

Operating System

BEG 373CO

Year: III

Semester: I

Teaching Schedule			Examination Scheme				
Hours/Week							
Theory	Tutorial	Practical	Internal		Final		Total
3	1	2	Theory	Practical	Theory	Practical	150
			20	50	80	-	

Course Objective:

After completion of this course, the students will acquire the concepts of Operating Systems and Design and Implement them.

1. Operating System Overview

(3 hrs) OS objectives and functions

- 1.1.1 OS as a user/computer interface
- 1.1.2 OS as Resource Manager
- 1.2 Evolution of Operating Systems
 - 1.2.1 Serial Processing
 - 1.2.2 Simple Batch Systems
 - 1.2.3 Multi-programmed Batch Systems
 - 1.2.4 Time-Sharing Systems

2. Process

(17

hrs)

- 2.1 Introduction to Process
 - 2.1.1 The process Model
 - 2.1.2 Implementation of Process
 - 2.1.3 Threads
- 2.2. Inter Process Communication (IPC)
 - 2.2.1 Race Conditions
 - 2.2.2 Critical Sections

- 2.2.3 Mutual Exclusion with Busy Waiting
- 2.2.4 Sleep and Wakeup
- 2.2.5 Semaphores
- 2.2.6 Monitors
- 2.2.7 Message Passing
- 2.3 Classical IPC problems
- 2.4 Process scheduling
 - 2.4.1 Preemptive Vs. Non Preemptive Scheduling
 - 2.4.2 Round Robin Scheduling
 - 2.4.3 Priority Scheduling
 - 2.4.4 Multiple Queues
 - 2.4.5 Shortest Job First
 - 2.4.6 Real time Scheduling
 - 2.4.7 Two-Level Scheduling

3. Input/Output (3 hrs)

- 3.1 Principles of I/O Hardware
- 3.2 Principles of I/O Software
- 3.3 Disks
- 3.4 Clocks
- 3.5 Terminals

4. Deadlocks (4 hrs)

- 4.1 Resources of Deadlock
- 4.2 Principles of Deadlock
- 4.3 Deadlock Detection and algorithm
- 4.4 Deadlock Avoidance

5. Memory Management (5 hrs)

- 5.1 Fixed and Variable partition systems
- 5.2 Bit maps
- 5.3 Memory management with linked list (First fit, best fit, next fit, quick fit and buddy system)
- 5.4 Multiprogramming memory management techniques
- 5.5 Virtual Memory

- 5.5.1 Paging and Segmentation
- 5.5.2 Swapping and page replacement

**6. Real Time Operating System
(2 hrs)**

- 6.1 Introduction and Example
- 6.2 Real Time Terminologies
 - 6.2.1 Soft Real Time
 - 6.2.2 Hard Real Time
 - 6.2.3 Real Real Time
 - 6.2.4 Firm Real Time

**7. Distributed Operating System
(3 hrs)**

- 7.1 Introduction
- 7.2 Communication and Synchronization
- 7.3 Process and Processor in Distributed OS

8. File Systems (3 hrs)

- 8.1 Files and Directories
- 8.2 File System Implementation
- 8.3 File Sharing and Locking

**9. Case Studies: Aspect of Different OS
(5 hrs)**
(Linux, Windows, Mac, iOS, Android OS)

Laboratory:

There shall be laboratories exercises covering following topics;

- i. Implementation of Process (Creation of process, Parent process, Child Process)

- ii. Interprocess Communication(Race Condition, Mutual Exclusion, Semaphores, Monitors, Message Passing)
- iii. Process Scheduling(Round Robin, Priority, Shortest Job first)
- iv. Implementation of Deadlocks
- v. Memory Management

Reference:

1. Operating Systems: Design and Implementation
 - Tanenbaum A.S. , Woodhull A.S. (Prentice-Hall)
2. Operating System: Internals and Design Principles
 - Stallings, William (prentice-Hall)
3. Operating System Concepts
 - Silberschatz A., Galvin P.B. (Addison- Wesley)
4. Mark Donovan: System Programming.

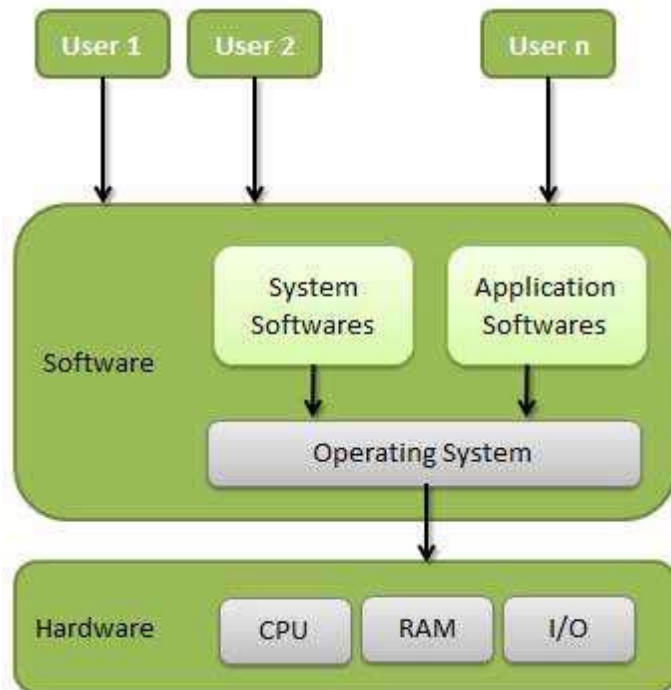
Marks Distribution:

Chapter	Marks
1	6
2	26
3	6
4	8
5	10
6,7	8
8,9	16
Total	80

Chapter-1: Operating System overview

Operating System:

An operating system is a collection of system programs that control computer and any other peripherals connected to it. 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.



The main objectives of an operating system are:

- **Convenience:** An OS makes a computer more convenient to use.
- **Efficiency:** An OS allows the computer system resources to be used in an efficient manner.
- **Ability to evolve:** An OS should be constructed in such a way as to permit the effective development, testing, and introduction of new system functions without interfering with service.

Some important functions of an operating System are:

- ✓ Memory Management
- ✓ Processor Management

- ✓ Device Management
- ✓ File Management
- ✓ Security
- ✓ Control over system performance
- ✓ Job accounting
- ✓ Error detecting
- ✓ Coordination between other software and users

Memory Management

Memory management refers to management of Primary Memory or Main Memory. Main memory is a large array of words or bytes where each word or byte has its own address.

Main memory provides a fast storage that can be access directly by the CPU. So for a program to be executed, it must in the main memory. Operating System does the following activities for memory management.

- ✓ Keeps tracks of primary memory i.e. what part of it are in use by whom, what part are not in use.
- ✓ In multiprogramming, OS decides which process will get memory when and how much.
- ✓ Allocates the memory when the process requests it to do so.
- ✓ De-allocates the memory when the process no longer needs it or has been terminated.

Processor Management

In multiprogramming environment, OS decides which process gets the processor when and how much time. This function is called process scheduling. Operating System does the following activities for processor management.

- ✓ Keeps tracks of processor and status of process. Program responsible for this task is known as traffic controller.
- ✓ Allocates the processor (CPU) to a process.
- ✓ De-allocates processor when processor is no longer required.

Device Management

OS manages device communication via their respective drivers. Operating System does the following activities for device management.

- ✓ Keeps tracks of all devices. Program responsible for this task is known as the I/O controller.

- ✓ Decides which process gets the device when and for how much time.
- ✓ Allocates the device in the efficient way.
- ✓ De-allocates devices.

File Management

A file system is normally organized into directories for easy navigation and usage. These directories may contain files and other directions. Operating System does the following activities for file management.

- ✓ Keeps track of information, location, uses, status etc. The collective facilities are often known as file system.
- ✓ Decides who gets the resources.
- ✓ Allocates the resources.
- ✓ De-allocates the resources.

Other Important Activities

Following are some of the important activities that Operating System does.

- ✓ **Security** -- By means of password and similar other techniques, preventing unauthorized access to programs and data.
- ✓ **Control over system performance** -- Recording delays between request for a service and response from the system.
- ✓ **Job accounting** -- Keeping track of time and resources used by various jobs and users.
- ✓ **Error detecting aids** -- Production of dumps, traces, error messages and other debugging and error detecting aids.
- ✓ **Coordination between other software and users** -- Coordination and assignment of compilers, interpreters, assemblers and other software to the various users of the computer systems.

We can call operating system as resource allocator. A computer has many resource hardware and software's, CPU, main memory I/O devices etc. The operating system acts as manager of these resources. Operating system is the control program. A control program manages the execution of user program to prevent error and improve use of computer.

1.1.1 Operating System as extended machine or virtual machine:

Generally, computer users sit in front of computer consisting of monitor, keyboard, mouse and system unit & use parts of computer system. For example, I/O device, system unit, memory etc. The computer system is design so that use of resource to

maximize what the user is performing. In this case the operating system is designed mostly for easy with same attention paid for performance, the performance is important to the user but it does not matter if most of the system is sitting idle, waiting for the slow i/o speed of the user. The operating system is design to maximize resource utilization. The program that hides the truth about the hardware from the programmer and present and a nice simple view a named file that can be read & written as “operating system”. Operating system shields the programmer from the interface, the abstraction offers by the operating system is slower & easier to use than the underlying hardware. The main function of operating system is to present the user with the equivalent of an extended machine or virtual machine that is easier to program than underlying hardware.

1.1.2 Operating system as resource manager:

Operating system is collection of software which is close to hardware. We can view operating system as a resource – hardware and software collector. A system has many hardware and software that may be required to solve the problem, cpu time, memory space, file storage space, I/O devices etc. the operating system acts as manager of these resources.

Modern computer consists of process, memories, times, disks, network, printer and wide varieties of other devices. The task of the operating system is to provide for an orderly and controlled allocation of the process, memories and I/O devices among the various programs competing for them. An operating system is a control program, a control program manages the execution of user program to prevent errors and improve use of computer. It is especially concerned with the operation and control of I/O devices. When a computer has multiple users the operating system manages and protects the memory I/O devices. The operating system keeps in trace that who is using which resource to grant resource required amount for usage and to mediate conflicting required different programs and users.

1.2 Evolution of Operating Systems

Operating systems are there from the very first computer generation. Operating systems keep evolving over the period of time. Following are few of the important types of operating system which are most commonly used.

1.2.1 Serial Processing

The serial processing operating systems are those which performs all the instructions into a sequential manner or the Instructions those are given by the user will be executed by using the FIFO Manner means First in First Out. All the instructions those are entered first in the system will be executed first and the instructions those

are entered later will be executed later. In serial processing, all the jobs are firstly prepared and stored on the card and after that card will be entered in the system and after that all the Instructions will be executed one by one. But the main problem is that a user doesn't interact with the system while he is working on the system, means the user can't be able to enter the data for execution.

1.2.2 Simple Batch Systems

The users of batch operating system do not interact with the computer directly. Each user prepares his job on an off-line device like punch cards and submits it to the computer operator. To speed up processing, jobs with similar needs are batched together and run as a group. Thus, the programmers leave their programs with the operator. The operator then sorts programs into batches with similar requirements.

The problems with Batch Systems are following.

- ✓ Lack of interaction between the user and job.
- ✓ CPU is often idle, because the speeds of the mechanical I/O devices are slower than CPU.
- ✓ Difficult to provide the desired priority.

1.2.3 Multi-programmed Batch Systems

Since several jobs are available on disk, OS can select which job to run, this essentially leads to job scheduling. Idea is to ensure that CPU is always busy. A single job may not be able to keep CPU busy.

In multiprogramming, multiple programs (or jobs) of different users can be executed simultaneously (i.e. at the same time). The multiple jobs that have to be run simultaneously must be kept in main memory and the operating system must manage them properly. If these jobs are ready to run, the processor must decide which one to run.

In multi-programmed batch system, the operating system keeps multiple jobs in main memory at a time. There may be many jobs that enter the system. Since in general, the main memory is too small to accommodate all jobs. So the jobs that enter the system to be executed are kept initially on the disk in the job pool. In other words, you can say that a job pool consists of all jobs residing on the disk awaiting allocation of main memory. When the operating system selects a job from a job pool, it loads that job into memory for execution.

The processor picks and begins to execute one of the jobs in main memory. Some jobs may have to wait for certain tasks (such as I/O operation), to complete. In a simple batch system or non-multi-programmed system, the processor would sit idle. In multi-programmed system, the CPU switches to second job and begins to execute it.

Similarly, when second job needs to wait, the processor is switched to third job, and so on. The processor also checks the status of previous jobs, whether they are completed or not.

The multi-programmed system takes less time to complete the same jobs than the simple batch system. The multi-programmed systems do not allow interaction between the processes (or jobs) when they are running on the computer.

Multiprogramming increases the CPU's utilization. Multi-programmed system provides an environment in which various computer resources are utilized effectively. The CPU always remains busy to run one of the jobs until all jobs complete their execution.

1.2.4 Time-Sharing Systems

Time sharing is a technique which enables many people, located at various terminals, to use a particular computer system at the same time. Time-sharing is a logical extension of multiprogramming. Processor's time which is shared among multiple users simultaneously is termed as time-sharing. The main difference between Multiprogrammed Batch Systems and Time-Sharing Systems is that in case of multiprogrammed batch systems, objective is to maximize processor use, whereas in Time-Sharing Systems objective is to minimize response time.

Multiple jobs are executed by the CPU by switching between them, but the switches occur so frequently. Thus, the user can receive an immediate response. For example, in a transaction processing, processor execute each user program in a short burst or quantum of computation. That is if n users are present, each user can get time quantum. When the user submits the command, the response time is in few seconds at most.

Operating system uses CPU scheduling and multiprogramming to provide each user with a small portion of a time. Computer systems that were designed primarily as batch systems have been modified to time-sharing systems.

Advantages of Timesharing operating systems are following

- ✓ Provide advantage of quick response.
- ✓ Avoids duplication of software.
- ✓ Reduces CPU idle time.

Disadvantages of Timesharing operating systems are following.

- ✓ Problem of reliability.
- ✓ Question of security and integrity of user programs and data.
- ✓ Problem of data communication.

Chapter-2: Process

A process is a program under execution. The execution of a process must progress in a sequential fashion. In other words, a process is defined as an entity which represents the basic unit of work to be implemented in the system.

To put it in simple terms, we write our computer programs in a text file and when we execute this program, it becomes a process which performs all the tasks mentioned in the program.

When a program is loaded into the memory and it becomes a process, it can be divided into four sections — stack, heap, text and data.

Components of a process are given as:

S.N.	Component	Description
1	Object Program	Code to be executed.
2	Data	Data to be used for executing the program.
3	Resources.	While executing the program, it may require some resources
4	Status	Verifies the status of the process execution. A process can run to completion only when all requested resources have been allocated to the process. Two or more processes could be executing the same program, each using their own data and resources

Program

A program by itself is not a process. It is a static entity made up of program statement while process is a dynamic entity. Program contains the instructions to be executed by processor.

A program takes a space at single place in main memory and continues to stay there.

A program does not perform any action by itself.

Process States

As a process executes, it changes state. The state of a process is defined as the current activity of the process.

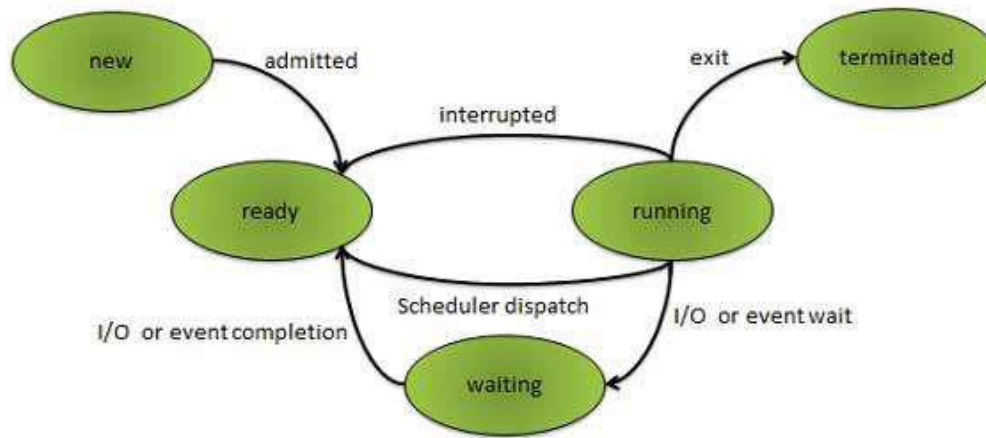


Figure: Process state

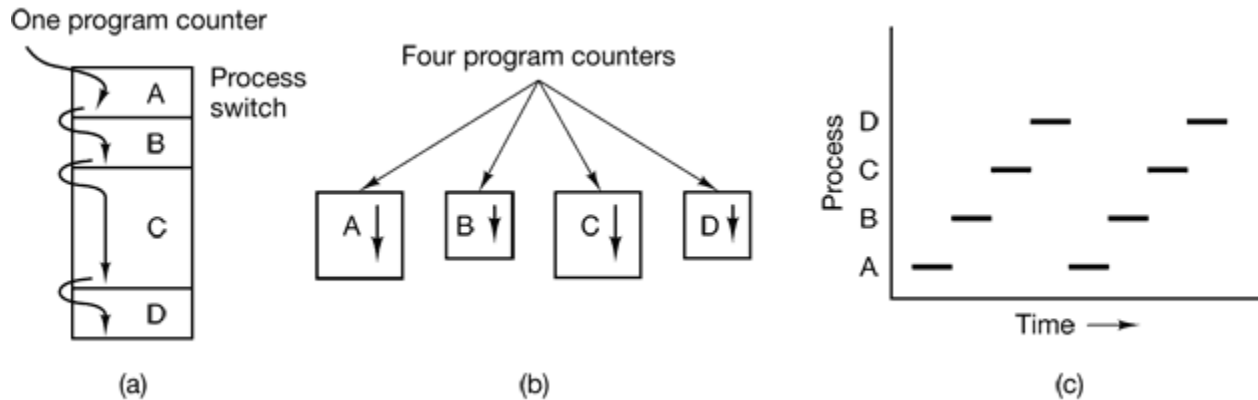
Process can have one of the following five states at a time.

S.N.	State	Description
1	New	The process is being created.
2	Ready	The process is waiting to be assigned to a processor. Ready processes are waiting to have the processor allocated to them by the operating system so that they can run.
3	Running	Process instructions are being executed (i.e. The process that is currently being executed).
4	Waiting	The process is waiting for some event to occur (such as the completion of an I/O operation).
5	Terminated	The process has finished execution.

2.1.1 The process Model

In the process model, all the runnable software on the computer, sometimes including the operating system, is organized into a number of **sequential processes**, or just processes for short. Conceptually, each process has its own virtual CPU

Fig. 2.a consists of multiprogramming with four programs in memory. In Fig. 2.b, there are four processes, each with its own flow of control (i.e., its own logical program counter), and each one running independently of the other ones. There is only one physical program counter, so when each process runs, its logical program counter is loaded into the real program counter.



When it is finished for the time being, the physical program counter is saved in the process' logical program counter in memory. In Fig. 2.c, there are we see that viewed over a long enough time interval, all the processes have made progress, but at any given instant only one process is actually running.

2.1.2 Implementation of Process

To implement the process model, the operating system maintains a table (an array of structures), called the **process table**, with one entry per process. (Some authors call these entries **process control blocks**.) This entry contains information about the process' state, its program counter, stack pointer, memory allocation, the status of its open files, its accounting and scheduling information, and everything else about the process that must be saved when the process is switched from *running* to *ready* or *blocked* state so that it can be restarted later as if it had never been stopped.

Figure below shows some of the more important fields in a typical system. The fields in the first column relate to process management. The other two columns relate to memory management and file management, respectively. It should be noted that precisely which fields the process table has is highly system dependent, but this figure gives a general idea of the kinds of information needed.

Process management	Memory management	File management
Registers	Pointer to text segment	Root directory
Program counter	Pointer to data segment	Working directory
Program status word	Pointer to stack segment	File descriptors
Stack pointer		User ID
Process state		Group ID
Priority		
Scheduling parameters		
Process ID		
Parent process		
Process group		
Signals		
Time when process started		
CPU time used		
Children's CPU time		
Time of next alarm		

Figure. 2.2 Structure of a typical process table entry i.e. Process Control Block (PCB)

2.1.3 Threads

A thread is a flow of execution through the process code, with its own program counter that keeps track of which instruction to execute next, system registers which hold its current working variables, and a stack which contains the execution history.

A thread shares with its peer threads few information like code segment, data segment and open files. When one thread alters a code segment memory item, all other threads see that.

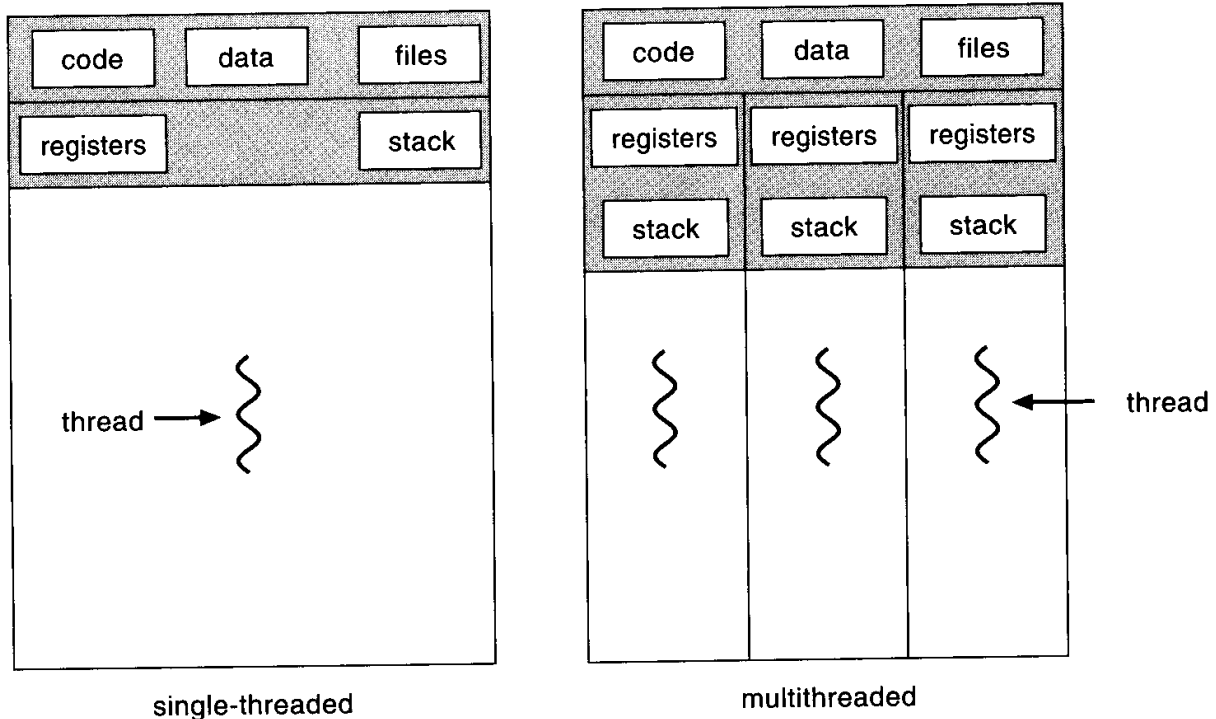


Figure: Single and multithreaded process

A thread is also called a **lightweight process**. Threads provide a way to improve application performance through parallelism. Threads represent a software approach to improving performance of operating system by reducing the overhead thread is equivalent to a classical process.

Difference between Process and Thread

S.N.	Process	Thread
1	Process is heavy weight or resource intensive.	Thread is light weight, taking lesser resources than a process.
2	Process switching needs interaction with operating system.	Thread switching does not need to interact with operating system.

3	In multiple processing environments, each process executes the same code but has its own memory and file resources.	All threads can share same set of open files, child processes.
4	If one process is blocked, then no other process can execute until the first process is unblocked.	While one thread is blocked and waiting, a second thread in the same task can run.
5	Multiple processes without using threads use more resources.	Multiple threaded processes use fewer resources.
6	In multiple processes each process operates independently of the others.	One thread can read, write or change another thread's data.

Advantages of Thread

- Threads minimize the context switching time.
- Use of threads provides concurrency within a process.
- Efficient communication.
- It is more economical to create and context switch threads.
- Threads allow utilization of multiprocessor architectures to a greater scale and efficiency.

Types of Thread

Threads are implemented in following two ways –

- **User Level Threads** – User managed threads.
- **Kernel Level Threads** – Operating System managed threads acting on kernel, an operating system core.

i. User Level Threads

In this case, the thread management kernel is not aware of the existence of threads. The thread library contains code for creating and destroying threads, for passing message and data between threads, for scheduling thread execution and for saving and restoring thread contexts. The application starts with a single thread.

Advantages

- Thread switching does not require Kernel mode privileges.
- User level thread can run on any operating system.
- Scheduling can be application specific in the user level thread.
- User level threads are fast to create and manage.

Disadvantages

- In a typical operating system, most system calls are blocking.
- Multithreaded application cannot take advantage of multiprocessing.

ii. Kernel Level Threads

In this case, thread management is done by the Kernel. There is no thread management code in the application area. Kernel threads are supported directly by the operating system. Any application can be programmed to be multithreaded. All of the threads within an application are supported within a single process.

The Kernel maintains context information for the process as a whole and for individuals threads within the process. Scheduling by the Kernel is done on a thread basis. The Kernel performs thread creation, scheduling and management in Kernel space. Kernel threads are generally slower to create and manage than the user threads.

Advantages

- Kernel can simultaneously schedule multiple threads from the same process on multiple processes.
- If one thread in a process is blocked, the Kernel can schedule another thread of the same process.
- Kernel routines themselves can be multithreaded.

Disadvantages

- Kernel threads are generally slower to create and manage than the user threads.
- Transfer of control from one thread to another within the same process requires a mode switch to the Kernel.

Multithreading Models

Some operating system provide a combined user level thread and Kernel level thread facility. Solaris is a good example of this combined approach. In a combined system, multiple threads within the same application can run in parallel on multiple processors and a blocking system call need not block the entire process. Multithreading models are three types

- Many to many relationship.
- Many to one relationship.
- One to one relationship.

i. Many to Many Model

The many-to-many model multiplexes any number of user threads onto an equal or smaller number of kernel threads.

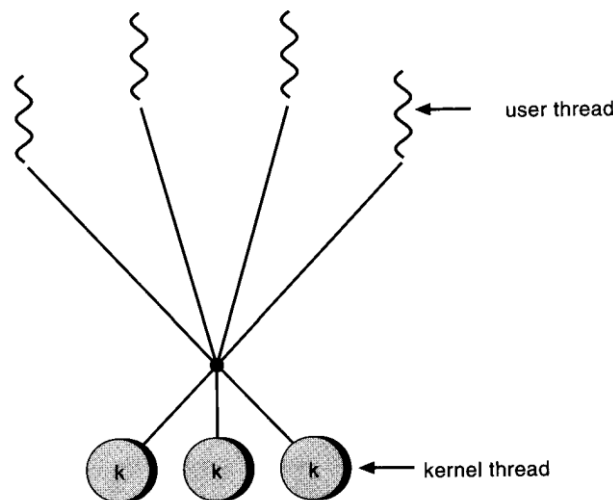


Figure: Many to Many Model

The diagram shows the many-to-many threading model where 6 user level threads are multiplexing with 6 kernel level threads. In this model, developers can create as many user threads as necessary and the corresponding Kernel threads can run in

parallel on a multiprocessor machine. This model provides the best accuracy on concurrency and when a thread performs a blocking system call, the kernel can schedule another thread for execution.

ii. Many to One Model

Many-to-one model maps many user level threads to one Kernel-level thread. Thread management is done in user space by the thread library. When thread makes a blocking system call, the entire process will be blocked. Only one thread can access the Kernel at a time, so multiple threads are unable to run in parallel on multiprocessors.

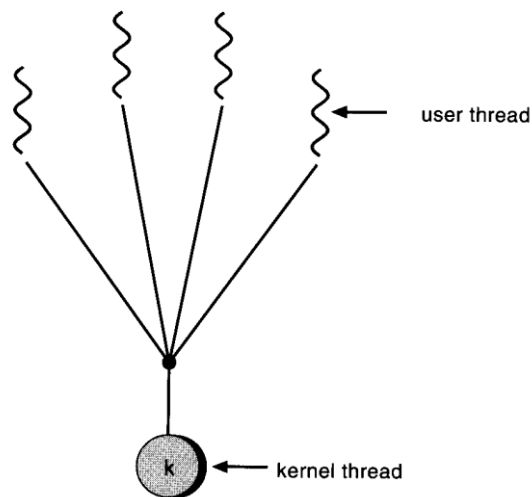


Figure: Many to One Model

If the user-level thread libraries are implemented in the operating system in such a way that the system does not support them, then the Kernel threads use the many-to-one relationship modes.

iii. One to One Model

There is one-to-one relationship of user-level thread to the kernel-level thread. This model provides more concurrency than the many-to-one model. It also allows another thread to run when a thread makes a blocking system call. It supports multiple threads to execute in parallel on microprocessors.

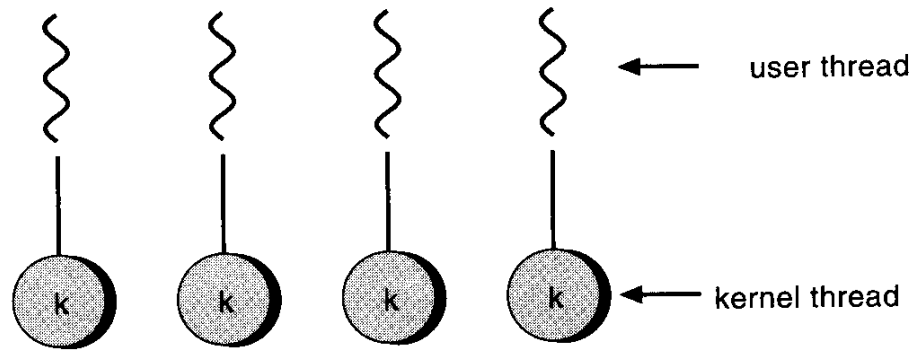


Figure: One to One Model

Disadvantage of this model is that creating user thread requires the corresponding Kernel thread. OS/2, windows NT and windows 2000 use one to one relationship model.

Difference between User-Level & Kernel-Level Thread

S.N.	User-Level Threads	Kernel-Level Thread
1	User-level threads are faster to create and manage.	Kernel-level threads are slower to create and manage.
2	Implementation is by a thread library at the user level.	Operating system supports creation of Kernel threads.
3	User-level thread is generic and can run on any operating system.	Kernel-level thread is specific to the operating system.
4	Multi-threaded applications cannot take advantage of multiprocessing.	Kernel routines themselves can be multithreaded.

2.2 Inter Process Communication (IPC)

Inter-process communication (IPC) is a mechanism that allows the exchange of data between processes. By providing a user with a set of programming interfaces, IPC helps a programmer organize the activities among different processes. IPC allows one application to control another application, thereby enabling data sharing without interference.

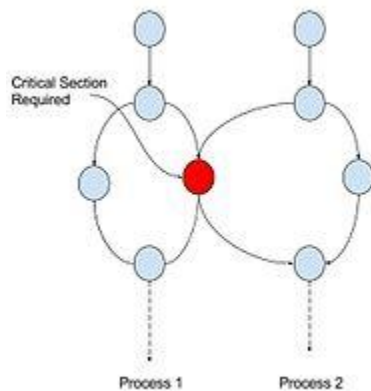
IPC enables data communication by allowing processes to use segments, semaphores, and other methods to share memory and information. IPC facilitates efficient message transfer between processes. The idea of IPC is based on Task Control Architecture (TCA). It is a flexible technique that can send and receive variable length arrays, data structures, and lists. It has the capability of using publish/subscribe and client/server data-transfer paradigms while supporting a wide range of operating systems and languages.

2.2.1 Race Conditions

In operating systems, processes that are working together share some common storage (main memory, file etc.) that each process can read and write. When two or more processes are reading or writing some shared data and the final result depends on who runs precisely when, are called race conditions. Concurrently executing threads that share data need to synchronize their operations and processing in order to avoid race condition on shared data. Only one ‘customer’ thread at a time should be allowed to examine and update the shared variable. Race conditions are also possible in Operating Systems. If the ready queue is implemented as a linked list and if the ready queue is being manipulated during the handling of an interrupt, then interrupts must be disabled to prevent another interrupt before the first one completes. If interrupts are not disabled then the linked list could become corrupt.

2.2.2 Critical Section

In concurrent programming, concurrent accesses to shared resources can lead to unexpected or erroneous behavior, so parts of the program where the shared resource is accessed is protected. This protected **section** is the **critical section** or **critical** region. It cannot be executed by more than one process.



Consider a system consisting of n processes $\{P_0, P_1, \dots, P_{n-1}\}$. Each process has a segment of code, called a **critical section**, in which the process may be changing common variables, updating a table, writing a file, and so on. The important feature of the system is that, when one process is executing in its critical section, no other process is to be allowed to execute in its critical section. Thus, the execution of critical sections by the processes is *mutually exclusive* in time. The *critical-section* problem is to design a protocol that the processes can use to cooperate. Each process must request permission to enter its critical section. The section of code implementing this request is the **entry section**. The critical section may be followed by an **exit section**. The remaining code is the **remainder section**.

do {

entry section

critical section

exit section

remainder section

 } while (1);

Figure: general structure of a typical process

A solution to the critical-section problem must satisfy the following three requirements:

1. **Mutual Exclusion:** If process P_i is executing in its critical section, then no other processes can be executing in their critical sections.
2. **Progress:** 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 section can participate in the decision on which will enter its critical section next, and this selection cannot be postponed indefinitely.
3. **Bounded Waiting:** There exists a bound on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

2.2.3 Mutual Exclusion with Busy Waiting

i. Disabling Interrupts

The most obvious way of achieving mutual exclusion is to allow a process to disable interrupts before it enters its critical section and then enable interrupts after it leaves its critical section.

By disabling interrupts the CPU will be unable to switch processes. This guarantees that the process can use the shared variable without another process accessing it.

But, disabling interrupts, is a major undertaking. At best, the computer will not be able to service interrupts for, maybe, a long time (who knows what a process is doing in its critical section?). At worst, the process may never enable interrupts, thus (effectively) crashing the computer.

ii. Lock Variables

Another method, which is obviously flawed, is to assign a lock variable. This is set to (say) 1 when a process is in its critical section and reset to zero when a processes exits its critical section.

It does not take a great leap of intuition to realize that this simply moves the problem from the shared variable to the lock variable.

iii. Strict Alternation

Process 0	Process 1
<pre>While (TRUE) { while (turn != 0); // wait critical_section(); turn = 1; Remainder_section(); }</pre>	<pre>While (TRUE) { while (turn != 1); // wait critical_section(); turn = 0; Remainder_section(); }</pre>

These code fragments offer a solution to the mutual exclusion problem.

Assume the variable *turn* is initially set to zero.

Process 0 is allowed to run. It finds that *turn* is zero and is allowed to enter its critical region. If process 1 tries to run, it will also find that *turn* is zero and will have to wait (the while statement) until *turn* becomes equal to 1.

When process 0 exits its critical region it sets *turn* to 1, which allows process 1 to enter its critical region.

If process 0 tries to enter its critical region again it will be blocked as *turn* is no longer zero.

iv. Algorithm 2:

The problem with strict alternation is that it does not retain sufficient information about the state of each process; it remembers only which process is allowed to enter into its critical section. To correct this problem, we can replace the variable *turn* with the following array:

```
boolean flag[2];
```

The elements of the array are initialized to false.

Process 0	Process 1
<pre> while (1) { flag[0]=TRUE; while (flag[1]=TRUE); // wait critical_section(); flag[0]=FALSE; Remainder_section(); } </pre>	<pre> while (1) { flag[1]=TRUE; while (flag[0]=TRUE); // wait critical_section(); flag[1]=FALSE; Remainder_section(); } </pre>

v. Peterson's Solution

A solution to the mutual exclusion problem that does not require strict alternation, but still uses the idea of lock (and warning) variables together with the concept of taking turns is described in (Dijkstra, 1965). In fact the original idea came from a Dutch mathematician (T. Dekker). This was the first time the mutual exclusion problem had been solved using a software solution. (Peterson, 1981), came up with a much simpler solution.

P0	P1
<pre> //global variables int turn; //0 or 1 boolean flag[2]; //interested to enter C.S. i.e critical section </pre>	
<pre> while(1) { flag[0] = TRUE;//P0 interested in C.S. turn = 1; while (turn==1 && flag[1]==True) ; //wait Critical_Section(); flag[0]=FALSE; Remainder_Section(); } </pre>	<pre> while(1) { flag[1] = TRUE;//P0 interested in C.S. turn = 0; while (turn==1 && flag[1]==True) ; //wait Critical_Section(); flag[1]=FALSE; Remainder_Section(); } </pre>

v. Test and Set Lock

If we are given assistance by the instruction set of the processor we can implement a solution to the mutual exclusion problem. The instruction we require is called test and set lock (TSL). This instructions reads the contents of a memory location, stores it in a register and then stores a non-zero value at the address. This operation is guaranteed to be indivisible. That is, no other process can access that memory location until the TSL instruction has finished.

This assembly (like) code shows how we can make use of the TSL instruction to solve the mutual exclusion problem.

```
enter_region:
    tsl register, flag ; copy flag to register and set flag to 1
    cmp register, #0 ; was flag zero?
    jnz enter_region ; if flag was non zero, lock was set , so loop
    ret ; return (and enter critical region)
```

```
leave_region:
    mov flag, #0 ; store zero in flag
    ret ; return
```

Assume, again, two processes.

Process 0 calls enter_region. The tsl instruction copies the flag to a register and sets it to a non-zero value. The flag is now compared to zero (cmp - compare) and if found to be non-zero (jnz - jump if non-zero) the routine loops back to the top. Only when process 1 has set the flag to zero (or under initial conditions), by calling leave_region, will process 0 be allowed to continue.

2.2.3 Sleep and Wakeup

Processes waiting to enter their critical sections waste processor time checking to see if they can proceed. A better solution to the mutual exclusion problem, which can be implemented with the addition of some new primitives, would be to block processes when they are denied access to their critical sections. Two primitives, Sleep and Wakeup, are often used to implement blocking in mutual exclusion.

Sleep:

When a process is not permitted to access its critical section, it uses a system call known as Sleep, which causes that process to block

Wakeup:

It is a system call that wakes up the process. The process will not be scheduled to run again, until another process uses the Wakeup system call. In most cases, Wakeup is called by a process when it leaves its critical section if any other processes have blocked.

2.2.5 Semaphores

Dijkstra proposed a significant technique for managing concurrent processes for complex mutual exclusion problems. He introduced a new synchronization tool called Semaphore.

A semaphore is hardware or a software tag variable whose value indicates the status of a common resource. Its purpose is to lock the resource being used. A process which needs the resource will check the semaphore for determining the status of the resource followed by the decision for proceeding. In multitasking operating systems, the activities are synchronized by using the semaphore techniques.

Semaphores are of two types –

1. **Binary semaphore:** Binary semaphore can take the value 0 & 1 only. Binary semaphores have 2 methods associated with it. (up, down / lock, unlock). Binary semaphores can take only 2 values (0/1). They are used to acquire locks. When a resource is available, the process in charge set the semaphore to 1 else 0.
2. **Counting semaphore:** Counting semaphore can take nonnegative integer values and can be used to allocate resources from a pool of identical resources

Two standard operations, **wait** and **signal** are defined on the semaphore. Entry to the critical section is controlled by the *wait* operation and exit from a critical region is taken care by *signal* operation. The **wait**, & **signal** operations are also called P and V operations. The manipulation of semaphore (S) takes place as following:

1. The wait command P(S) decrements the semaphore value by 1. If the resulting value becomes negative then P command is delayed until the condition is satisfied.
2. The V(S) i.e. signals operation increments the semaphore value by 1.

Mutual exclusion on the semaphore is enforced within P(S) and V(S). If a number of processes attempt P(S) simultaneously, only one process will be allowed to proceed & the other processes will be waiting. These operations are defined as under –

```
wait(S){
while (S<=0); // No operation i.e. block the calling process (i.e. wait on S)
S--;
}

signal(S)
{
if any processes are waiting on S
    Start one of these processes
else
    Set S to S+1
```

The semaphore operation are implemented as operating system services and so wait and signal are atomic in nature i.e. once started, execution of these operations cannot be interrupted.

Thus semaphore is a simple yet powerful mechanism to ensure mutual exclusion among concurrent processes.

2.2.6 Monitors

Monitors were developed in the 1970s to make it easier to avoid deadlocks.

- A monitor is a collection of procedures, variables, and data structures grouped together.
- Processes can call the monitor procedures but cannot access the internal data structures.
- Only one process at a time may be be **active** in a monitor.
Active in a monitor means in ready queue or CPU with the program counter somewhere in a monitor method.
- A monitor is a language construct.

- The compiler usually enforces mutual exclusion.
- Condition variables allow for blocking and unblocking.
 - ✓ `cv.wait()` blocks a process.
The process is said to be waiting for (or waiting on) the condition variable `cv`.
 - ✓ `cv.signal()` (also called `cv.notify()`) unblocks a process waiting for the condition variable `cv`.

Monitor Implementation

Monitors are implemented by using queues to keep track of the processes attempting to become active in the monitor.

To be active, a monitor must obtain a **lock** to allow it to execute the monitor code. Processes that are blocked are put in a queue of processes waiting for an unblocking event to occur.

These are the queues that might be used:

- **The entry queue** contains processes attempting to call a monitor procedure from outside the monitor.
Each monitor has one entry queue.
- **The signaller queue** contains processes that have executed a notify operation.
Each monitor has at most one signaller queue. In some implementations, a notify leaves the process active and no signaller queue is needed.
- **The waiting queue** contains processes that have been awakened by a notify operation. Each monitor has one waiting queue.
- **Condition variable queues** contain processes that have executed a condition variable wait operation. There is one such queue for each condition variable.

2.2.6 Message Passing

Message passing is a method of interprocess communication that uses two primitives, send and receive, which, like semaphores and unlike monitors, are system calls rather than language constructs. They can easily be put into library procedures, such as

```
send(destination, &message);  
and  
receive(source, &message);
```

The former call sends a message to a given destination and the latter one receives a message from a given source (or from *ANY*, if the receiver does not care). If no message is available, the receiver can block until one arrives. Alternatively, it can return immediately with an error code.

Design Issues for Message Passing Systems

Message passing systems have many challenging problems and design issues that do not arise with semaphores or monitors, especially if the communicating processes are on different machines connected by a network. For example, messages can be lost by the network. To guard against lost messages, the sender and receiver can agree that as soon as a message has been received, the receiver will send back a special **acknowledgement** message. If the sender has not received the acknowledgement within a certain time interval, it retransmits the message.

Now consider what happens if the message itself is received correctly, but the acknowledgement is lost. The sender will retransmit the message, so the receiver will get it twice. It is essential that the receiver be able to distinguish a new message from the retransmission of an old one. Usually, this problem is solved by putting consecutive sequence numbers in each original message. If the receiver gets a message bearing the same sequence number as the previous message, it knows that the message is a duplicate that can be ignored.

Message systems also have to deal with the question of how processes are named, so that the process specified in a send or receive call is unambiguous. **Authentication** is also an issue in message systems: how can the client tell that he is communicating with the real file server, and not with an imposter?

2.3 Classical IPC Problems:

Following are some of the classical problem faced while process synchronization in systems where cooperating processes are present.

Bounded Buffer Problem

- This problem is generalized in terms of the Producer-Consumer problem.
- Solution to this problem is, creating two counting semaphores "full" and "empty" to keep track of the current number of full and empty buffers respectively.


```

/* program boundedbuffer */
const int sizeofbuffer = /* buffer size */;
semaphore s = 1, n = 0, e = sizeofbuffer;
void producer()
{
    while (true) {
        produce();
        semWait(e);
        semWait(s);
        append();
        semSignal(s);
        semSignal(n);
    }
}
void consumer()
{
    while (true) {
        semWait(n);
        semWait(s);
        take();
        semSignal(s);
        semSignal(e);
        consume();
    }
}
void main()
{
    parbegin (producer, consumer);
}

```

A Solution to the Bounded-Buffer Producer/Consumer Problem Using Semaphores

The producer and consumer functions can be expressed as follows (variable *in* and *out* are initialized to 0 and *n* is the size of the buffer):

<pre> producer: while (true) { /* produce item v */ while ((in + 1) % n == out) /* do nothing */; b[in] = v; in = (in + 1) % n; } </pre>	<pre> consumer: while (true) { while (in == out) /* do nothing */; w = b[out]; out = (out + 1) % n; /* consume item w */; } </pre>
--	--

The Readers Writers Problem

- In this problem there are some processes(called readers) that only read the shared data, and never change it, and there are other processes(called writers) who may change the data in addition to reading or instead of reading it.
- There are various type of the readers-writers problem, most centred on relative priorities of readers and writers

The dining philosophers problem is useful for modeling processes that are competing for exclusive access to a limited number of resources, such as I/O devices. Another famous problem is the readers and writers problem which models access to a database. For example, an airline reservation system, with many competing processes wishing to read and write it. It is acceptable to have multiple processes reading the database at the same time, but if one process is updating (writing) the database, no other process may have access to the database, not even a reader.

Solution to Readers Writer problems

```
typedef int semaphore;           /* use your imagination */
semaphore mutex = 1;            /* controls access to 'rc' */
semaphore db = 1;               /* controls access to the database */
int rc = 0;                     /* # of processes reading or wanting to */
void reader(void)
{
    while (TRUE){               /* repeat forever */
        down(&mutex);           /* get exclusive access to 'rc' */
        rc = rc + 1;            /* one reader more now */
        if (rc == 1) down(&db); /* if this is the first reader ... */
        up(&mutex);             /* release exclusive access to 'rc' */
        read_data_base();       /* access the data */
        down(&mutex);           /* get exclusive access to 'rc' */
        rc = rc - 1;           /* one reader fewer now */
        if (rc == 0) up(&db);   /* if this is the last reader ... */
        up(&mutex);             /* release exclusive access to 'rc' */
        use_data_read();        /* noncritical region */
    }
}

void writer(void)
{
    while (TRUE){               /* repeat forever */
        think_up_data();        /* noncritical region */
        down(&db);             /* get exclusive access */
        write_data_base();      /* update the data */
        up(&db);               /* release exclusive access */
    }
}
```

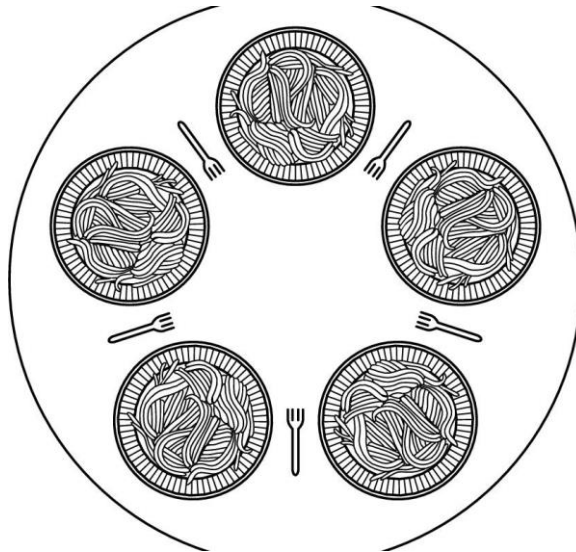
In this solution, the first reader to get access to the data base does a down on the semaphore db.

Subsequent readers merely have to increment a counter, rc. As readers leave, they decrement the counter and the last one out does an up on the semaphore, allowing a blocked writer, if there is one, to get in.

Dining Philosophers Problem

There are N philosophers sitting around a circular table eating noodles and discussing philosophy. The problem is that each philosopher needs 2 forks to eat, and there are only N forks, one between each 2 philosophers. It is required to design an algorithm that the philosophers can follow that insures that none starves as long as each philosopher eventually stops eating, and such that the maximum number of philosophers can eat at once.

- Philosophers eat/think
- Eating needs 2 forks
- Pick one fork at a time
- How to prevent deadlock?



One idea is to instruct each philosopher to behave as follows:

- think until the left fork is available; when it is, pick it up
- think until the right fork is available; when it is, pick it up
- eat
- put the left fork down

- put the right fork down
- repeat from the start

This solution is incorrect: it allows the system to reach deadlock. Suppose that all five philosophers take their left forks simultaneously. None will be able to take their right forks, and there will be a deadlock.

The solution presented below is deadlock-free and allows the maximum parallelism for an arbitrary number of philosophers. It uses an array, state, to keep track of whether a philosopher is eating, thinking, or hungry (trying to acquire forks). A philosopher may move into eating state only if neither neighbor is eating. Philosopher *i*'s neighbors are defined by the macros LEFT and RIGHT. In other words, if *i* is 2, LEFT is 1 and RIGHT is 3.

Solution:

```
#define N      5      /* number of philosophers */
#define LEFT   (i+N-1)%N /* number of i's left neighbor */
#define RIGHT  (i+1)%N /* number of i's right neighbor */
#define THINKING 0      /* philosopher is thinking */
#define HUNGRY  1      /* philosopher is trying to get forks */
#define EATING   2      /* philosopher is eating */
typedef int semaphore; /* semaphores are a special kind of int */
int state[N];          /* array to keep track of everyone's state */
semaphore mutex = 1;   /* mutual exclusion for critical regions */
semaphore s[N];        /* one semaphore per philosopher */
void philosopher(int i) /* i: philosopher number, from 0 to N1 */
{
    while (TRUE){      /* repeat forever */
        think();       /* philosopher is thinking */
        take_forks(i); /* acquire two forks or block */
        eat();          /* yum-yum, spaghetti */
        put_forks(i);  /* put both forks back on table */
    }
}
```

```

void take_forks(int i)          /* i: philosopher number, from 0 to N1 */
{
    down(&mutex);               /* enter critical region */
    state[i] = HUNGRY;          /* record fact that philosopher i is hungry */
    test(i);                   /* try to acquire 2 forks */
    up(&mutex);                 /* exit critical region */
    down(&s[i]);                /* block if forks were not acquired */
}

void put_forks(i)              /* i: philosopher number, from 0 to N1 */
{
    down(&mutex);               /* enter critical region */
    state[i] = THINKING;        /* philosopher has finished eating */

    test(LEFT);                /* see if left neighbor can now eat */
    test(RIGHT);               /* see if right neighbor can now eat */
    up(&mutex);                 /* exit critical region */
}

void test(i)                   /* i: philosopher number, from 0 to N1 */
{
    if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
        state[i] = EATING;
        up(&s[i]);
    }
}

```

2.4 Process Scheduling

Basic Concepts:

Principally, process scheduling means **CPU scheduling**. It is the basic function of a multiprogramming system. By switching the CPU between different processes, the operating system helps to increase the CPU utilization.

The idea of multiprogramming is quite simple. Several processes are kept in memory at time. When one process has to wait, the operating system takes the CPU from that process, and gives it to another process. Assume we have two processes A and B to be executed as shown in Figure.

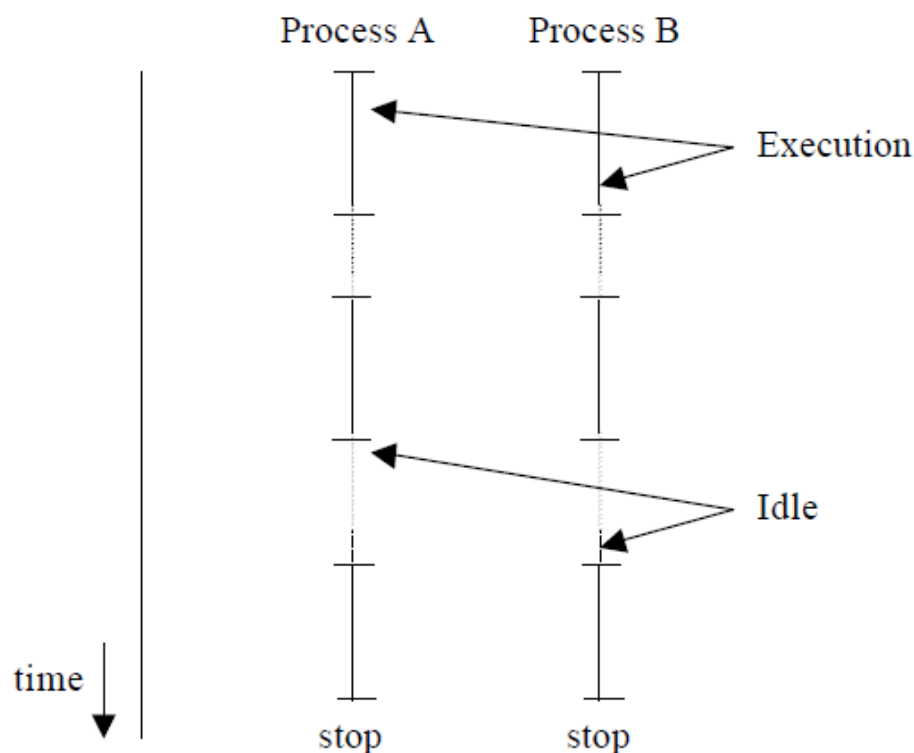


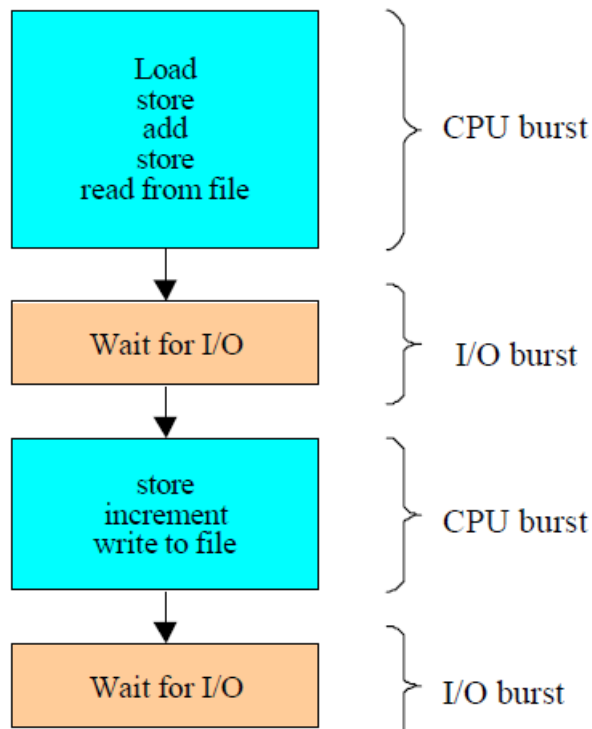
Figure: Two processes A and B for execution

Each process executes for one second, then waits for one second. It is repeated 60 times. If we run process A first and then process B, one after the other, it will take four minutes to run the two processes: A takes two minutes to run and then B takes two minutes to run. We actually compute only two minutes of this time. The other two minutes are idle time. Our CPU utilization is only 50%.

If we multi-program A and B, we can increase the CPU utilization considerably. Let us start with process A first, which executes for one second. Then, while A waits for one minute, we start process B to execute. When process B waits, A is ready to run again and so on. In that manner, we only need two

minutes to execute two processes A and B, and there is no idle time. The CPU utilization is increased from 50% to 100%.

The CPU scheduling is based on the following observed property of process: the process execution is a cycle of CPU execution and I/O wait. So process executions alternate between these two states. Normally, process execution begins with a CPU burst. It is followed by an I/O burst, which is followed by another CPU burst, and so on as shown in figure below:



CPU Scheduling Criteria:

The CPU scheduling criteria are given as:

i. CPU utilization

To make out the best use of CPU and not to waste any CPU cycle, CPU would be working most of the time (Ideally 100% of the time). Considering a real system, CPU usage should range from 40% (lightly loaded) to 90% (heavily loaded.)

ii. Throughput

It is the total number of processes completed per unit time or rather say total amount of work done in a unit of time. This may range from 10/second to 1/hour depending on the specific processes.

iii. Turnaround time

It is the amount of time taken to execute a particular process, i.e. The interval from time of submission of the process to the time of completion of the process(Wall clock time).

iv. Waiting time

The sum of the periods spent waiting in the ready queue amount of time a process has been waiting in the ready queue to acquire get control on the CPU.

v. Response time

Amount of time it takes from when a request was submitted until the first response is produced. Remember, it is the time till the first response and not the completion of process execution (final response).

In general CPU utilization and Throughput are maximized and other factors are reduced for proper optimization.

Schedulers:

Schedulers are special system software which handles 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:

i. Long Term Scheduler:

Long Term Scheduler It is also called **job scheduler**. Long term scheduler determines which programs are admitted to the system for processing. Job scheduler selects processes from the queue and loads them into memory for execution. The primary objective of the job scheduler is to provide a balanced mix of jobs, such as I/O bound and processor bound.

ii. Short Term Scheduler:

It is also called **CPU scheduler**. Main objective is increasing 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 process among the processes that are ready to execute and allocates CPU to one of them.

Short term scheduler also known as **dispatcher**, execute most frequently and makes the fine grained decision of which process to execute next. Short term scheduler is faster than long term scheduler.

iii. Medium Term Scheduler

Medium term scheduling is part of the 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.

Comparison between 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 in time sharing system	It is also minimal in time sharing system	It is a part of Time sharing 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 context switch saves the content of all processor registers for the process being removed from the CPU, in its process descriptor. The context of a process is represented in the process control block of a process. Context switch time is pure overhead. Context switching can significantly affect performance as modern computers have a lot of general and status registers to be saved.

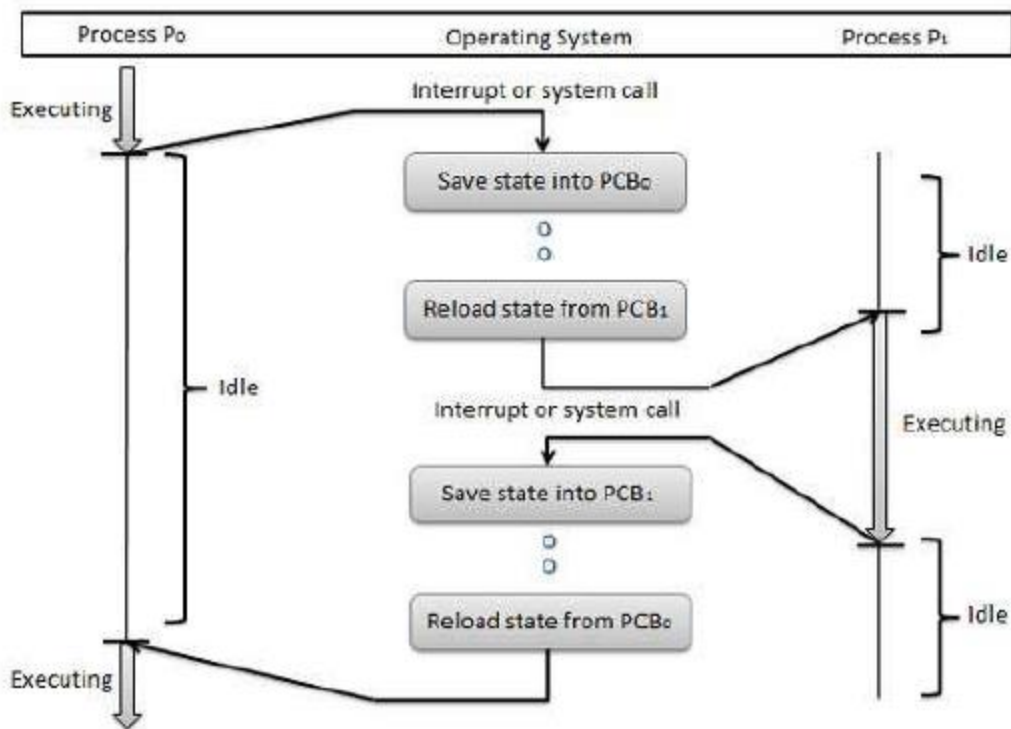


Figure: Context switch

Preemptive VS Non-Preemptive Scheduling

BASIS FOR COMPARISON	PREEMPTIVE SCHEDULING	NON PREEMPTIVE SCHEDULING
Basic	The resources are allocated to a process for a limited time.	Once resources are allocated to a process, the process holds it till it completes its burst time or switches to waiting state.

BASIS FOR COMPARISON	PREEMPTIVE SCHEDULING	NON PREEMPTIVE SCHEDULING
Interrupt	Process can be interrupted in between.	Process can not be interrupted till it terminates or switches to waiting state.
Starvation	If a high priority process frequently arrives in the ready queue, low priority process may starve.	If a process with long burst time is running CPU, then another process with less CPU burst time may starve.
Overhead	Preemptive scheduling has overheads of scheduling the processes.	Non-preemptive scheduling does not have overheads.
Flexibility	Preemptive scheduling is flexible.	Non-preemptive scheduling is rigid.

CPU Scheduling Algorithms:


CPU scheduling algorithms are explained below :

1. First Come First Serve(FCFS) Scheduling
 2. Shortest-Job-First(SJF) Scheduling
 3. Priority Scheduling
 4. Round Robin(RR) Scheduling
 5. Multilevel Queue Scheduling
-

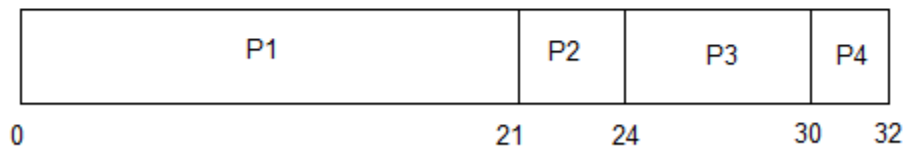
First Come First Serve(FCFS) Scheduling

- Jobs are executed on first come, first serve basis.
- Easy to understand and implement.
- Poor in performance as average wait time is high.

PROCESS	BURST TIME
P1	21
P2	3
P3	6
P4	2



The average waiting time will be = $(0 + 21 + 24 + 30) / 4 = \underline{18.75 \text{ ms}}$



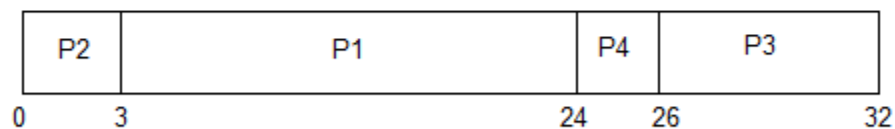
This is the GANTT chart for the above processes

Priority Scheduling

- Priority is assigned for each process.
- Process with highest priority is executed first and so on.
- Processes with same priority are executed in FCFS manner.
- Priority can be decided based on memory requirements, time requirements or any other resource requirement.

PROCESS	BURST TIME	PRIORITY
P1	21	2
P2	3	1
P3	6	4
P4	2	3

The GANTT chart for following processes based on Priority scheduling will be,



The average waiting time will be, $(0 + 3 + 24 + 26) / 4 = \underline{13.25 \text{ ms}}$

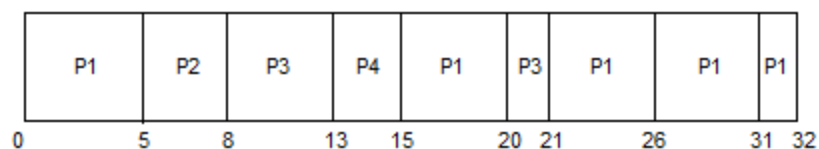
Round Robin(RR) Scheduling

- A fixed time is allotted to each process, called **quantum**, for execution.
- Once a process is executed for given time period that process is preempted and other process executes for given time period.
- Context switching is used to save states of preempted processes.

PROCESS	BURST TIME
P1	21
P2	3
P3	6
P4	2



The GANTT chart for round robin scheduling will be,



The average waiting time will be, 11 ms.

Shortest-Job-First(SJF) Scheduling

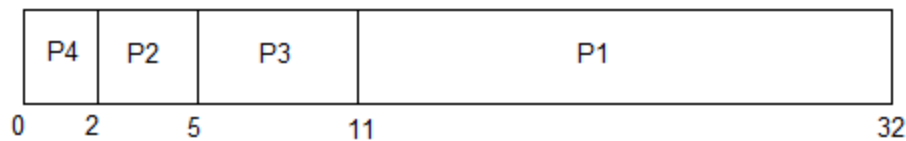
- Best approach to minimize waiting time.
- Actual time taken by the process is already known to processor.
- Impossible to implement.

PROCESS	BURST TIME
P1	21
P2	3
P3	6
P4	2



In Shortest Job First Scheduling, the shortest Process is executed first.

Hence the GANTT chart will be following :

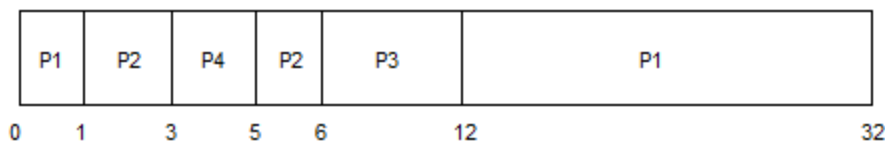


Now, the average waiting time will be = $(0 + 2 + 5 + 11)/4 = 4.5$ ms

In Preemptive Shortest Job First Scheduling, jobs are put into ready queue as they arrive, but as a process with short burst time arrives, the existing process is preempted.

PROCESS	BURST TIME	ARRIVAL TIME
P1	21	0
P2	3	1
P3	6	2
P4	2	3

The GANTT chart for Preemptive Shortest Job First Scheduling will be,



The average waiting time will be, $((5-3) + (6-2) + (12-1))/4 = \underline{4.25 \text{ ms}}$

The average waiting time for preemptive shortest job first scheduling is less than both, non-preemptive SJF scheduling and FCFS scheduling.

Real-Time CPU Scheduling

A real-time system is one in which time plays an essential role. Typically, one or more physical devices external to the computer generate stimuli, and the computer must react appropriately to them within a fixed amount of time.

Real-time systems are generally categorized as **hard real time**, meaning there are absolute deadlines that must be met, or else, and **soft real time**, meaning that missing an occasional deadline is undesirable, but nevertheless tolerable. In both cases, real-time behavior is achieved by dividing the program into a number of processes, each of whose behavior is predictable and known in advance. These processes are generally short lived and can run to completion in well under a second. When an external event is detected, it is the job of the scheduler to schedule the processes in such a way that all deadlines are met.

The events that a real-time system may have to respond to can be further categorized as periodic (occurring at regular intervals) or aperiodic (occurring unpredictably). A system may have to respond to multiple periodic event streams. Depending on how much time each event requires for processing, it may not even be possible to handle them all.

Two Level Scheduling:

Two-level scheduling is a process scheduling that involves swapped out processes.

Consider this problem: A system contains 50 running processes all with equal priority. However, the system's memory can only hold 10 processes in memory simultaneously. Therefore, there will always be 40 processes swapped out written on virtual memory on the hard disk. The time taken to swap out and swap in a process is 50 ms respectively.

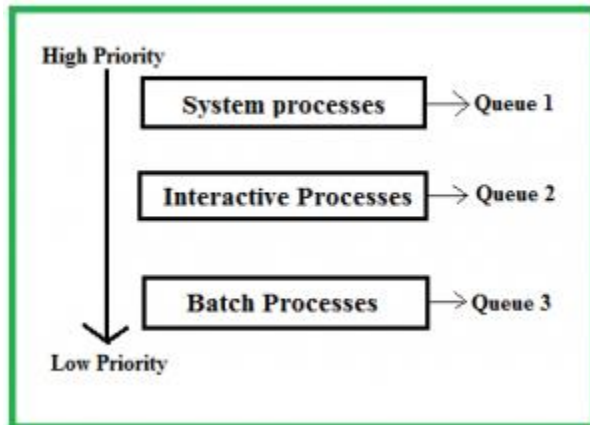
With straightforward Round-robin scheduling, every time a context switch occurs, a process would need to be swapped in (because only the 10 least recently used processes are swapped in). Choosing randomly among the processes would diminish the probability to 80% (40/50). If that occurs, then obviously a process also need to be swapped out. Swapping in and out of is costly, and the scheduler would waste much of its time doing unneeded swaps.

That is where **two-level scheduling** enters the picture. It uses two different schedulers, one lower-level scheduler which can only select among those processes in memory to run. That scheduler could be a Round-robin scheduler. The other scheduler is the higher-level scheduler whose only concern is to swap in and swap out processes from memory. It does its scheduling much less often than the lower-level scheduler since swapping takes so much time.

Thus, the higher-level scheduler selects among those processes in memory that have run for a long time and swaps them out. They are replaced with processes on disk that have not run for a long time.

Multiple Queues (Multilevel Queue) Scheduling

Ready Queue is divided into separate queues for each class of processes. For example, let us take three different types of process System processes, Interactive processes and Batch Processes. All three process have there own queue as shown in the figure below:



All three different type of processes have there own queue. Each queue have its own Scheduling algorithm. For example, queue 1 and queue 2 uses **Round Robin** while queue 3 can use **FCFS** to schedule there processes.

Scheduling among the queues : What will happen if all the queues have some processes? Which process should get the cpu? To determine this Scheduling among the queues is necessary. There are two ways to do so –

1. **Fixed priority preemptive scheduling method** – Each queue has absolute priority over lower priority queue. Let us consider following priority order **queue 1 > queue 2 > queue 3**. According to this algorithm no process in the batch queue(queue 3) can run unless queue 1 and 2 are empty. If any batch process (queue 3) is running and any system (queue 1) or Interactive process(queue 2) entered the ready queue the batch process is preempted.
2. **Time slicing** – In this method each queue gets certain portion of CPU time and can use it to schedule its own processes. For instance, queue 1 takes 50 percent of CPU time queue 2 takes 30 percent and queue 3 gets 20 percent of CPU time.

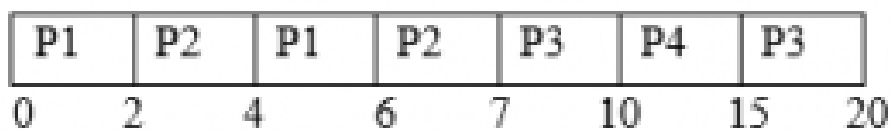
Example:

Consider below table of four processes under Multilevel queue scheduling. Queue number denotes the queue of the process.

Process	Arrival time	CPU Burst Time	Queue number
P1	0	4	1
P2	0	3	1
P3	0	8	2
P4	10	5	1

Priority of queue 1 is greater than queue 2. queue 1 uses Round Robin (Time Quantum = 2) and queue 2 uses FCFS.

Below is the Gantt chart of the problem:



At starting both queues have process so process in queue 1 (P1, P2) runs first (because of higher priority) in the round robin fashion and completes after 7 units then process in queue 2 (P3) starts running (as there is no process in queue 1) but while it is running P4 comes in queue 1 and interrupts P3 and start running for 5 second and after its completion P3 takes the CPU and completes its execution.

Comparison of Scheduling Algorithms

By now, you must have understood how CPU can apply different scheduling algorithms to schedule processes. Now, let us examine the advantages and disadvantages of each scheduling algorithm.

First Come First Serve (FCFS)

Advantages:

- FCFS algorithm doesn't include any complex logic, it just puts the process requests in a queue and executes it one by one.
- Hence, FCFS is pretty simple and easy to implement.
- Eventually, every process will get a chance to run, so starvation doesn't occur.

Disadvantages:

- There is no option for pre-emption of a process. If a process is started, then CPU executes the process until it ends.
 - Because there is no pre-emption, if a process executes for a long time, the processes in the back of the queue will have to wait for a long time before they get a chance to be executed.
-

Shortest Job First (SJF)

Advantages:

- According to the definition, short processes are executed first and then followed by longer processes.
- The throughput is increased because more processes can be executed in less amount of time.

Disadvantages:

- The time taken by a process must be known by the CPU beforehand, which is not possible.
- Longer processes will have more waiting time, eventually they'll suffer starvation.

Note: Preemptive Shortest Job First scheduling will have the same advantages and disadvantages as those for SJF.

Round Robin (RR)

Advantages:

- Each process is served by the CPU for a fixed time quantum, so all processes are given the same priority.
- Starvation doesn't occur because for each round robin cycle, every process is given a fixed time to execute. No process is left behind.

Disadvantages:

- The throughput in RR largely depends on the choice of the length of the time quantum. If time quantum is longer than needed, it tends to exhibit the same behavior as FCFS.
 - If time quantum is shorter than needed, the number of times that CPU switches from one process to another process, increases. This leads to decrease in CPU efficiency.
-

Priority based Scheduling**Advantages:**

- The priority of a process can be selected based on memory requirement, time requirement or user preference. For example, a high end game will have better graphics, that means the process which updates the screen in a game will have higher priority so as to achieve better graphics performance.

Disadvantages:

- A second scheduling algorithm is required to schedule the processes which have same priority.

- In preemptive priority scheduling, a higher priority process can execute ahead of an already executing lower priority process. If lower priority process keeps waiting for higher priority processes, starvation occurs.

Chapter-3: Input/output

3.1 PRINCIPLES OF I/O HARDWARE

One of the important jobs of an Operating System is to manage various I/O devices including mouse, keyboards, touch pad, disk drives, display adapters, USB devices, Bit-mapped screen, LED, Analog-to-digital converter, On/off switch, network connections, audio I/O, printers etc.

3.1.1 I/O Devices

An I/O system is required to take an application I/O request and send it to the physical device, then take whatever response comes back from the device and send it to the application. I/O devices can be divided into two categories –

- **Block devices** – A block device is one with which the driver communicates by sending entire blocks of data. Common block sizes range from 512 bytes to 32,768 bytes. For example, Hard disks, USB cameras, Disk-On-Key etc.

- **Character devices** – A character device is one with which the driver communicates by sending and receiving single characters (bytes, octets). For example, serial ports, parallel ports, sounds cards etc

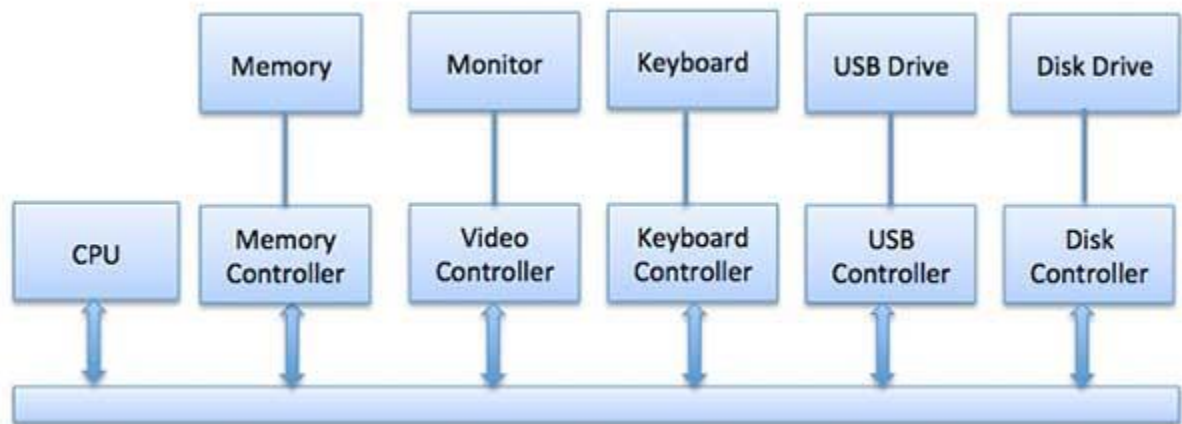
3.1.2 Device Controllers

Device drivers are software modules that can be plugged into an OS to handle a particular device. Operating System takes help from device drivers to handle all I/O devices.

The Device Controller works like an interface between a device and a device driver. I/O units (Keyboard, mouse, printer, etc.) typically consist of a mechanical component and an electronic component where electronic component is called the device controller.

There is always a device controller and a device driver for each device to communicate with the Operating Systems. A device controller may be able to handle multiple devices. As an interface its main task is to convert serial bit stream to block of bytes, perform error correction as necessary.

Any device connected to the computer is connected by a plug and socket, and the socket is connected to a device controller. Following is a model for connecting the CPU, memory, controllers, and I/O devices where CPU and device controllers all use a common bus for communication.



3.1.3 Synchronous vs asynchronous I/O

- **Synchronous I/O** – In this scheme CPU execution waits while I/O proceeds

- **Asynchronous I/O** – I/O proceeds concurrently with CPU execution

3.1.4 Communication to I/O Devices

The CPU must have a way to pass information to and from an I/O device. There are three approaches available to communicate with the CPU and Device.

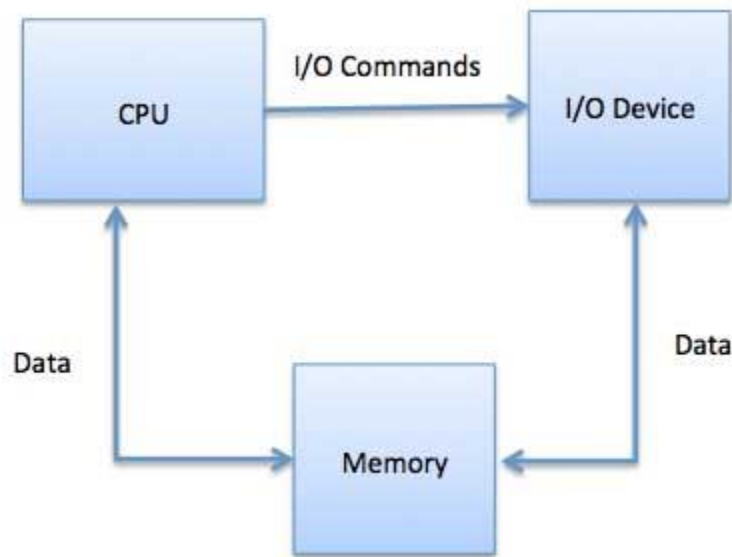
- Special Instruction I/O
- Memory-mapped I/O
- Direct memory access (DMA)

Special Instruction I/O

This uses CPU instructions that are specifically made for controlling I/O devices. These instructions typically allow data to be sent to an I/O device or read from an I/O device.

Memory-mapped I/O

When using memory-mapped I/O, the same address space is shared by memory and I/O devices. The device is connected directly to certain main memory locations so that I/O device can transfer block of data to/from memory without going through CPU.



While using memory mapped IO, OS allocates buffer in memory and informs I/O device to use that buffer to send data to the CPU. I/O device operates asynchronously with CPU, interrupts CPU when finished.

The advantage to this method is that every instruction which can access memory can be used to manipulate an I/O device. Memory mapped IO is used for most high-speed I/O devices like disks, communication interfaces.

3.1.5 Direct Memory Access (DMA)

Slow devices like keyboards will generate an interrupt to the main CPU after each byte is transferred. If a fast device such as a disk generated an interrupt for each byte, the operating system would spend most of its time handling these interrupts. So a typical computer uses direct memory access (DMA) hardware to reduce this overhead.

Direct Memory Access (DMA) means CPU grants I/O module authority to read from or write to memory without involvement. DMA module itself controls exchange of data between main memory and the I/O device. CPU is only involved at the beginning and end of the transfer and interrupted only after entire block has been transferred.

Direct Memory Access needs a special hardware called DMA controller (DMAC) that manages the data transfers and arbitrates access to the system bus. The controllers are programmed with source and destination pointers (where to read/write the data), counters to track the number of transferred bytes, and settings, which includes I/O and memory types, interrupts and states for the CPU cycles.

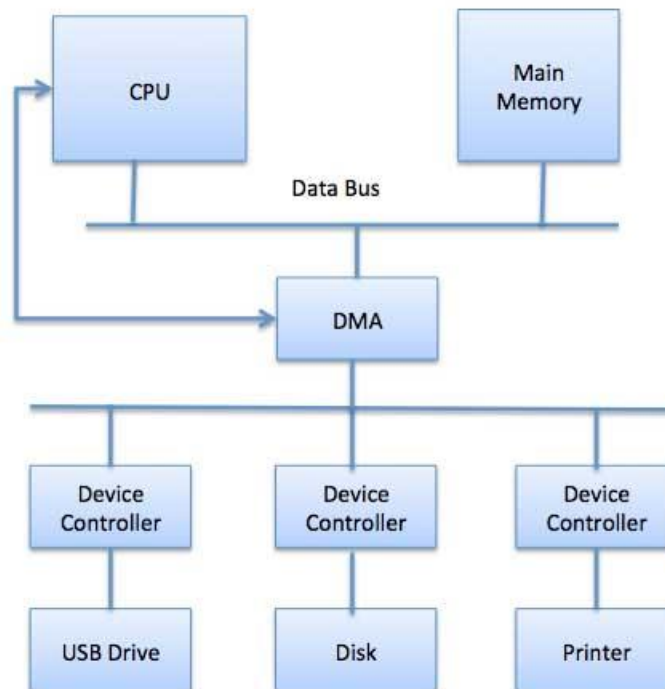


Figure: DMA

The operating system uses the DMA hardware as follows –

Step	Description
1	Device driver is instructed to transfer disk data to a buffer address X.
2	Device driver then instruct disk controller to transfer data to buffer.
3	Disk controller starts DMA transfer.
4	Disk controller sends each byte to DMA controller.
5	DMA controller transfers bytes to buffer, increases the memory address, decreases the counter C until C becomes zero.
6	When C becomes zero, DMA interrupts CPU to signal transfer completion.

3.1.6 Polling vs Interrupts I/O

A computer must have a way of detecting the arrival of any type of input. There are two ways that this can happen, known as **polling** and **interrupts**. Both of these techniques allow the processor to deal with events that can happen at any time and that are not related to the process it is currently running.

Polling I/O

Polling is the simplest way for an I/O device to communicate with the processor the processor. The process of periodically checking status of the device to see if it is time for the next I/O operation, is called polling. The I/O device simply puts the information in a Status register, and the processor must come and get the information.

Most of the time, devices will not require attention and when one does it will have to wait until it is next interrogated by the polling program. This is an inefficient method and much of the processors time is wasted on unnecessary polls.

Compare this method to a teacher continually asking every student in a class, one after another, if they need help. Obviously the more efficient method would be for a student to inform the teacher whenever they require assistance.

Interrupts I/O

An alternative scheme for dealing with I/O is the interrupt-driven method. An interrupt is a signal to the microprocessor from a device that requires attention.

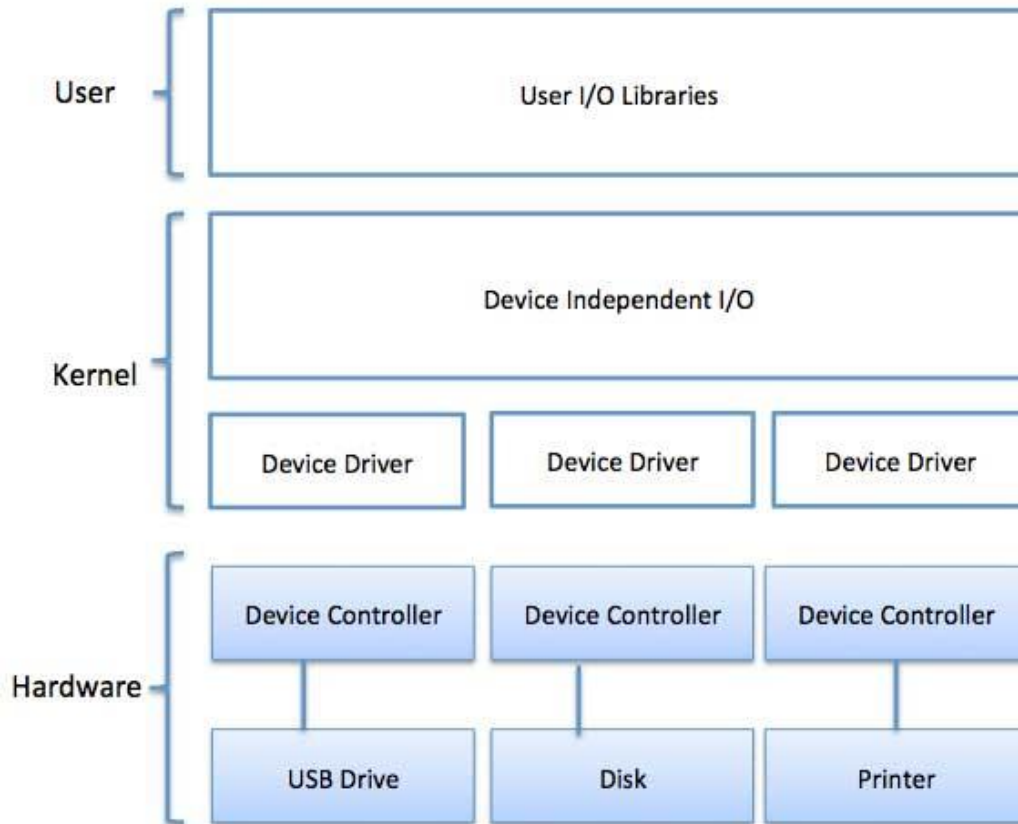
A device controller puts an interrupt signal on the bus when it needs CPU's attention when CPU receives an interrupt, It saves its current state and invokes the appropriate interrupt handler using the interrupt vector (addresses of OS routines to handle various events). When the interrupting device has been dealt with, the CPU continues with its original task as if it had never been interrupted.

3.2 Principles of I/O software

I/O software is often organized in the following layers:

- **User Level Libraries** – This provides simple interface to the user program to perform input and output. For example, **stdio** is a library provided by C and C++ programming languages.
- **Kernel Level Modules** – This provides device driver to interact with the device controller and device independent I/O modules used by the device drivers.
- **Hardware** – This layer includes actual hardware and hardware controller which interact with the device drivers and makes hardware alive.

A key concept in the design of I/O software is that it should be **device independent** where it should be possible to write programs that can access any I/O device without having to specify the device in advance. For example, a program that reads a file as input should be able to read a file on a floppy disk, on a hard disk, or on a CD-ROM, without having to modify the program for each different device.



3.2.1 Device Drivers

Device drivers are software modules that can be plugged into an OS to handle a particular device. Operating System takes help from device drivers to handle all I/O devices. Device drivers encapsulate device-dependent code and implement a standard interface in such a way that code contains device-specific register reads/writes. Device driver, is generally written by the device's manufacturer and delivered along with the device on a CD-ROM.

A device driver performs the following jobs –

- To accept request from the device independent software above to it.
- Interact with the device controller to take and give I/O and perform required error handling
- Making sure that the request is executed successfully

How a device driver handles a request is as follows: Suppose a request comes to read a block N. If the driver is idle at the time a request arrives, it starts carrying out the request immediately.

Otherwise, if the driver is already busy with some other request, it places the new request in the queue of pending requests.

3.2.2 Interrupt handlers

An interrupt handler, also known as an interrupt service routine or ISR, is a piece of software or more specifically a callback function in an operating system or more specifically in a device driver, whose execution is triggered by the reception of an interrupt.

When the interrupt happens, the interrupt procedure does whatever it has to in order to handle the interrupt, updates data structures and wakes up process that was waiting for an interrupt to happen.

The interrupt mechanism accepts an address — a number that selects a specific interrupt handling routine/function from a small set. In most architectures, this address is an offset stored in a table called the interrupt vector table. This vector contains the memory addresses of specialized interrupt handlers.

3.2.3 Device-Independent I/O Software

The basic function of the device-independent software is to perform the I/O functions that are common to all devices and to provide a uniform interface to the user-level software. Though it is difficult to write completely device independent software but we can write some modules which are common among all the devices. Following is a list of functions of device-independent I/O Software:

- Uniform interfacing for device drivers
- Device naming - Mnemonic names mapped to Major and Minor device numbers
- Device protection
- Providing a device-independent block size
- Buffering because data coming off a device cannot be stored in final destination.
- Storage allocation on block devices
- Allocation and releasing dedicated devices
- Error Reporting

3.2.4 User-Space I/O Software

These are the libraries which provide richer and simplified interface to access the functionality of the kernel or ultimately interactive with the device drivers. Most of the user-level I/O

software consists of library procedures with some exception like spooling system which is a way of dealing with dedicated I/O devices in a multiprogramming system.

I/O Libraries (e.g., stdio) are in user-space to provide an interface to the OS resident device-independent I/O SW. For example putchar(), getchar(), printf() and scanf() are example of user level I/O library stdio available in C programming.

3.2.5 Kernel I/O Subsystem

Kernel I/O Subsystem is responsible to provide many services related to I/O. Following are some of the services provided.

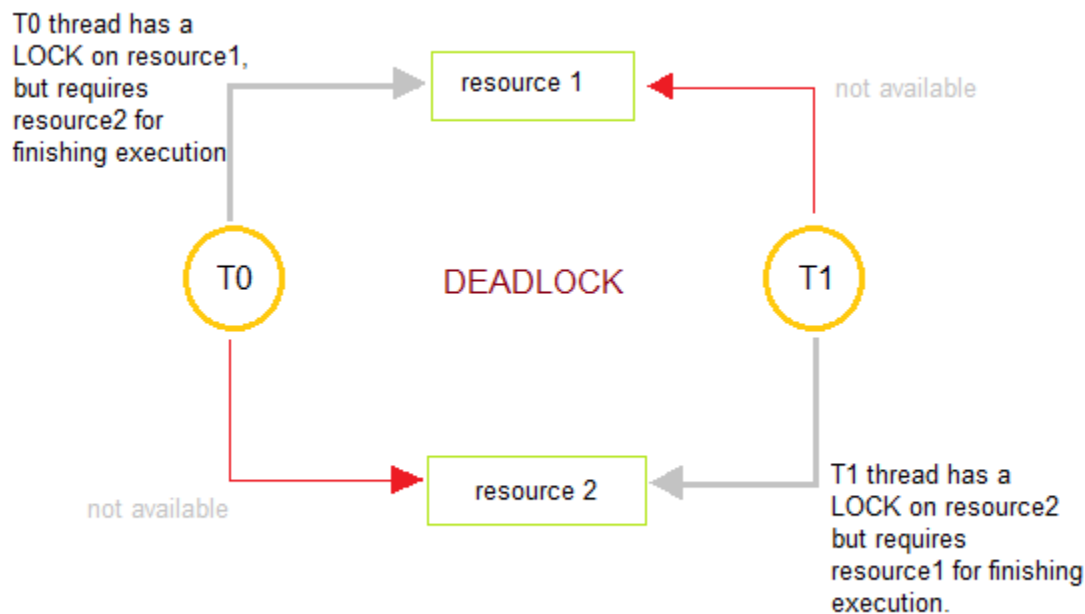
- **Scheduling** – Kernel schedules a set of I/O requests to determine a good order in which to execute them. When an application issues a blocking I/O system call, the request is placed on the queue for that device. The Kernel I/O scheduler rearranges the order of the queue to improve the overall system efficiency and the average response time experienced by the applications.
- **Buffering** – Kernel I/O Subsystem maintains a memory area known as **buffer** that stores data while they are transferred between two devices or between a device with an application operation. Buffering is done to cope with a speed mismatch between the producer and consumer of a data stream or to adapt between devices that have different data transfer sizes.
- **Caching** – Kernel maintains cache memory which is region of fast memory that holds copies of data. Access to the cached copy is more efficient than access to the original.
- **Spooling and Device Reservation** – A spool is a buffer that holds output for a device, such as a printer, that cannot accept interleaved data streams. The spooling system copies the queued spool files to the printer one at a time. In some operating systems, spooling is managed by a system daemon process. In other operating systems, it is handled by an in kernel thread.
- **Error Handling** – An operating system that uses protected memory can guard against many kinds of hardware and application errors.

Chapter- 4: DEADLOCKS

Computer systems are full of resources that can only be used by one process at a time. Common examples include printers, tape drives, and slots in the system's internal tables. Having two processes simultaneously writing to the printer leads to gibberish. Having two processes using the same file system table slot will invariably lead to a corrupted file system. Consequently, all operating systems have the ability to (temporarily) grant a process exclusive access to certain resources.

For many applications, a process needs exclusive access to not one resource, but several. Suppose, for example, two processes each want to record a scanned document on a CD. Process *A* requests permission to use the scanner and is granted it. Process *B* is programmed differently and requests the CD recorder first and is also granted it. Now *A* asks for the CD recorder, but the request is denied until *B* releases it. Unfortunately, instead of releasing the CD recorder *B* asks for the scanner. At this point both processes are blocked and will remain so forever. This situation is called a **deadlock**.

Deadlocks are a set of blocked processes each holding a resource and waiting to acquire a resource held by another process.



3.1 RESOURCES

Deadlocks can occur when processes have been granted exclusive access to devices, files, and so forth. A resource can be a hardware device (e.g., a tape drive) or a piece of information (e.g., a locked record in a database). The **resource** is the object granted to a process. A computer can have many different resources that can be acquired. For some resources, several identical instances may be available, such as three printers. When several copies of a resource are available, any one of them can be used to satisfy any request for the resource. In short, a resource is anything that can be used by only a single process at any instant of time.

Resources come in two types: **preemptable** and **nonpreemptable**. A **preemptable** resource is one that can be taken away from the process owning it with no ill effects. Memory is an example of a preemptable resource. A **nonpreemptable resource**, in contrast, is one that cannot be taken away from its current owner without causing the computation to fail. If a process has begun to burn a CD-ROM, suddenly taking the CD recorder away from it and giving it to another process will result in a garbled CD, CD recorders are not preemptable at an arbitrary moment.

The sequence of events required to use a resource is given below:

1. Request the resource.
2. Use the resource.
3. Release the resource.

3.2 Conditions (i.e. principle) for Deadlock

Deadlock can arise if following four conditions hold simultaneously:

- ✓ **Mutual exclusion:** Only one process at a time can use a resource.
- ✓ **Hold and wait:** A process holding at least one resource isv waiting to acquire additional resources held by other processes.
- ✓ **No preemption:** A resource can be released onlyv voluntarily by the process holding it, after that process has completed its task.
- ✓ **Circular wait:** There exists a set $\{P_0, P_1, \dots, P_0\}$ of waitingv processes such that P_0 is waiting for a resource that is held by P_1 , P_1 is waiting for a resource that is held by P_2 , ..., P_{n-1} is waiting for a resource that is held by P_n , and P_0 is waiting for a resource that is held by P_0 .

Resource-Allocation Graph

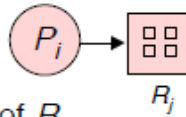
- Process



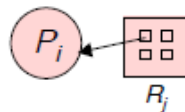
- Resource Type with 4 instances



- P_i requests instance of R_j



- P_i is holding an instance of R_j



A set of vertices V and a set of edges E .

- V is partitioned into two types:
 - $P = \{P_1, P_2, \dots, P_n\}$, the set consisting of all the processes in the system.
 - $R = \{R_1, R_2, \dots, R_m\}$, the set consisting of all resource types in the system.
- request edge – directed edge $P_i \rightarrow R_j$
- assignment edge – directed edge $R_j \rightarrow P_i$

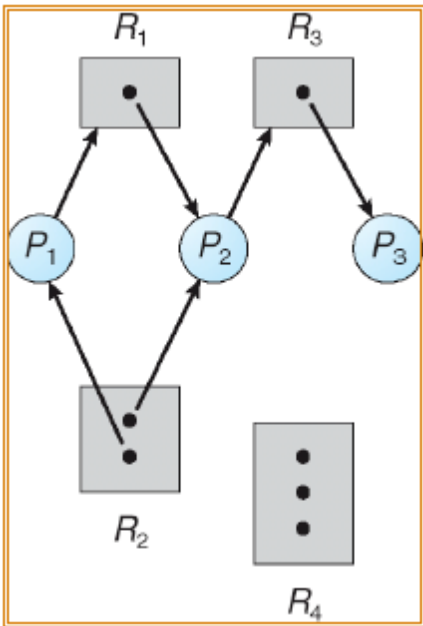


Fig. a Resource Allocation Graph

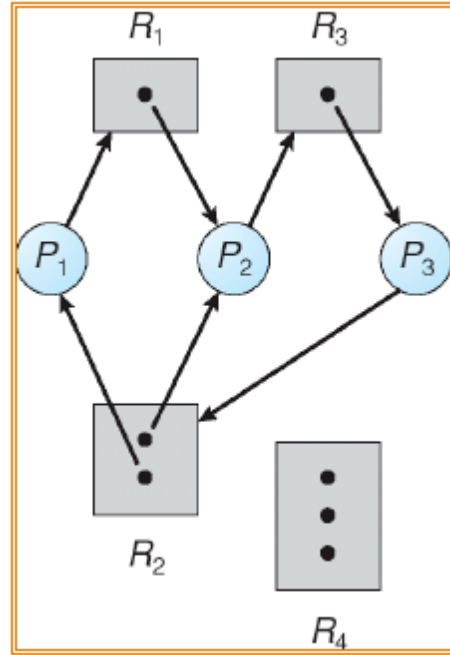


Fig. b Resource Allocation Graph with a deadlock:

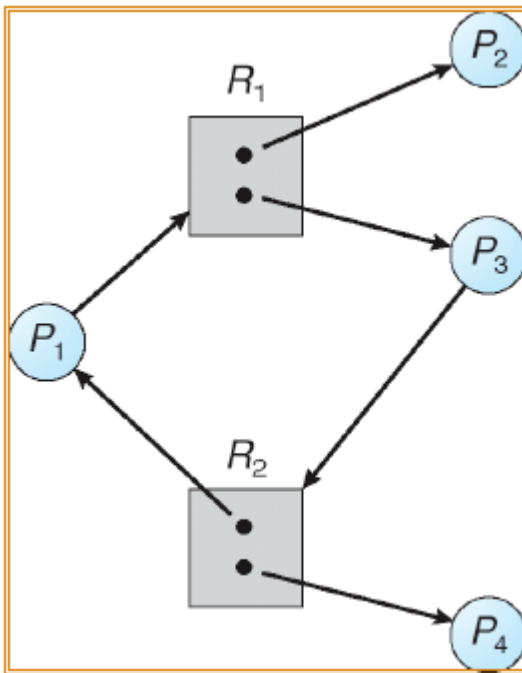


Fig. c Graph with a cycle but no deadlock

Methods for Handling Deadlocks

- ✓ Ensure that the system will never enter a deadlock state.
- ✓ Allow the system to enter a deadlock state and then recover.
- ✓ Ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems, including *UNIX*.

3.3 Deadlock Prevention

We can prevent the occurrence of a deadlock by ensuring at least one of these four conditions cannot hold:

1. **Mutual Exclusion** – not required for sharable resources; must hold for non-sharable resources.
2. **Hold and Wait** – must guarantee that whenever a process requests a resource, it does not hold any other resources.
 - ✓ Require process to request and be allocated all its resources before it begins execution, or allow process to request resources only when the process has none.
 - ✓ Low resource utilization; starvation possible.
3. **No Preemption:**
 - ✓ If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released.
 - ✓ Preempted resources are added to the list of resources for which the process is waiting.
 - ✓ Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting.
4. **Circular Wait** – impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration.

3.4 Deadlock Avoidance:

Requires that the system has some additional a priori information available.

- Simplest and most useful model requires that each process declare the maximum number of resources of each type that it may need.
- The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition.
- Resource-allocation state is defined by the number of available and allocated resources, and the maximum demands of the processes.

■ If graph contains no cycles \Rightarrow no deadlock.

■ If graph contains a cycle \Rightarrow

- if only one instance per resource type, then deadlock.
- if several instances per resource type, possibility of deadlock.

Safe State

- When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state.
- System is in **safe state** if there exists a sequence $\langle P_1, P_2, \dots, P_n \rangle$ of ALL the processes such that for each P_i , the resources that P_i can still request can be satisfied by currently available resources + resources held by all the P_j with $j < i$.
- That is:
 - If P_i resource needs are not immediately available, then P_i can wait until all P_j have finished.
 - When P_j is finished, P_i can obtain needed resources, execute, return allocated resources, and terminate.
 - When P_i terminates, P_{i+1} can obtain its needed resources, and so on.

Basic Facts:

- If a system is in safe state \Rightarrow no deadlocks.
- If a system is in unsafe state \Rightarrow possibility of deadlock.
- Avoidance \Rightarrow ensure that a system will never enter an unsafe state.

Deadlock Avoidance Algorithm:

a. **Single instance of a resource type:** Use a resource allocation graph

b. **Multiple instances of a resource type:** Use the banker's algorithm

Banker's Algorithm consists of :

- i. *Safety algorithm*
- ii. *Resource Request Algorithm*

a. Resource Allocation Graph Algorithm:

Suppose that process P_i requests a resource R_j . The request can be granted only if converting the request edge to an assignment edge does not result in the formation of a cycle in the resource allocation graph

- Claim edge $P_i \rightarrow R_j$ indicated that process P_i may request resource R_j ; represented by a dashed line.
- Claim edge converts to request edge when a process requests a resource.
- Request edge converted to an assignment edge when the resource is allocated to the process.
- When a resource is released by a process, assignment edge reconverts to a claim edge.
- Resources must be claimed *a priori* in the system.

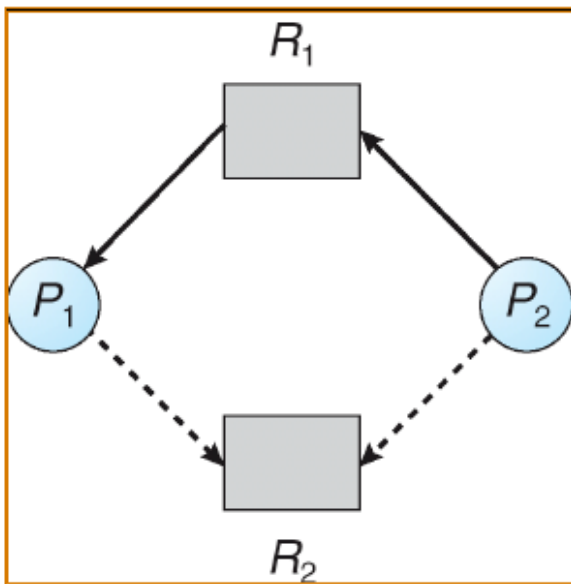


Fig. Resource-Allocation Graph

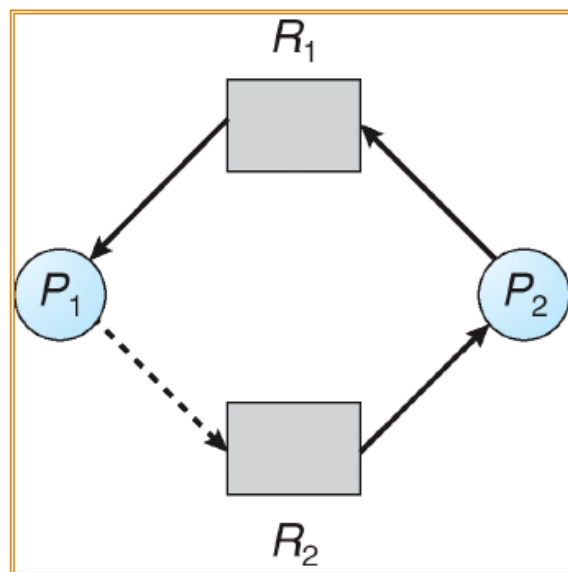


Fig. Unsafe state in resource allocation graph

b. Banker's Algorithm

The Resource Allocation Graph algorithm is not applicable to a resource allocation system with multiple instances of each resource type. Banker's Algorithm for deadlock avoidance involves:

- ✓ There can be multiple instances of each resource type.
- ✓ Each process must a priori claim maximum use.
- ✓ When a process requests a resource it may have to wait.
- ✓ When a process gets all its resources it must return them in a finite amount of time.

Data structure for a Banker's Algorithm:

Let n = number of processes, and m = number of resources types.

- **Available:** Vector of length m . If available $[j] = k$, there are k instances of resource type R_j available.
- **Max:** $n \times m$ matrix. If $Max[i, j] = k$, then process P_i may request at most k instances of resource type R_j .
- **Allocation:** $n \times m$ matrix. If $Allocation[i, j] = k$ then P_i is currently allocated k instances of R_j .
- **Need:** $n \times m$ matrix. If $Need[i, j] = k$, then P_i may need k more instances of R_j to complete its task.

$$Need[i, j] = Max[i, j] - Allocation[i, j].$$

i. Safety Algorithm:

1. Let **Work** and **Finish** be vectors of length m and n , respectively. Initialize:

$$Work = Available$$

$$Finish[i] = false \text{ for } i = 0, 1, \dots, n-1.$$

2. Find and i such that both:

$$(a) \text{ } Finish[i] = false$$

$$(b) \text{ } Need_i \leq Work$$

If no such i exists, go to step 4.

3. $Work = Work + Allocation_i$
 $Finish[i] = true$
go to step 2.

4. If $Finish[i] == true$ for all i , then the system is in a safe state.

ii. Resource Request Algorithm for Process P_i

$Request$ = request vector for process P_i . If $Request_i[j] = k$ then process P_i wants k instances of resource type R_j .

1. If $Request_i \leq Need_i$ go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim.
2. If $Request_i \leq Available$, go to step 3. Otherwise P_i must wait, since resources are not available.
3. Pretend to allocate requested resources to P_i by modifying the state as follows:

$$Available = Available - Request;$$

$$Allocation_i = Allocation_i + Request_i;$$

$$Need_i = Need_i - Request_i;$$

- If safe \Rightarrow the resources are allocated to P_i .
- If unsafe $\Rightarrow P_i$ must wait, and the old resource-allocation state is restored

Example of Banker's Algorithm:

- 5 processes P_0 through P_4 ;

3 resource types:

A (10 instances), B (5 instances), and C (7 instances).

- Snapshot at time T_0 :

	<u>Allocation</u>			<u>Max</u>			<u>Available</u>		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	7	5	3	3	3	2
P_1	2	0	0	3	2	2			
P_2	3	0	2	9	0	2			
P_3	2	1	1	2	2	2			
P_4	0	0	2	4	3	3			

- The content of the matrix *Need* is defined to be *Max* – *Allocation*.

	<u>Need</u>
	<i>A B C</i>
P_0	7 4 3
P_1	1 2 2
P_2	6 0 0
P_3	0 1 1
P_4	4 3 1

- The system is in a safe state since the sequence $\langle P_1, P_3, P_4, P_2, P_0 \rangle$ satisfies safety criteria.

3.4 Deadlock Detection

If a system does not employ either a deadlock-prevention or a deadlock-avoidance algorithm, then a deadlock situation may occur. In this environment, the system must provide:

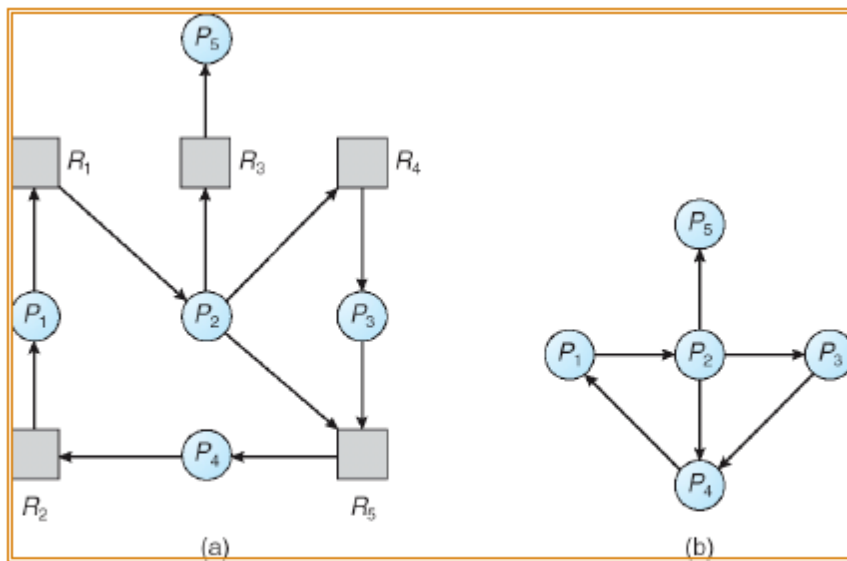
- An algorithm that examines the state of the system to determine whether a deadlock has occurred
- An algorithm to recover from the deadlock

Deadlock detection involves:

- Allowing system to enter deadlock state
- Detection algorithm
- Recovery scheme

3.4.1 Single Instance of each Resource Type:

- Maintain *wait-for* graph
 - Nodes are processes.
 - $P_i \rightarrow P_j$ if P_i is waiting for P_j .
- Periodically invoke an algorithm that searches for a cycle in the graph. If there is a cycle, there exists a deadlock.
- An algorithm to detect a cycle in a graph requires an order of n^2 operations, where n is the number of vertices in the graph.



Resource-Allocation Graph Corresponding wait-for graph

3.4.2 Several Instances of a Resource Type: Its data structure involves:

- **Available:** A vector of length m indicates the number of available resources of each type.
- **Allocation:** An $n \times m$ matrix defines the number of resources of each type currently allocated to each process.
- **Request:** An $n \times m$ matrix indicates the current request of each process. If $Request[i_j] = k$, then process P_i is requesting k more instances of resource type. R_j .

Detection Algorithm:

1. Let *Work* and *Finish* be vectors of length *m* and *n*, respectively
Initialize:

(a) *Work* = *Available*

(b) For $i = 1, 2, \dots, n$, if $Allocation_i \neq 0$, then
 $Finish[i] = false$; otherwise, $Finish[i] = true$.

2. Find an index *i* such that both:

(a) $Finish[i] == false$

(b) $Request_i \leq Work$

If no such *i* exists, go to step 4.

3. $Work = Work + Allocation_i$
 $Finish[i] = true$
go to step 2.

4. If $Finish[i] == false$, for some $i, 1 \leq i \leq n$, then the system is in deadlock state. Moreover, if $Finish[i] == false$, then P_i is deadlocked.

Example of Detection Algorithm:

- Five processes P_0 through P_4 ; three resource types
A (7 instances), B (2 instances), and C (6 instances).

- Snapshot at time T_0 :

	<u>Allocation</u>			<u>Request</u>			<u>Available</u>		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	0	0	0	0	0	0
P_1	2	0	0	2	0	2			
P_2	3	0	3	0	0	0			
P_3	2	1	1	1	0	0			
P_4	0	0	2	0	0	2			

- Sequence $\langle P_0, P_2, P_3, P_1, P_4 \rangle$ will result in $Finish[i] = true$ for all *i*.

Recovery from Deadlock:

When a detection algorithm determines that a deadlock exists, several alternatives exist. One possibility is to inform the operator that a deadlock has occurred, and to let the operator deal with the deadlock manually. The other possibility is to let the system *recover* from the deadlock automatically. There are two options for breaking a deadlock. One solution is simply to abort one or more processes to break the circular wait. The second option is to preempt some resources from one or more of the deadlocked processes.

a. Process Termination

- Abort all deadlocked processes.
- Abort one process at a time until the deadlock cycle is eliminated.
- In which order should we choose to abort?
- Priority of the process.
- How long process has computed, and how much longer to completion.
- Resources the process has used.
- Resources process needs to complete.
- How many processes will need to be terminated.
- Is process interactive or batch?

b. Resource Preemption

- Selecting a victim – minimize cost.
- Rollback – return to some safe state, restart process for that state.
- Starvation – same process may always be picked as victim, include number of rollback in cost factor.

Chapter-5: Memory Management

Main Memory refers to a physical memory that is the internal memory to the computer. The word main is used to distinguish it from external mass storage devices such as disk drives. Main memory is also known as RAM. The computer is able to change only data that is in main memory. Therefore, every program we execute and every file we access must be copied from a storage device into main memory.

All the programs are loaded in the main memory for execution. Sometimes complete program is loaded into the memory, but sometimes a certain part or routine of the program is loaded into the main memory only when it is called by the program, this mechanism is called **Dynamic Loading**, this enhance the performance.

Sometimes one program is dependent on some other program. In such a case, rather than loading all the dependent programs, CPU links the dependent programs to the main executing program when its required. This mechanism is known as **Dynamic Linking**.

Memory Management

- Is the task carried out by the OS and hardware to accommodate multiple processes in main memory
- If only a few processes can be kept in main memory, then much of the time all processes will be waiting for I/O and the CPU will be idle
- Hence, memory needs to be allocated efficiently in order to pack as many processes into memory as possible
- In most schemes, the kernel occupies some fixed portion of main memory and the rest is shared by multiple processes

Logical versus Physical Address Space

An address generated by the CPU is a logical address whereas address actually available on memory unit is a physical address. Logical address is also known a Virtual address. 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 run-time 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.

5.1 Fixed and variable partition system

Fixed partitioning:

Main memory is divided into a no. of static partitions at system generation time. A process may be loaded into a partition of equal or greater size. Memory Manager will allocate a region to a process that best fits it. Unused memory within an allocated partition called **internal fragmentation**.

Advantages:

- ✓ Simple to implement
- ✓ Little OS overhead

Disadvantages:

There is inefficient use of memory due to internal fragmentation. Main memory utilization is extremely inefficient. Any program, no matter how small, occupies an entire partition. This phenomenon, in which there is wasted space internal to a partition due to the fact that the block of data loaded is smaller than the partition, is referred to as internal fragmentation.

Two possibilities:

- Equal size partitioning
- Unequal size Partition

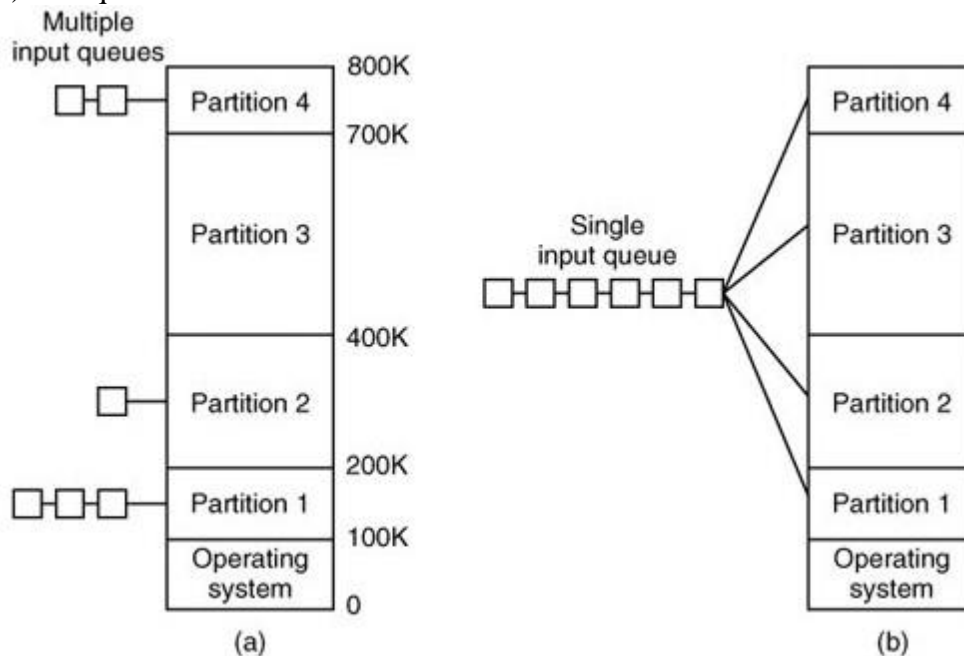


Fig. (a) Fixed memory partitions with separate input queues for each partition.

Fig. (b) Fixed memory partitions with a single input queue.

When the queue for a large partition is empty but the queue for a small partition is full, as is the case for partitions 1 and 3. Here small jobs have to wait to get into memory, even though plenty of memory is free. An alternative organization is to maintain a single queue as in Fig. 4-

2(b). Whenever a partition becomes free, the job closest to the front of the queue that fits in it could be loaded into the empty partition and run.

Dynamic/Variable Partitioning:

To overcome some of the difficulties with fixed partitioning, an approach known as dynamic partitioning was developed. The partitions are of variable length and number. When a process is brought into main memory, it is allocated exactly as much memory as it requires and no more. An example, using 64 Mbytes of main memory, is shown in Figure below. Eventually it leads to a situation in which there are a lot of small holes in memory. As time goes on, memory becomes more and more fragmented, and memory utilization declines. This phenomenon is referred to as **external fragmentation**, indicating that the memory that is external to all partitions becomes increasingly fragmented.

One technique for overcoming external fragmentation is compaction: From time to time, the operating system shifts the processes so that they are contiguous and so that all of the free memory is together in one block. For example, in Figure h, compaction will result in a block of free memory of length 16M. This may well be sufficient to load in an additional process. The difficulty with compaction is that it is a time consuming procedure and wasteful of processor time.

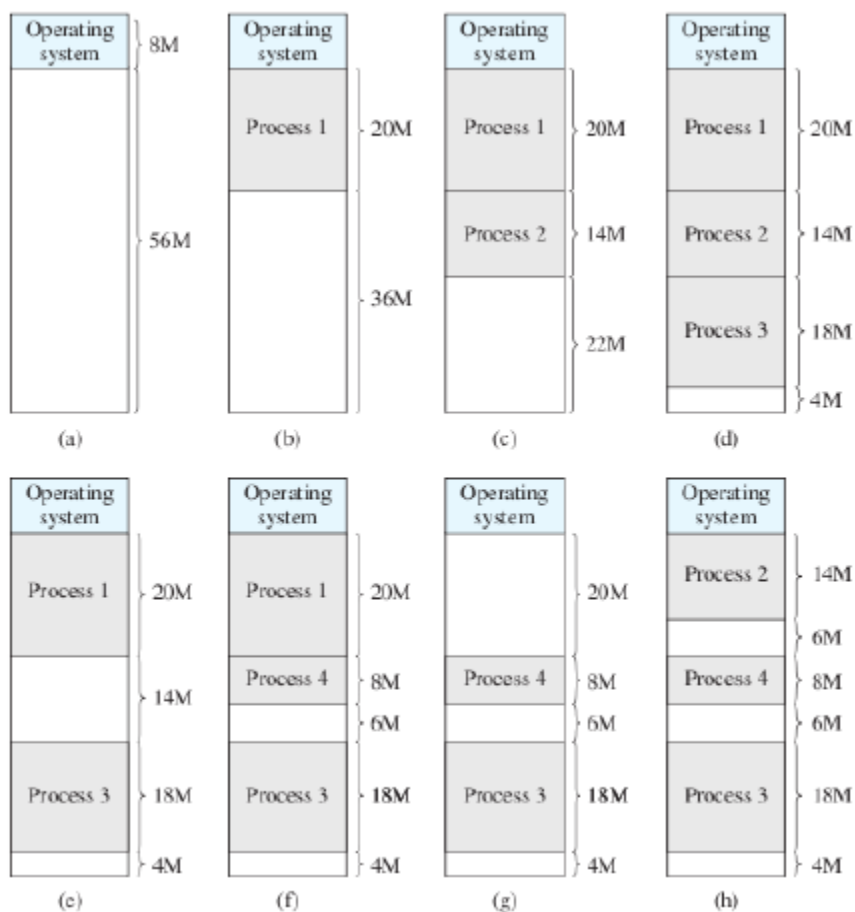


Fig. The effects of variable sized partitioning

5.2 Bitmaps

When memory is assigned dynamically, the operating system must manage it. With a bitmap, memory is divided up into allocation units. Corresponding to each allocation unit is a bit in the bitmap, which is 0 if the unit is free and 1 if it is occupied (or vice versa). Figure below shows part of memory and the corresponding bitmap.

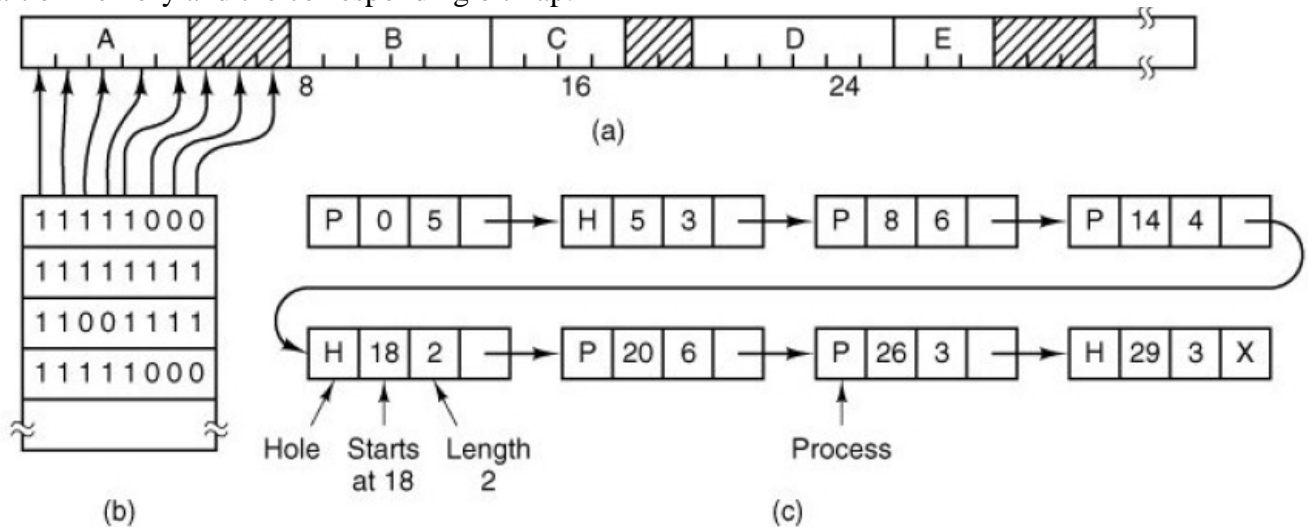


Fig:(a) A part of memory with five processes and three holes. The tick marks show the memory allocation units. The shaded regions (0 in the bitmap) are free. (b) The corresponding bitmap. (c) The same information as a list.

The size of the allocation unit is an important design issue. The smaller the allocation unit, the larger the bitmap. A bitmap provides a simple way to keep track of memory words in a fixed amount of memory because the size of the bitmap depends only on the size of memory and the size of the allocation unit.

5.3 Memory Management with linked list

Another way of keeping track of memory is to maintain a linked list of allocated and free memory segments, where a segment is either a process or a hole between two processes.

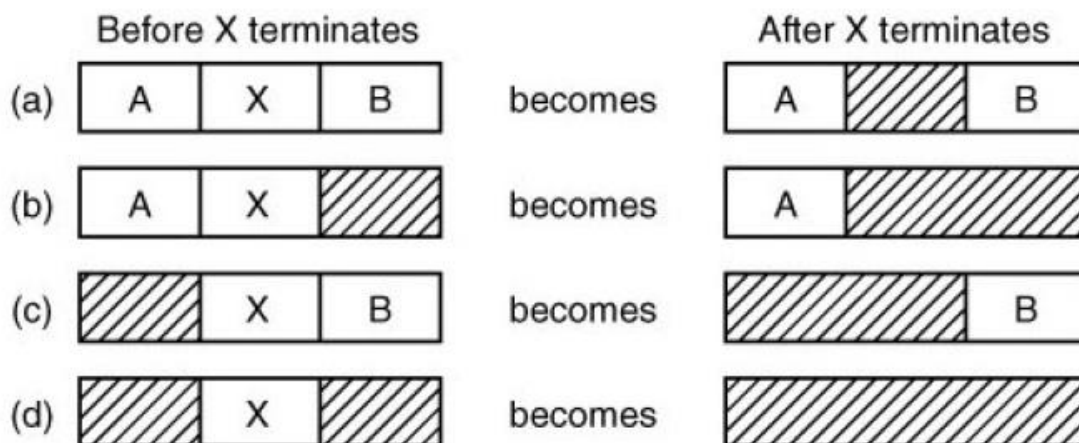


Fig. Four neighbors combinations for the terminating Process, X

Each entry in the list specifies a hole (H) or process (P), the address at which it starts, the length, and a pointer to the next entry. In this example, the segment list is kept sorted by address. Sorting this way has the advantage that when a process terminates or is swapped out, updating the list is straightforward. A terminating process normally has two neighbors (except when it is at the very top or very bottom of memory). These may be either processes or holes, leading to the four combinations shown in fig above. When the processes and holes are kept on a list sorted by address, several algorithms can be used to allocate memory for a newly created process or an existing process being swapped in from disk.

First Fit:

The simplest algorithm is first fit. The process manager scans along the list of segments until it finds a hole that is big enough. The hole is then broken up into two pieces, one for the process and one for the unused memory, except in the statistically unlikely case of an exact fit. First fit is a fast algorithm because it searches as little as possible.

Next Fit:

It works the same way as first fit, except that it keeps track of where it is whenever it finds a suitable hole. The next time it is called to find a hole, it starts searching the list from the place where it left off last time, instead of always at the beginning, as first fit does.

Best Fit:

Best fit searches the entire list and takes the smallest hole that is adequate. Rather than breaking up a big hole that might be needed later, best fit tries to find a hole that is close to the actual size needed.

Worst Fit:

It always take the largest available hole, so that the hole broken off will be big enough to be useful. It should be noted that worst fit is not a very good idea either.

Quick Fit:

It maintains separate lists for some of the more common sizes requested. For example, it might have a table with n entries, in which the first entry is a pointer to the head of a list of 4-KB holes, the second entry is a pointer to a list of 8-KB holes, the third entry a pointer to 12-KB holes, and so on. With quick fit, finding a hole of the required size is extremely fast, but it has the same disadvantage as all schemes that sort by hole size, namely, when a process terminates or is swapped out, finding its neighbors to see if a merge is possible is expensive. If merging is not done, memory will quickly fragment into a large number of small holes into which no processes fit.

Buddy-system:

Both fixed and dynamic partitioning schemes have drawbacks. A fixed partitioning scheme limits the number of active processes and may use space inefficiently if there is a poor match between available partition sizes and process sizes. A dynamic partitioning scheme is more complex to maintain and includes the overhead of compaction. An interesting compromise is the buddy system. In a buddy system, the entire memory space available for allocation is initially treated as a single block whose size is a power of 2. When the first request is made, if its size is greater than half of the initial block, then the entire block is allocated. Otherwise, the block is split in two equal companion buddies. If the size of the request is greater than half of one of the buddies, then allocate one to it. Otherwise, one of the buddies is split in half again. This method continues until the smallest block greater than or equal to the size of the request is found and

allocated to it. In this method, when a process terminates the buddy block that was allocated to it is freed. Whenever possible, an unallocated buddy is merged with a companion buddy in order to form a larger free block. Two blocks are said to be companion buddies if they resulted from the split of the same direct parent block.

5.4 Multiprogramming memory management techniques

The memory management function keeps track of the status of each memory location, either allocated or free. It determines how memory is allocated among competing processes, deciding which gets memory, when they receive it, and how much they are allowed. When memory is allocated it determines which memory locations will be assigned. It tracks when memory is freed or unallocated and updates the status.

i. Single contiguous allocation

Single allocation is the simplest memory management technique. All the computer's memory, usually with the exception of a small portion reserved for the operating system, is available to the single application. MS-DOS is an example of a system which allocates memory in this way. An embedded system running a single application might also use this technique.

A system using single contiguous allocation may still multitask by swapping the contents of memory to switch among users. Early versions of the Music operating system used this technique.

ii. Partitioned allocation

Partitioned allocation divides primary memory into multiple memory partitions, usually contiguous areas of memory. Each partition might contain all the information for a specific job or task. Memory management consists of allocating a partition to a job when it starts and unallocating it when the job ends. Partitioned allocation usually requires some hardware support to prevent the jobs from interfering with one another or with the operating system.

Partitions may be either static, that is defined at Initial Program Load (IPL) or boot time or by the computer operator, or dynamic, that is automatically created for a specific job.

iii. Paged memory management

Paged allocation divides the computer's primary memory into fixed-size units called page frames, and the program's virtual address space into pages of the same size. The hardware memory management unit maps pages to frames. The physical memory can be allocated on a page basis while the address space appears contiguous.

Usually, with paged memory management, each job runs in its own address space. Paged memory can be demand-paged when the system can move pages as required between primary and secondary memory.

iv. Segmented memory management

Segmented memory is the only memory management technique that does not provide the user's program with a linear and contiguous address space. Segments are areas of memory that usually correspond to a logical grouping of information such as a code procedure or a data array. Segments require hardware support in the form of a segment table which usually contains the physical address of the segment in memory, its size, and other data such as access protection bits and status (swapped in, swapped out, etc.)

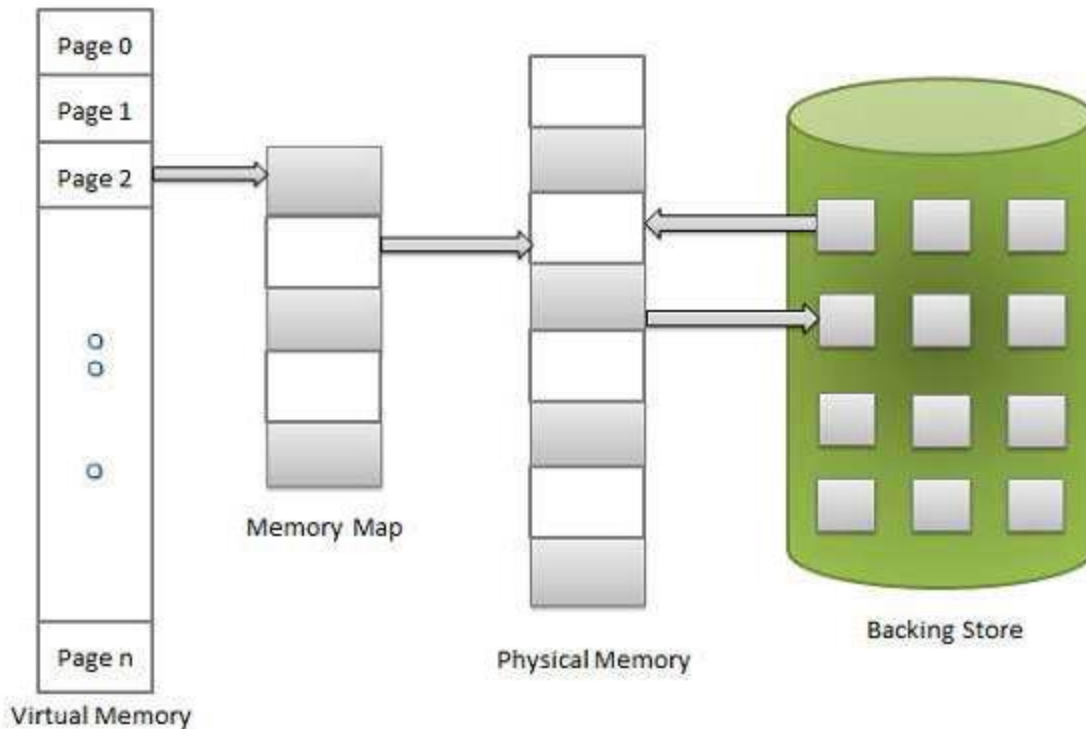
It is possible to implement segmentation with or without paging. Without paging support the segment is the physical unit swapped in and out of memory if required. With paging support the pages are usually the unit of swapping and segmentation only adds an additional level of security.

5.5 Virtual Memory

Virtual memory is a technique that allows the execution of processes which are not completely available in memory. The main visible advantage of this scheme is that programs can be larger than physical memory. Virtual memory is the separation of user logical memory from physical memory. This separation allows an extremely large virtual memory to be provided for programmers when only a smaller physical memory is available.

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.



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.

5.5.1 Paging and Segmentation

Paging:

External fragmentation is avoided by using paging technique. Paging is a technique in which physical memory is broken into blocks of the same size called pages (size is power of 2, between 512 bytes and 8192 bytes). When a process is to be executed, its corresponding pages are loaded into any available memory frames.

Logical address space of a process can be non-contiguous and a process is allocated physical memory whenever the free memory frame is available. Operating system keeps track of all free frames. Operating system needs n free frames to run a program of size n pages.

Address generated by CPU is divided into:

- *Page number (p)*: Page number is used as an index into a page table which contains base address of each page in physical memory.
- *Page offset (d)*: Page offset is combined with base address to define the physical memory address.

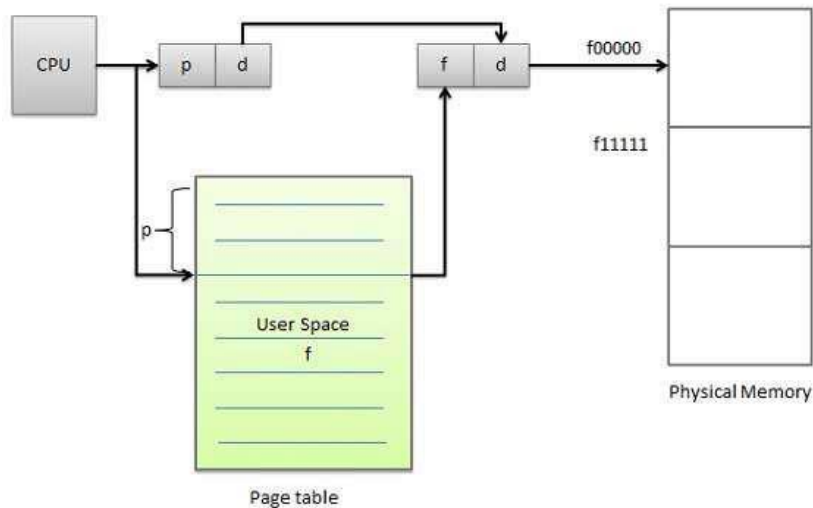
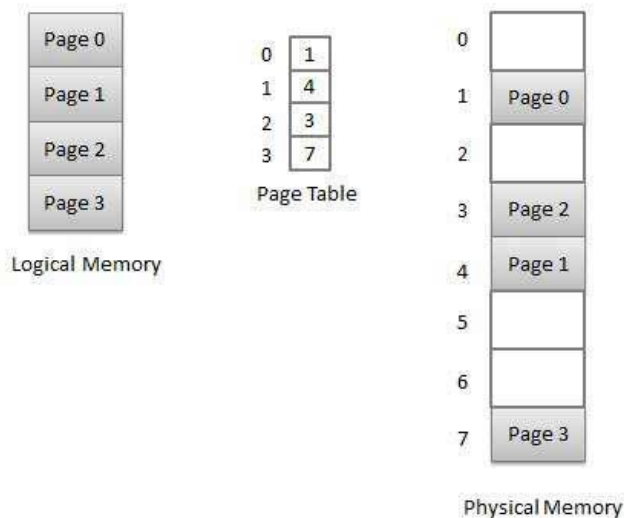


Figure below shows the paging table architecture.

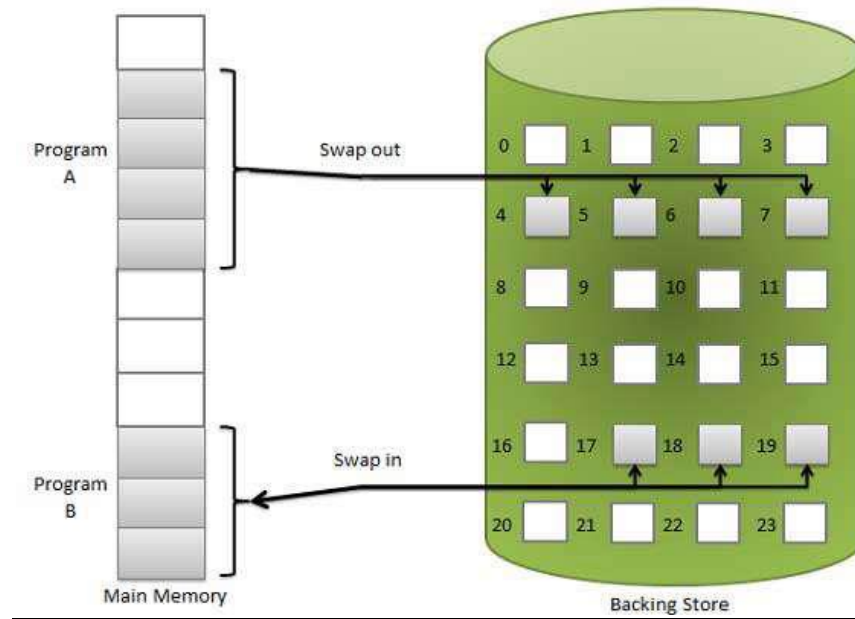


Demand Paging

A demand paging system is quite similar to a paging system with swapping. When we want to execute a process, we swap it into memory. Rather than swapping the entire process into memory, however, we use a lazy swapper called pager.

When a process is to be swapped in, the pager guesses which pages will be used before the process is swapped out again. Instead of swapping in a whole process, the pager brings only those necessary pages into memory. Thus, it avoids reading into memory pages that will not be used in anyway, decreasing the swap time and the amount of physical memory needed.

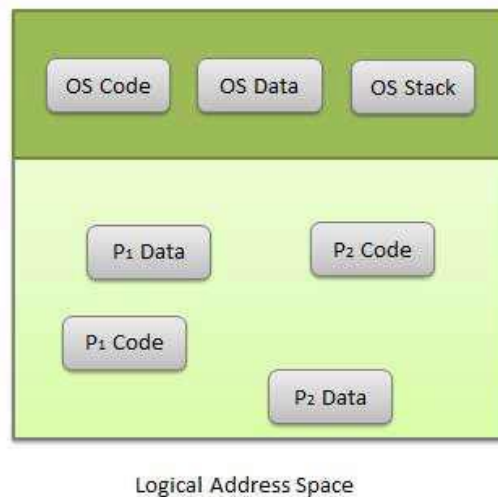
Hardware support is required to distinguish between those pages that are in memory and those pages that are on the disk using the valid-invalid bit scheme where valid and invalid pages can be checked by checking the bit. Marking a page will have no effect if the process never attempts to access the page. While the process executes and accesses pages that are memory resident, execution proceeds normally.



Segmentation

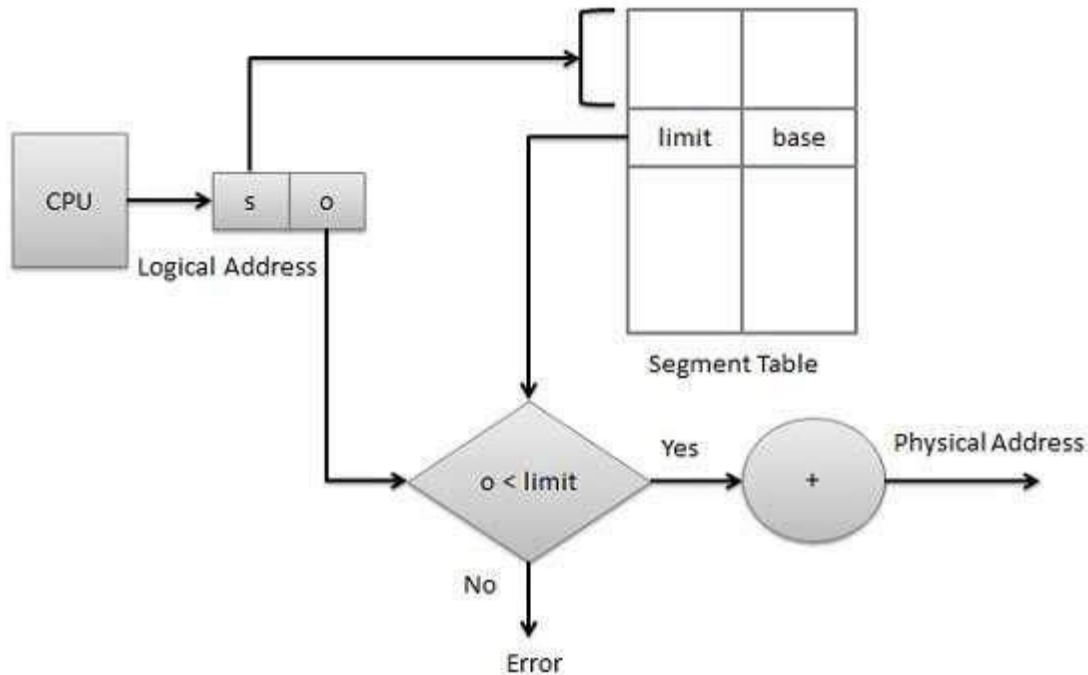
Segmentation is a technique to break memory into logical pieces where each piece represents a group of related information. For example, data segments or code segment for each process, data segment for operating system and so on. Segmentation can be implemented using or without using paging.

Unlike paging, segment is having varying sizes and thus eliminates internal fragmentation. External fragmentation still exists but to lesser extent.



Address generated by CPU is divided into:

- **Segment number (s):** Segment number is used as an index into a segment table which contains base address of each segment in physical memory and a limit of segment.
- **Segment offset (o):** Segment offset is first checked against limit and then is combined with base address to define the physical memory address.



5.5.2 Swapping and Page Replacement

Swapping:

Swapping is a mechanism in which a process can be swapped temporarily out of main memory to a backing store, and then brought back into memory for continued execution.

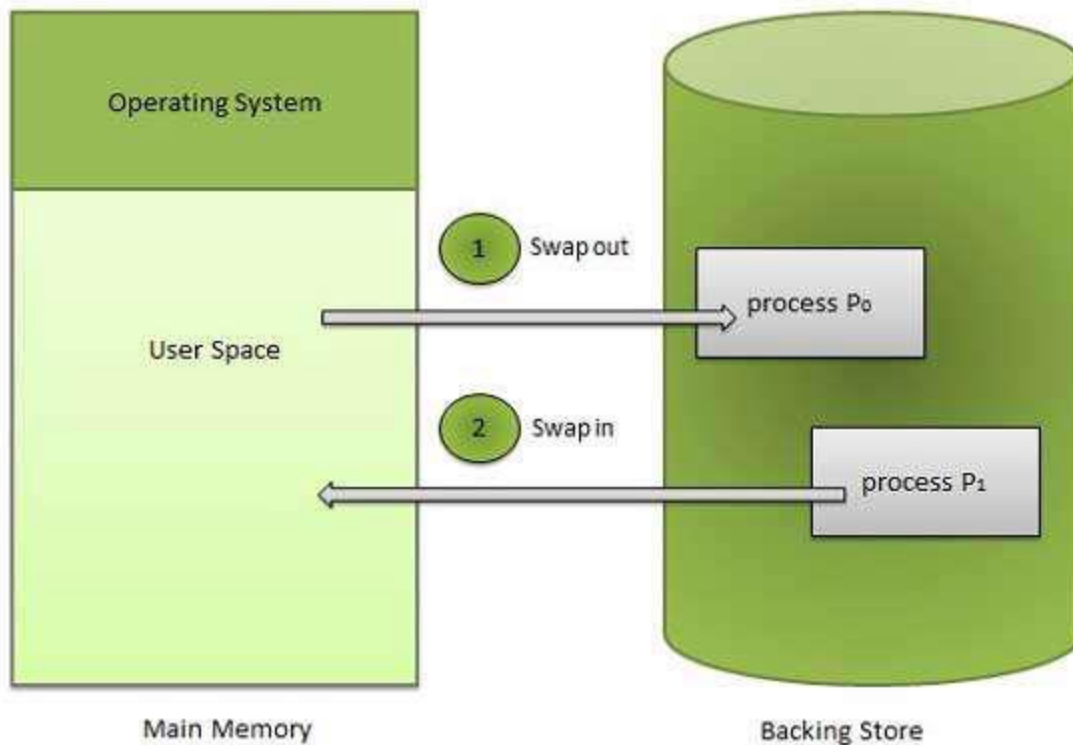
Backing store is a usually a hard disk drive or any other secondary storage which fast in access and large enough to accommodate copies of all memory images for all users. It must be capable of providing direct access to these memory images.

Major time consuming part of swapping is transfer time. Total transfer time is directly proportional to the amount of memory swapped. Let us assume that the user process is of size 100KB and the backing store is a standard hard disk with transfer rate of 1 MB per second. The actual transfer of the 100K process to or from memory will take

100KB / 1000KB per second

= 1/10 second

= 100 milliseconds



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 i.e. internal and external

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.

Page Replacement Algorithm

Page replacement algorithms are the techniques using which 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 then 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.

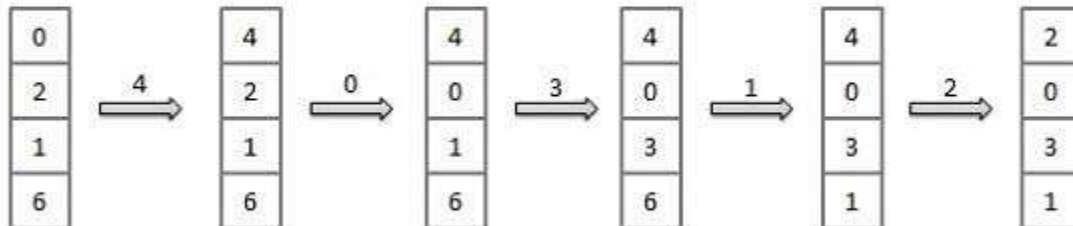
The string of memory references is called reference string. Reference strings are generated artificially or by tracing a given system and recording the address of each memory reference.

First In First Out (FIFO) Page Replacement 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

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

Misses : x x x x x x x x x



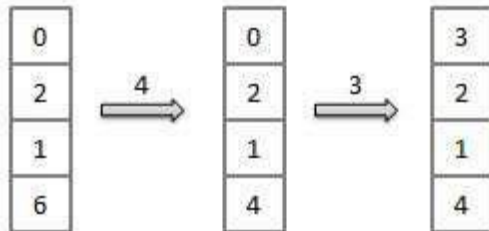
Fault Rate = $9 / 12 = 0.75$

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 x



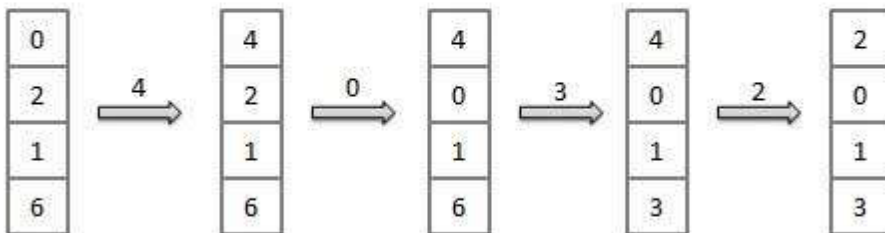
$$\text{Fault Rate} = 6 / 12 = 0.50$$

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



$$\text{Fault Rate} = 8 / 12 = 0.67$$

Chapter-6: Real Time Operating System

A **real-time operating system (RTOS)** is an **operating system (OS)** intended to serve real-time applications that process data as it comes in, typically without buffering delays. Processing time requirements (including any OS delay) are measured in tenths of seconds or shorter increments of time.

Real-time systems are those systems in which the correctness of the system depends not only on the logical result of computation, but also on the time at which the results are produced.

RTOS is therefore an operating system that supports real-time applications by providing logically correct result within the deadline required. Basic Structure is similar to regular OS but, in addition, it provides mechanisms to allow real time scheduling of tasks.

Though real-time operating systems may or may not increase the speed of execution, they can provide much more precise and predictable timing characteristics than general-purpose OS.

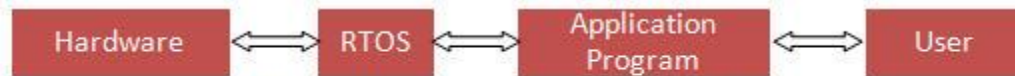


Fig. Real time embedded system with RTOS

RTOS is key to many embedded systems and provides a platform to build applications. All embedded systems are not designed with RTOS. Embedded systems with relatively simple/small hardware/code might not require an RTOS. Embedded systems with moderate-to-large software applications require some form of scheduling, and hence RTOS.

Important terminologies used in context of real time systems

- **Determinism:** An application is referred to as deterministic if its timing can be guaranteed within a certain margin of error.
- **Jitter:** Timing error of a task over subsequent iterations of a program or loop is referred to as jitter. RTOS are optimized to minimize jitter.

DIFFERENCE: RTOS v/s General Purpose OS

- **Determinism** - The key difference between general-computing operating systems and real-time operating systems is the "deterministic" timing behavior in the real-time operating systems. "Deterministic" timing means that OS consume only known and expected amounts of time. RTOS have their worst case latency defined. Latency is not of a concern for General Purpose OS.
- **Task Scheduling** - General purpose operating systems are optimized to run a variety of applications and processes simultaneously, thereby ensuring that all tasks receive at least some

processing time. As a consequence, low-priority tasks may have their priority boosted above other higher priority tasks, which the designer may not want. However, RTOS uses priority-based preemptive scheduling, which allows high-priority threads to meet their deadlines consistently. All system calls are deterministic, implying time bounded operation for all operations and ISRs. This is important for embedded systems where delay could cause a safety hazard. The scheduling in RTOS is time based. In case of General purpose OS, like Windows/Linux, scheduling is process based.

- **Preemptive kernel** - In RTOS, all kernel operations are preemptible
- **Priority Inversion** - RTOS have mechanisms to prevent priority inversion
- **Usage** - RTOS are typically used for embedded applications, while General Purpose OS are used for Desktop PCs or other generally purpose PCs.

Types of Real Time Operating System

RTOS specifies a known maximum time for each of the operations that it performs. Based upon the degree of tolerance in meeting deadlines, RTOS are classified into following categories

i. Hard real-time: Degree of tolerance for missed deadlines is negligible. A missed deadline can result in catastrophic failure of the system. For a **Hard real-time system**, if the system fails to meet the deadline even once, the system is considered to have failed. The **hard real-time** definition considers any missed deadline to be a system failure. This scheduling is used extensively in mission critical systems where failure to conform to timing constraints results in a loss of life or property.

For a life saving device, automatic parachute opening device for skydivers, delay can be fatal. Parachute opening device deploys the parachute at a specific altitude based on various conditions. If it fails to respond in specified time, parachute may not get deployed at all leading to casualty. Similar situation exists during inflation of air bags, used in cars, at the time of accident. If airbags don't get inflated at appropriate time, it may be fatal for a driver. So such systems must be hard real time systems

Examples:

Air France Flight 447 crashed into the ocean after a sensor malfunction caused a series of system errors. The pilots stalled the aircraft while responding to outdated instrument readings. All 12 crew and 216 passengers were killed.

ii. Firm real-time: For a **Firm real-time system**, even if the system fails to meet the deadline, possibly more than once (i.e. for multiple requests), the system is not considered to have failed. The **firm real-time** definition allows for infrequently missed deadlines.

Examples:

Manufacturing systems with robot assembly lines where missing a deadline results in improperly assembling a part. As long as ruined parts are infrequent enough to be caught by quality control and not too costly, then production continues.

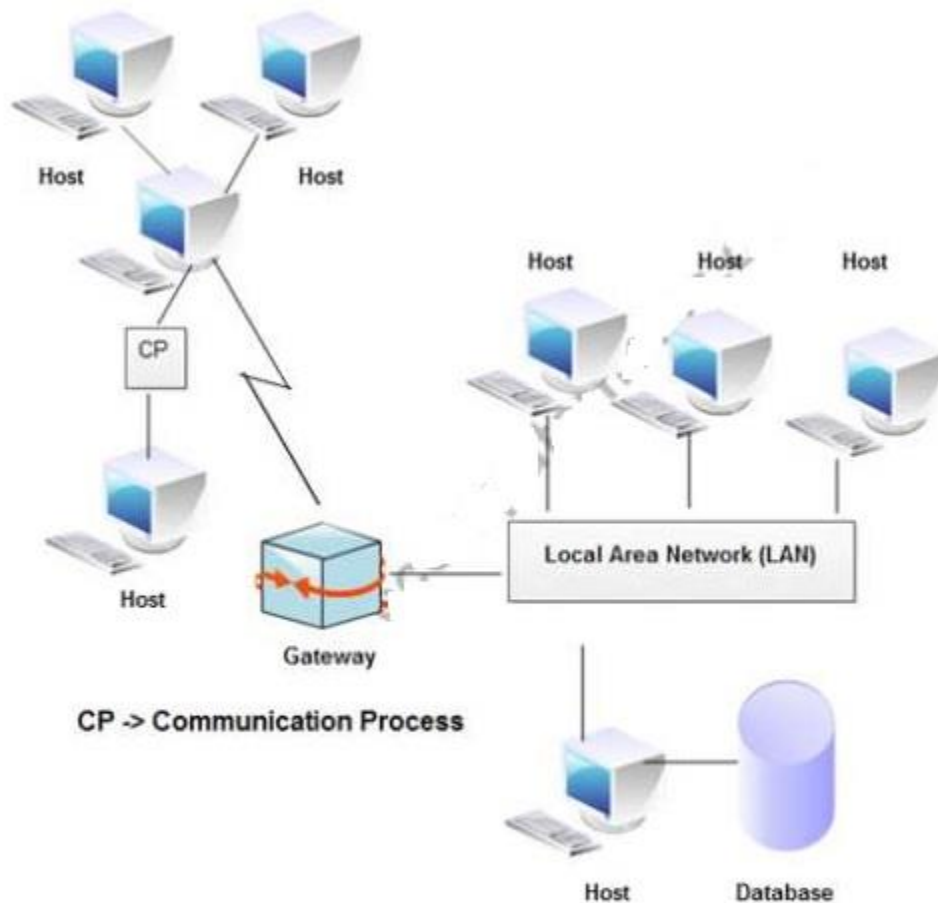
iii. **Soft real-time:** The **soft real-time** system allows for frequently missed deadlines, and as long as tasks are timely executed their results continue to have value. In a **soft real time** system, even if the result is obtained after the deadline, the results are still considered as valid.

Example: Web browser- We request for certain URL, it takes some time in loading the page. If the system takes more than expected time to provide us with the page, the page obtained is not considered as invalid, we just say that the system's performance wasn't up to the mark.

Another example is TV live broadcast in which delay can be acceptable.

Chapter-7: Distributed Operating System

Distributed Operating System is a model where distributed applications are running on multiple computers linked by communications. A distributed operating system is an extension of the network operating system that supports higher levels of communication and integration of the machines on the network.



A Typical View of Distributed System

These systems are referred as loosely coupled systems where each processor has its own local memory and processors communicate with one another through various communication lines, such as high speed buses or telephone lines. By loosely coupled systems, we mean that such computers possess no hardware connections at the CPU - memory bus level, but are connected by external interfaces that run under the control of software.

The Distributed Os involves a collection of autonomous computer systems, capable of communicating and cooperating with each other through a LAN / WAN. A Distributed Os provides a virtual machine abstraction to its users and wide sharing of resources like as computational capacity, I/O and files etc.

The structure shown in fig contains a set of individual computer systems and workstations connected via communication systems, but by this structure we cannot say it is a distributed system because it is the software, not the hardware, that determines whether a system is distributed or not.

The users of a true distributed system should not know, on which machine their programs are running and where their files are stored. LOCUS and MICROS are the best examples of distributed operating systems.

Distributed systems provide the following advantages:

- 1 Sharing of resources.
- 2 Reliability.
- 3 Communication.
- 4 Computation speedup.

Distributed systems are potentially more reliable than a central system because if a system has only one instance of some critical component, such as a CPU, disk, or network interface, and that component fails, the system will go down. When there are multiple instances, the system may be able to continue in spite of occasional failures. In addition to hardware failures, one can also consider software failures. Distributed systems allow both hardware and software errors to be dealt with.

A distributed system is a set of computers that communicate and collaborate each other using software and hardware interconnecting components. Multiprocessors (MIMD computers using shared memory architecture), multicomputers connected through static or dynamic interconnection networks (MIMD computers using message passing architecture) and workstations connected through local area network are examples of such distributed systems.

A distributed system is managed by a distributed operating system. A distributed operating system manages the system shared resources used by multiple processes, the process scheduling activity (how processes are allocating on available processors), the communication and synchronization between running processes and so on. The software for parallel computers could be also tightly coupled or loosely coupled. The loosely coupled software allows computers and users of a distributed system to be independent each other but having a limited possibility to cooperate. An example of such a system is a group of computers connected through a local network. Every computer has its own memory, hard disk. There are some shared resources such files and printers. If the interconnection network broke down, individual computers could be used but without some features like printing to a non-local printer

7.1 Communication and Synchronization

In a distributed system there is no shared memory and thus the whole nature of the communication between processes should be reconsidered. The processes, to communicate, must adhere to rules known as protocols. For distributed systems in a wide area, these protocols often take the form of several layers and each layer has its own goals and rules. Messages are exchanged in various ways, and there are many design options in this regard,

an important option is the procedure call. It is also important to consider the possibilities of communication between groups of processes, not only between two processes.

A distributed semaphore is a distributed synchronization mechanism that behaves in much the same way as a semaphore. Two operations are defined on distributed semaphores: P and V. Execution of a P operation consists of making a transition to a P-phase, and, similarly, for a V operation transition to a V-phase is attempted. A semaphore is a synchronization mechanism that ensures that, for every completed P-phase transition, a unique V-phase transition has been made by some process.

7.2 Processes and Processors in Distributed OS

A Distributed Processes program consists of a fixed number of modules residing on separate logical machines. Each module contains a single process. Modules do not nest. Processes communicate by calling entry procedures (called common procedures) defined in other modules. Communication is thus by means of implicit receipt and remote-invocation send. Data can be shared between entry procedures, but not across module boundaries. An entry procedure is free to block itself on an arbitrary Boolean condition. The main body of code for a process may do likewise. Each process alternates between executing its main code and serving external requests. It jumps from one body of code to another only when a blocking statement is encountered. The executions of entry procedures thus exclude each other in time, much as they do in a monitor. Nested calls block the outer modules; a process remains idle while waiting for its remote requests to complete. There is a certain amount of implementation cost in the repeated evaluation of blocking conditions

Processor Allocation

Two basic allocation models:

- a. Non-migratory: once process is placed, it cannot be moved
 - b. Migratory: process can move in the middle of execution. It must restore state at new CPU.
- It provides better load balancing, but more complex design

Distributed processor management

If local processor is idle or underutilized, use it otherwise execute it remotely. Resource manager must:

- keep track of idle processors
- find one when a request is received
- send the process to a remote computer
- receive results from remote process

Chapter-8: File System

8.1 Files and directories

File:

A file is a named collection of related information that is recorded on secondary storage such as magnetic disks, magnetic tapes and optical disks. In general, a file is a sequence of bits, bytes, lines or records whose meaning is defined by the files creator and user.

Attributes of a File

Following are some of the attributes of a file :

- **Name** . It is the only information which is in human-readable form.
- **Identifier**. The file is identified by a unique tag(number) within file system.
- **Type**. It is needed for systems that support different types of files.
- **Location**. Pointer to file location on device.
- **Size**. The current size of the file.
- **Protection**. This controls and assigns the power of reading, writing, executing.
- **Time, date, and user identification**. This is the data for protection, security, and usage monitoring.

File Structure

A file has various kinds of structure. Some of them can be :

- **Simple Record Structure** with lines of fixed or variable lengths.
- **Complex Structures** like formatted document or reloadable load files.
- **No Definite Structure** like sequence of words and bytes etc.

File Access Methods

The way that files are accessed and read into memory is determined by Access methods. Usually a single access method is supported by systems while there are OS's that support multiple access methods.

i. Sequential Access

- Data is accessed one record right after another in an order.
- Read command cause a pointer to be moved ahead by one.
- Write command allocate space for the record and move the pointer to the new End Of File.
- Such a method is reasonable for tape.

ii. Direct Access

- This method is useful for disks.
- The file is viewed as a numbered sequence of blocks or records.
- There are no restrictions on which blocks are read/written, it can be done in any order.
- User now says "read n" rather than "read next".
- "n" is a number relative to the beginning of file, not relative to an absolute physical disk location.

iii. Indexed Sequential Access

- It is built on top of Sequential access.
- It uses an Index to control the pointer while accessing files.

Directory

Information about files is maintained by Directories. A directory can contain multiple files. It can even have directories inside of them. In Windows we also call these directories as folders.

Following is the information maintained in a directory :

- **Name** : The name visible to user.
- **Type** : Type of the directory.
- **Location** : Device and location on the device where the file header is located.
- **Size** : Number of bytes/words/blocks in the file.
- **Position** : Current next-read/next-write pointers.
- **Protection** : Access control on read/write/execute/delete.
- **Usage** : Time of creation, access, modification etc.

Single-Level Directory Systems

The simplest form of directory system is having one directory containing all the files. Sometimes it is called the **root directory**, but since it is the only one, the name does not matter much.

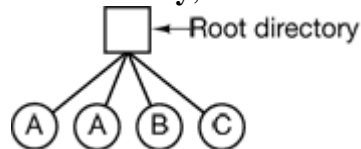


Figure . A single-level directory system containing four files, owned by three different people, A, B, and C

The problem with having only one directory in a system with multiple users is that different users may accidentally use the same names for their files. For example, if user A creates a file called *mailbox*, and then later user B also creates a file called *mailbox*, B's file will overwrite A's file. Consequently, this scheme is not used on multiuser systems any more, but could be used on

a small embedded system, for example, a system in a car that was designed to store user profiles for a small number of drivers.

Two-level Directory Systems

To avoid conflicts caused by different users choosing the same file name for their own files, the next step up is giving each user a private directory. In that way, names chosen by one user do not interfere with names chosen by a different user and there is no problem caused by the same name occurring in two or more directories. This design leads to the system as shown in Figure below. This design could be used, for example, on a multiuser computer or on a simple network of personal computers that shared a common file server over a local area network.

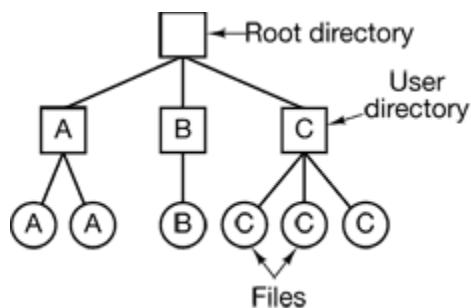


Figure A two-level directory system. The letters indicate the owners of the directories and files.

Hierarchical Directory Systems

The two-level hierarchy eliminates name conflicts among users but is not satisfactory for users with a large number of files. Even on a single-user personal computer, it is inconvenient. It is quite common for users to want to group their files together in logical ways. A professor for example, might have a collection of files that together form a book that he is writing for one course, a second collection of files containing student programs submitted for another course, a third group of files containing the code of an advanced compiler-writing system he is building, a fourth group of files containing grant proposals, as well as other files for electronic mail, minutes of meetings, papers he is writing, games, and so on. Some way is needed to group these files together in flexible ways chosen by the user.

What is needed is a general hierarchy (i.e., a tree of directories). With this approach, each user can have as many directories as are needed so that files can be grouped together in natural ways. This approach is shown in Fig. below. Here, the directories A, B, and C contained in the root directory each belong to a different user, two of whom have created subdirectories for projects they are working on.

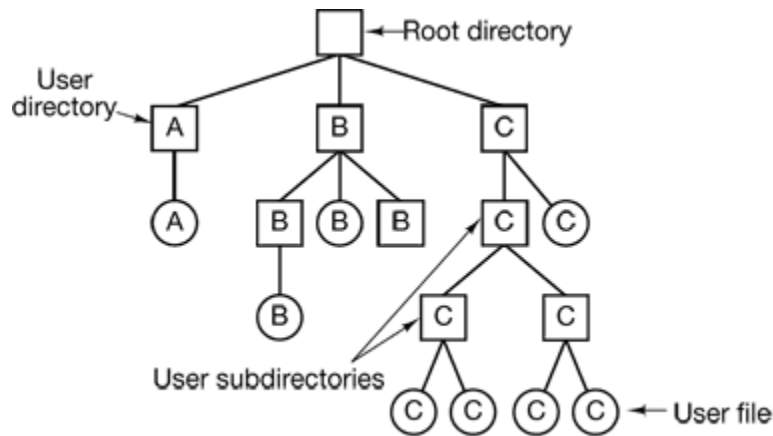


Figure A hierarchical directory system

8.2. FILE SYSTEM IMPLEMENTATION

Users are concerned with how files are named, what operations are allowed on them, what the directory tree looks like, and similar interface issues. Implementors are interested in how files and directories are stored, how disk space is managed, and how to make everything work efficiently and reliably.

8.2.1 File System Layout

File systems are stored on disks. Most disks can be divided up into one or more partitions, with independent file systems on each partition. Sector 0 of the disk is called the **MBR (Master Boot Record)** and is used to boot the computer. The end of the MBR contains the partition table. This table gives the starting and ending addresses of each partition. One of the partitions in the table is marked as active. When the computer is booted, the BIOS reads in and executes the MBR. The first thing the MBR program does is locate the active partition, read in its first block, called the **boot block**, and execute it. The program in the boot block loads the operating system contained in that partition. For uniformity, every partition starts with a boot block, even if it does not contain a bootable operating system.

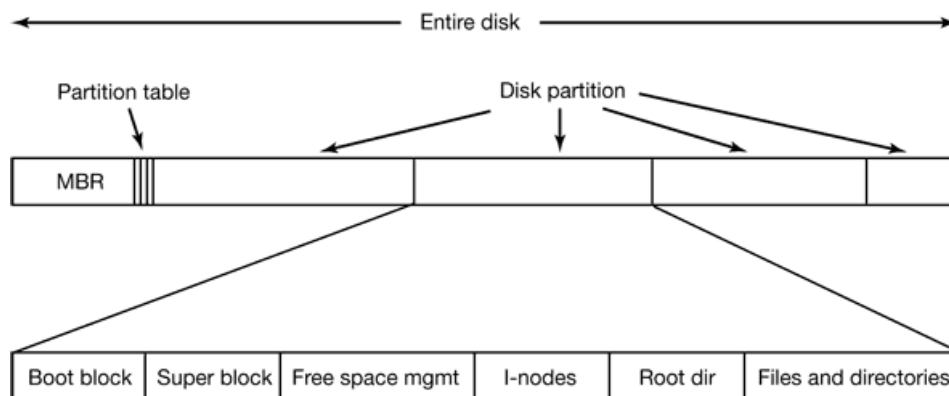


Figure A possible file system layout

8.2.2 Implementing Files

Probably the most important issue in **implementing file storage** is keeping track of which disk blocks go with which file. Various methods are used in different operating systems. Some of them are explained below:

i. Contiguous Allocation

The simplest allocation scheme is to store each file as a contiguous run of disk blocks. Thus on a disk with 1-KB blocks, a 50-KB file would be allocated 50 consecutive blocks. With 2-KB blocks, it would be allocated 25 consecutive blocks.

An example of contiguous storage allocation is shown in Fig. below. Here the first 40 disk blocks are shown, starting with block 0 on the left. Initially, the disk was empty. Then a file A, of length four blocks was written to disk starting at the beginning (block 0). After that a six-block file, *B*, was written starting right after the end of file A. Note that each file begins at the start of a new block, so that if file A was really $3\frac{1}{2}$ blocks, some space is wasted at the end of the last block. In the figure, a total of seven files are shown, each one starting at the block following the end of the previous one. Shading is used just to make it easier to tell the files apart.

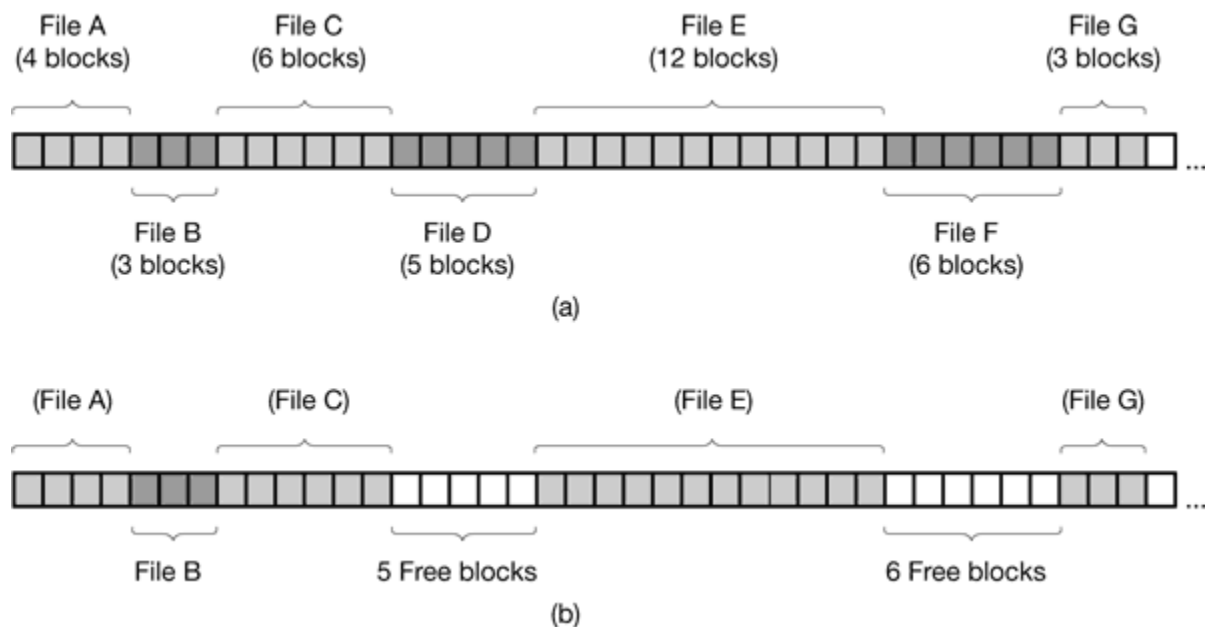
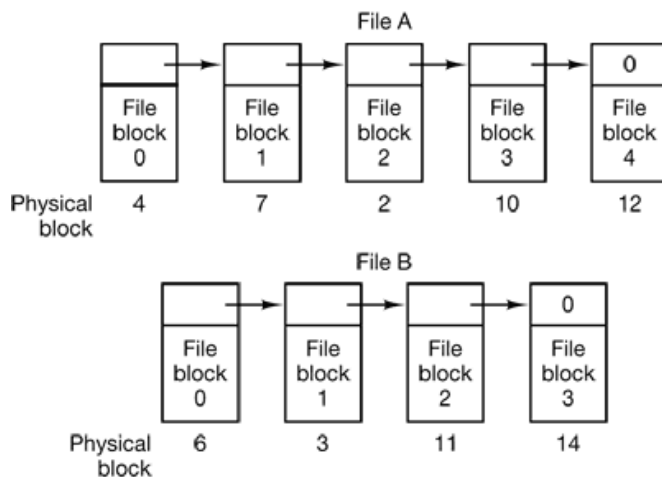


Figure. (a) Contiguous allocation of disk space for seven files. (b) The state of the disk after files *D* and *F* have been removed

ii. Linked List Allocation

The second method for storing files is to keep each one as a linked list of disk blocks, as shown in Fig. below. The first word of each block is used as a pointer to the next one. The rest of the block is for data.

Unlike contiguous allocation, every disk block can be used in this method. No space is lost to disk fragmentation (except for internal fragmentation in the last block). Also, it is sufficient for the directory entry to merely store the disk address of the first block. The rest can be found starting there.



iii. Linked List Allocation Using a Table in Memory (i.e. Indexed Allocation)

Disadvantages of the linked list allocation can be eliminated by taking the pointer word from each disk block and putting it in a table in memory. Such a table in main memory is called a **FAT (File Allocation Table)**.

- Provides solutions to problems of contiguous and linked allocation.
- An index block is created having all pointers to files.
- Each file has its own index block which stores the addresses of disk space occupied by the file.
- Directory contains the addresses of index blocks of files.

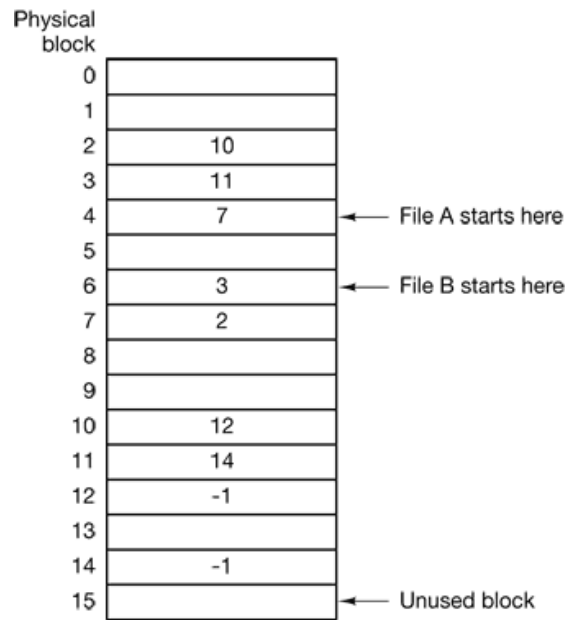


Figure Linked list allocation using a file allocation table in main memory.

8.3 File Sharing and Locking

The Concept of File Sharing and Locking is used in the Multi-users Environment, or in which there are many users who are requesting for a single file, to perform some operations.

With the help of file sharing, many users can perform the operations on the single file but there must be some mechanisms for controlling the access to the single file.

File locking is the act of ensuring that when you access a file, usually via a software application, no one can change the file until you are done examining it. If you want to modify the file, then file locking ensures that no one else can examine or modify the file until you are done modifying it.

File Locking is used when a single user is performing operations on the file so that other users cannot change the contents of the file. There are three types of Lock:-

- 1) **Read Only :** This Lock is used when all the users are reading the contents from the single file and no one can be able to change the contents on the file.
- 2) **Linked Shared:** The Linked Shared Locked is used when there are many users working on a single file and when the various users are requesting to change the contents of the file, then the change is performed on the file in a **Linear Order**.
- 3) **Exclusive Lock:** This is very important Lock. When the multiple users are working on the single file, the file is locked when any user is going to change the data of the file. Only single user can change file's content and that file will be locked for other users.