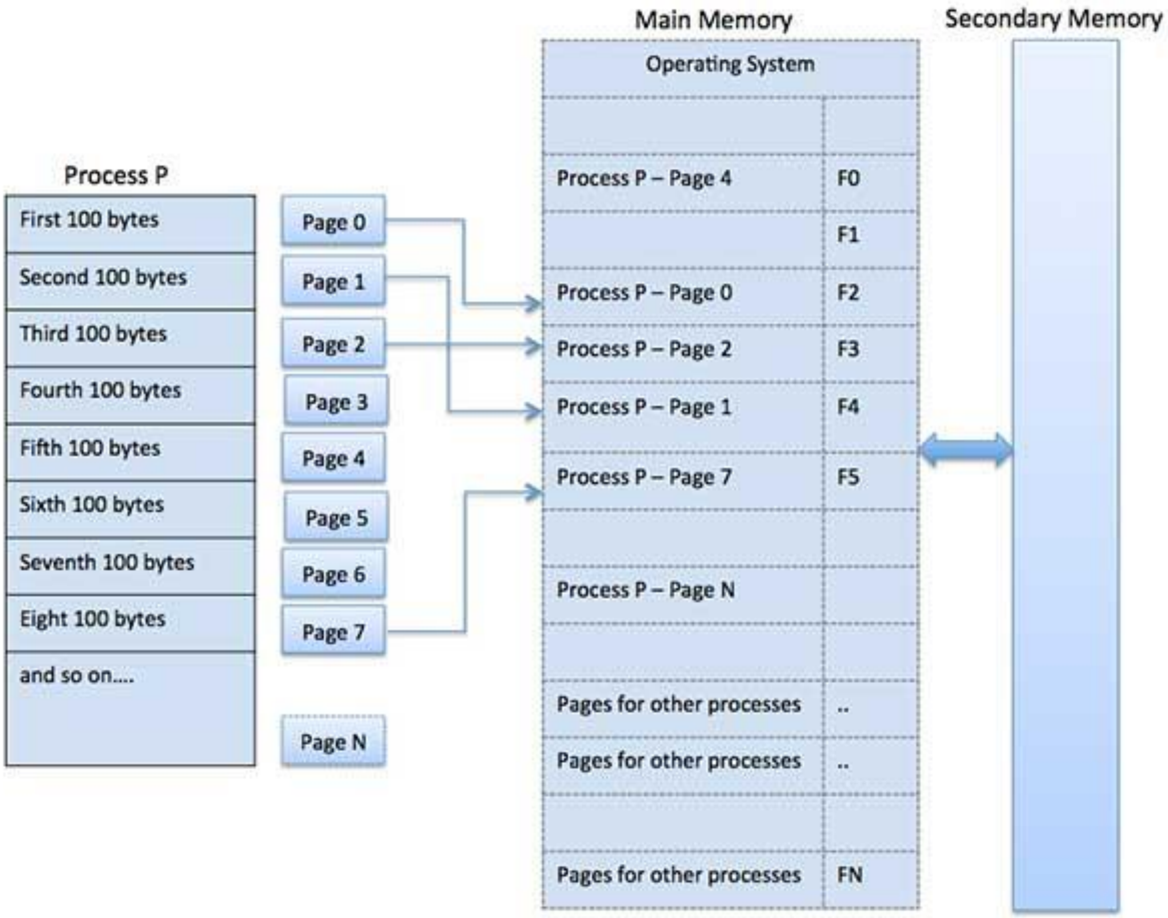
The internal fragmentation can be reduced by effectively assigning the smallest partition but large enough for the process.

Paging

A computer can address more memory than the amount physically installed on the system. This extra memory is actually called virtual memory and it is a section of a hard that's set up to emulate the computer's RAM. Paging technique plays an important role in implementing virtual memory.

Paging is a memory management technique in which process address space is broken into blocks of the same size called **pages** (size is power of 2, between 512 bytes and 8192 bytes). The size of the process is measured in the number of pages.

Similarly, main memory is divided into small fixed-sized blocks of (physical) memory called **frames** and the size of a frame is kept the same as that of a page to have optimum utilization of the main memory and to avoid external fragmentation.



Address Translation

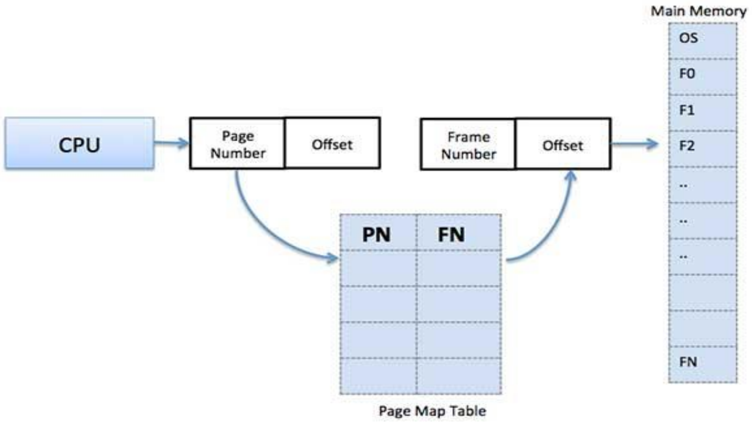
Page address is called **logical address** and represented by **page number** and the **offset**.

Logical Address = Page number + page offset

Frame address is called **physical address** and represented by a **frame number** and the **offset**.

Physical Address = Frame number + page offset

A data structure called **page map table** is used to keep track of the relation between a page of a process to a frame in physical memory.



When the system allocates a frame to any page, it translates this logical address into a physical address and create entry into the page table to be used throughout execution of the program.

When a process is to be executed, its corresponding pages are loaded into any available memory frames. Suppose you have a program of 8Kb but your memory can accommodate only 5Kb at a given point in time, then the paging concept will come into picture. When a computer runs out of RAM, the operating system (OS) will move idle or unwanted pages of memory to secondary memory to free up RAM for other processes and brings them back when needed by the program.

This process continues during the whole execution of the program where the OS keeps removing idle pages from the main memory and write them onto the secondary memory and bring them back when required by the program.

Advantages and Disadvantages of Paging

Here is a list of advantages and disadvantages of paging −

- Paging reduces external fragmentation, but still suffer from internal fragmentation.
- Paging is simple to implement and assumed as an efficient memory management technique.
- Due to equal size of the pages and frames, swapping becomes very easy.
- Page table requires extra memory space, so may not be good for a system having small RAM.

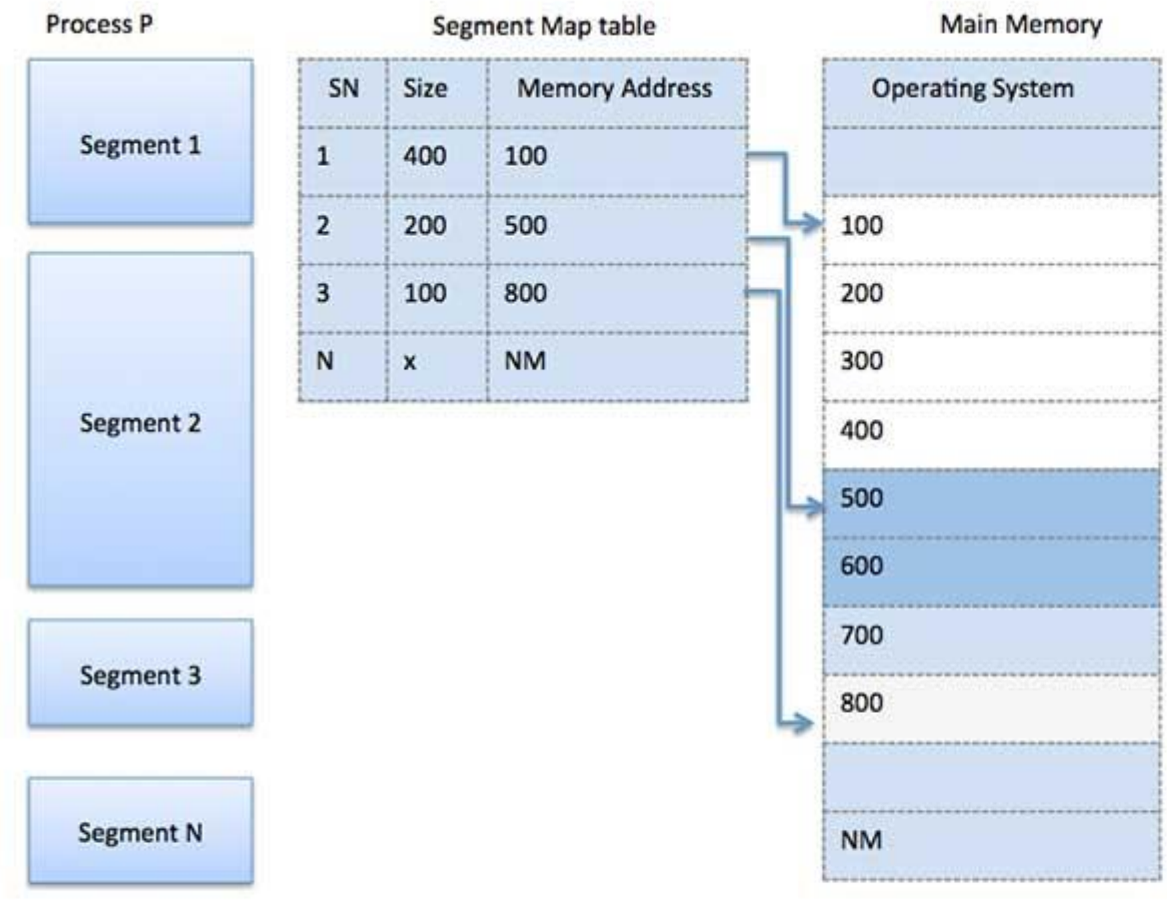
Segmentation

Segmentation is a memory management technique in which each job is divided into several segments of different sizes, one for each module that contains pieces that perform related functions. Each segment is actually a different logical address space of the program.

When a process is to be executed, its corresponding segmentation are loaded into non-contiguous memory though every segment is loaded into a contiguous block of available memory.

Segmentation memory management works very similar to paging but here segments are of variable-length where as in paging pages are of fixed size.

A program segment contains the program's main function, utility functions, data structures, and so on. The operating system maintains a **segment map table** for every process and a list of free memory blocks along with segment numbers, their size and corresponding memory locations in main memory. For each segment, the table stores the starting address of the segment and the length of the segment. A reference to a memory location includes a value that identifies a segment and an offset.



SN	Size	Memory Address
1	400	100
2	200	500
3	100	800
N	x	NM

.

- The memory manager determines how available memory space is allocated to processes and how to respond to changes in a process's memory usage.
- It also interacts with special-purpose memory management hardware (if any is available) to improve performance.

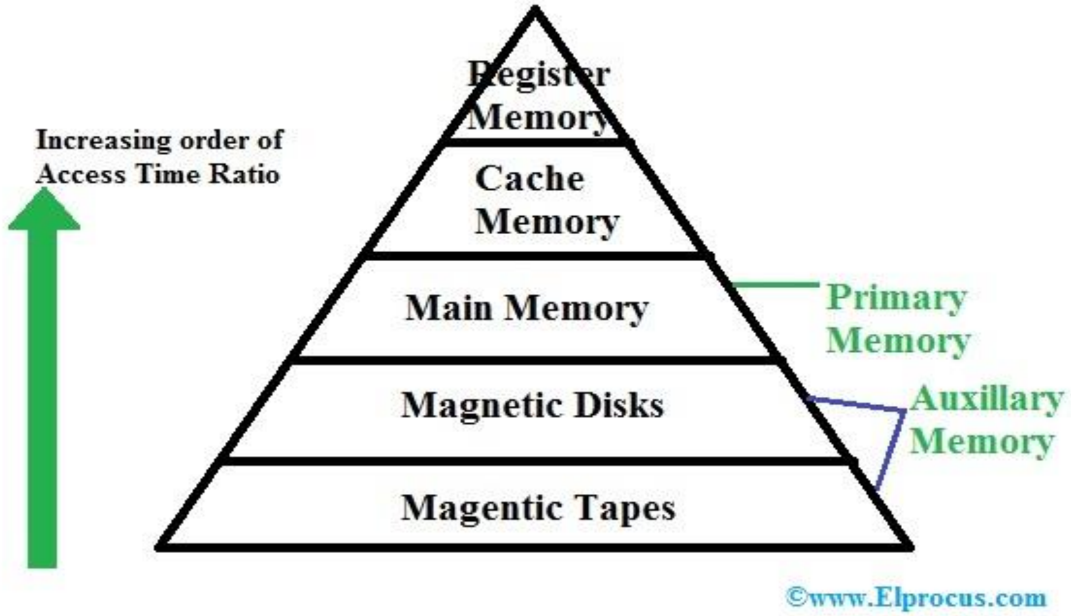
  – Performed by memory manager

  - Which process will stay in memory?
  - How much memory will each process have access to?
  - Where in memory will each process go?

MEMORY HIERARCHY

Memory is categorized into **volatile and nonvolatile memories**, with the former requiring constant power ON of the system to maintain data storage.

Furthermore, a typical computer system provides a hierarchy of different times of memories for data storage.



Different levels of the memory hierarchy

Cache (MB): Cache is the fastest accessible memory of a computer system. It's access speed is in the order of a few nanoseconds. It is volatile and expensive, so the typical cache size is in the order of megabytes.

Main memory (GB): Main memory is arguably the most used memory. When discussing computer algorithms such as quick sort, balanced binary sorted trees, or fast Fourier transform, one typically assumes that the algorithm operates on data stored in the main memory. The main memory is reasonably fast, with access speed around 100 nanoseconds. It also offers larger capacity at a lower cost. Typical main memory is in the order of 10 GB. However, the main memory is volatile.

Secondary storage (TB): Secondary storage refers to nonvolatile data storage units that are external to the computer system. Hard drives and solid state drives are examples of secondary storage. They offer very large storage capacity in the order of terabytes at very low cost. Therefore, database servers typically have an array of secondary storage devices with data stored distributed and redundantly across these devices. Despite the continuous improvements in access speed of hard drives, secondary storage devices are several magnitudes slower than main memory. Modern hard drives have access speed in the order of a few milliseconds.

Tertiary storage (PB): Tertiary storage refers storage designed for the purpose data backup. Examples of tertiary storage devices are tape drives are robotic driven disk arrays. They are capable of petabyte range storage, but have very slow access speed with data access latency in seconds or minutes.

MEMORY MANAGEMENT STRATEGIES

They are divided into:

1. Fetch strategies
2. Placement strategies
3. Replacement strategies

Fetch strategies determine when to move the next piece of a program or data to main memory from secondary storage. We divide them into two types.

- i. **Demand fetch strategies**
- ii. **Anticipatory fetch strategies.**

Placement strategies determine where in main memory the system should place incoming program or data pieces. We consider the **first-fit**, **best-fit,** and **worst-fit** memory placement strategies.

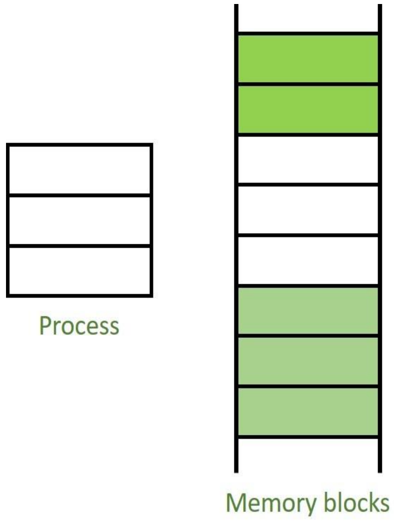
Replacement strategies

- When memory is too full to accommodate a new program, the system must remove some (or all) of a program or data that currently resides in memory.
- The system's **replacement strategy** determines which piece to remove.

CONTIGUOUS VS. NONCONTIGUOUS MEMORY ALLOCATION

1. Contiguous Memory Allocation :

Contiguous memory allocation is basically a method in which a single contiguous section/part of memory is allocated to a process or file needing it. Because of this all the available memory space resides at the same place together, which means that the freely/unused available memory partitions are not distributed in a random fashion here and there across the whole memory space.

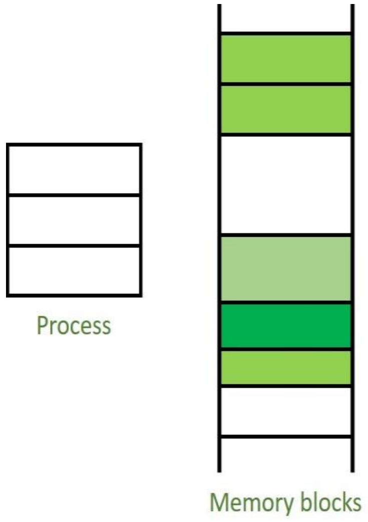


Contiguous Memory Allocation

10 The main memory is a combination of two main portions- one for the operating system and other for the user program. We can implement/achieve contiguous memory allocation by dividing the memory partitions into fixed size partitions.

2. Non-Contiguous Memory Allocation :

Non-Contiguous memory allocation is basically a method on the contrary to contiguous allocation method, allocates the memory space present in different locations to the process as per it's requirements. As all the available memory space is in a distributed pattern so the freely available memory space is also scattered here and there.
This technique of memory allocation helps to reduce the wastage of memory, which eventually gives rise to Internal and external fragmentation.



Noncontiguous Memory Allocation

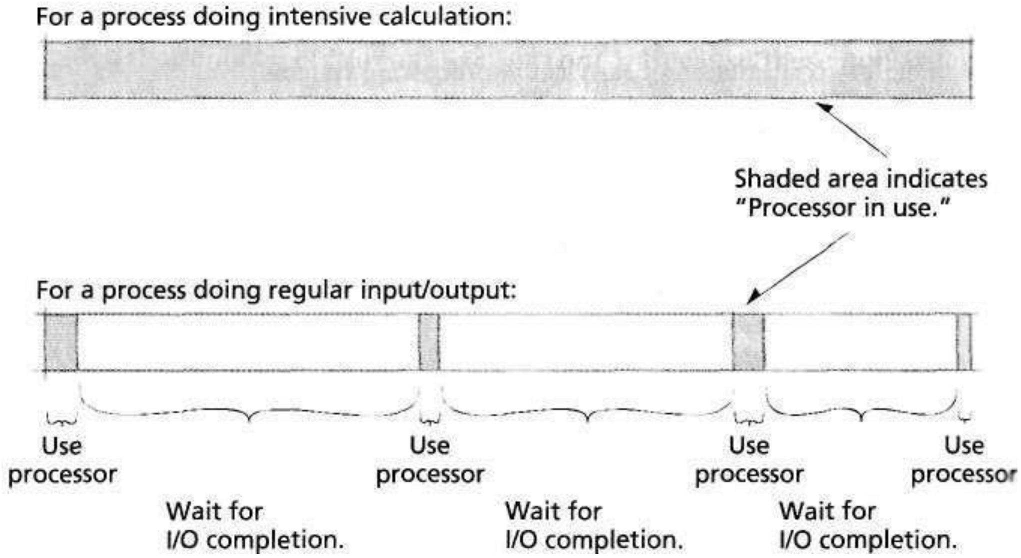
Difference between Contiguous and Non-contiguous Memory Allocation :

S.NO.	CONTIGUOUS MEMORY ALLOCATION	NON-CONTIGUOUS MEMORY ALLOCATION
1.	Contiguous memory allocation allocates consecutive blocks of memory to a file/process.	Non-Contiguous memory allocation allocates separate blocks of memory to a file/process.
2.	Faster in Execution.	Slower in Execution.
3.	It is easier for the OS to control.	It is difficult for the OS to control.
4.	Overhead is minimum as not much address translations are there while executing a process.	More Overheads are there as there are more address translations.
5.	Internal fragmentation occurs in Contiguous memory allocation method.	External fragmentation occurs in Non-Contiguous memory allocation method.
6.	It includes single partition allocation and multi-partition allocation.	It includes paging and segmentation.
7.	Wastage of memory is there.	No memory wastage is there.
8.	In contiguous memory allocation, swapped-in processes are arranged in the originally allocated space.	In non-contiguous memory allocation, swapped-in processes can be arranged in any place in the memory.

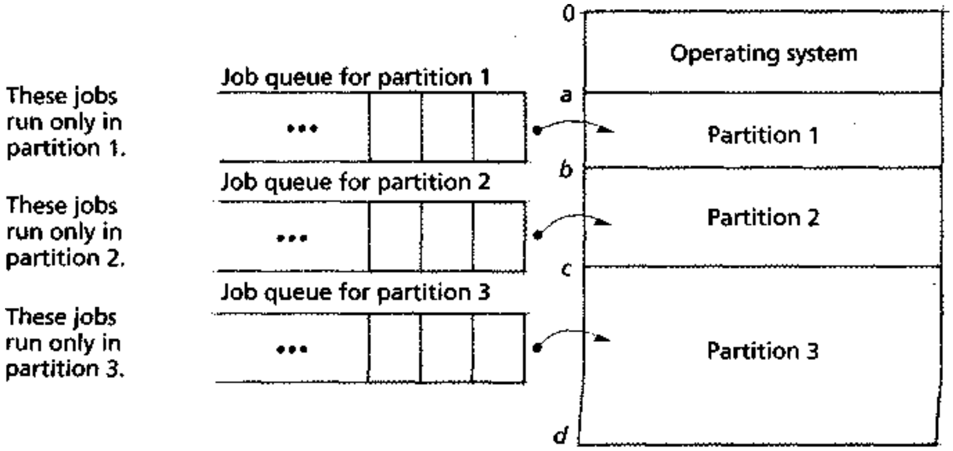
FIXED-PARTITION MULTIPROGRAMMING

- A typical process would consume the processor time it needed to generate an input/output request; the process could not continue until the I/O finished.
- To take maximum advantage of multiprogramming, several processes must reside in the computer's main memory at the same time.
- Thus, when one process requests input/output, the processor may switch to

another process and continue to perform calculations without the delay.

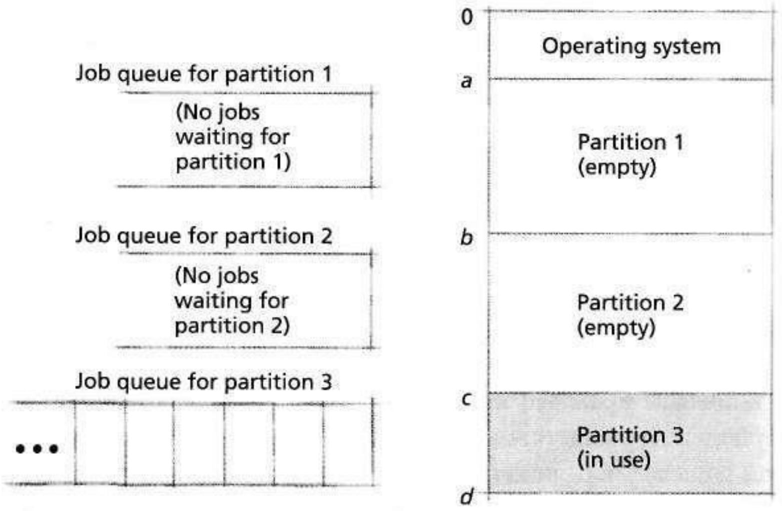


- The system divides main memory into a number of fixed size partitions.
- Each partition holds a single job, and the system switches the processor rapidly between jobs to create the illusion of simultaneity.
- This technique enables the system to provide simple multiprogramming capabilities.

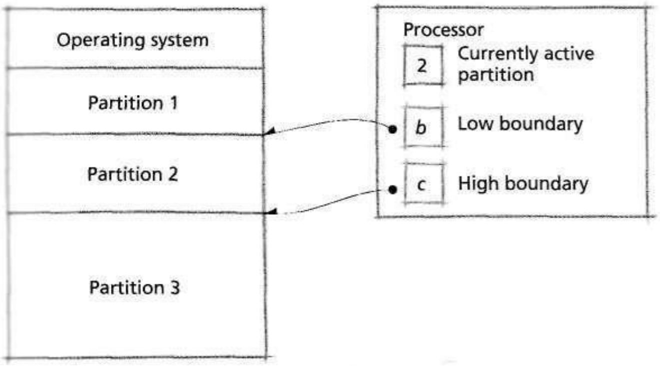


- This restriction led to wasted memory.

- If a job was ready to run and the program's partition was occupied, then that job had to wait, even if other partitions were available.


0
Operating system
a
Job queue for partition 1
(No jobs waiting for partition 1)
Partition 1 (empty)
b
Job queue for partition 2
(No jobs waiting for partition 2)
Partition 2 (empty)
Job queue for partition 3
c
Partition 3 (in use)
d

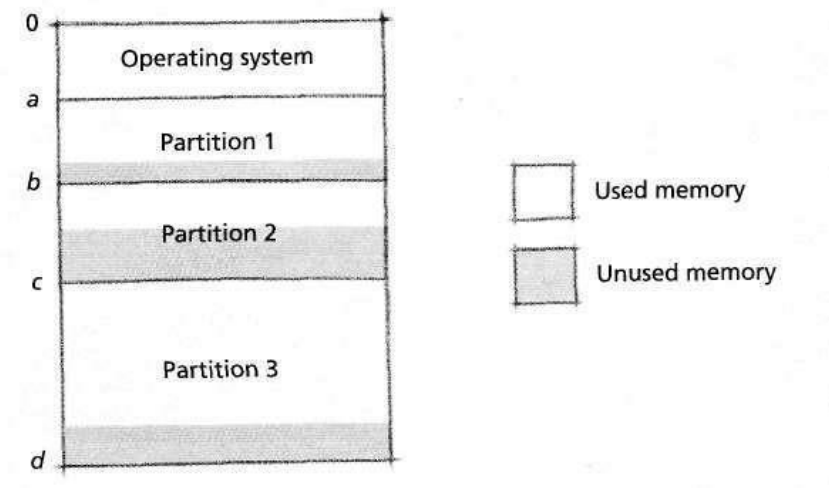
- All the jobs in the system must run in partition 3 (i.e., the programs' instructions all begin at address c).
- Because this partition currently is in use. all other jobs are forced to wait, even though the system has two other partitions in which the jobs could run (if they had been compiled for these partitions).
- This scheme eliminates some of the memory waste inherent in multipro- operating system from the user process.
- In a multiprogramming system, the system must protect the operating system from all user processes and protect each process from all the others.



- The system can delimit each partition with two boundary registers low and high, also called the **base** and **limit** registers.
- When a process issues a memory request, the system checks whether the requested address is greater than or equal to the process's low boundary register value and less than the process's high boundary register value (see the Anecdote, Compartmentalization).
- If so, the system honors the request; otherwise, the system terminates the program with an error message.

Fixed-partition multiprogramming suffers from **internal frag**mentation, which occurs when the size of a process's memory and data is smaller than that of the partition in which the process executes.

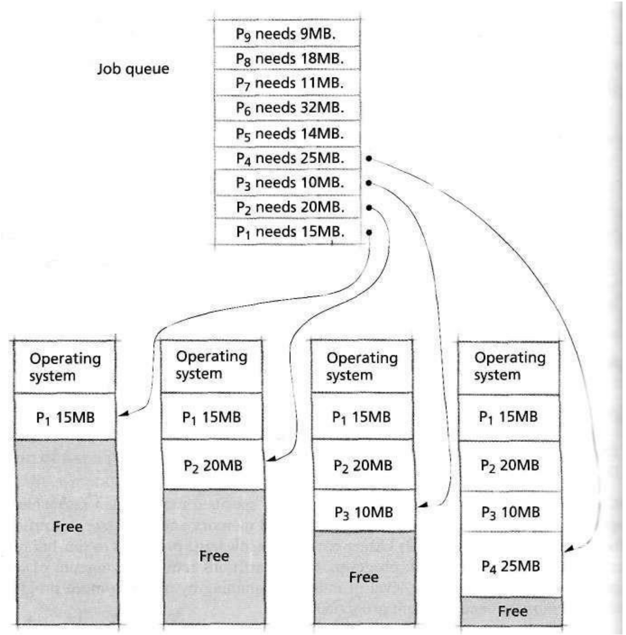
INTERNAL FRAGMENTATION



VARIABLE-PARTITION MULTIPROGRAMMING

- The operating system designers decided, would be to allow a process to occupy only as much space as needed (up to the amount of available main memory).
- This scheme is called **variable-partition multiprogramming**.

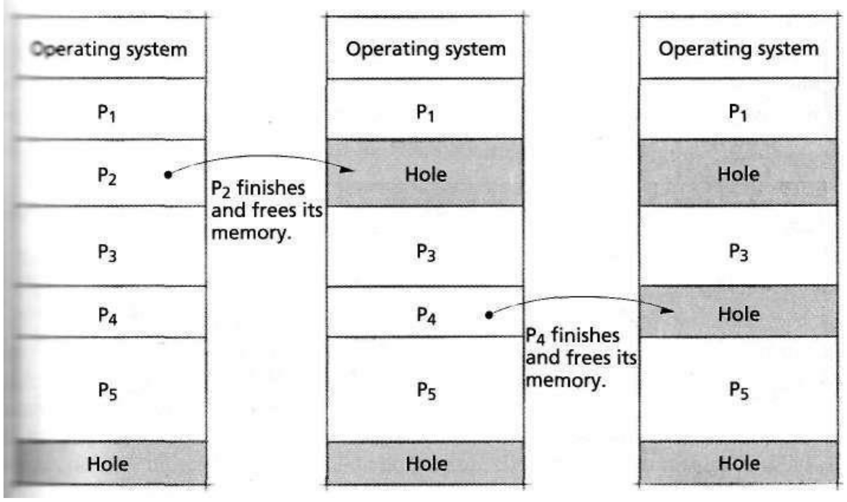
VARIABLE-PARTITION CHARACTERISTICS



- The queue at the top contains available jobs and information about their memory requirements.
- The operating system makes no assumption about the size of a job.

Holes

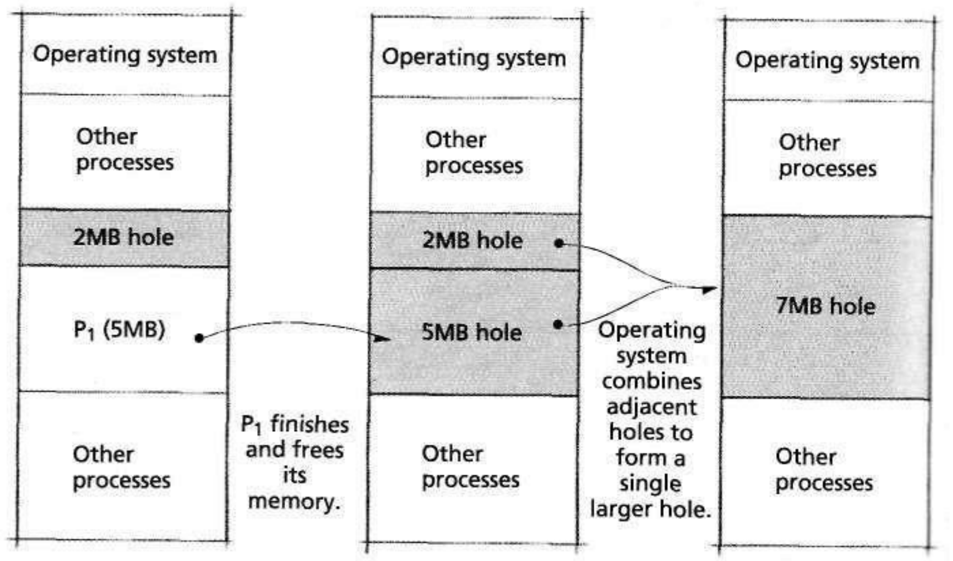
- Variable-partition multiprogramming organizations do not suffer from internal fragmentation.
- In variable partition multiprogramming, the waste does not become obvious until processes finish and leave **holes** in main memory.



- The system can continue to place new processes in these holes.
- However, as processes continue to complete, the holes get smaller, until every hole eventually becomes too small to hold a new process.
- This is called **external fragmentation.**

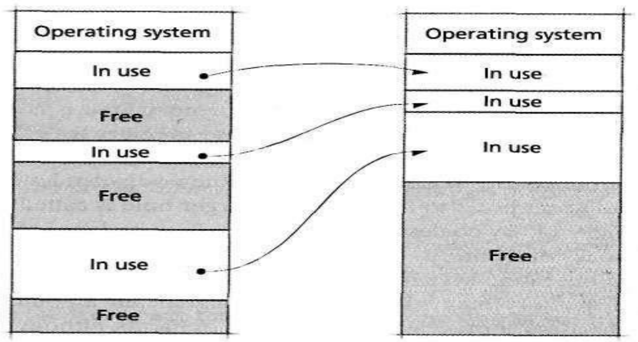
The system then records in a **free memory list** either (1) that the system now has an additional hole or (2) that an existing hole has been enlarged.

By coalescing holes, the system reclaims the largest possible contiguous blocks of memory.



- Another technique for reducing external fragmentation is called **memory compaction,** which relocates all occupied areas of memory to one end or the other of main memory.

- Now all of the available free memory is contiguous, so that an available process can run if its memory requirement is met by the single hole that results from compaction.
- Sometimes memory compaction is colorfully referred to as **burping the memory.**
- More conventionally, it is called **garbage collection.**



Operating system places all "in use" blocks together leaving free memory as a single large hole.

VIRTUAL MEMORY MANAGEMENT

INTRODUCTION

- Replacement strategy
  - – Technique a system employs to select pages for replacement when memory is full
  - – Determines where in main memory to place an incoming page or segment
- Fetch strategy
  - – Determines when pages or segments should be loaded into main memory
  - – Anticipatory fetch strategies
- Use heuristics to predict which pages a process will soon reference and load those pages or segments.

A computer can address more memory than the amount physically installed on the system. This extra memory is actually called **virtual memory** and it is a section of a hard disk that's set up to emulate the computer's RAM.

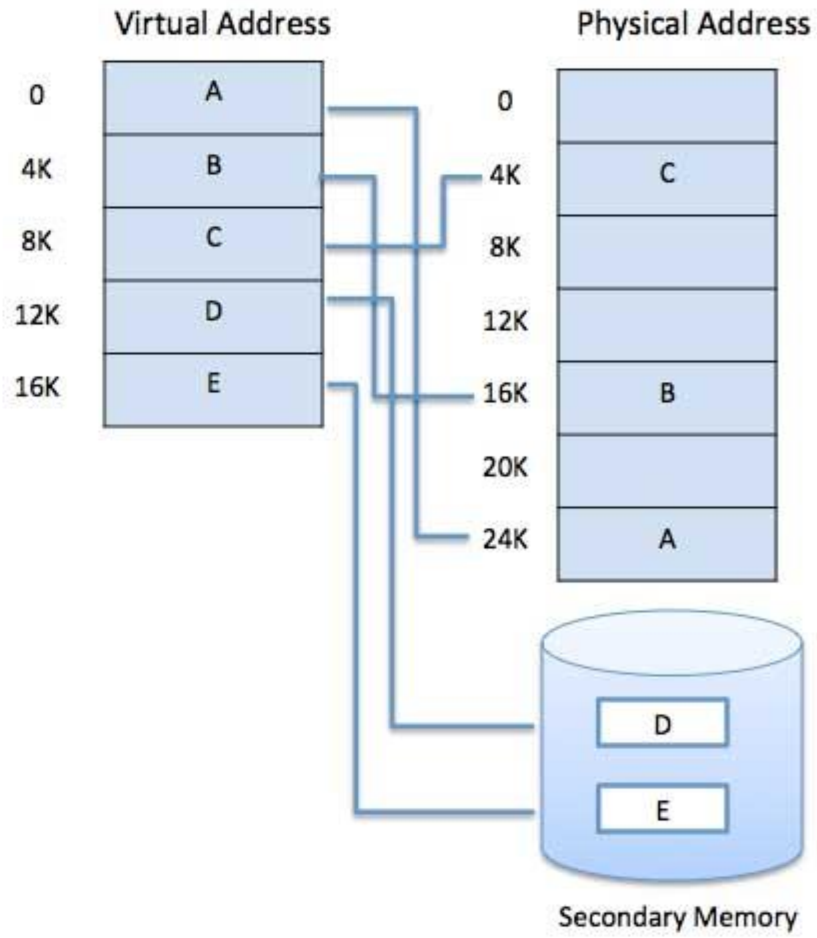
The main visible advantage of this scheme is that programs can be larger than physical memory. Virtual memory serves two purposes. First, it allows us to extend the use of physical memory by using disk. Second, it allows us to have memory protection, because each virtual address is translated to a physical address.

Following are the situations, when entire program is not required to be loaded fully in main memory.

- User written error handling routines are used only when an error occurred in the data or computation.
- Certain options and features of a program may be used rarely.

- Many tables are assigned a fixed amount of address space even though only a small amount of the table is actually used.
- The ability to execute a program that is only partially in memory would counter many benefits.
- Less number of I/O would be needed to load or swap each user program into memory.
- A program would no longer be constrained by the amount of physical memory that is available.
- Each user program could take less physical memory, more programs could be run the same time, with a corresponding increase in CPU utilization and throughput.

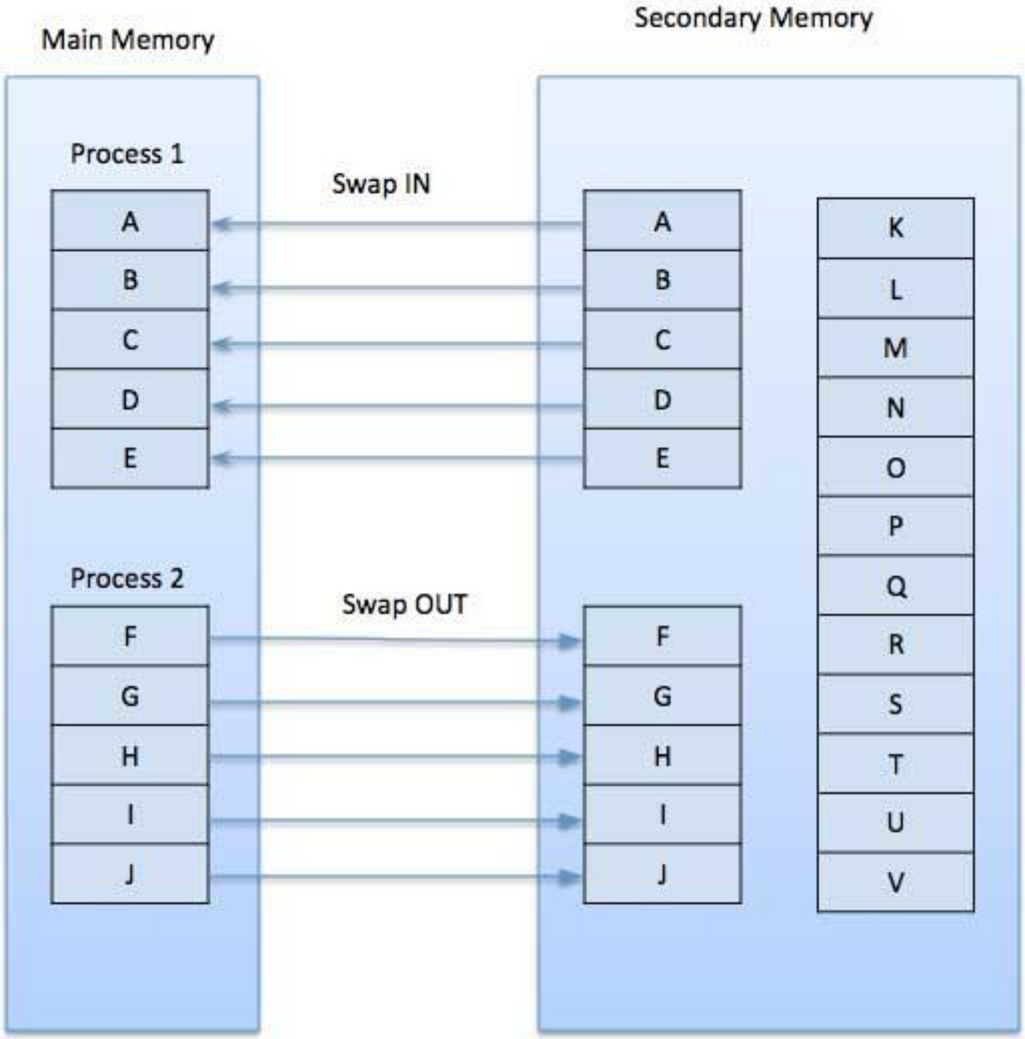
Modern microprocessors intended for general-purpose use, a memory management unit, or MMU, is built into the hardware. The MMU's job is to translate virtual addresses into physical addresses. A basic example is given below −



Virtual memory is commonly implemented by demand paging. It can also be implemented in a segmentation system. Demand segmentation can also be used to provide virtual memory.

Demand Paging

A demand paging system is quite similar to a paging system with swapping where processes reside in secondary memory and pages are loaded only on demand, not in advance. When a context switch occurs, the operating system does not copy any of the old program's pages out to the disk or any of the new program's pages into the main memory Instead, it just begins executing the new program after loading the first page and fetches that program's pages as they are referenced.



While executing a program, if the program references a page which is not available in the main memory because it was swapped out a little ago, the processor treats this invalid memory reference as a **page fault** and transfers control from the program to the operating system to demand the page back into the memory.

Advantages

Following are the advantages of Demand Paging −

- Large virtual memory.

- More efficient use of memory.
- There is no limit on degree of multiprogramming.

Disadvantages

- Number of tables and the amount of processor overhead for handling page interrupts are greater than in the case of the simple paged management techniques.

Page Replacement Algorithm

Page replacement algorithms are the techniques using which an Operating System decides which memory pages to swap out, write to disk when a page of memory needs to be allocated. Paging happens whenever a page fault occurs and a free page cannot be used for allocation purpose accounting to reason that pages are not available or the number of free pages is lower than required pages.

When the page that was selected for replacement and was paged out, is referenced again, it has to read in from disk, and this requires for I/O completion. This process determines the quality of the page replacement algorithm: the lesser the time waiting for page-ins, the better is the algorithm.

A page replacement algorithm looks at the limited information about accessing the pages provided by hardware, and tries to select which pages should be replaced to minimize the total number of page misses, while balancing it with the costs of primary storage and processor time of the algorithm itself. There are many different page replacement algorithms. We evaluate an algorithm by running it on a particular string of memory reference and computing the number of page faults,

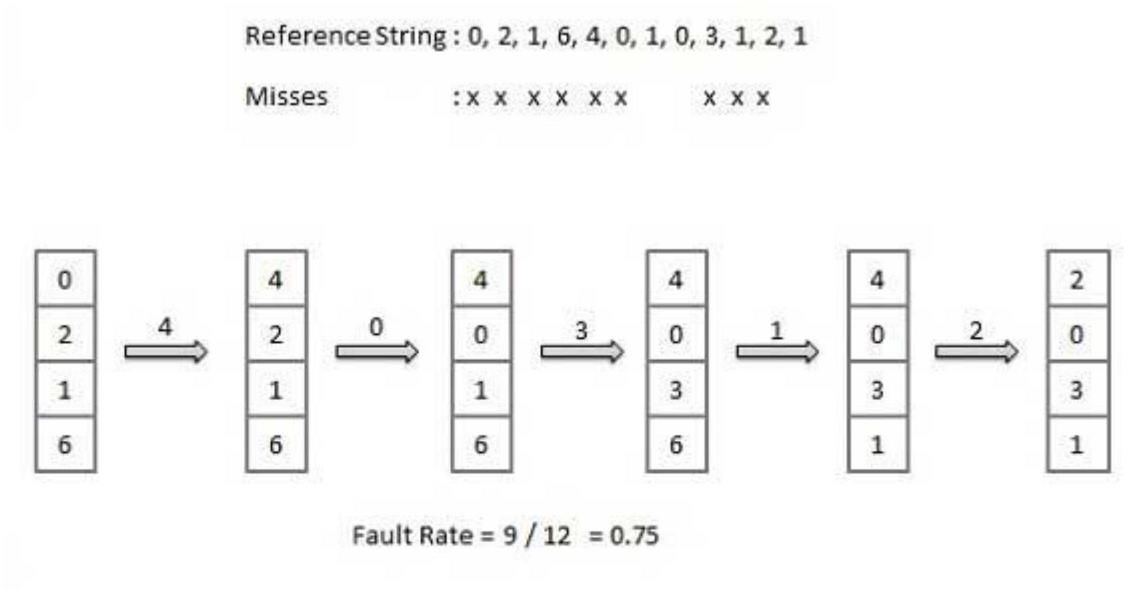
Reference String

The string of memory references is called reference string. Reference strings are generated artificially or by tracing a given system and recording the address of each memory reference. The latter choice produces a large number of data, where we note two things.

- For a given page size, we need to consider only the page number, not the entire address.
- If we have a reference to a page **p**, then any immediately following references to page **p** will never cause a page fault. Page p will be in memory after the first reference; the immediately following references will not fault.
- For example, consider the following sequence of addresses − 123,215,600,1234,76,96
- If page size is 100, then the reference string is 1,2,6,12,0,0

First In First Out (FIFO) algorithm

- Oldest page in main memory is the one which will be selected for replacement.
- Easy to implement, keep a list, replace pages from the tail and add new pages at the head.



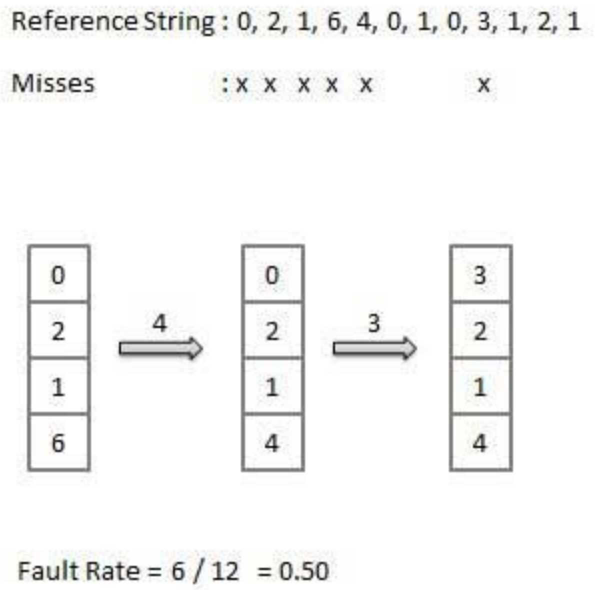
Reference String : 0, 2, 1, 6, 4, 0, 1, 0, 3, 1, 2, 1

Misses : x x x x x x x x x

Fault Rate = 9 / 12 = 0.75

Optimal Page algorithm

- An optimal page-replacement algorithm has the lowest page-fault rate of all algorithms. An optimal page-replacement algorithm exists, and has been called OPT or MIN.
- Replace the page that will not be used for the longest period of time. Use the time when a page is to be used.



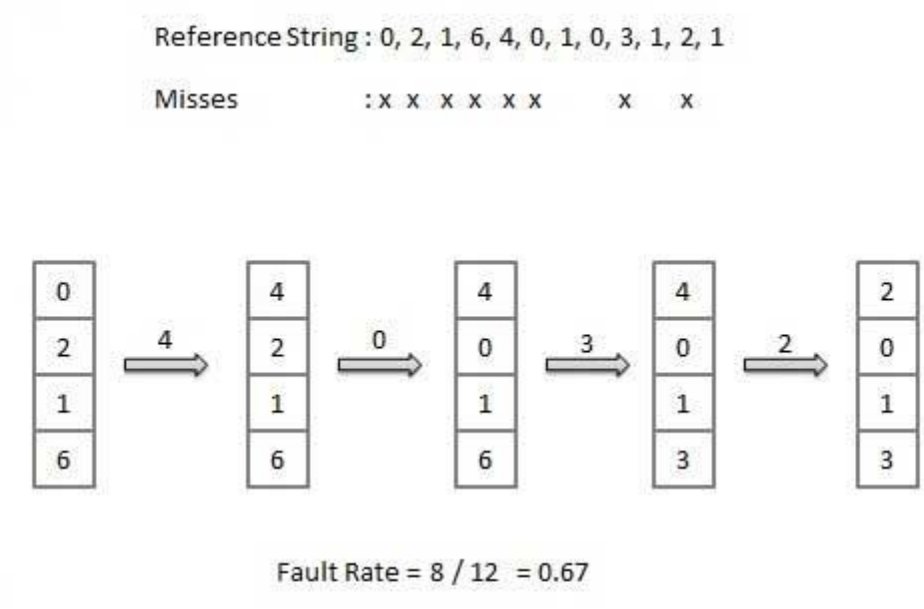
Reference String : 0, 2, 1, 6, 4, 0, 1, 0, 3, 1, 2, 1

Misses : x x x x x x

Fault Rate = 6 / 12 = 0.50

Least Recently Used (LRU) algorithm

- Page which has not been used for the longest time in main memory is the one which will be selected for replacement.
- Easy to implement, keep a list, replace pages by looking back into time.



Reference String : 0, 2, 1, 6, 4, 0, 1, 0, 3, 1, 2, 1

Misses : x x x x x x x x

Fault Rate = 8 / 12 = 0.67

Page Buffering algorithm

- To get a process start quickly, keep a pool of free frames.
- On page fault, select a page to be replaced.
- Write the new page in the frame of free pool, mark the page table and restart the process.
- Now write the dirty page out of disk and place the frame holding replaced page in free pool.

Least frequently Used(LFU) algorithm

- The page with the smallest count is the one which will be selected for replacement.
- This algorithm suffers from the situation in which a page is used heavily during the initial phase of a process, but then is never used again.

Most frequently Used(MFU) algorithm

- This algorithm is based on the argument that the page with the smallest count was probably just brought in and has yet to be used.

PAGE-REPLACEMENT STRATEGIES

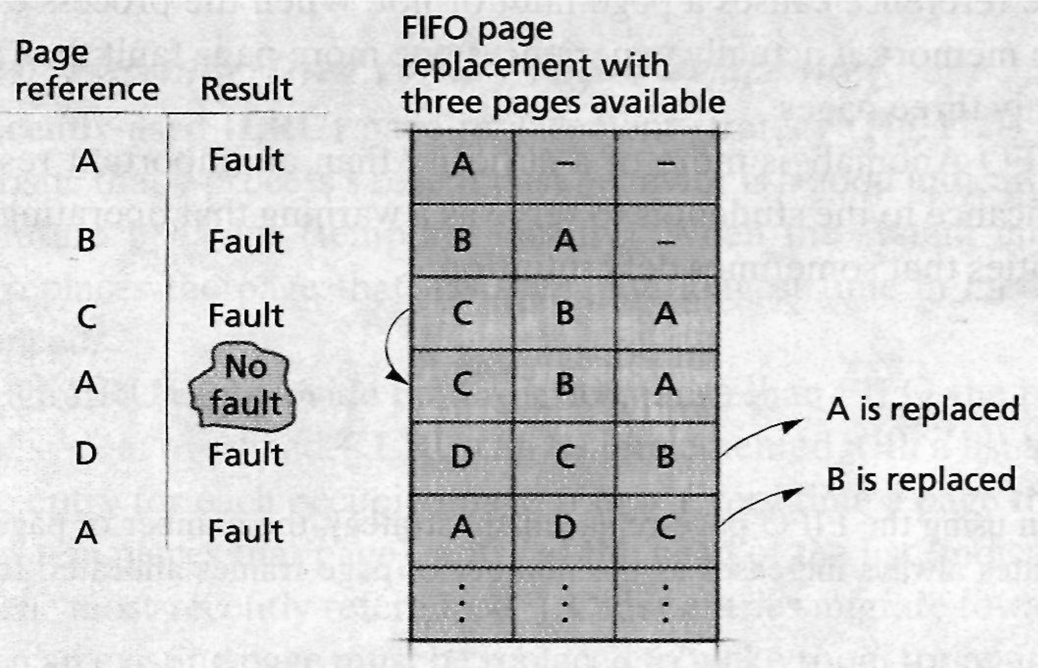
Each strategy is characterized by the heuristic it uses to select a page for replacement and the overhead it incurs. Some replacement strategies are intuitively appealing but lead to poor performance.

RANDOM PAGE REPLACEMENT

- Random page replacement
  - – Low-overhead page-replacement strategy that does not discriminate against particular processes.
  - – Each page in main memory has an equal likelihood of being selected for replacement.
  - – Could easily select as the next page to replace the page that will be referenced next.

FIRST-IN-FIRST-OUT (FIFO) PAGE REPLACEMENT

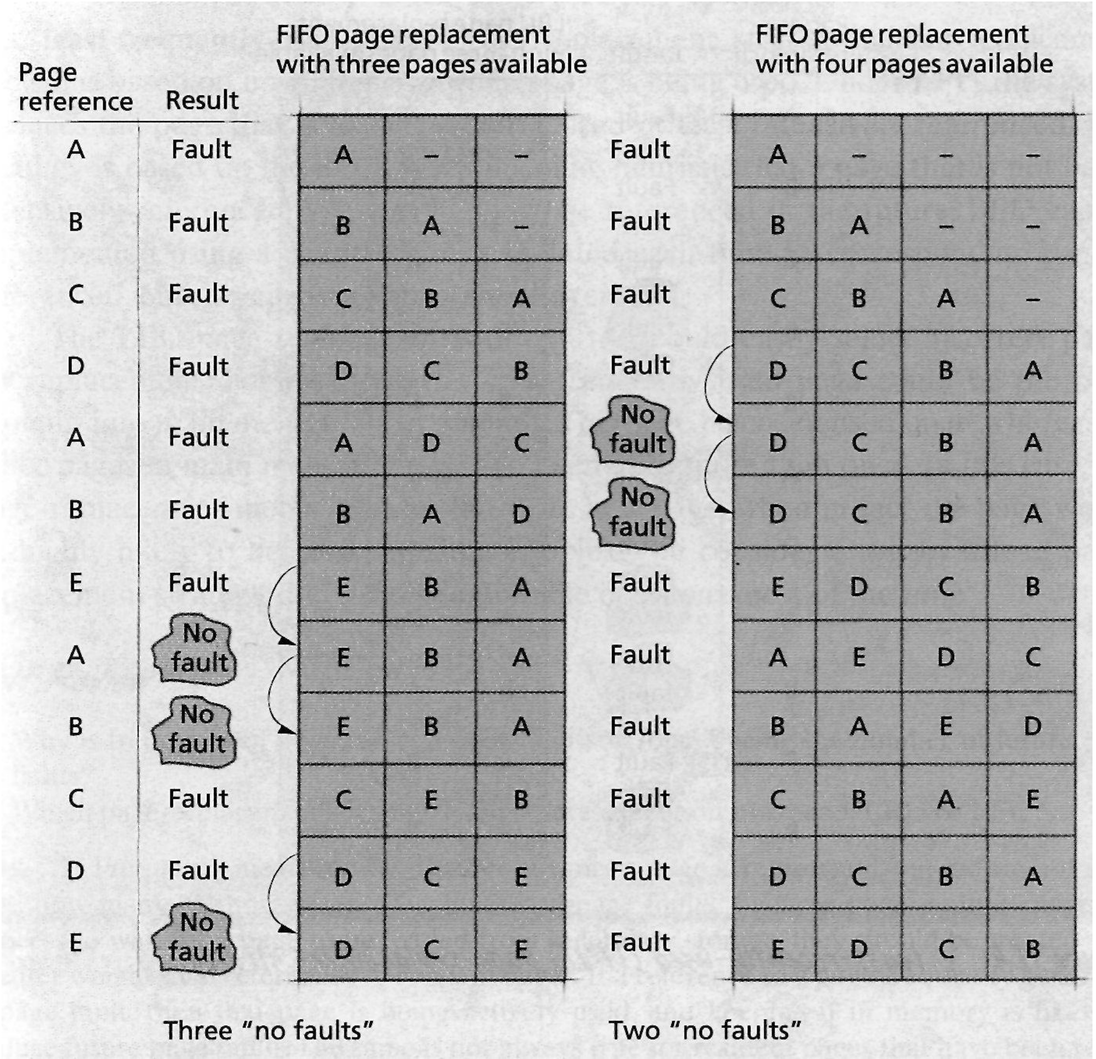
- **A** simple example of the FIFO strategy for a process which has been allocated three page frames.
- The leftmost column contains the process's page-reference pattern.
- Each row shows the state of the FIFO queue after each new page arrives; pages enter the tail of the queue on the left and exit the head on the right.
- This would be a poor choice, because the page would be recalled to main memory
- almost immediately, resulting in an increased page-fault rate. Modifications to FIFO: Second- Chance and Clock Page Replacement, FIFO forms the basis of various implemented page-replacement schemes.



Page reference	Result	FIFO page replacement with three pages available		
A	Fault	A	–	–
B	Fault	B	A	–
C	Fault	C	B	A
A	No fault	C	B	A
D	Fault	D	C	B
A	Fault	A	D	C
		⋮	⋮	⋮

FIFO ANOMALY

- The first table demonstrates how the reference pattern causes the system to load and replace pages (using FIFO) when the system allocates three page frames to the process.
- The second table shows how the system behaves in response to the same reference pattern, but when four page frames have been allocated.
- When the process executes with four pages in memory, it actually experiences one more page fault than when it executes with only three pages.



Page reference	Result	FIFO page replacement with three pages available				FIFO page replacement with four pages available			
A	Fault	A	–	–	Fault	A	–	–	–
B	Fault	B	A	–	Fault	B	A	–	–
C	Fault	C	B	A	Fault	C	B	A	–
D	Fault	D	C	B	Fault	D	C	B	A
A	Fault	A	D	C	No fault	D	C	B	A
B	Fault	B	A	D	No fault	D	C	B	A
E	Fault	E	B	A	Fault	E	D	C	B
A	No fault	E	B	A	Fault	A	E	D	C
B	No fault	E	B	A	Fault	B	A	E	D
C	Fault	C	E	B	Fault	C	B	A	E
D	Fault	D	C	E	Fault	D	C	B	A
E	No fault	D	C	E	Fault	E	D	C	B

Three "no faults"

Two "no faults"

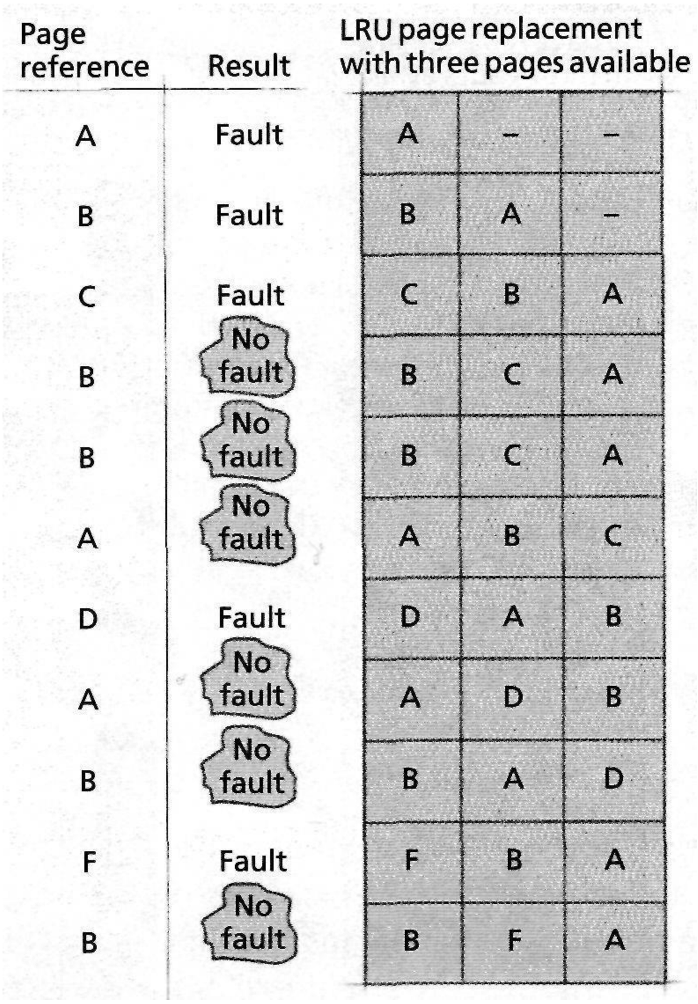
LEAST-RECENTLY USED (LRU) PAGE REPLACEMENT

– Exploits temporal locality by replacing the page that has spent the longest time in memory without being referenced

– Can provide better performance than FIFO

– Increased system overhead

– LRU can perform poorly if the least-recently used page is the next page to be referenced by a program that is iterating inside a loop that references several pages



Page reference	Result	LRU page replacement with three pages available		
A	Fault	A	–	–
B	Fault	B	A	–
C	Fault	C	B	A
B	No fault	B	C	A
B	No fault	B	C	A
A	No fault	A	B	C
D	Fault	D	A	B
A	No fault	A	D	B
B	No fault	B	A	D
F	Fault	F	B	A
B	No fault	B	F	A

LEAST-FREQUENTLY-USED (LFU) PAGE REPLACEMENT

– Replaces page that is least intensively referenced

– Based on the heuristic that a page not referenced often is not likely to be referenced in the future

– Could easily select wrong page for replacement

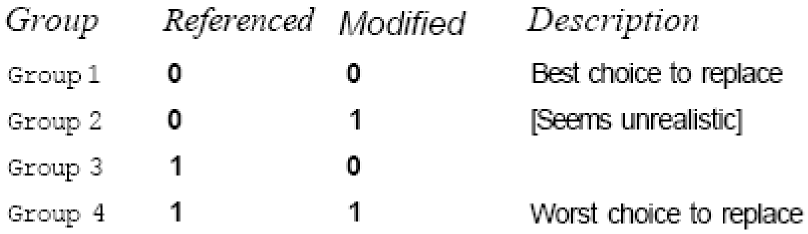
– A page that was referenced heavily in the past may never be referenced again, but will stay in memory while newer, active pages are replaced.

NOT-USED-RECENTLY (NUR) PAGE REPLACEMENT

The NUR strategy is implemented using the following two hardware bits per page table entry:

- **referenced bit**—set to 0 if the page has not been referenced and set to one if the page has been referenced.
- **modified bit**—set to 0 if the page has not been modified and set to 1 if the page has been modified.

- The referenced bit is sometimes called the **accessed bit.**
- The pages in the lowest-numbered groups should be replaced first, and those in the highest-numbered groups last. Pages within a group are selected randomly for replacement.
- Note that Group 2 seems to describe an unrealistic situation—namely, pages that have been modified but not referenced.
- This occurs because of the periodic resetting of the referenced bits.



Group	*Referenced*	*Modified*	*Description*
Group 1	0	0	Best choice to replace
Group 2	0	1	[Seems unrealistic]
Group 3	1	0	
Group 4	1	1	Worst choice to replace

MODIFICATION TO FIFO: SECOND-CHANCE AND CLOCK PAGE REPLACEMENT

- **Second chance page replacement**
  - Examines referenced bit of the oldest page
    - If it's off
  - The strategy selects that page for replacement
    - If it's on
  - The strategy turns off the bit and moves the page to tail of FIFO queue

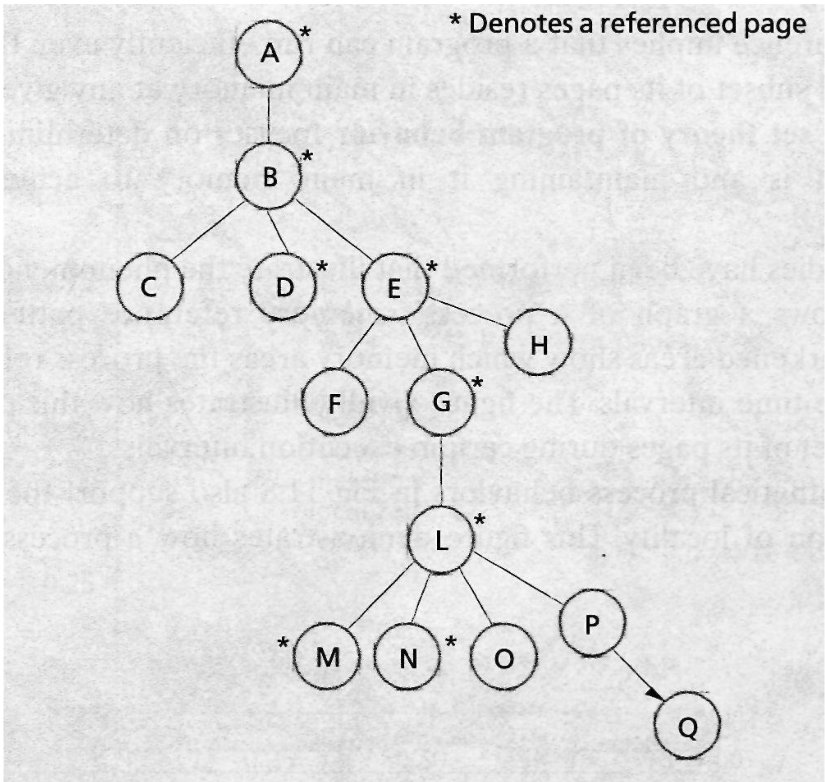
– Ensures that active pages are the least likely to be replaced

- **Clock page replacement**

  – Similar to second chance, but arranges the pages in circular list instead of linear list.

FAR PAGE REPLACEMENT

- Each vertex in the access graph represents one of the process's pages.
- An edge from vertex v to vertex *w* means that the process can reference page *w* after it has referenced page v.
- For example, if an instruction on page *v* references data on page *w*, there will be a directed edge from vertex *v* to vertex *w*.
- Similarly, if a function call to page *x* returns to page y, there will be an edge from vertex *x* to vertex *y*.
- The access graph indicates that, after the process references page B, it will next reference either page A, C, D or E, but it will not reference page G before it has referenced
- page E.



Q is chosen for replacement

- The field of graph theory provides algorithms for building and searching the
- kinds of graphs in the far strategy.
- However, largely due t its complexity and execution-time overhead, far has not been implemented in real systems.

PAGE-FAULT-FREQUENCY (PFF) PAGE REPLACEMENT

- The **page-fault-frequency (PFF)** algorithm adjusts a process's **resident page set**
- based on the frequency at which the process is faulting.
- Alternatively, PFF may adjust a process's resident page set based on the time between page faults, called the process's **interfault time.**
- PFF has a lower overhead than working set page replacement because it adjusts the resident page set only after each page fault; a working set mechanism must operate after each memory reference.
- A benefit of PFF is that it adjusts a process's resident page set dynamically, in
- response to the process's changing behavior.
- If a process is switching to a larger working set, then it will fault frequently, and PFF will allocate more page frames.

PAGE RELEASE

• Inactive pages can remain in main memory for a long time until the management strategy detects that the process no longer needs them

– One way to solve the problem

• Process issues a voluntary page release to free a page frame that it knows it no longer needs

• Eliminate the delay period caused by letting process gradually pass the page from its working set

• The real hope is in compiler and operating system support.

PAGE SIZE

• Some systems improve performance and memory utilization by providing multiple page sizes

– Small page sizes

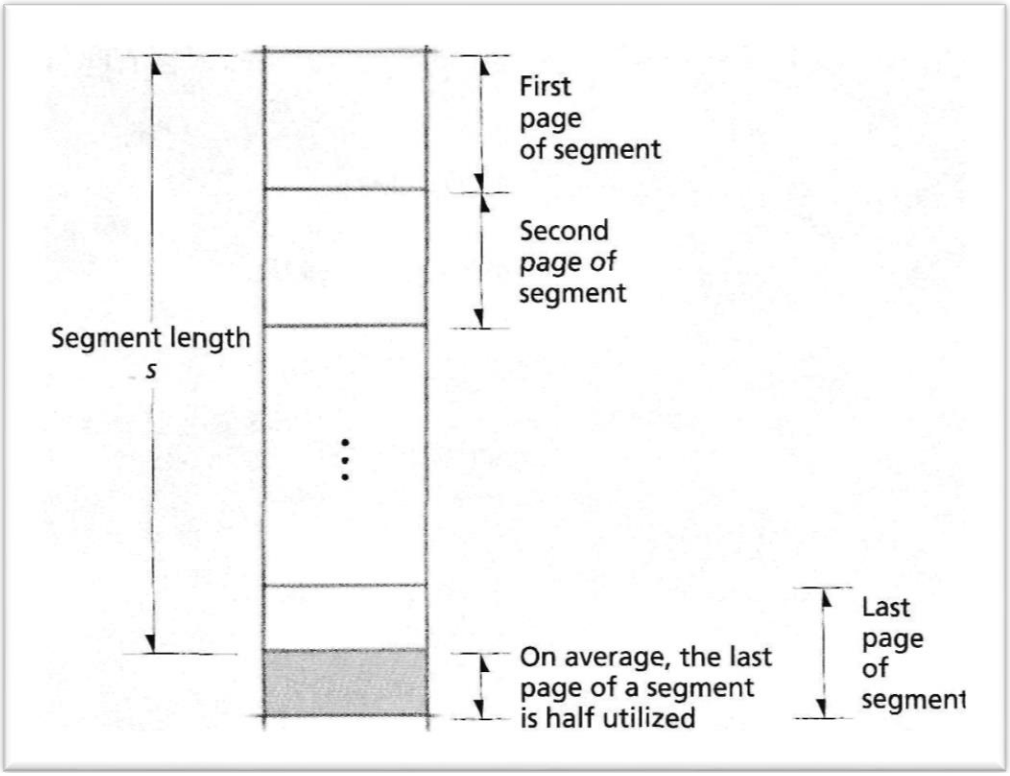
- Reduce internal fragmentation
- Can reduce the amount of memory required to contain a process's working set
- More memory available to other processes

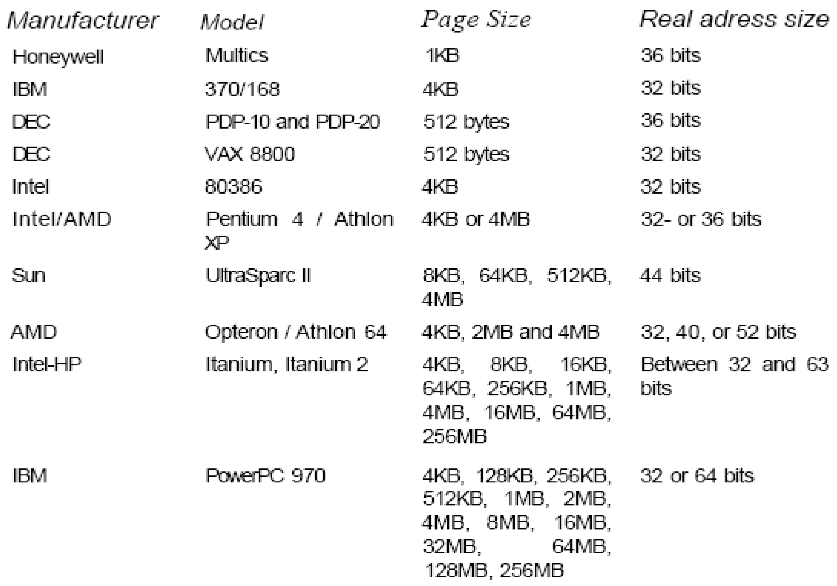
– Large page size

- Reduce wasted memory from table fragmentation
- Enable each TLB entry to map larger region of memory, improving performance
- Reduce number of I/O operations the system performs to load a process's working set into memory

– Multiple page size

- Possibility of external fragmentation



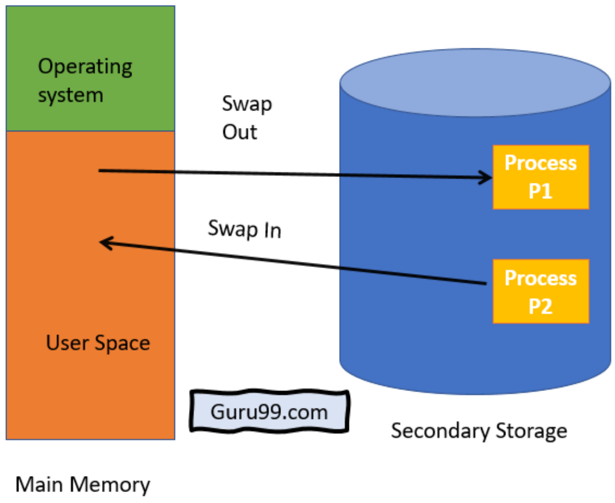


Manufacturer	*Model*	*Page Size*	*Real adress size*
Honeywell	Multics	1KB	36 bits
IBM	370/168	4KB	32 bits
DEC	PDP-10 and PDP-20	512 bytes	36 bits
DEC	VAX 8800	512 bytes	32 bits
Intel	80386	4KB	32 bits
Intel/AMD	Pentium 4 / Athlon XP	4KB or 4MB	32- or 36 bits
Sun	UltraSparc II	8KB, 64KB, 512KB, 4MB	44 bits
AMD	Opteron / Athlon 64	4KB, 2MB and 4MB	32, 40, or 52 bits
Intel-HP	Itanium, Itanium 2	4KB, 8KB, 16KB, 64KB, 256KB, 1MB, 4MB, 16MB, 64MB, 256MB	Between 32 and 63 bits
IBM	PowerPC 970	4KB, 128KB, 256KB, 512KB, 1MB, 2MB, 4MB, 8MB, 16MB, 32MB, 64MB, 128MB, 256MB	32 or 64 bits

What is Swapping?

Swapping is a method in which the process should be swapped temporarily from the main memory to the backing store. It will be later brought back into the memory for continue execution.

Backing store is a hard disk or some other secondary storage device that should be big enough inorder to accommodate copies of all memory images for all users. It is also capable of offering direct access to these memory images.



Benefits of Swapping

Here, are major benefits/pros of swapping:

- It offers a higher degree of multiprogramming.
- Allows dynamic relocation. For example, if address binding at execution time is being used, then processes can be swap in different locations. Else in case of compile and load time bindings, processes should be moved to the same location.
- It helps to get better utilization of memory.
- Minimum wastage of CPU time on completion so it can easily be applied to a priority-based scheduling method to improve its performance.

What is Memory allocation?

Memory allocation is a process by which computer programs are assigned memory or space.

Here, main memory is divided into two types of partitions

1. **Low Memory** - Operating system resides in this type of memory.
2. **High Memory-** User processes are held in high memory.

Partition Allocation

Memory is divided into different blocks or partitions. Each process is allocated according to the requirement. Partition allocation is an ideal method to avoid internal fragmentation.

Below are the various partition allocation schemes :

- **First Fit**: In this type fit, the partition is allocated, which is the first sufficient block from the beginning of the main memory.
- **Best Fit:** It allocates the process to the partition that is the first smallest partition among the free partitions.
- **Worst Fit:** It allocates the process to the partition, which is the largest sufficient freely available partition in the main memory.
- **Next Fit:** It is mostly similar to the first Fit, but this Fit, searches for the first sufficient partition from the last allocation point.

What is Paging?

Paging is a storage mechanism that allows OS to retrieve processes from the secondary storage into the main memory in the form of pages. In the Paging method, the main memory is divided into small fixed-size blocks of physical memory, which is called frames. The size of a frame should be kept the same as that of a page to have maximum utilization of the main memory and to avoid external fragmentation. Paging is used for faster access to data, and it is a logical concept.

What is Fragmentation?

Processes are stored and removed from memory, which creates free memory space, which are too small to use by other processes.

After sometimes, that processes not able to allocate to memory blocks because its small size and memory blocks always remain unused is called fragmentation. This type of problem happens during a dynamic memory allocation system when free blocks are quite small, so it is not able to fulfill any request.

Two types of Fragmentation methods are:

1. External fragmentation
2. Internal fragmentation

- External fragmentation can be reduced by rearranging memory contents to place all free memory together in a single block.
- The internal fragmentation can be reduced by assigning the smallest partition, which is still good enough to carry the entire process.

What is Segmentation?

Segmentation method works almost similarly to paging. The only difference between the two is that segments are of variable-length, whereas, in the paging method, pages are always of fixed size.

A program segment includes the program's main function, data structures, utility functions, etc. The OS maintains a segment map table for all the processes. It also includes a list of free memory blocks along with its size, segment numbers, and its memory locations in the main memory or virtual memory.

What is Dynamic Loading?

Dynamic loading is a routine of a program which is not loaded until the program calls it. All routines should be contained on disk in a relocatable load format. The main program will be loaded into memory and will be executed. Dynamic loading also provides better memory space utilization.

What is Dynamic Linking?

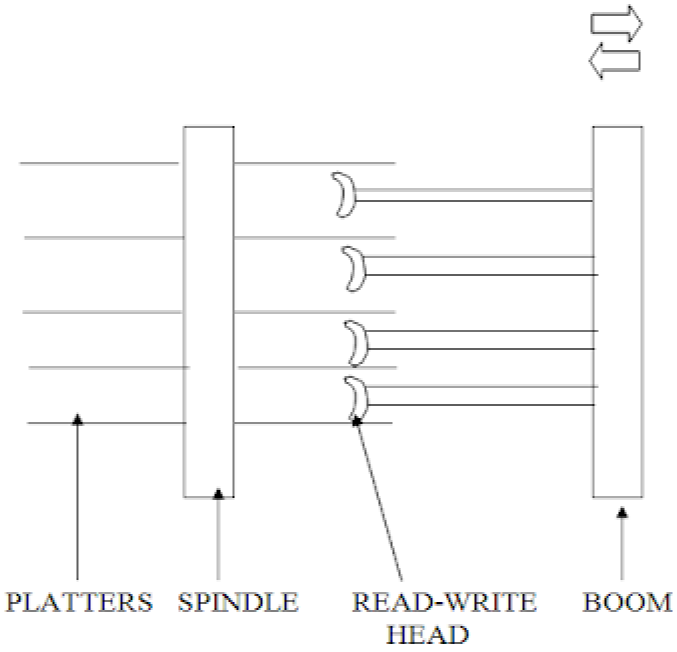
Linking is a method that helps OS to collect and merge various modules of code and data into a single executable file. The file can be loaded into memory and executed. OS can link system-level libraries into a program that combines the libraries at load time. In Dynamic linking method, libraries are linked at execution time, so program code size can remain small.

UNIT V

DISK PERFORMANCE OPTIMIZATION

INTRODUCTION

In multi programmed computing systems, inefficiency is often caused by improper use of rotational storage devices such as disks and drums.



This is a schematic representation of the side view of a moving-head disk. Data is recorded on a series of magnetic disk or platters. These disks are connected by a common spindle that spins at very high speed. The data is accessed (ie., either read or written) by a series of read-write heads, one head per disk surface. A read-write head can access only data immediately adjacent to it.

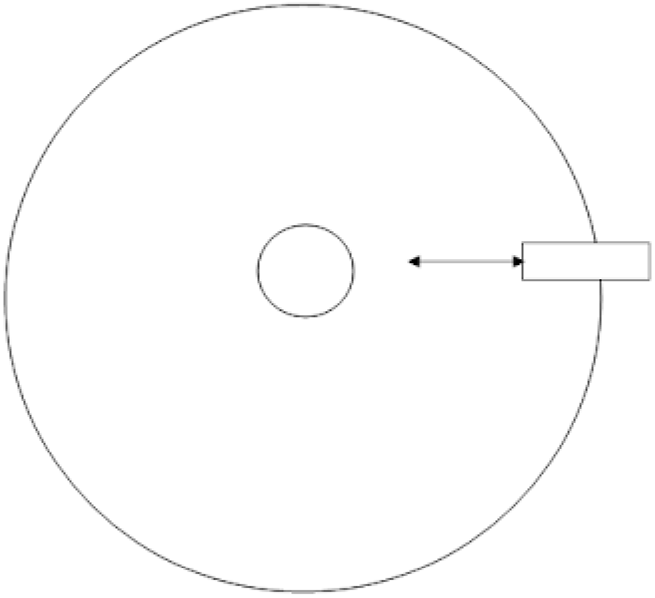
Therefore, before data can be accessed, the portion of the disk surface from which the data is to be read (or the portion on which the data is to be written) must rotate until it is immediately below (or above) the read-write head. The time it takes for data to rotate from its current position to a position adjacent to the read-write head is called latency time.

Each of the several read-write heads, while fixed in position, sketches out in circular track of data on a disk surface. All read-write heads are attached to a single boom or moving arm

assembly. The boom may move in or out. When the boom moves the read-write heads to a new position, a different set of tracks becomes accessible. For a particular position of the boom, the set of tracks sketched out by all the read-write heads forms a vertical cylinder. The process of moving the boom **to a new cylinder is called a seek operation.**

Thus, in order to access a particular record of data on a moving-head disk, several operations are usually necessary. First, the boom must be moved to the appropriate cylinder. Then the portion of the disk on which the data record is stored must rotate until it is immediately under(or over) the read-write head (ie., latency time).

COMPONENTS OF A DISK ACCESS



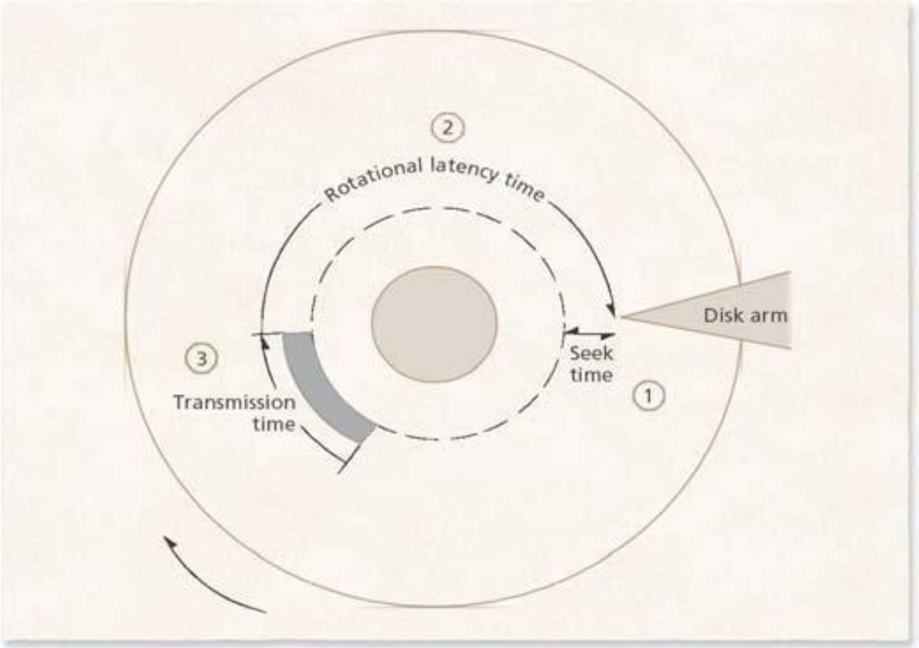
Then the record, which is of arbitrary size must be made to spin by the read-write head. This is called transmission time. This is tediously slow compared with the high processing speeds of the central computer system.

WHY DISK SCHEDULING IS NECESSARY

- First-come-first-served (FCFS) scheduling has major drawbacks
  - Seeking to randomly distributed locations results in long waiting times
  - Under heavy loads, system can become overwhelmed
- Requests must be serviced in logical order to minimize delays
  - Service requests with least mechanical motion
- The first disk scheduling algorithms concentrated on minimizing seek times, the

component of disk access that had the highest latency

- Modern systems perform rotational optimization as well



DISK SCHEDULING STRATEGIES

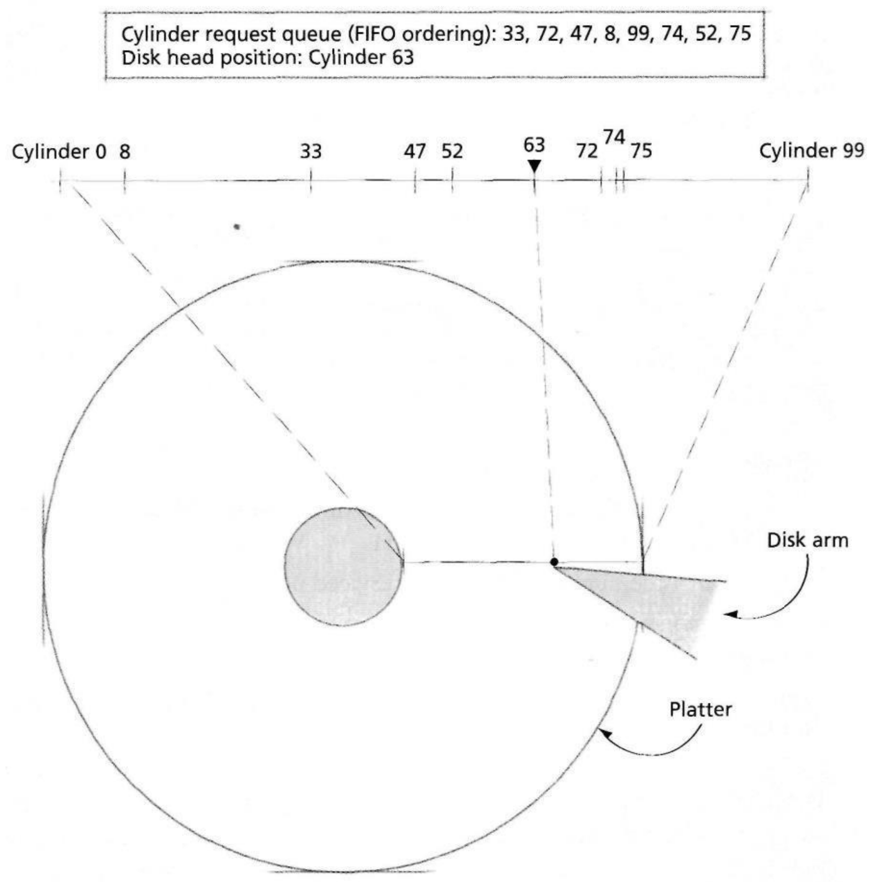
- A system's disk scheduling strategy depends on the system objectives, but most strategies are evaluated by the following criteria:

- *Throughput*—the number of requests serviced per unit time
- ***Mean response time*—**the average time spent waiting for a request to be serviced
- *Variance of response times—&* measure of the predictability of response times.

- Each disk request should be serviced within an acceptable time period.

- To demonstrate the result of each policy on an arbitrary series of requests.

- The arbitrary series of requests is intended to demonstrate how each policy orders disk requests; it does not necessarily indicate the relative performance of each policy in a real system.


Cylinder request queue (FIFO ordering): 33, 72, 47, 8, 99, 74, 52, 75
Disk head position: Cylinder 63
Cylinder 0
8
33
47
52
63
72
74
75
Cylinder 99
Disk arm
Platter

In the examples that follow, we assume that the disk contains 100 cylinders, numbered 0-99, and that the read/write head is initially located at cylinder 63, unless stated otherwise.

Disk Scheduling Algorithms

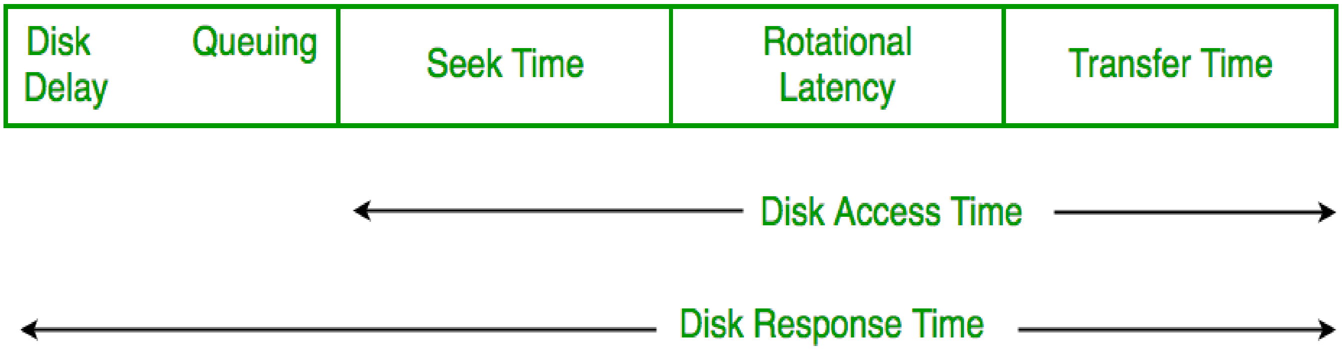
Disk scheduling is done by operating systems to schedule I/O requests arriving for the disk. Disk scheduling is also known as I/O scheduling.
Disk scheduling is important because:

- Multiple I/O requests may arrive by different processes and only one I/O request can be served at a time by the disk controller. Thus other I/O requests need to wait in the waiting queue and need to be scheduled.
- Two or more request may be far from each other so can result in greater disk arm movement.
- Hard drives are one of the slowest parts of the computer system and thus need to be accessed in an efficient manner.

There are many Disk Scheduling Algorithms but before discussing them let’s have a quick look at some of the important terms:

- **Seek Time:**Seek time is the time taken to locate the disk arm to a specified track where the data is to be read or write. So the disk scheduling algorithm that gives minimum average seek time is better.
- **Rotational Latency:** Rotational Latency is the time taken by the desired sector of disk to rotate into a position so that it can access the read/write heads. So the disk scheduling algorithm that gives minimum rotational latency is better.
- **Transfer Time:** Transfer time is the time to transfer the data. It depends on the rotating speed of the disk and number of bytes to be transferred.
- **Disk Access Time:** Disk Access Time is:

Disk Access Time = Seek Time +
Rotational Latency +
Transfer Time



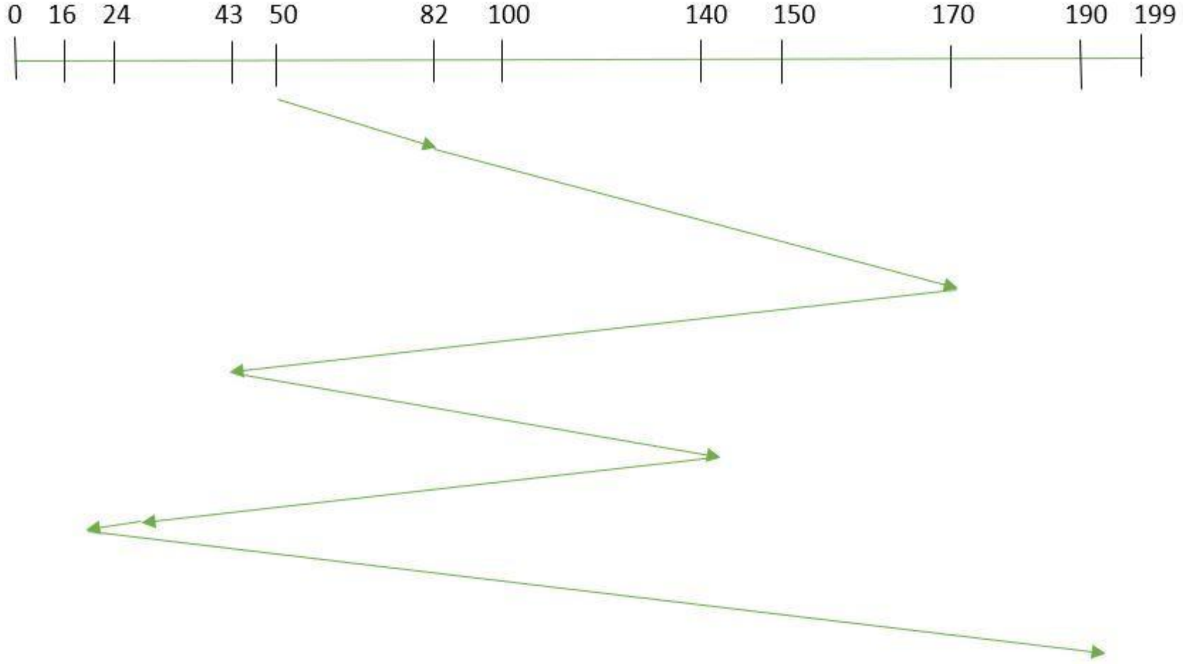
- **Disk Response Time:** Response Time is the average of time spent by a request waiting to perform its I/O operation. *Average Response time* is the response time of the all requests. *Variance Response Time* is measure of how individual request are serviced with respect to average response time. So the disk scheduling algorithm that gives minimum variance response time is better.

Disk Scheduling Algorithms

1. **FCFS:** FCFS is the simplest of all the Disk Scheduling Algorithms. In FCFS, the requests are addressed in the order they arrive in the disk queue.Let us understand this with the help of an example.

   Example:

   Suppose the order of request is- (82,170,43,140,24,16,190)
   And current position of Read/Write head is : 50



So, total seek time:
=(82-50)+(170-82)+(170-43)+(140-43)+(140-24)+(24-16)+(190-16)
=642

Advantages:

- Every request gets a fair chance
- No indefinite postponement

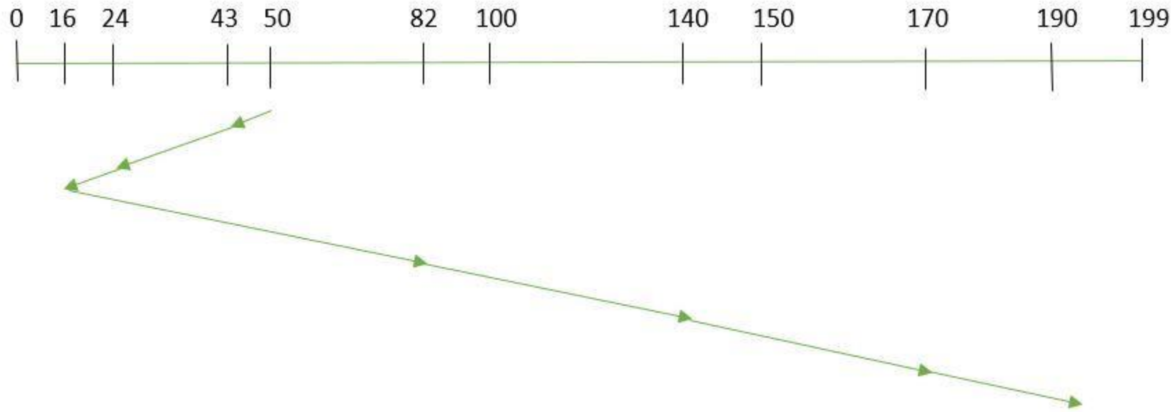
Disadvantages:

- Does not try to optimize seek time
- May not provide the best possible service

SSTF: In SSTF (Shortest Seek Time First), requests having shortest seek time are executed first. So, the seek time of every request is calculated in advance in the queue and then they are scheduled according to their calculated seek time. As a result, the request near the disk arm will get executed first. SSTF is certainly an improvement over FCFS as it decreases the average response time and increases the throughput of system.Let us understand this with the help of an example.

Example:

Suppose the order of request is- (82,170,43,140,24,16,190)
And current position of Read/Write head is : 50



So, total seek time:

=(50-43)+(43-24)+(24-16)+(82-16)+(140-82)+(170-40)+(190-170)
=208

Advantages:

- Average Response Time decreases
- Throughput increases

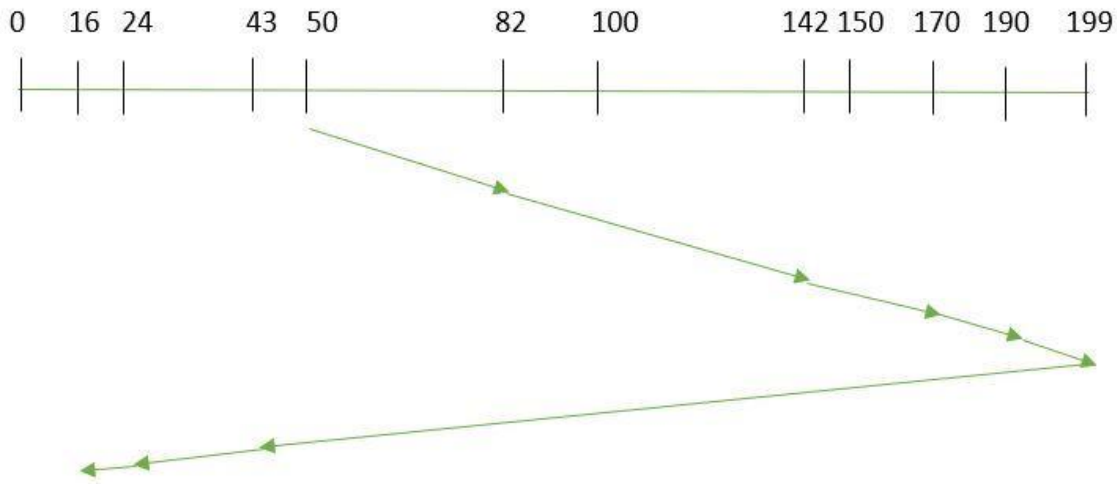
Disadvantages:

- Overhead to calculate seek time in advance
- Can cause Starvation for a request if it has higher seek time as compared to incoming requests
- High variance of response time as SSTF favours only some requests

SCAN: In SCAN algorithm the disk arm moves into a particular direction and services the requests coming in its path and after reaching the end of disk, it reverses its direction and again services the request arriving in its path. So, this algorithm works as an elevator and hence also known as **elevator algorithm.** As a result, the requests at the midrange are serviced more and those arriving behind the disk arm will have to wait.

Example:

Suppose the requests to be addressed are-82,170,43,140,24,16,190. And the Read/Write arm is at 50, and it is also given that the disk arm should move **"towards the larger value".**



Therefore, the seek time is calculated as:

=(199-50)+(199-16)
=332

Advantages:

- High throughput
- Low variance of response time
- Average response time

Disadvantages:

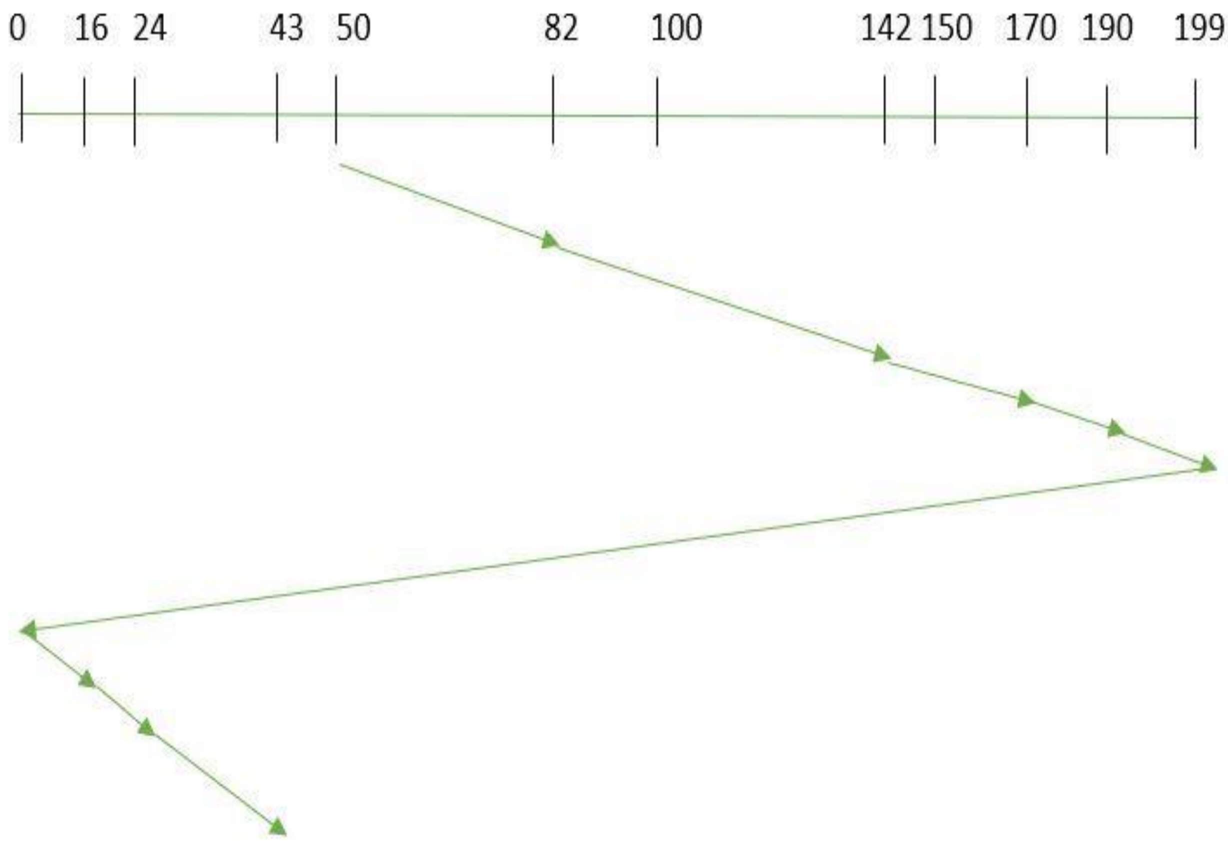
- Long waiting time for requests for locations just visited by disk arm

CSCAN: In SCAN algorithm, the disk arm again scans the path that has been scanned, after reversing its direction. So, it may be possible that too many requests are waiting at the other end or there may be zero or few requests pending at the scanned area.

These situations are avoided in *CSCAN* algorithm in which the disk arm instead of reversing its direction goes to the other end of the disk and starts servicing the requests from there. So, the disk arm moves in a circular fashion and this algorithm is also similar to SCAN algorithm and hence it is known as C-SCAN (Circular SCAN).

Example:

Suppose the requests to be addressed are-82,170,43,140,24,16,190. And the Read/Write arm is at 50, and it is also given that the disk arm should move **“towards the larger value”.**



Seek time is calculated as:

=(199-50)+(199-0)+(43-0)
=391

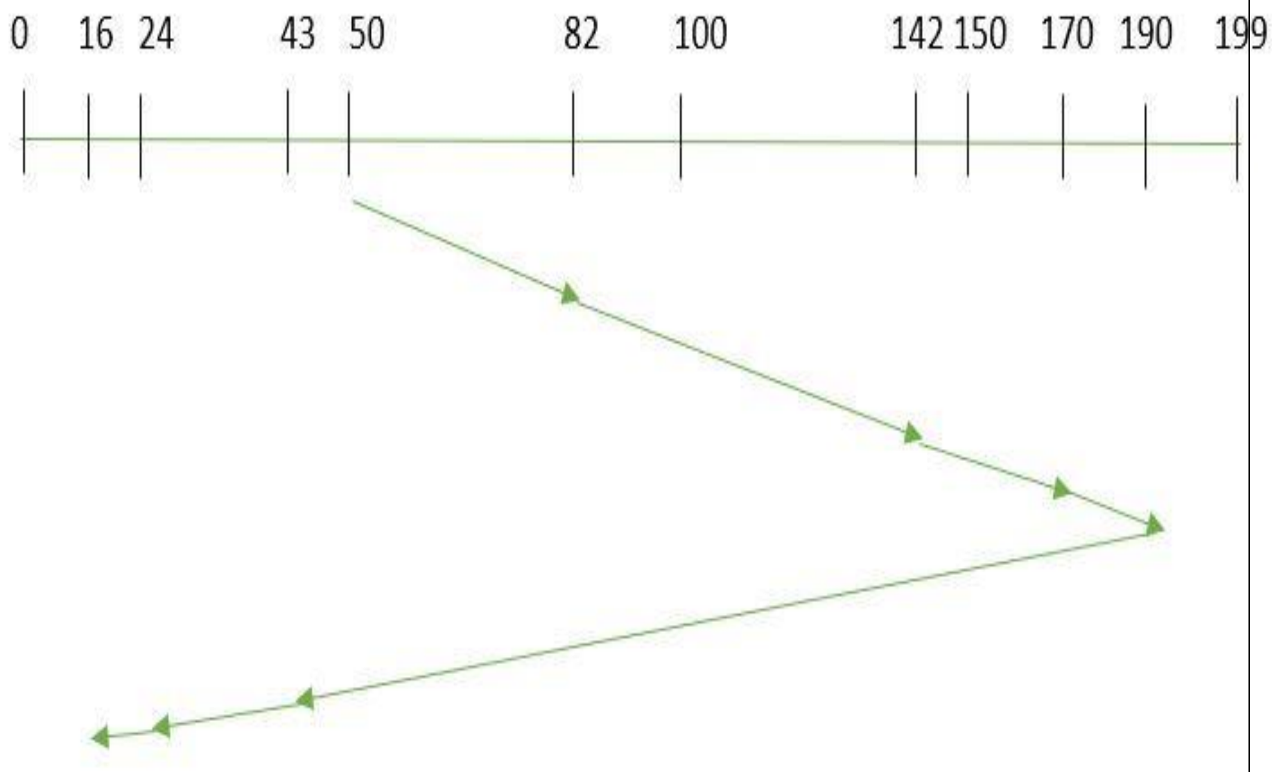
Advantages:

- Provides more uniform wait time compared to SCAN

LOOK: It is similar to the SCAN disk scheduling algorithm except for the difference that the disk arm in spite of going to the end of the disk goes only to the last request to be serviced in front of the head and then reverses its direction from there only. Thus it prevents the extra delay which occurred due to unnecessary traversal to the end of the disk.

Example:

Suppose the requests to be addressed are-82,170,43,140,24,16,190. And the Read/Write arm is at 50, and it is also given that the disk arm should move **"towards the larger value".**



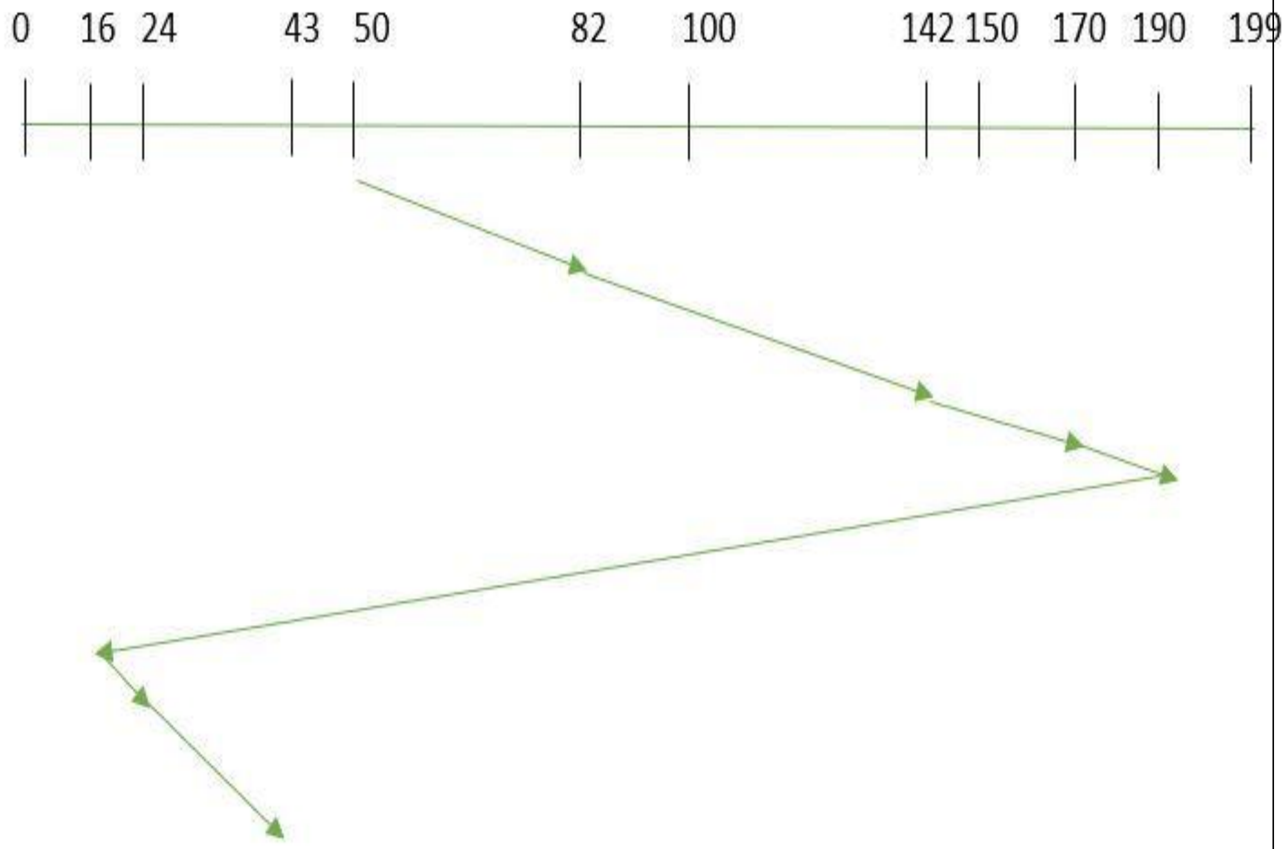
So, the seek time is calculated as:

=(190-50)+(190-16)
=314

CLOOK: As LOOK is similar to SCAN algorithm, in similar way, CLOOK is similar to CSCAN disk scheduling algorithm. In CLOOK, the disk arm in spite of going to the end goes only to the last request to be serviced in front of the head and then from there goes to the other end's last request. Thus, it also prevents the extra delay which occurred due to unnecessary traversal to the end of the disk.

Example:

Suppose the requests to be addressed are-82,170,43,140,24,16,190. And the Read/Write arm is at 50, and it is also given that the disk arm should move **"towards the larger value"**



So, the seek time is calculated as:

=(190-50)+(190-16)+(43-16)
=341

RSS– It stands for random scheduling and just like its name it is nature. It is used in situations where scheduling involves random attributes such as random processing time, random due dates, random weights, and stochastic machine breakdowns this algorithm sits perfect. Which is why it is usually used for and analysis and simulation.

LIFO– In LIFO (Last In, First Out) algorithm, newest jobs are serviced before the existing ones i.e. in order of requests that get serviced the job that is newest or last entered is serviced first and then the rest in the same order.

Advantages

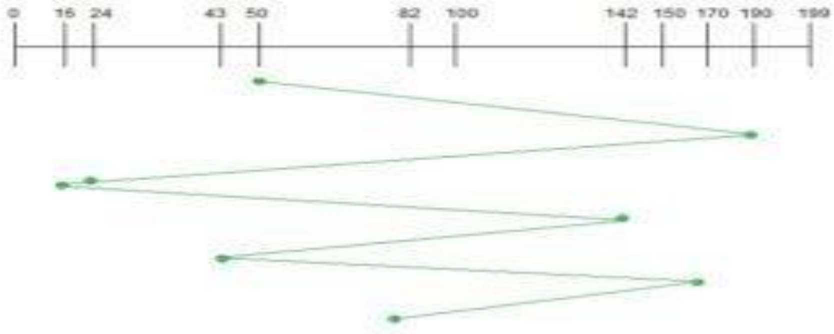
- Maximizes locality and resource utilization

Disadvantages

- Can seem a little unfair to other requests and if new requests keep coming in, it cause starvation to the old and existing ones.

Example

Suppose the order of request is- (82,170,43,142,24,16,190)
And current position of Read/Write head is : 50



N-STEP SCAN – It is also known as N-STEP LOOK algorithm. In this a buffer is created for N requests. All requests belonging to a buffer will be serviced in one go. Also once the buffer is full no new requests are kept in this buffer and are sent to another one. Now, when these N requests are serviced, the time comes for another top N requests and this way all get requests get a guaranteed service

Advantages

- It eliminates starvation of requests completely

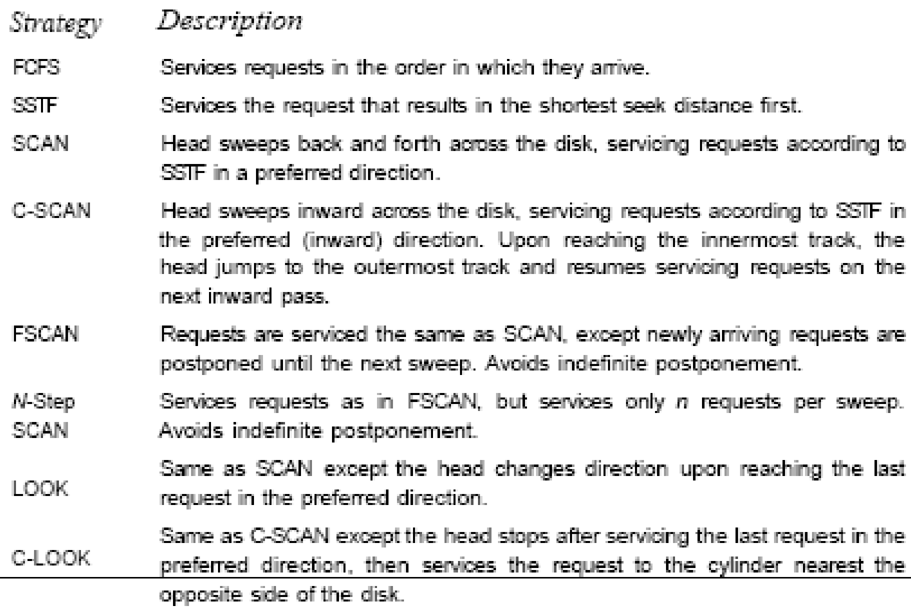
FSCAN– This algorithm uses two sub-queues. During the scan all requests in the first queue are serviced and the new incoming requests are added to the second queue. All new requests are kept on halt until the existing requests in the first queue are serviced.

Advantages

- FSCAN along with N-Step-SCAN prevents "arm stickiness" (phenomena in I/O scheduling where the scheduling algorithm continues to service requests at or near the current sector and thus prevents any seeking)

.

SEEK OPTIMIZATION STRATEGIES SUMMARY



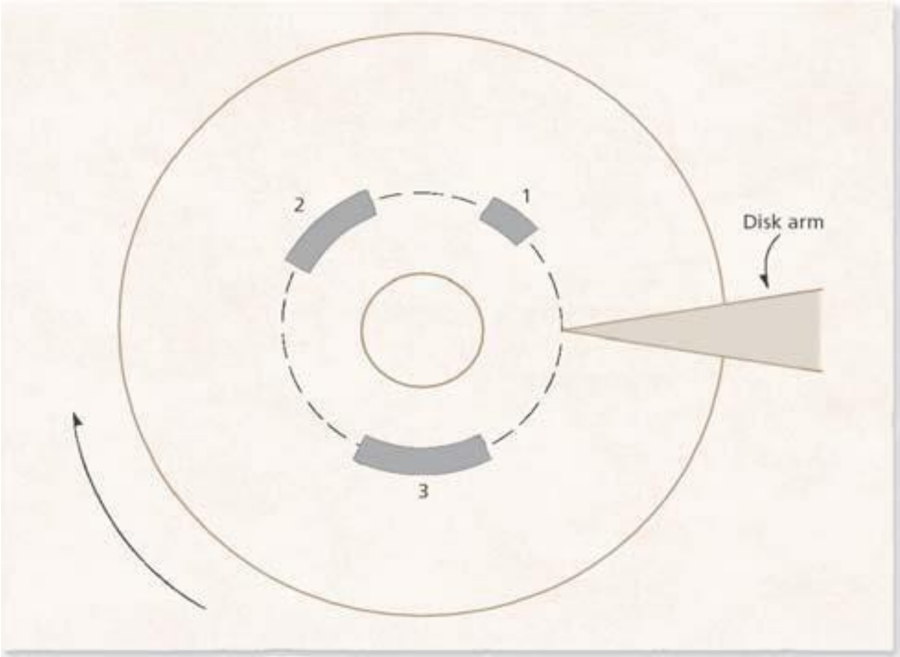
Strategy	*Description*
FCFS	Services requests in the order in which they arrive.
SSTF	Services the request that results in the shortest seek distance first.
SCAN	Head sweeps back and forth across the disk, servicing requests according to SSTF in a preferred direction.
C-SCAN	Head sweeps inward across the disk, servicing requests according to SSTF in the preferred (inward) direction. Upon reaching the innermost track, the head jumps to the outermost track and resumes servicing requests on the next inward pass.
FSCAN	Requests are serviced the same as SCAN, except newly arriving requests are postponed until the next sweep. Avoids indefinite postponement.
N-Step SCAN	Services requests as in FSCAN, but services only *n* requests per sweep. Avoids indefinite postponement.
LOOK	Same as SCAN except the head changes direction upon reaching the last request in the preferred direction.
C-LOOK	Same as C-SCAN except the head stops after servicing the last request in the preferred direction, then services the request to the cylinder nearest the opposite side of the disk.

ROTATIONAL OPTIMIZATION

- Seek time formerly dominated performance concerns
  - Today, seek times and rotational latency are the same order of magnitude
- Recently developed strategies attempt to optimization disk performance by reducing rotational latency
- Important when accessing small pieces of data distributed throughout the disk surfaces

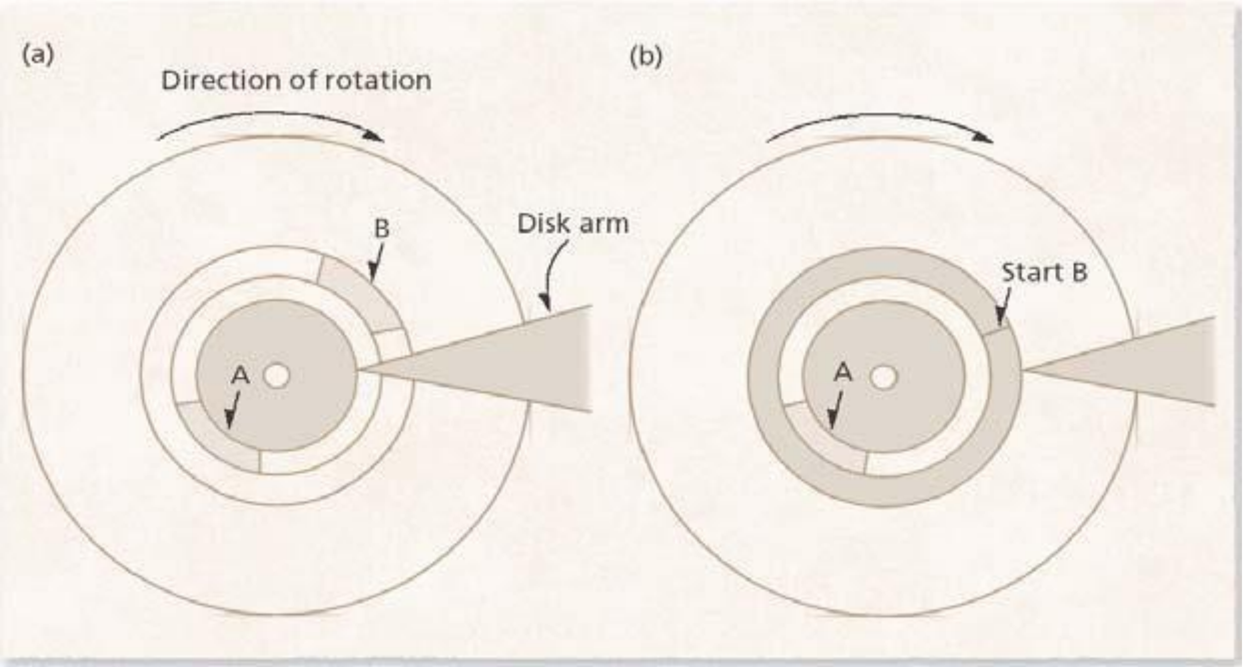
SLTF SCHEDULING

- Shortest-latency-time-first scheduling
  - On a given cylinder, service request with shortest rotational latency first
  - Easy to implement
  - Achieves near-optimal performance for rotational latency



SPTF AND SATF SCHEDULING

- Shortest-positioning-time-first scheduling
  - Positioning time: Sum of seek time and rotational latency
  - SPTF first services the request with the shortest positioning time
  - Yields good performance
  - Can indefinitely postpone requests



- Shortest-access-time-first scheduling
  - Access time: positioning time plus transmission time
  - High throughput
- Again, possible to indefinitely postpone requests
- Both SPTF and SATF can implement LOOK to improve performance
- Weakness
  - Both SPTF and SATF require knowledge of disk performance characteristics which might not be readily available due to error correcting data and transparent reassignment of bad sectors

FILE AND DATABASE SYSTEMS

INTRODUCTION

- Files
  - Named collection of data that is manipulated as a unit
  - Reside on secondary storage devices

- Operating systems can create an interface that facilitates navigation of a user's files
  - – File systems can protect such data from corruption or total loss from disasters
  - – Systems that manage large amounts of shared data can benefit from databases as an alternative to files.

DATA HIERARCHY

- Information is stored in computers according to a data hierarchy.
- Lowest level of data hierarchy is composed of bits
  - – Bit patterns represent all data items of interest in computer systems
- Next level in the data hierarchy is fixed-length patterns of bits such as bytes, characters and words
  - – Byte: typically 8 bits
  - – Word: the number of bits a processor can operate on at once
  - – Characters map bytes (or groups of bytes) to symbols such as letters, numbers, punctuation and new lines
- Three most popular character sets in use today: ASCII, EBCDIC and Unicode
  - – Field: a group of characters
  - – Record: a group of fields
  - – File: a group of related records
- Highest level of the data hierarchy is a file system or database
- A volume is a unit of data storage that may hold multiple files.

FILES

A file is a named collection of data that may be manipulated as a unit by operations such as

- **open** —Prepare a file to be referenced.

- **close** —Prevent further reference to a file until it is reopened.
- **create** — Create a new file.
- **destroy** —Remove a file.
- **copy—**Copy the contents of one file to another.
- **rename**—Change the name of a file.
- **list—**Print or display the contents of a file.

Individual data items within the file may be manipulated by operations like

- **read—**Copy data from a file to a process's memory.
- **write** —Copy data from a process's memory to a file.
- **update—**Modify an existing data item in a file.
- **insert—**Add a new data item to a file.
- **delete** —Remove a data item from a

file. Files may be characterized by attributes such as

- size —the amount of data stored in the file.
- **location—**the location of the file (in a storage device or in the system's logical file organization).
- **accessibility—**restrictions placed on access to file data.
- **type—**how the file data is used. For example, an executable file contains machine instructions for a process. A data file may specify the application that is used to access its data.
- **volatility**—the frequency with which additions and deletions are made to a file.
- **activity—**the percentage of a file's records accessed during a given period of time.

FILE SYSTEMS

A file system organizes files and manages access to data.3 File systems are responsible for:

- **File management—**providing the mechanisms for files to be stored, referenced, shared, and secured.

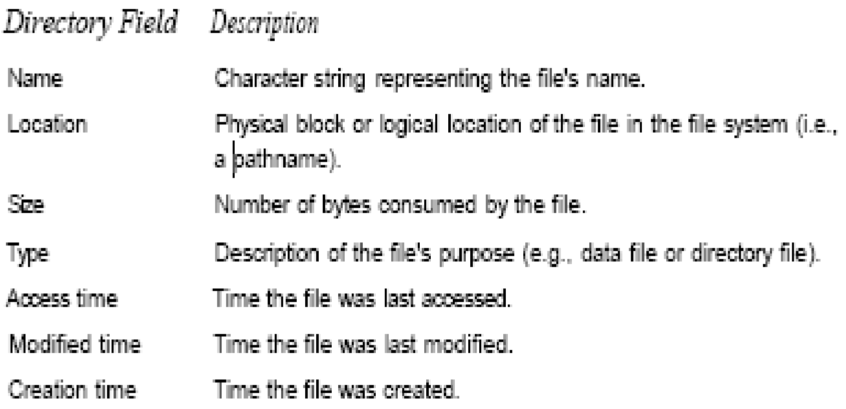
- **Auxiliary storage management** —allocating space for files on secondary or tertiary storage devices.
- **File integrity mechanisms** —ensuring that the information stored in a file is uncorrupted. When file integrity is assured, files contain only the information that they are intended to have.
- **Access methods—**how the stored data can be accessed.

The mechanism tor sharing files should provide various types of controlled access such as **read access, write access, execute access** or various combinations of these.

- **File system characteristics**
  - – Should exhibit device independence:
  - • Users should be able to refer to their files by symbolic names rather than having to use physical device names
  - – Should also provide backup and recovery capabilities to prevent either accidental loss or malicious destruction of information
  - – May also provide encryption and decryption capabilities to make information useful only to its intended audience

DIRECTORIES

- Directories:
  - – Files containing the names and locations of other files in the file system, to organize and quickly locate files
- Directory entry stores information such as:
  - – File name
  - – Location
  - – Size
  - – Type
  - – Accessed
  - – Modified and creation times



Directory Field	*Description*
Name	Character string representing the file's name.
Location	Physical block or logical location of the file in the file system (i.e., a pathname).
Size	Number of bytes consumed by the file.
Type	Description of the file's purpose (e.g., data file or directory file).
Access time	Time the file was last accessed.
Modified time	Time the file was last modified.
Creation time	Time the file was created.

Single-level (or flat) file system:

– Simplest file system organization

– Stores all of its files using one directory

– No two files can have the same name

– File system must perform a linear search of the directory contents to locate each file, which can lead to poor performance

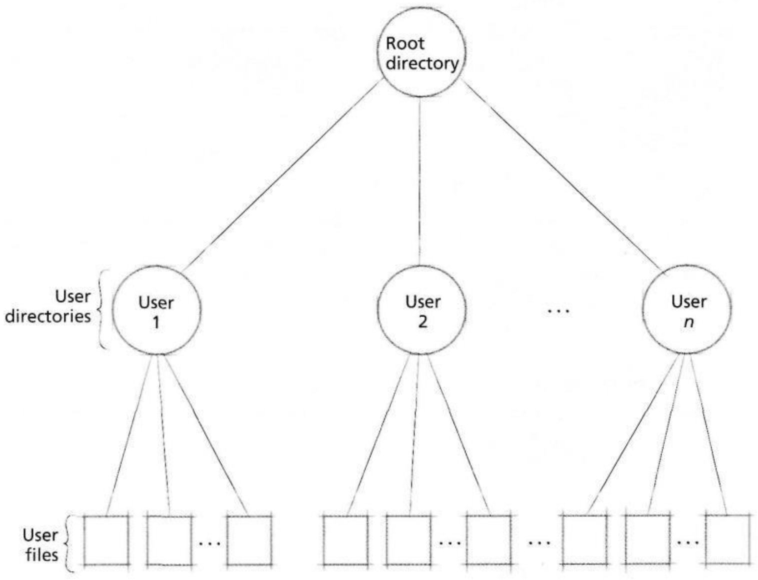
- Hierarchical file system:

– A root indicates where on the storage device the root directory begins

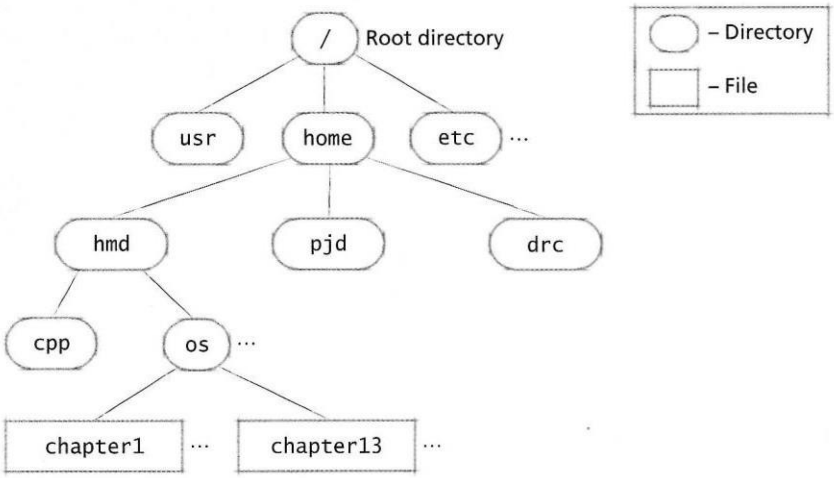
– The root directory points to the various directories, each of which contains an entry for each of its files

– File names need be unique only within a given user directory

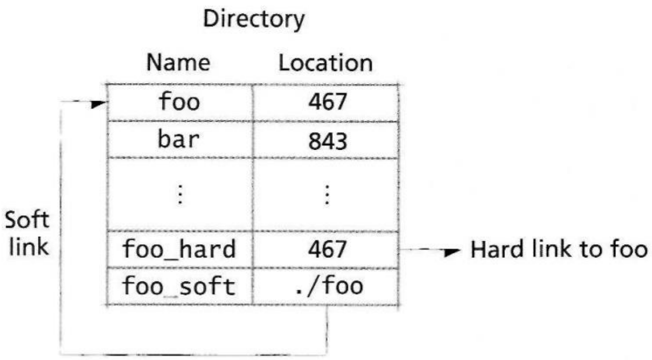
– The name of a file is usually formed as the pathname from the root directory to the file



- Working directory
  - Simplifies navigation using pathnames
  - Enables users to specify a pathname that does not begin at the root directory (i.e., a relative path)
  - Absolute path (i.e., the path beginning at the root) = working directory + relative path



- Link: a directory entry that references a data file or directory located in a different directory
  - Facilitates data sharing and can make it easier for users to access files located throughout a file system’s directory structure
  - Soft link: directory entry containing the pathname for another file
  - Hard link: directory entry that specifies the location of the file (typically a block number) on the storage device
  - Because a hard link specifies a physical location of a file, it references invalid data when the physical location of its corresponding file changes
  - Because soft links store the logical location of the file in the file system, they do not require updating when file data is moved
  - However, if a user moves a file to different directory or renames the file, any soft links to that file are no longer valid



Directory

Name	Location
foo	467
bar	843
⋮	⋮
foo_hard	467
foo_soft	./foo

METADATA

- Metadata
  - – Information that protects the integrity of the file system
  - – Cannot be modified directly by users
- Many file systems create a superblock to store critical information that protects the integrity of the file system
  - – A superblock might contain:
- The file system identifier
- The location of the storage device's free blocks
  - – To reduce the risk of data loss, most file systems distribute redundant copies of the superblock throughout the storage device
- File open operation returns a file descriptor
  - – A non-negative integer index into the open-file table
- From this point on, access to the file is directed through the file descriptor
- To enable fast access to file-specific information such as permissions, the open-file table often contains file control blocks, also called file attributes:
  - – Highly system-dependent structures that might include the file's symbolic name, location in secondary storage, access control data and so on

MOUNTING

- Mount operation
  - – Combines multiple file systems into one namespace so that they can be referenced from a single root directory

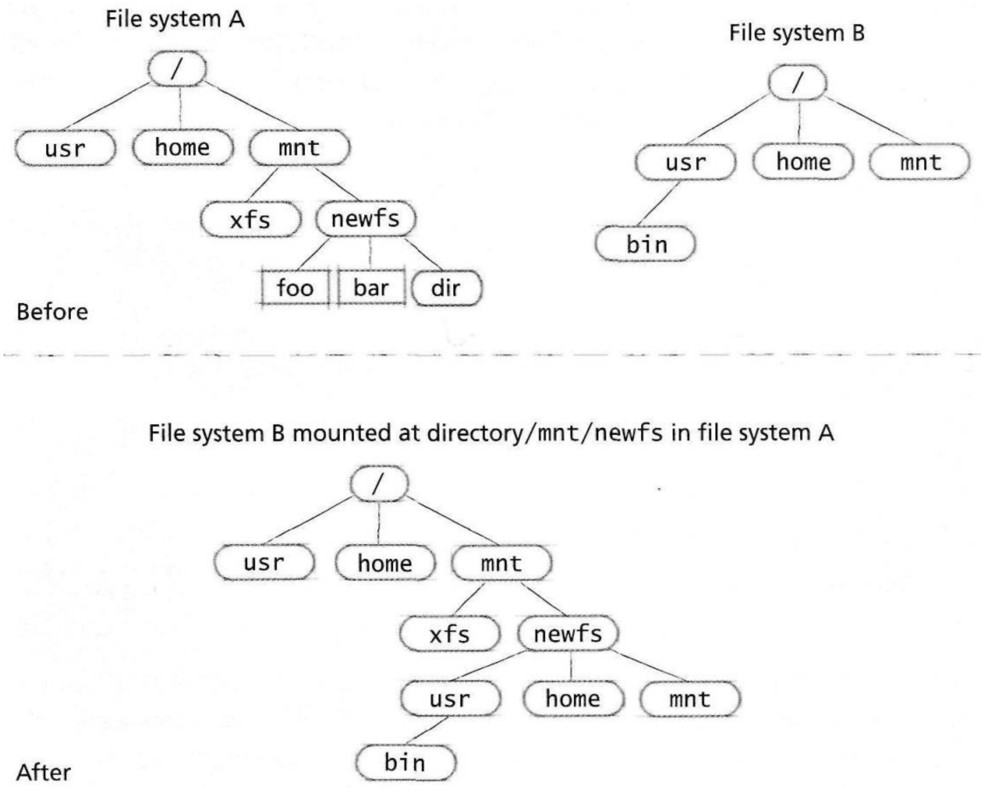
– Assigns a directory, called the mount point, in the native file system to the root of the mounted file system

- File systems manage mounted directories with mount tables:

  – Contain information about the location of mount points and the devices to which they point

- When the native file system encounters a mount point, it uses the mount table to determine the device and type of the mounted file system

- Users can create soft links to files in mounted file systems but cannot create hard links between file systems



FILE ORGANIZATION:

- **Sequential—**Records are placed in physical order. The "next" record is the one that physically follows the previous record. This organization is natural for files stored on magnetic tape, an inherently sequential medium.

- **Direct—**Records are directly (randomly) accessed by their physical addresses on a direct access storage device (DASD).

- **Indexed sequential**—Records on disk are arranged in logical sequence according to a key contained in each record.
- **Partitioned**—This is essentially a file of sequential sub files. Each sequential subfile is called a **member.** The starting address of each member is stored in the file's directory. Partitioned files have been used to store program libraries or macro libraries.

FILE ALLOCATION

- File allocation
  - – Problem of allocating and freeing space on secondary storage is somewhat like that experienced in primary storage allocation under variable-partition multiprogramming
  - – Contiguous allocation systems have generally been replaced by more dynamic noncontiguous allocation systems
- Files tend to grow or shrink over time
- Users rarely know in advance how large their files will be used.

CONTIGUOUS FILE ALLOCATION

- Contiguous allocation
  - – Place file data at contiguous addresses on the storage device

– Advantages

- Successive logical records typically are physically adjacent to one another

– Disadvantages

- External fragmentation
- Poor performance can result if files grow and shrink over time
- If a file grows beyond the size originally specified and no contiguous free blocks are available, it must be transferred to a new area of adequate size, leading to additional I/O operations.

LINKED-LIST NONCONTIGUOUS FILE ALLOCATION

- Sector-based linked-list noncontiguous file allocation scheme:

– A directory entry points to the first sector of a file

- The data portion of a sector stores the contents of the file
- The pointer portion points to the file's next sector

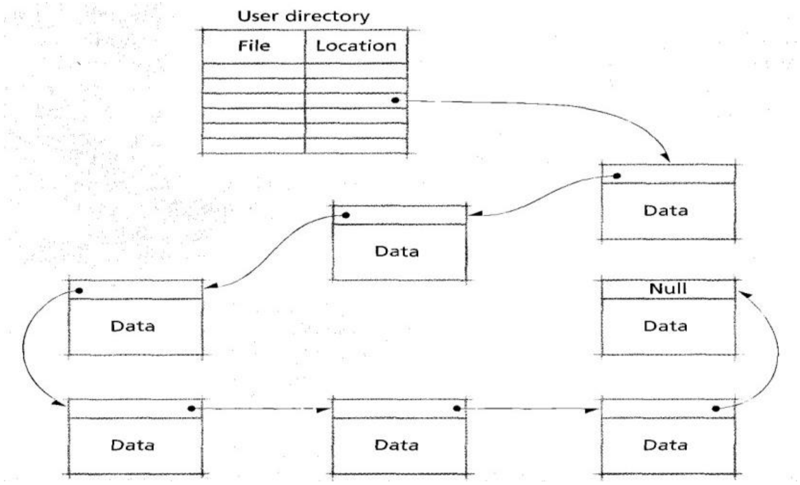
– Sectors belonging to a common file form a linked list

- When performing block allocation, the system allocates blocks of contiguous sectors (sometimes called extents)
- Block chaining

– Entries in the user directory point to the first block of each file

– File blocks contain:

- A data block
- A pointer to the next block



- When locating a record

– The chain must be searched from the beginning

– If the blocks are dispersed throughout the storage device (which is normal), the search process can be slow as block-to-block seeks occur

- Insertion and deletion are done by modifying the pointer in the previous block
- Large block sizes

– Can result in significant internal fragmentation

- Small block sizes

– May cause file data to be spread across multiple blocks dispersed throughout the storage device

– Poor performance as the storage device performs many seeks to access all the records of a file

TABULAR NONCONTIGUOUS FILE ALLOCATION

– Uses tables storing pointers to file blocks

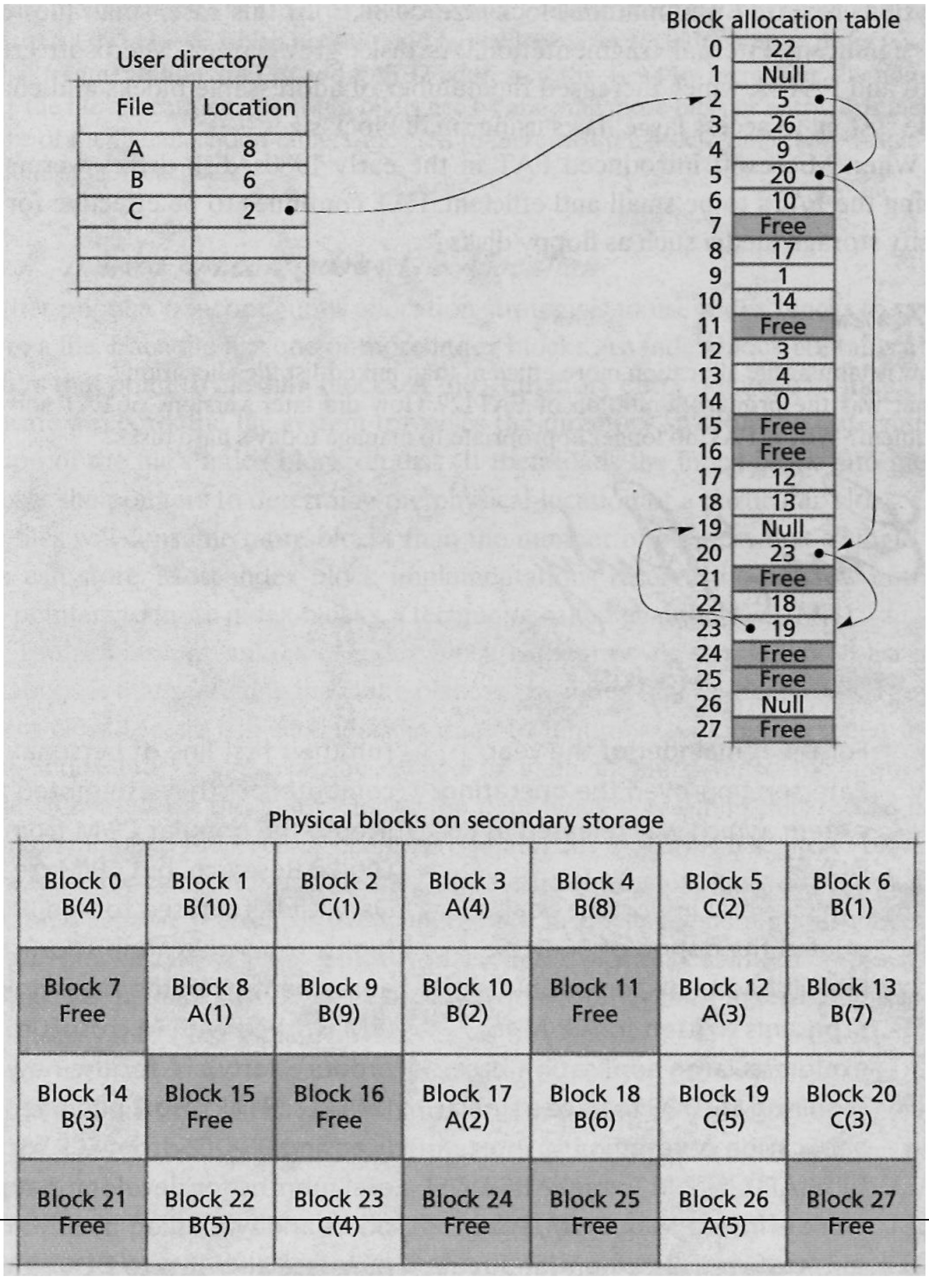
- Reduces the number of lengthy seeks required to access a particular record

– Directory entries indicate the first block of a file

– Current block number is used as an index into the block allocation table to determine the location of the next block.

- If the current block is the file's last block, then its block allocation

table entry is null



User directory

File	Location
A	8
B	6
C	2 •

Block allocation table

0	22
1	Null
2	5 •
3	26
4	9
5	20 •
6	10
7	Free
8	17
9	1
10	14
11	Free
12	3
13	4
14	0
15	Free
16	Free
17	12
18	13
19	Null
20	23 •
21	Free
22	18
23	• 19
24	Free
25	Free
26	Null
27	Free

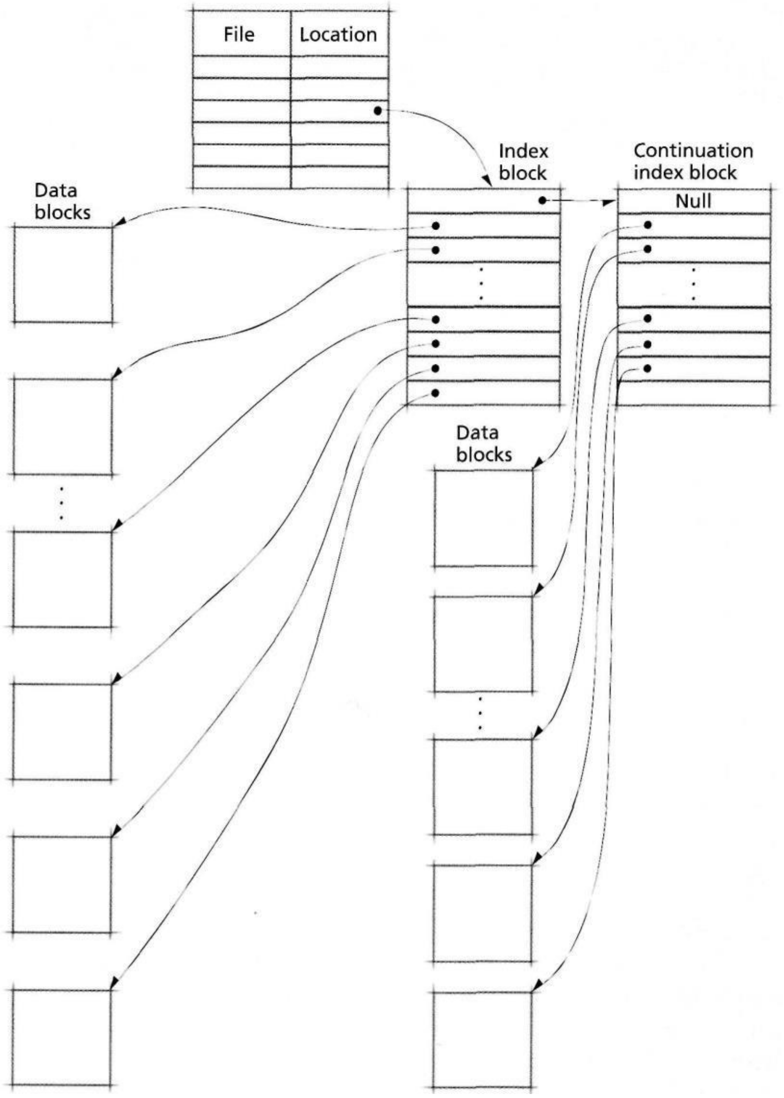
Physical blocks on secondary storage

Block 0 B(4)	Block 1 B(10)	Block 2 C(1)	Block 3 A(4)	Block 4 B(8)	Block 5 C(2)	Block 6 B(1)
Block 7 Free	Block 8 A(1)	Block 9 B(9)	Block 10 B(2)	Block 11 Free	Block 12 A(3)	Block 13 B(7)
Block 14 B(3)	Block 15 Free	Block 16 Free	Block 17 A(2)	Block 18 B(6)	Block 19 C(5)	Block 20 C(3)
Block 21 Free	Block 22 B(5)	Block 23 C(4)	Block 24 Free	Block 25 Free	Block 26 A(5)	Block 27 Free

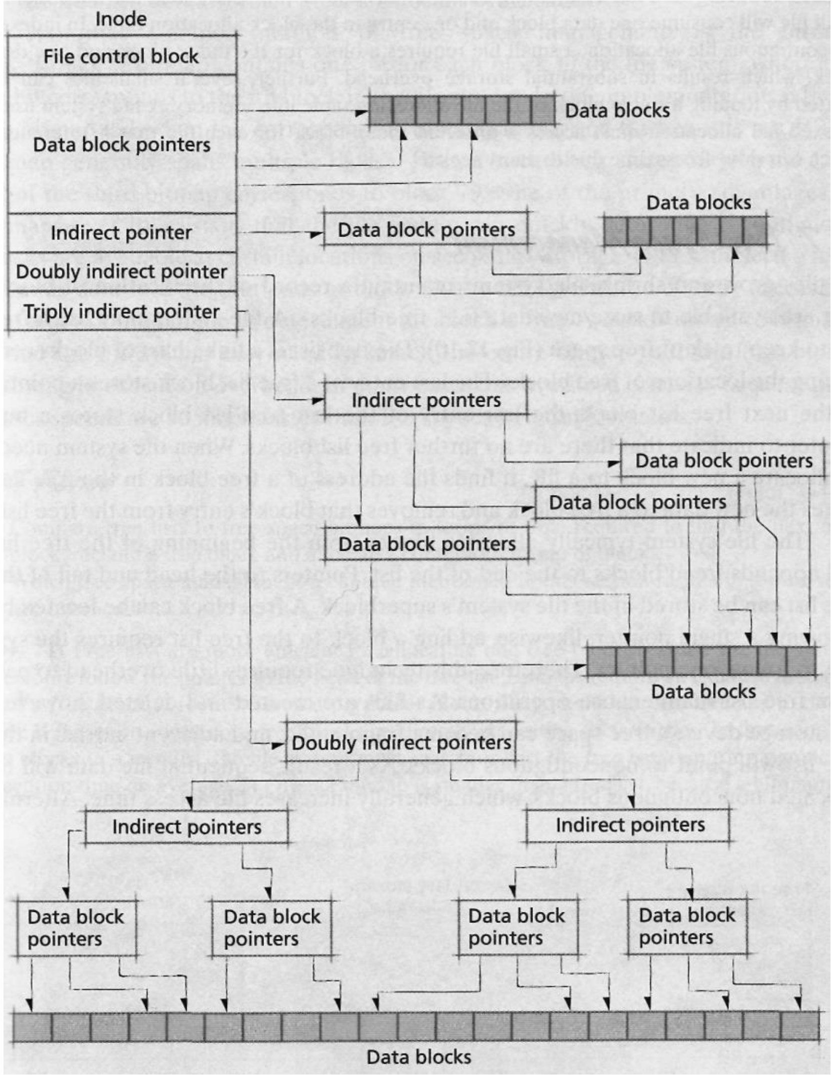
- Pointers that locate file data are stored in a central location
  - – The table can be cached so that the chain of blocks that compose a file can be traversed quickly
  - – Improves access times
- To locate the last record of a file, however:
  - – The file system might need to follow many pointers in the block allocation table
  - – Could take significant time
- When a storage device contains many blocks:
  - – The block allocation table can become large and fragmented
  - – Reduces file system performance
- A popular implementation of tabular noncontiguous file allocation is Microsoft's FAT file system.

INDEXED NONCONTIGUOUS FILE ALLOCATION

- Indexed noncontiguous file allocation:
  - – Each file has an index block or several index blocks
  - – Index blocks contain a list of pointers that point to file data blocks
  - – A file's directory entry points to its index block, which may reserve the last few entries to store pointers to more index blocks, a technique called chaining
- Primary advantage of index block chaining over simple linked-list implementations:
  - – Searching may take place in the index blocks themselves.
  - – File systems typically place index blocks near the data blocks they reference, so the data blocks can be accessed quickly after their index block is loaded.

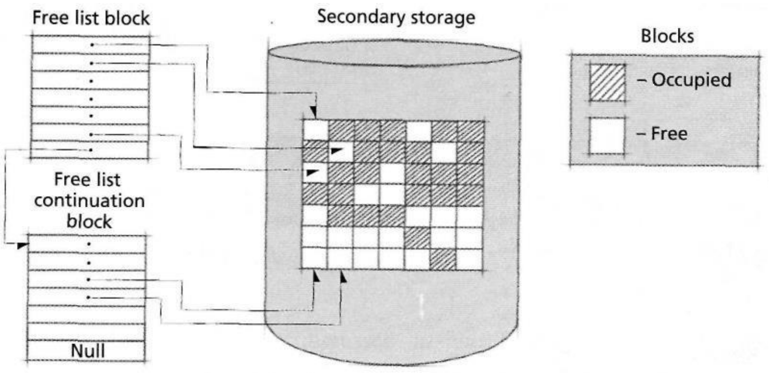


- Index blocks are called inodes (i.e., index nodes) in UNIX-based operating systems

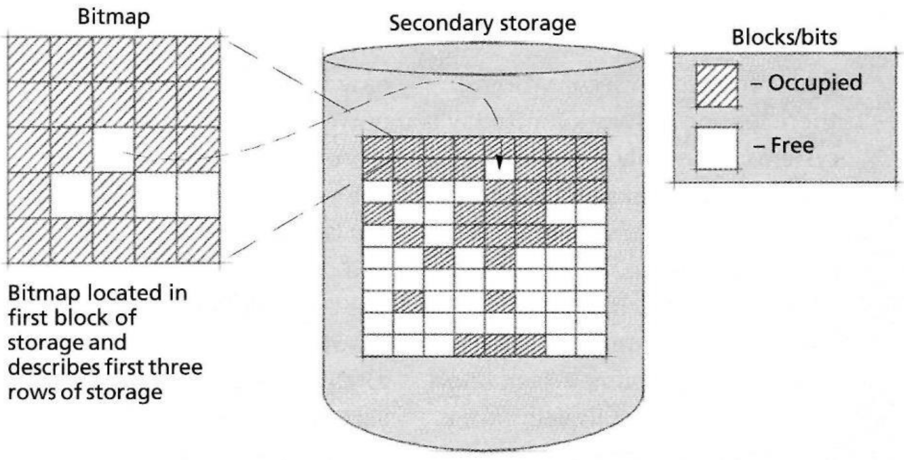


FREE SPACE MANAGEMENT

- Some systems use a free list to manage the storage device’s free space
  - Free list: Linked list of blocks containing the locations of free blocks
  - Blocks are allocated from the beginning of the free list
  - Newly freed blocks are appended to the end of the list
- Low overhead to perform free list maintenance operations
- Files are likely to be allocated in noncontiguous blocks
  - Increases file access time



- A bitmap contains one bit for each block in memory
  - *i*th bit corresponds to the *i*th block on the storage device
- Advantage of bitmaps over free lists:
  - The file system can quickly determine if contiguous blocks are available at certain locations on secondary storage
- Disadvantage of bitmaps:
  - The file system may need to search the entire bitmap to find a free block, resulting in substantial execution overhead


Bitmap
Secondary storage
Blocks/bits
– Occupied
– Free
Bitmap located in first block of storage and describes first three rows of storage

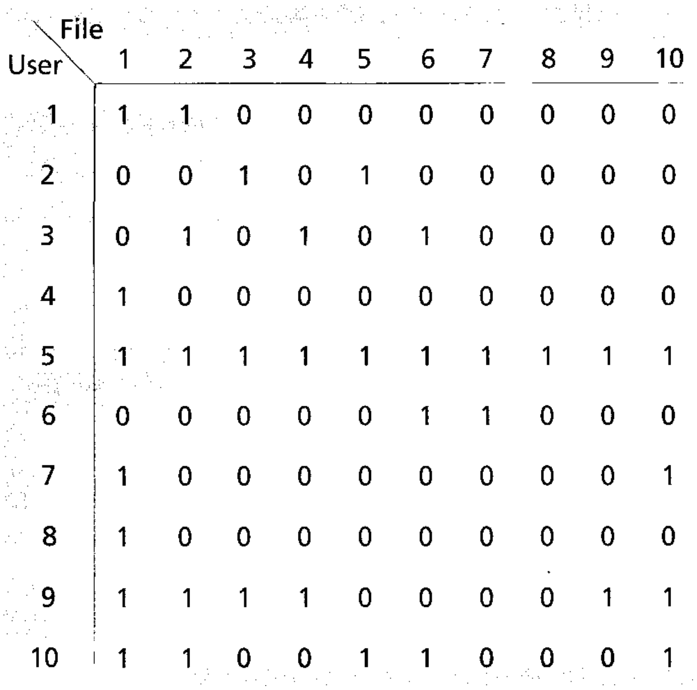
FILE ACCESS CONTROL

- Files are often used to store sensitive data such as:
  - Credit card numbers
  - Passwords
  - Social security numbers

- Therefore, they should include mechanisms to control user access to data.
  - Access control matrix
  - Access control by user classes

ACCESS CONTROL MATRIX

- Two-dimensional access control matrix:
  - Entry a*ij* is 1 if user *i* is allowed access to file *j*
  - Otherwise a*ij* = 0
- In an installation with a large number of users and a large number of files, this matrix generally would be large and sparse
- Inappropriate for most systems



User \ File	1	2	3	4	5	6	7	8	9	10
1	1	1	0	0	0	0	0	0	0	0
2	0	0	1	0	1	0	0	0	0	0
3	0	1	0	1	0	1	0	0	0	0
4	1	0	0	0	0	0	0	0	0	0
5	1	1	1	1	1	1	1	1	1	1
6	0	0	0	0	0	1	1	0	0	0
7	1	0	0	0	0	0	0	0	0	1
8	1	0	0	0	0	0	0	0	0	0
9	1	1	1	1	0	0	0	0	1	1
10	1	1	0	0	1	1	0	0	0	1

ACCESS CONTROL BY USER CLASSES

- A technique that requires considerably less space is to control access to various user classes
- User classes can include:
  - The file owner
  - A specified user
  - Group

    – Project

    – Public

- Access control data

– Can be stored as part of the file control block

– Often consumes an insignificant amount of space