Early history

-1940s to 1950s

- 1940: the earliest electronic digital computers did not include operating system. Machines of the time were so primitive (ancient).
- 1950: the systems generally executed one job at a time. A job constituted the set of program instructions. These computers were called single-stream batch-processing systems. The operating systems reduced interjob transition times; programmers were required to directly control system resources.

- The 1960s

- It is also called the batch processing systems but using resources more efficiently by running several jobs at once.
- The systems improved resource utilization by allowing one job to use the processor while other jobs used peripheral devices.
- Processor bound job or compute bound job means jobs that mainly used the processor.
- I/O bound job means mainly used peripheral devices.
- Multiprogramming: systems that managed several jobs at once. The operating system rapidly switches the processor from job to job. Degree of multiprogramming or level of multiprogramming indicates how many jobs can be managed at once. Resources are shared among a set of processes.
- Interactive users: communicate with their jobs during execution via dumb terminals which were online.
- Timesharing: systems were developed to support simultaneous interactive users.
- Real-time systems: attempt to supply a response within a certain bounded time period.
- Virtual machine (VM) operating system: these systems were designed to perform basic interactive computing tasks for individuals, but their real value proved to be the manner in which they shared programs and data and demonstrated the value of interactive computing in program development environment.
- Process: to describe a program in execution in the context of operating system.
- Concurrent processes: execute independently but multiprogrammed systems enable multiple processes to cooperate to perform a common task.
- Turnaround time: the time between submission of a job and the return of its results, was reduced to minutes or seconds.

- Virtual memory: programs are able to address more memory locations than are actually provided in main memory, also called real memory or physical memory.

- 1970s

- The systems were primarily multimode multiprogramming systems that supported batch processing, time sharing and real-time applications
- Personal computers posted by early and continuing developments in microprocessor technology
- Communications between computers in local area networks (LANs) was made practical and economical by the Ethernet standard
- Security problems increased with growing volumes of information passing over vulnerable communications lines. Encryption received much attention

-1980s

- It was the decade of the personnel computers and the workstation
- Software such as spreadsheet programs, word processors, database packages and graphics packages
- Personnel computers proved to be easy to learn and use partially because of GUI(windows, icons, menus)
- Distributed computing became wide spread under client/server model. Clients request services and servers perform the requested services
- The software engineering field continued to evolve

-The 1990s

- Operating system designers developed techniques to protect computers from attacks
- Microsoft became the dominant in the 1990s. In 1981 Microsoft released DOS operating system. 1n the mid 1980 Microsoft developed its windows operating system, and then in 1990s released windows 3.0. 1993 release of Windows 3.1. After, that Windows 95, Windows 98, Windows NT, and Windows XP.
- Object technology: each software object encapsulates a set of attributes and methods. This allows applications to be built with components that can be reused in many applications. In object-oriented operating system objects represent components of the operating system and system resources. Object-oriented concepts were exploited to create modular operating system that were easier to maintain
- Open-source movement: open-source software is distributed with the source code, allowing individuals to examine and modify the software before compiling and executing (Linux operating system)
- Operating system became increasingly user friendly (GUI features)

-2000 and beyond

- Middleware: is a software that links two separate applications to communicate and exchange data via the internet
- Massive parallelism: number of systems has large of processors so that many independent parts of computations can be performed in parallel.
- Computing on mobile devices which are used for e-mail, web browsing

-application bases

- The operating system provides a series of application programming interface (API) calls which applications programmers and other operations use to accomplish detailed hardware manipulations and other operations. API provides system calls by which a user program instructs the operating system to do the work.

Application base is the combination of the hardware and the operating system environment in which applications are developed

−operating system environment

- Embedded systems are characterized by a small set of specialized resources that provide functionality to devices (phones). In embedded environments, efficient resource management is the key to building a successful operating system
- Real-time systems require tasks to be performed within a particular time frame. Real-time operating system must enable processes to respond immediately to critical events. Soft real-time systems ensure that real-time tasks execute with higher priority. Hard real-time system guarantee that all of their tasks complete on time
- Virtual machine (VM) is a software abstraction of a computer that often executes as a user application on the top of the native operating system. VM tend to be less efficient than real machines because they access the hardware indirectly or simulate hardware that is not actually connected to the computer. This increases the number of software instructions required to perform each hardware action
- Portability is the ability for software to run on multiple platforms

Definition of Operating System (OS) ch.1

OS is a set of programs that controls effectively the computer resources and makes them conveniently available to users i.e easy to use. Os is rather complicated software and hence designed usually by professional software companies and sold with computer system as part of it. During computer operation, some basic OS programs (Called Os Core or Kernel) are resident in main memory while others are stored on hard disk and loaded into memory when needed.

O/ S goals

1- The primary goal of an o/s is to make o/s convenient to use
2- A secondary goal is to use the computer H/W in an efficient manner.
3- Provide a connection between the user and computer resources.

Computer System Components

An o/s is an important part of almost every computer system . A computer system can be divided roughly into four components.

1- The hardware (CPU, Memory , I/O devices) .
2- Operating system(O/S).
3- Application programs(Assemply , Database compiler text , Editor)
4- Users

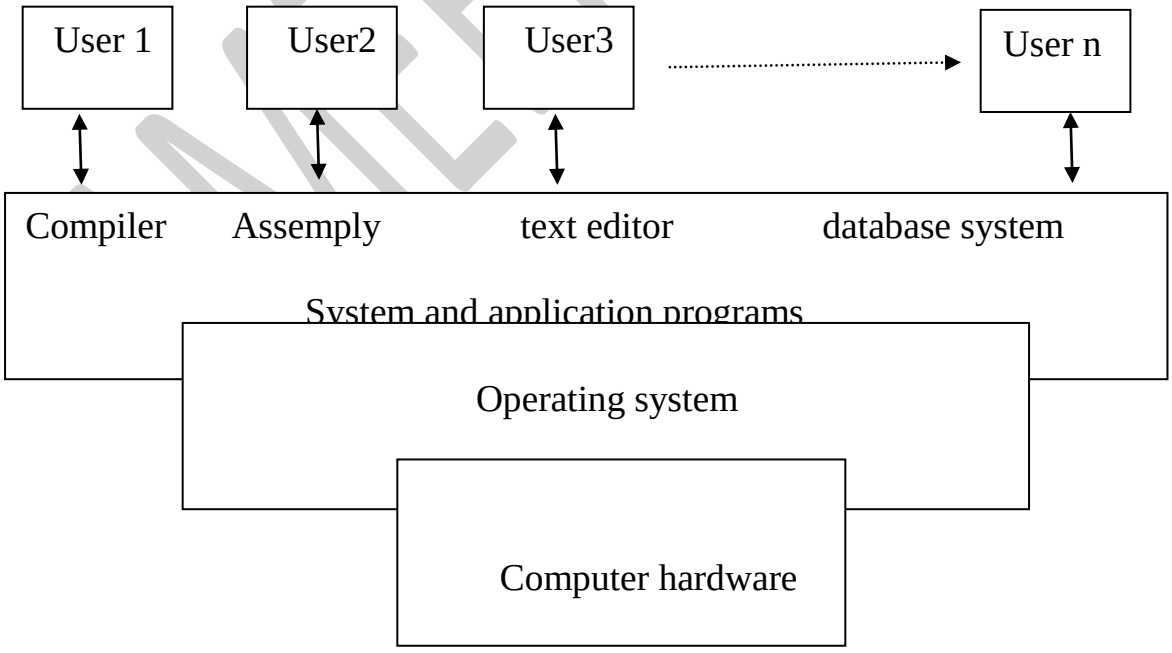


Fig 1 Abstracted view of the components of computer system

Functions of OS

The functions can be summarized as follows (will be explained later in more details):

1- Management of computer resources (processors, memory, disks, I/O devices, programs, etc.).
2-Scheduling resources among users (time sharing).
3-Protection of programs being executed in memory from one another.
4-Providing a proper user interface e.g Graphics User Interface (GUI).
5-File management.
6-Network communication.
7-Many others.

O/S Categories

The main categories of modern o/s may be classified into Meny groups , which are distinguished by the nature of inter action that take place between the computer and the users .

1- Batch system

In this type of o/s, users submit jobs on regular schedule (e.g, daily, weekly, monthly) to a central place where the user of such system did not interact directly with o/s. to speed up processing, jobs with similar needs were batched together and were run through the computer as a group. thus, the programs would have the programs with the operator, the major task of this type was to transfer control automatically from one job to the next. the o/s always resident in memory as in fig2

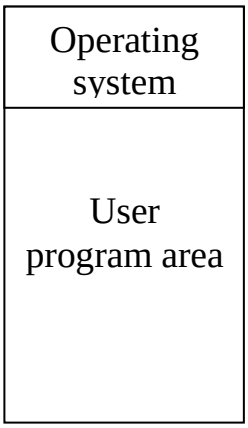


Fig 2: Memory layout for simple batch system

The output from each job would be sent back to the appropriate programmer.

- Advantage of batch system is very simple
- Disadvantages

-There is no direct interaction between the user and the job while the job is executing

- The delay between the job submission and the job completion (called turn around time) may result from amount of computing time needed.

Performance Development

o/s attempted to schedule computational activities to ensure good performance , where many facilities had been added to o/s some of these are :

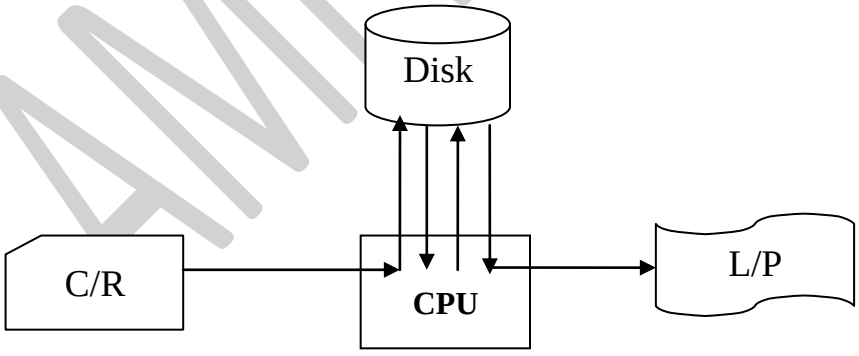
a- Spooling (Simulataneous Peripheral operation On- Line)

1- Spooling uses the disk as a very large buffer for reading as far a Head as possible on input devices and for storing output files until the output devices are able to accept them.

2- Spooling is now a standard feature of most O/S .

3- Spooling allows the computation of one job can overlap with the I/O of another jobs , therefore spooling can keep both CPU and I/O devices working as much higher rates.

4- The figure below show the spooling layout



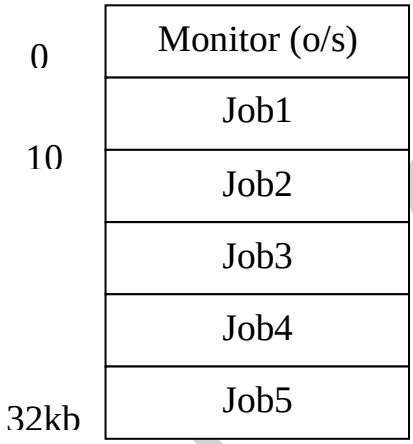
The spooling layout

b- Multiprogramming batch system

1- Spooling provides an important data structure called a job pool kept on disk. The O/S picks one job from the pool and begin to execute it.

2- In multiprogramming system , when the job may have to wait for any reason such as an I/O regrets , the O/S simply switches to and executes another job .when the second job need to wait the CPU is switches to another job and so on . Then the CPU will never be idle.

3- The figure below show the multiprogramming layout where the O/S keeps several jobs in memory at a time . This set of jobs is a subset of the jobs kept in the job pool.



The multiprogramming layout

2-Time Sharing System

This type of o/s provides on- line communication between the user and the system , where the user will give instruction to the o/s or to the program directly (usually from terminal) and receivers an intermediate response , therefore some time called an interactive system .

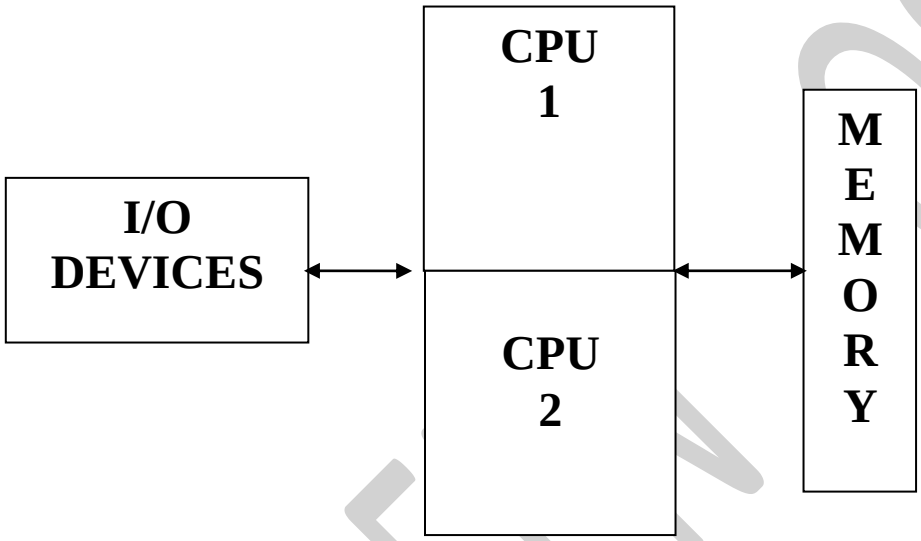
The time sharing system allows many users simultaneously share the computer system where little CPU time is needed for each user.

As the system switches rapidly from one user to the next user is given the impression that they each have their own computer , while actually one o/s shared among the many users.

- Advantage : reduce the CPU ideal time
- Disadvantage : more complex.

3-Parallel systems

1- Most systems to date are simple –processor system that is they have one main CPU.
2- There is a trend to have multiprocessor system , where such systems have more than one processor in close communication sharing the computer Bus , the clock , and some times memory and peripheral devices, in the figure below
3- The advantage of this type of systems :



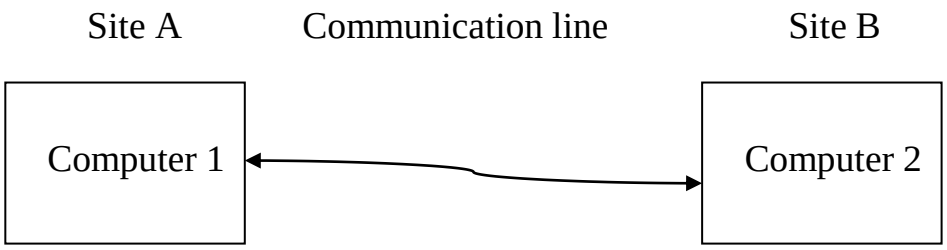
Parallel system layout

1- Increase throughput
2- The cost
3- Increase reliability

There are two types of processors in multiprocessors systems:-
a- Symmetric multiprocessor
b- Asymmetric multiprocessor

4-Distributed systems

1- A recent trend in C/S is to distribute computation among several processor.
2- In contrast to the parallel system , the processors do not share memory and clock.
3- The processors communicate with one another through various communication lines, such as high speed buses or telephone lines. This type of systems called a distributed system.



Distributed system

There are many reasons to build this type of system :-
1- Resource sharing
2- Computation speed up
3- Reliability
4- communication

5-Real time system

A real time system is used when there are rigid time requirement on the operation of a processor or the flow of data. A real time system guarantees that critical tasks complete on time . The secondary storage of any sort is usually limited , data instead being stored in short term memory (ROM)

There are two categories of real time system :
1- hard real time systems
2- soft real time systems

Computer System Operation:

A modern, general-purpose computer system consists of CPU and a number of device controllers that connected through a common bus that provides access to shared memory system, CPU other devices can execute concurrently competing for memory cycles.

Booting:

It is the operation of bringing operating system kernel from the secondary storage and put it in main storage to execute it in CPU. There is a program bootstrap which is performing this operation when computer is powered up or rebooted.
Bootstrap software: it is an initial program and simple it is stored in read-only memory (ROM) such as firmware or EEPROM within the computer hardware.
Jobs of Bootstrap program:

1- Initialize all the aspect of the system, from CPU registers to device controllers to memory contents.
2- Locate and load the operating system kernel into memory then the operating system starts executing the first process, such as "init" and waits for some event to occur.
The operating system then waits for some event to occur
Types of events are either software events (system call) or hardware events (signals from the hardware devices to the CPU through the system bus and known as an interrupt).
Note: all modern operating system are "interrupt driven".
Trap (exception): it is a software-generated interrupt caused either by an error (ex: division by zero or invalid memory access) or by a specific request from a user program that an operating system service be performed.
Interrupt vector (IV): it is a fixed locations (an array) in the low memory area (first 100 locations of RAM) of operating system when the interrupt occur the CPU stops what its doing and transfer execution to a fixed location (IV) contain starting address of the interrupt service routine(ISR), on completion the CPU resumes the interrupted computation.
Interrupt Service Routine: is it a routine provided to be responsible for dealing with the interrupt.

I/O Structure

Each I/O device connected to the C/S through its controller . A device controller maintains some local buffer storage and a set of special purpose registers [It is responsible for moving the data between the peripheral devices that is controls and its local buffer storage]

I/O Interrupts

To start an I/O operation the CPU loads the appropriate registers within the device controller . The controller examines the contents of these registers to determine what action to take. For example if it finds a read request the controller will start the transfer of data from the device to its local buffer . Once the transfer of data is complete the device controller informs the CPU that it has finished its operation.

DMA Structure

A high _speed device such as a tape , disk , or communication network may be able to transmit information at close to memory speeds , the CPU would need 2 microseconds to respond to each interrupt . That would not leave much time for process execution . To solve this problem Direct Memory Access (DMA) is used for high speed I/O devices After setting up buffer , pointers , and counters for I/O device , the device controller transfers an entire block of data directly to or from its own buffer storage to memory with no intervention by the CPU Only one interrupt is generated per block rather than one interrupt per byte (or word) generated for low speed devices
The DMA controller interrupts the CPU when the transfer has been completed

Storage Structure

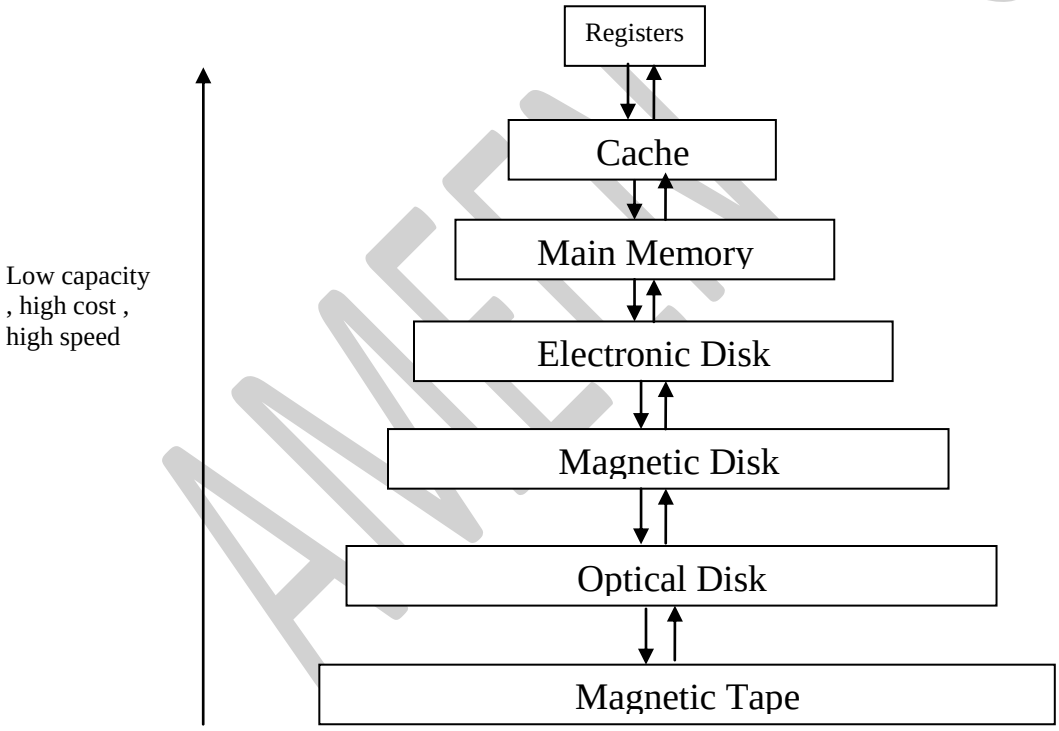
The programs must be in main memory to be executed . Main memory is the only large storage area that the processor can access directly . Each word in memory has its own address.
Interaction is achieved through a sequence of load or store instruction to specific memory addresses. The load instruction moves a word from main memory to an internal register within the CPU where as the store instruction moves the content of register to main memory.

We want the programs and data to reside in memory permanently .
This arrangement is not possible for the following two reasons:
A: Main memory is usually too small to store all needed programs and data permanently.
B: Main memory is volatile storage device that loses its contents when power is turned off or otherwise lost.

Therefore most C/S provide secondary storage as an extension of main memory. It be able to hold large quantities of data permanently . The most common secondary storage device is a magnetic disk which provides storage of both programs and data . There are other many media such as floppy disks , CD, ROMs , and DVDs.

Storage Hierarchy

The variety of storage systems in a C/S can be organized in hierarchy according to speed and their cost . figure below the higher levels are expensive , but are fast .



Hardware protection:

when we have single user any error occur to the system then we could determined that this error must be caused by the user program ,but when we begin to dealing with spooling ,multiprogramming, and sharing disk to hold many users data this sharing both improved utilization and increase problems .
In multiprogramming system, where one erroneous program might modify the program or data of another program, or even the resident monitor itself. MS-DOS and the Macintosh OS both allow this kind of error.
A properly designed operating system must ensure that an incorrect (or malicious) program cannot cause other program to execute incorrectly.
Many programming error are detected by the hardware these error are normally handled by the operating system.

Dual-Mode Operation:

To ensure proper operation, we must protect the operating system and all other programs and their data from any malfunctioning program.
The approach taken by many operating systems provides hardware support that allows us to differentiate among various modes of execution.
A bit, called the mode bit is added to the hardware of the computer to indicates the current mode: monitor (0) or user (1) with mode bit we could distinguish between a task that is executed on behalf of the operating system , and one that is executed on behalf of the user.

I/O Operation Protection:

A use program may disrupt the normal operation of the system by issuing illegal I/O instruction we can use various mechanisms to ensure that such disruption can not take place in the system.
One of them is by defining all I/O instructions to be privileged instructions. Thus users cannot issue I/O instructions directly they must do it through the operating system, by execute a system call to request that the operating system performing I/O in its behalf. The operating system, executing in monitor mode, check that the request is valid, and (if the request is valid) does the I/O requested. The operating system then returns to the user.

Memory Protection:

To insure correct operation, we must protect the interrupt vector and interrupt service routine from modification by a user program. This protection must be provided by the hardware, we need the ability to determine the range of legal addresses that the program may access, and to protect the memory outside that space. We could provided the protection by using two registers a base register and limit register

Base register hold the smallest legal physical memory address.

Limit register: contains the size of the range.

This protection is accomplished by the CPU hardware comparing every address generated in user mode with the registers. Any attempt by a program executing in user mode to access monitor memory or other users' memory results in a trap to the monitor, which treats the attempts as a fatal error.

CPU Protection:

In addition to protecting I/O and memory we must insure that the operating system maintains control. We must prevent the user from getting stuck in an infinite loop or not calling system services, and never returning control to the operating system. To accomplish this goal, we can use a timer.

Timer can be set to interrupt the computer after a specified period. The period may be fixed (for example, 1/60 second) or variable (for example, from 1 millisecond to 1 second) A variable timer is generally implemented by a fixed rate clock and a counter.

We can use the timer to prevent a user program from running too long Simple technique is to initialize a counter with the mount of time that a program is allowed to run.

Amore common use of timer is to implement time sharing. In the most case, the timer could be set to interrupt every N millisecond, where N is the time slice that each user is allowed to execute before the next user get control of the CPU. The operating system is invoked to perform housekeeping tasks.

This procedure is known as a context switching, following a context switch, the next program continues with its execution from the point at which it left off.

Operating System Structure ch 2

In the following lectures we will consider the components and services that are provided by different operating systems.

System Components

Many modern computer systems share the goal of supporting the following components:

- **Process management**

A process can be thought of a program in execution. A process needs certain resources to accomplish its task. Also the process various initialization values.
A process is the unit of work in a system. Such a system consists of a collection of processes, some of which are system processes others are user processes. All processes execute concurrently by multiplexing the CPU among them.
The OS responsible for the following activities in connection with process management:

- Creation and deletion both user and system processes.
- Suspending and resuming processes.
- Providing mechanisms for process synchronization.
- Providing mechanisms for process communication.
- Providing mechanisms for deadlock handling.

- **Main Memory Management**

The main memory is the central to the operation of a modern computer system. For a program to be executed it must mapped to absolute addresses and loaded to the M.M.
The OS responsible for the following activities in connection with M.M management:

- Keeping track of which parts of memory are currently being used and by whom.

• Deciding which processes are to be loaded into memory when memory space become available.
• Allocating and deallocating memory space as needed.

- **File Management**

For convenient use of the computer, the OS provides a uniform logical view of information storage. The OS abstracts from the physical properties of its storage device to define the logical storage unit, the file. A file is acollection of related information defined by its creator. These files are organized in directories to ease their use.
The OS responsible for the following activities in connection with file management:
• Creating and deleting files.
• Creating and deleting directories.
• Supporting primitives for manipulating files and directories.
• Mapping files onto secondary storage.
• Backing up files on stable storage media.

- **I/O System Management**

One of the purposes of OS is to hide the peculiarities of specific hardware devices. The OS responsible for the following activities in connection with I/O system management:
• A memory management component that includes buffering, caching and spooling.
• A general device driver interface.
• Derivers for specific hardware devices.

- **Secondary Storage Management**

The computer system must provide secondary storage to back up main memory because that are hold by MM are lost when power is switched of f and the MM is too small to accommodate all data programs. The OS

responsible for the following activities in connection with disk management:

- Free space management
- Storage allocation
- Disk scheduling

- **Networking**

A distributed system collects physically separate heterogeneous system into a single coherent system, providing the user with the access to various resources that the system maintain. Access to a shared resource allows computation speed up, increase functionality, increase data arability, and enhance reliability.

- **Protection System**

Protection is any mechanism for controlling the access programs, processes, or users to the resources defined by the computer system. This mechanism must provide means for specification of the controls to be imposed and means for enforcement. Protection can improve reliability by detecting latent errors at the interfaces between component subsystems.

- **Command Interpreter System**

Command Interpreter System is the interface between the user and the OS. Some of these Command Interpreter System are user friendly such as mouse based window and menus. In other shells commands are typed on a keyboard.

Operating System Services

An operating system provides an environment for the execution of programs. It provides certain services to programs and to the users of these programs. The specific services provided differ from one operating system to another but we can identify common classes. These operating system services are provided for the convenience of the programmer, to make the programming task easier.

1. Program execution
2. I/O operation
3. File system manipulation
4. Communications
5. Error detection
6. Resource allocation
7. Accounting
8. Protection

System Calls

System calls provide the interface between a process and the operating system. These calls are generally available as assembly language instructions and they are usually listed in the various manuals used by assembly language.

System Programs

System programs provide a convenient environment for program development and execution. Some of them are simply user interfaces to system calls others are considerably more complex. They can be divided into these categories:

- File management
- Status information
- File modification
- Programming language support
- Program loading and execution
- Communications

System Structure

A system as large and complex as a modern operating system must be engineered carefully if it is to function properly and to be modified easily. There are three different system structures:

- Simple structure
- Layered Approach
- Microkernal

System Design and Implementation

The problems and steps of system design and implementation are as follows:

- Design Goals
- Mechanisms and Policies
- Implementation

Processes ch3

In the following lectures we will consider the concepts of process.

Process Concepts

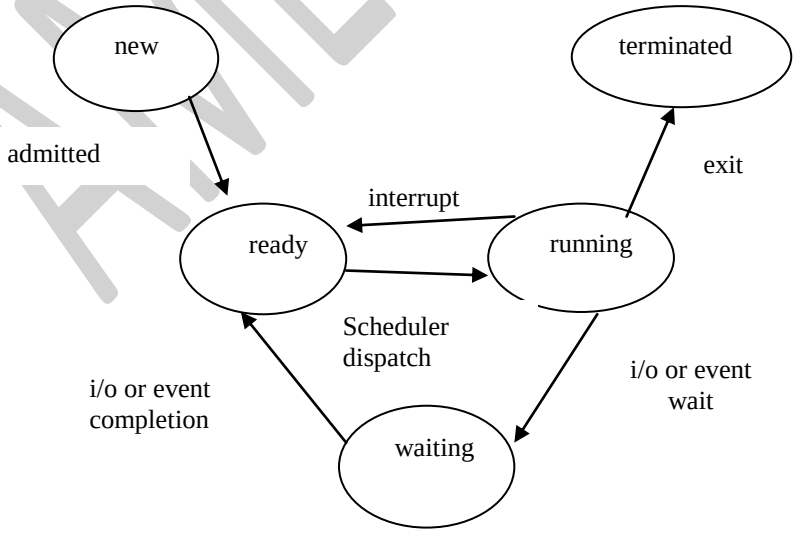
A process is a program in execution. A process is more than the program code, which is sometimes known as the text section. It also includes the current activity, as represented by the value of the program counter and the contents of the processor's registers.

Process state

The state of a process is defined in part by the current activity of the process. Each process may be in one of the following states:

- New
- Running
- Waiting
- Ready
- Terminated

This diagram is shown in fig 1 where we notice the followings:



1. At any instant of time, there is only one process running i.e allocated CPU time.
2. Exit from Running state may occur as a result of any of following events:

· Completion of process.
· Request of I/O service by a process.
· Time slice determined by interval timer has expired and hence an interrupt is activated which forces CPU to run OS instructions.

3. The transfer from Ready to Running state (dispatch) is carried out by OS according to certain criteria as will be shown later when studying " Processor Scheduling".

4. The term " Execution " means generally, " Ready", " Running", or " waiting".

Process Control Block (PCB)

Each process is represented by a process control block (PCB). A PCB contains many pieces of information associated with a specific process, such as:

- Process states
- Program counter
- CPU registers
- CPU scheduling information
- Memory management information
- Accounting information
- I/O status information

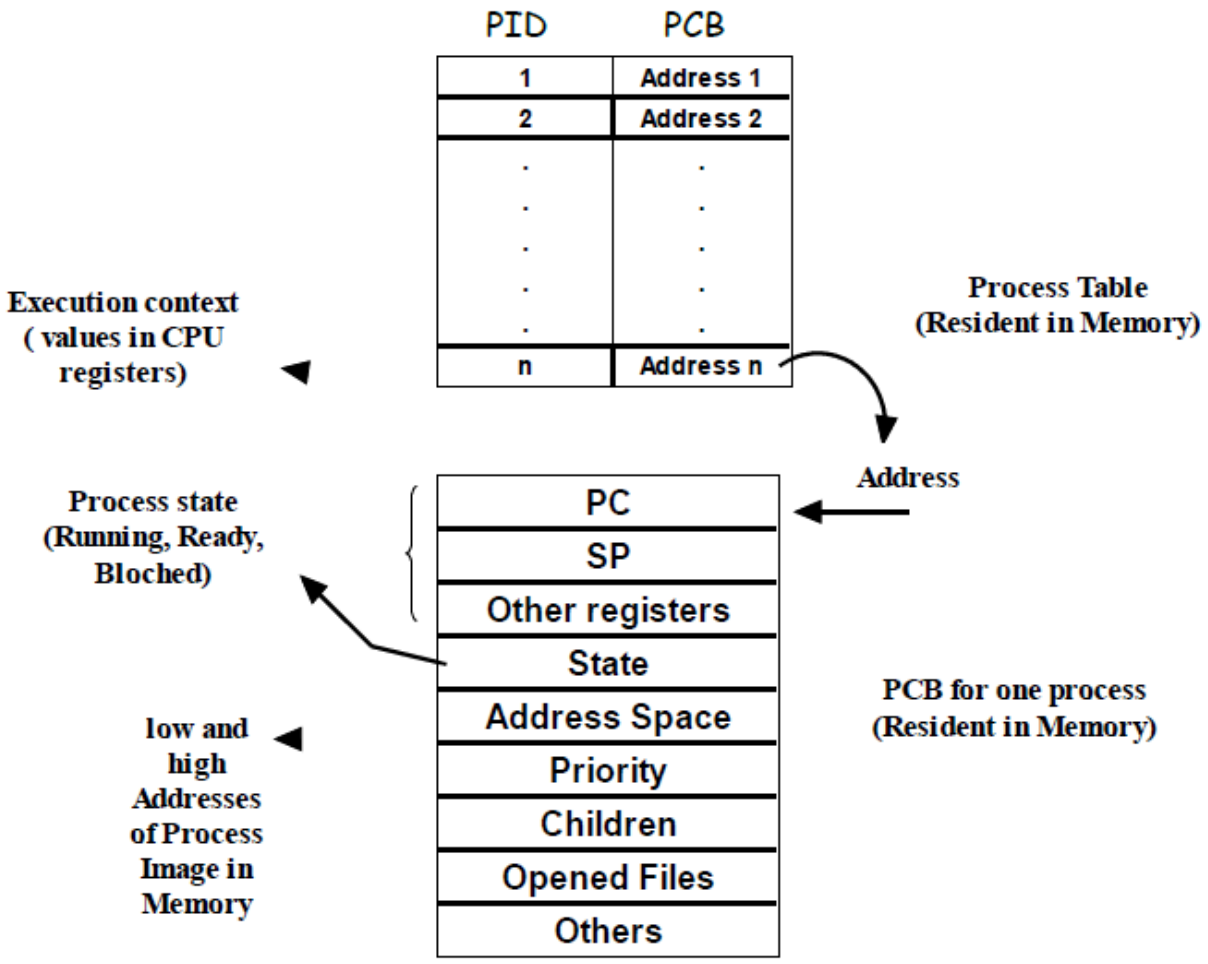


Fig 3.5 Process Table and PCB

Now, it is very useful to show the different components resident in memory in a from called " Memory Map" as shown in fig 3.6.

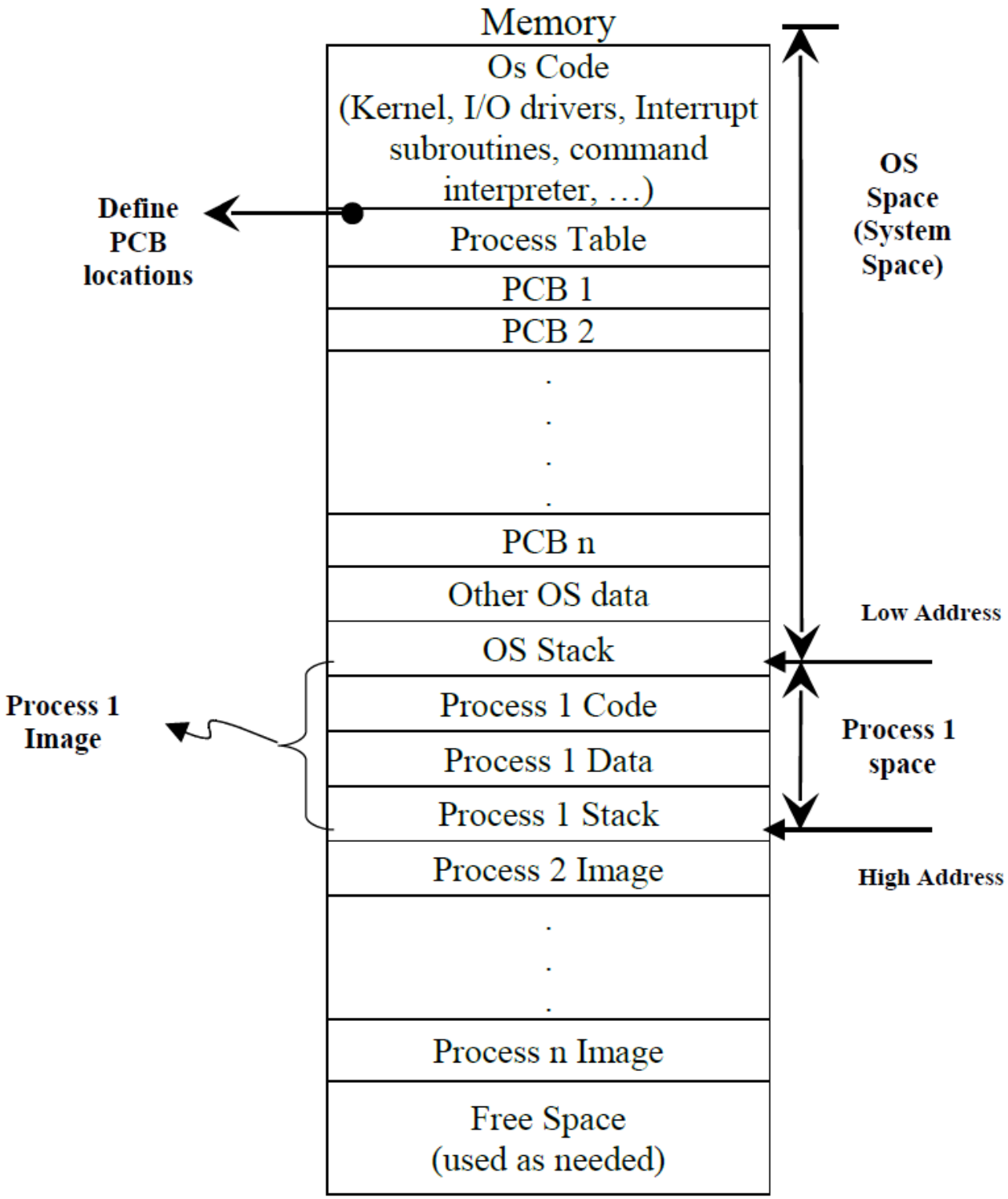


Fig 3.6 Memory Map.

Process Scheduling

A uniprocessor system can have only one running process. If more processes exist, as in multiprogramming system, there will be only one process running and the rest must wait until the CPU is free and can be rescheduled.

- **Scheduling Queues**

A new process as enter the system is put in a queue called ready queue. It waits in the ready queue until it is selected for execution. Once the process is assigned to the CPU and it is executing, one of the several event could occur:
The process could issue an I/O request, and then be placed in an I/O queue.
The process could create a new subprocess and wait for the termination.
The process could be removed forcibly from the CPU, as a result of an interrupt and be put back in the ready queue.

- **Scheduler**

A process migrates between the various scheduling queues throughout its lifetime. The operating system must select processes from these queues in some fashion. The selection process is carried out by the appropriate scheduler. There are two types of scheduling algorithms categorized according to the frequency of their execution.
• Long term scheduler (job scheduler) which selects a process from the job pool and load them into the MM.
• Short term scheduler (CPU scheduler) which select a process from the ready queue and allocate it to the CPU.

- **Context Switch**

Switching the CPU to another process requires saving the state of the old process and loading the saved state for the process. This task is known as a context switch.

Operation on Processes

The process in the system can execute concurrently, and they must be created and deleted dynamically.

- **Process Creation**

A process may create several new processes during the course of execution. The creating process is called a parent process, whereas the new processes are called the children.
When a process is created it obtains various resources and initialization values that may be passed along from the parent process to the child process.

- **Process Termination**

A process terminates when it finishes executing its final statement and asks the operating system to delete it. At that point the process may return data to its parent process and the OS deallocate all the physical and logical resources that are previously allocated to that process.

Cooperating Processes

The concurrent process executing in the operating system may be either independent processes that does not share any data or cooperating that affects each others.
We may provide an environment that allows process cooperation for several reasons:

- Information sharing
- Computation speedup
- Modularity
- Convenience

Inter process Communication

The cooperating processes can communicate in a shared memory environment. The scheme requires that these processes
share a common buffer pool. Another way to achieve the same effect for the operating system is provided via an interprocess communication (IPC).
IPC provides a mechanism to allow processes to communicate and synchronize their actions without sharing the same address space. This technique is useful for distributed systems. IPC is provided by a message passing system.

Inter Prcess Communication (IPC)

IPC is some times necessary but it presents two main problems:

1. Address violation problem :IPC means sharing some data (access common locations in memory). The shared data will be outside the address space of at least one process which, in turn, creates address violation problem. This problem may be solved by using "System Calls" for shared variables.
2. Write Access Problem : If the shared variable is of type Read/Write then another problem has to be solved in order to keep data integrity. This topic will be discussed later when studying "Asynchronous Concurrent Execution".

Definition of Thread in OS ch 4

A thread is a stream of instructions (line of control) within a process that can be executed independently of other threads. This means that a process may create at sometimes several threads that can be executed concurrently by several processors or each thread is dispatched for one time slice. Another definition of thread is a "Light Weight Process LWP" as it simulates the original process "Called Heavy Weight Process HWP" in the running for one time slice when it is dispatched.
As the thread runs in a process environment, therefore, it shares a process address space which means that communication between threads is very simple and variable sharing is possible without causing address violation problem

Motivations of Threads

From above discussion, we deduce that thread motivations can be summarized as follows:
1- Fast execution of a program as it can make use of several processors at the same time (case of multiprocessing) or dispatched more time slices (Case of Single Processor CPU)
2-Easy communication between threads as they share the same process address space that created them.
3- Easier design of some applications which have a lot of parallel activities such as a "Word" program.
 Note: The usefulness of multithreading can be made clear by considering a "Word" program. Each time a user types a character at the keyboard, OS receives a keyboard interrupt and issues a signal to the word program (process). The word process responds by storing the character in memory and displaying it on the screen. Because today's computers can execute hundreds of millions of instructions between successive keystroke, a word process can execute several other threads between keyboard interrupts. For example, a word process may detect misspelled words as being typed and periodically save a copy of document to disk. Each feature may be implemented by separate thread. As a result, the Word process (processor) can respond to keyboard interrupts even if one or more of its threads are blocked due to I/0 operation (e.g. saving copy of file to disk)

There are two types of programming languages
١-single threaded: Allows single thread of control in the program and hence concurrent activities are not possible within the same program. Examples of such language are: C, C++, VB, etc.

2-Multithreaded: Allows several threads of control in the program and hence concurrent activities are quite possible within the same program. Examples of such languages are: C#, VB.NET, JAVA, ADA, etc.

It is worth noting that "single threaded languages" are also called "non-threaded" or "sequential" while "multithreaded" are called "threaded" or "parallel."

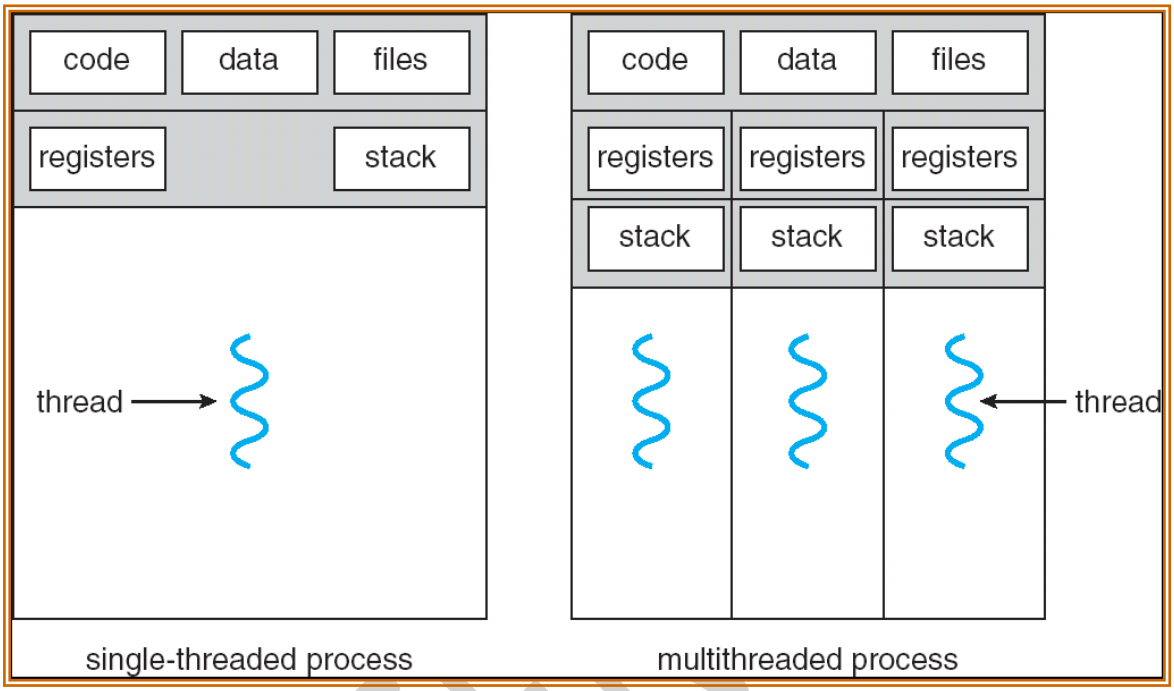


Figure 1: Single-threaded and multithreaded processes

Non-Threaded and Threaded Algorithms

Suppose we want to calculate the following expressions:

$Y = (a1+x)^3 + (a2+x)4$

where a1, a2 are constants and x is input variable. This calculation can be done as follows:

1 -Using Non-Threaded Algorithm:

The calculation is shown in fig bellow and we notice that it takes a total of 7 arithmetic operations

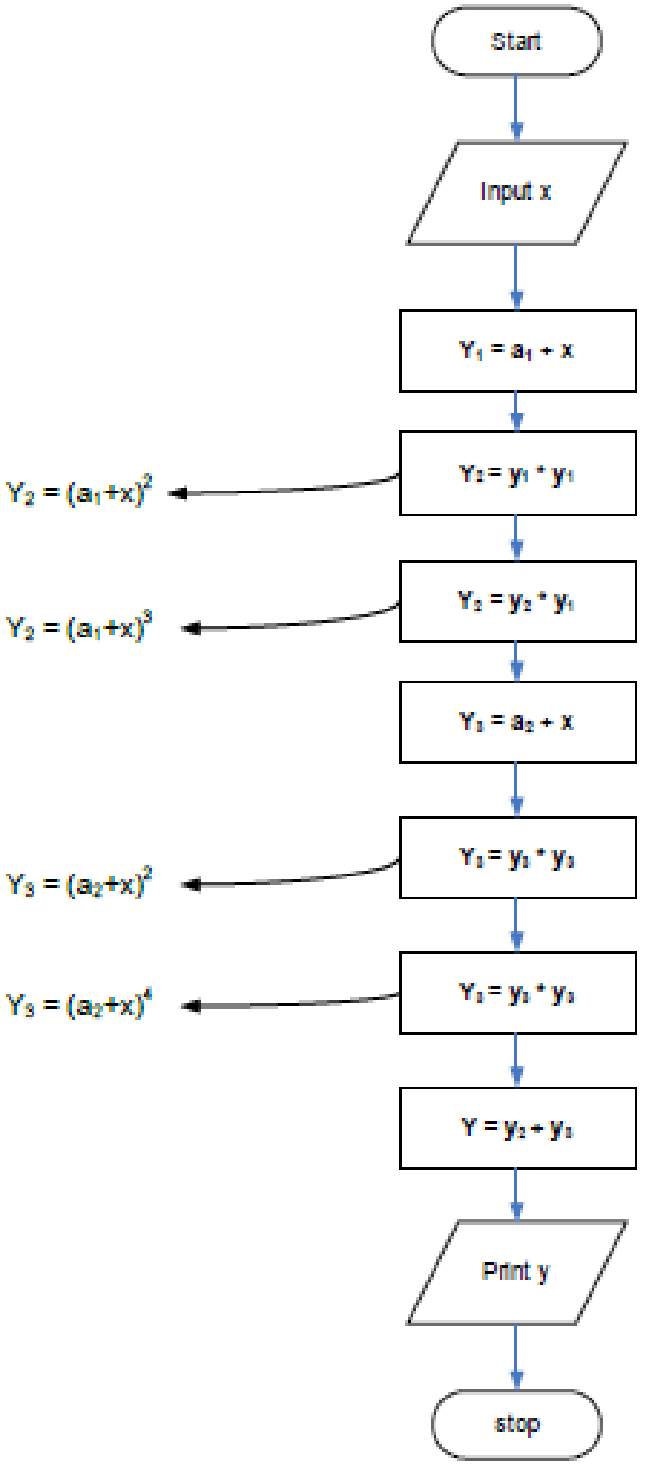


Fig2. Non Threaded Algorithm

2- Using Threaded Algorithm:

The calculation is shown in fig below The number of arithmetic operations in Thread1 is 3, and in Thread2 is 3.

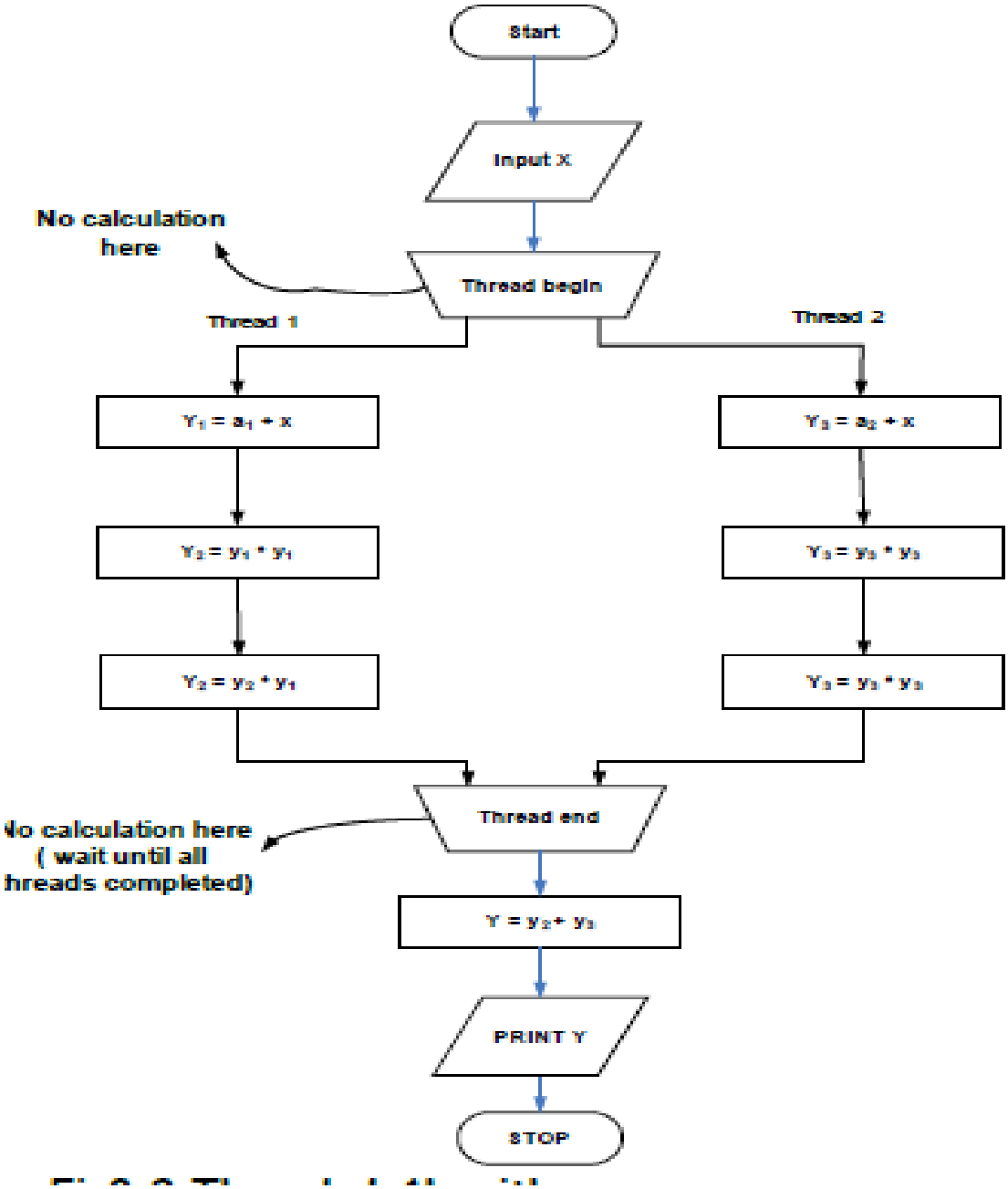


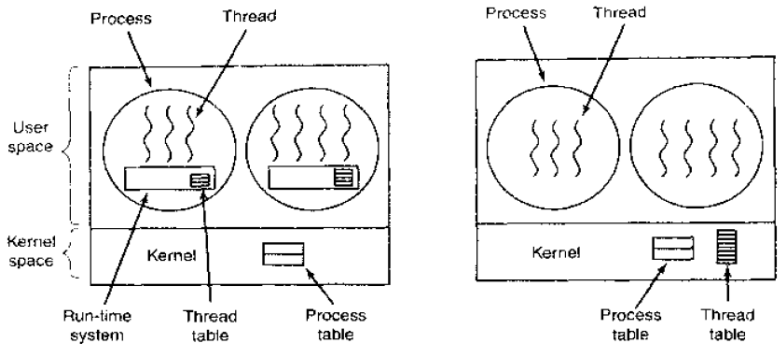
Fig3: Threaded Algorithm

Thread1 and Thread2 can be executed concurrently and hence the equivalent number of operations is 3 only. The total number of operations is 4 which is less than 7 needed in non threaded algorithm

Threading Models

1- User-Level Threads

The first method is to put the threads package entirely in user space, the kernel knows nothing about them, so User level threads perform threading operations in the user space, meaning that threads are created by runtime libraries that cannot execute privileged instructions or access kernel primitives directly.



(a) (b)

Figure 4: (a) A user-level threads package, (b) A threads package managed by the kernel.

2- Kernel-Level Threads

Kernel-level threads attempt to address the limitations of user-level threads by mapping each thread to its own execution context.

3- Combining User- level Threads and kernel-level Threads

Also called hybrid Threads, various ways have been investigated to try to combine the advantages of user-level threads with kernel-level threads. One way is use kernel-level threads and then multiplex user-level threads onto some or all of the kernel threads, as shown in figure bellow

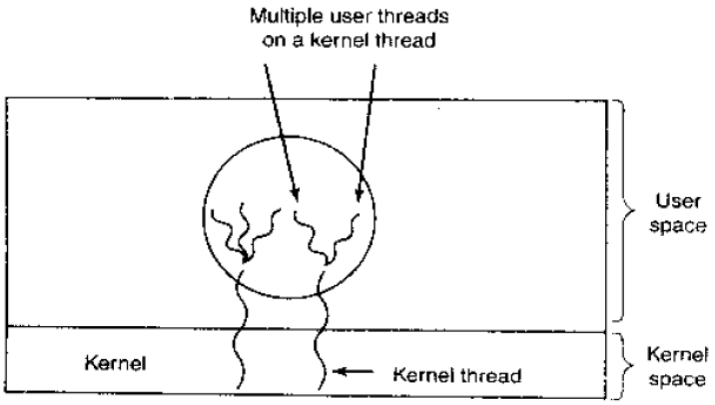


Figure 5: Multiplexing user-level threads onto kernel-level threads.

Comparison among three models

a-user-level threads

1- threads are created by runtime libraries that cannot execute privileged instructions or access kernel primitives directly.

2- The operating system treats each multithreaded process as a single execution context.

3- The user-level thread implementations are also called many-to-one thread mappings because the operating system maps all threads in a multithreaded process to a single execution context.

4- When a process employs user-level threads, user-level libraries perform scheduling and dispatching operations on the process's threads.

5- User level threads do not require the operating system support threads.

6- User level threads are more portable because of not relying on a particular operating system's threading API.

7- user level threads consume less resource than Kernel level threads

8- user-level threads don’t require that the operating system manage all threads in the system.

9- User level thread performance varies depending on the system and on process behavior and in multiprocessor User level threads do not scale well to

multiprocessor systems, so can result in suboptimal performance in multiprocessor systems.

b-kernel-level threads

1- threads are can execute privileged instructions or access kernel primitives directly.

2- The operating system creates a kernel thread that executes the user thread's instructions.

3- Kernel-level threads are often described as one-to-one threads mapping

4- mapping require operating system to provide each user thread with a kernel thread that the operating system can dispatch.

5- Kernel level threads require the operating system support threads.

6- software that employs kernel-level threads is often less portable than software employs user-level threads

7- Kernel level threads consume more resource than user level threads

8- Kernel-level threads require that the operating system manage all threads in the system.

9- kernel can manage each thread individually, meaning that the operating system can dispatch a process's ready threads even if one of its threads is blocked (improve performance).

c-hybrid threads

1- threads are can execute privileged instructions or access kernel primitives directly.

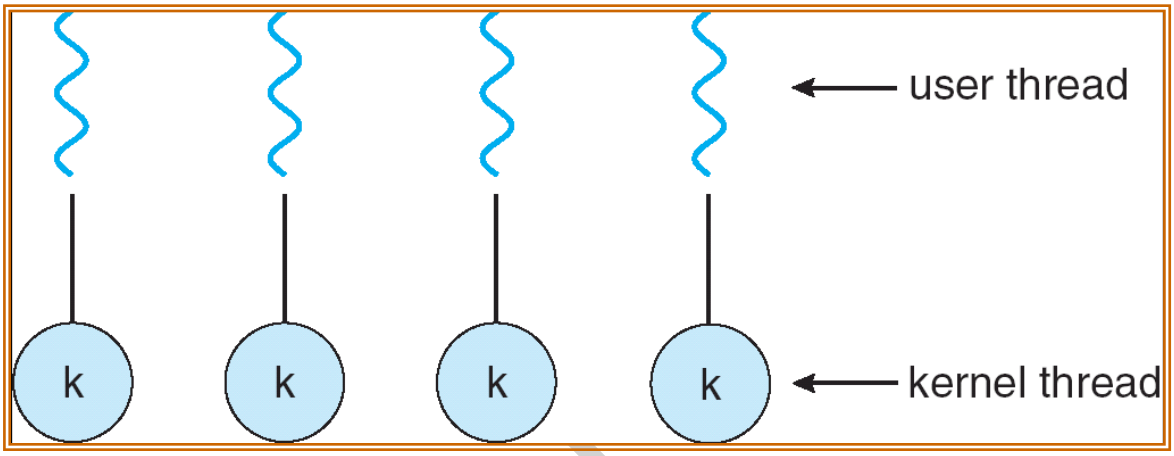
2- The operating system creates a kernel thread that executes the user thread's instructions.

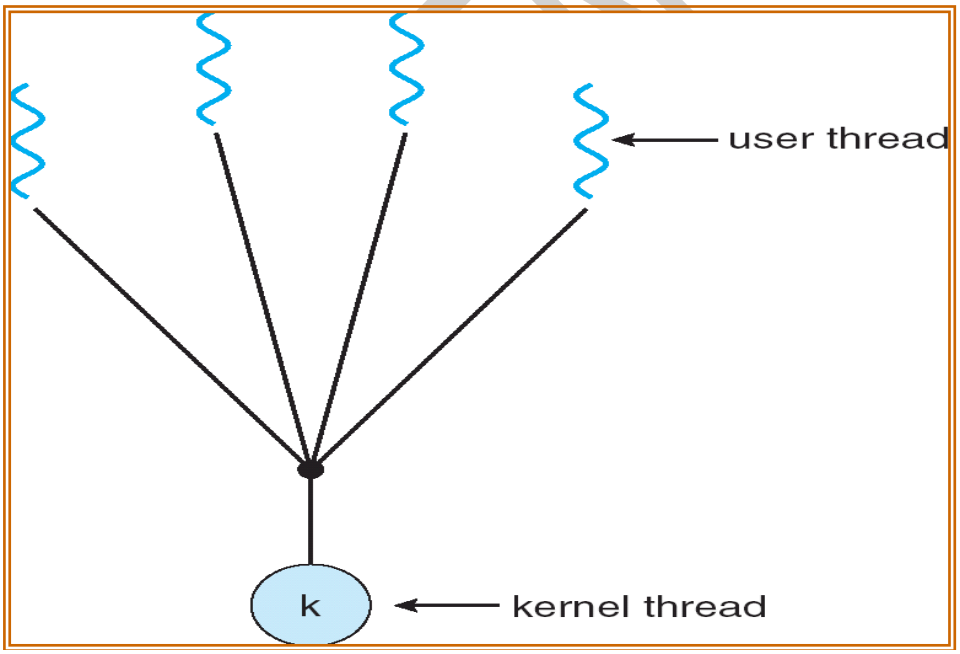
3- The combination is known as the many-to-many thread mapping as its name, this implementation maps many user-level-threads to a set of kernel-level

threads. Some refer to this technique as (m-to-n threads mapping)

4-Hybrid threads require the operating system support threads.

5- applications can improve performance by customizing the threading library's scheduling algorithm.





CPU Scheduling ch5

In the following lectures we will introduce the basic scheduling concepts and present several different CPU scheduling algorithms.

Scheduling Concepts
Scheduling is a fundamental operating system function. Almost all computer resources are scheduled before use. The CPU scheduling is central to operating systems.

CPU-I/O Burst Cycle

The success of CPU scheduling depends on the following observed property of processes: process execution consists of a cycle of CPU execution and I/O wait. Processes alternate between these two states. Process execution begins with a CPU burst. That is followed by I/O burst, then another CPU burst and so on. The last CPU burst will end with a system request to terminate execution.

CPU Scheduler
Whenever the CPU becomes idle, the operating system must select one of the processes in the ready queue to be executed. The selection process is carried out by the short term scheduler(CPU scheduler). The scheduler selects from among the processes in in memory that are ready to execute and allocates the CPU to one of them.

Scheduling Schemes

There are two scheduling schemes can be recognized:
- Preemptive scheduling
- Nonpreemptive scheduling

Under the nonpreemptive scheduling, once the CPU has been allocated to a process, the process keeps the CPU until it release the CPU either by terminating or by switching to the waiting state. On the other hand

Preemptive scheduling occure when the CPU has been allocted to a process and this process is interrupted by higher priority process. At this moment the executing process is stopped and returned back to the ready queue, the CPU is allocated to the higher priority process.

Dispatcher

It is the module that gives control of the CPU to the process selected by the CPU scheduler. This function involves:

- Switching Context
- Switching to user mode
- Jumping to the proper location in the user program to restart the program.

Scheduling Criteria

Many criteria have been suggested for comparing CPU scheduling algorithms. The criteria include the following:

- CPU Utilization
- Throughput
- Turned around Time
- Waiting time
- Response time

The optimization criteria are as follow:

- Max CPU utilization
- Max throughput
- Min turnaround time
- Min waiting time
- Min response time

Scheduling Algorithm

Here we will mention some of the CPU scheduling algorithms that are used in different operating systems

1• First Come First Served (FCFS)

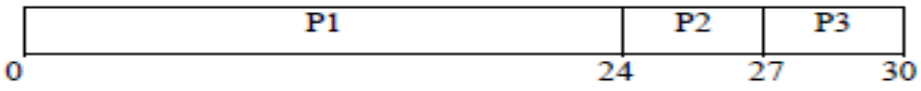
With this algorithm the process that requests the CPU first is allocated the CPU first. The implementation of the FCFS policy is easily managed with FIFO queue. When a process enters the ready queue, its PCB is linked onto the tail of the queue. When the CPU is free, it is allocated to the process at the head of the queue.

The average waiting time under the FCFS policy is often quite long. Consider the following set of processes that arrive at time 0, with the length of CPU burst time given in millisecond:

Example 1:

Process	Burst time
P1	24
P2	3
P3	3

The Gantt Chart is as follows:

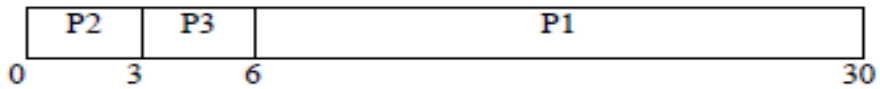


The average waiting time = (0+24+27)/ 3=17 millisecond

The average completion time = (24+27+30)/3 = 27 millisecond

Example 2:

If the processes arrive in the order P2, P3, P1 the result will be shown in the following Gantt Chart:



The average waiting time = (0+3+6)/ 3=3 millisecond

The average completion time = (3+6+30)/3 = 13 millisecond (compared to 27)

Thus the average waiting time under FCFS policy is not the minimal.

2. Shortest Job First Scheduling (SJF)

This algorithm associate with each process the length of the latter's next CPU burst. When the CPU is available, it is assigned to the process has the smallest next CPU burst. If two processes have the same length , FCFS scheduling is used to break this tie.
As an example consider the following set of processes with the length of the CPU burst given in millisecond:

Example **3** using **SJF:**

Process	Burst time
P1	6
P2	8
P3	7
P4	3

Gantt chart:

0 3 9 16 24

P4	**P1**	**P3**	**P2**

W. T p1 = 3-0 = 3
W. T p2 = 16-0 = 16
W. T p3 = 9-0 = 9
W. T p4 = 0-0 = 0

Average wait time = (3 + 16 + 9 + 0) / 4 = 7 milliseconds
The average completion time = (9+24+16+3)/4 =13 milliseconds
The average waiting time in SJF is the optimal that it gives the minimum average waiting time.
The SJF is either preemptive or non-preemptive.

- ***Non preemptive:*** once CPU given to the process it cannot be preempted until completes its CPU burst.
- ***Preemptive:*** if a new process arrives with CPU burst length less than remaining time of current executing process, preempt. This scheme is known as the **Shortest Remaining Time First (SRTF)**.

As an example consider the following set of processes with the length of the CPU burst given in millisecond:

Example **4** using **SJF:**

Process	Burst Time	Arrival Time
P0	4	0
P1	4	1
P2	2	2
P3	1	5
P4	3	7

Gantt chart:

0		4		6		7		10		14
P0		**P2**		**P3**		**P4**		**P1**		

W. T p0 = 0-0 = 0
W. T p1 = 10-1 = 9
W. T p2 = 4-2 = 2
W. T p3 = 6-5 = 1
W. T p4 = 7-7 = 0
Average wait time = (0 + 9 + 2 +1+ 0) / 5 = 2.4 milliseconds

3. Priority Scheduling Algorithm

In this algorithm a priority is associated with each process and the CPU is allocated to the process of the highest priority. We use the low numbers to represent high priority.

As an example consider the following set of processes with the length of the CPU burst given in millisecond

Example **5** using **priority**:

Process	Burst time	Priority
P1	10	3
P2	1	1
P3	2	4
P4	1	5
P5	5	2

Gantt chart:

P2	P5	P1	P3	P4

0 1 6 16 18 19

W. T p1 = 6-0= 6
W. T p2 = 0-0 = 0
W. T p3 = 16-0 = 16
W. T p4 = 18-0 = 18
W. T p5 = 1-0 = 1
Average wait time = (6 + 0 + 16 +18+ 1) / 5 = 8.2 milliseconds

Example **6** using **non-preemptive priority:**

Process	Burst time	Arrival	Priority
P0	4	0	3
P1	2	3	2
P2	5	4	1
P3	6	10	0
P4	8	11	5

Gantt chart:

0 4 9 11 17 25

P0	P2	P1	P3	P4

W. T p0 = 0- 0= 0
W. T p1 = 9-3 = 6
W. T p2 = 4 - 4 = 0
W. T p3 = 11-10 = 1
W. T p4 = 17- 11 = 6

Average wait time = (0 + 6 + 0 +1+ 6) / 5 = 2.6 millisecond

Example **7** using **Preemptive Priority:**

Process	Burst time	Arrival	Priority
P0	5	0	3
P1	8	2	2
P2	3	3	1
P3	6	4	0
P4	2	8	4

Gantt chart:

0	**2**	**3**	**4**	**10**	**12**	**19**	**22**
P0	**P1**	**P2**	**P3**	**P2**	**P1**	**P0**	**P4**

W. T p0 = 19-2-0 = 17
W. T p1 = 12 -2-1 = 9
W. T p2 = 10 – 3- 3 = 4
W. T p3 = 4- 4 = 0
W. T p4 = 22- 8 = 14
Average wait time = (17+9+ 4 + 0 +14) / 5 = 8.8 milliseconds

Priority scheduling can be either Preemptive or non preemptive, when a process arrives the ready queue, its priority is compared with the priority of the currently running process. A preemptive priority will preempt the CPU if the priority of the newly arrived process is higher than the priority of the currently running process. A non preemptive priority scheduling will put the new
process with the higher priority than the priority of the currently running process at the head of the ready queue.

4.Round Robin Scheduling Algorithm

The Round Robin algorithm is designed especially for time sharing system. It is similar to FCFS but preemption is added switch between processes. A small unit of time called time quantum (or time slice) is defined. A time quantum is generally from 10 to 100 milliseconds. The ready queue is treated as a circular queue, allocated the CPU to each process for a time interval of up to 1 time quantum.
The CPU scheduler picks the first process from the ready queue, sets a timer to interrupt after 1 time quantum, and dispatch the processes. One

of two things will then happen. The processes may have a CPU burst of less than 1 time quantum. In this case, the processes itself will release the CPU. The scheduler will then proceed to the next process in the ready queue. Otherwise, if the CPU burst of the currently running processes is longer than 1 time quantum, the timer will go off and will cause an interrupt to the operating system. A context switch will be executed, and the process will be put at the tail of the ready queue. The CPU scheduler will then select the next process in the ready queue.

As an example consider the following set of processes with the length of the CPU burst given in millisecond:

Example 8 using RR: quantum=4

Process	Burst time
P1	24
P2	3
P3	3

Gantt chart:

0 4 7 10 14 18 22 26 30

P1	**P2**	**P3**	**P1**	**P1**	**P1**	**P1**	**P1**

W. T p1 = 26-4-4-4-4-4 = 6

W. T p2 = 4-0 = 4

W. T p3 = 7-0 = 7

Average wait time = (6 + 4 +7) / 3 = 5.66 milliseconds

Example 9 using **preemptive RR**: quantum=2

Process	Burst time	Arrival time
P0	6	0
P1	8	1
P2	4	7
P3	2	9
P4	10	11

0	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30

P0	P1	P0	P1	P2	P3	P4	P0	P1	P2	P4	P1	P4	P4	P4

W. T p0 = 14- 2 – 2- 0= 10
W. T p1 = 22 -2-2-2-1 = 15
W. T p2 = 18 -2-7 = 9
W. T p3 = 10 -9 = 1
W. T p4 = 28 -2-2-2-2-11 = 9

Average wait time = (10 + 15 +9+1+9) / 5 = 8.8 milliseconds

Comparison among Scheduling Algorithms

Algorithms	Policy type	Dis advantages	Advantages
FCFS	Non preemptive	Average waiting time is often quite long.	Easy to implement.
SJF	Non preemptive Or preemptive	Knowing the length of the next CPU request.	Gives minimum average waiting time.
Priority	Non preemptive Or preemptive	Blocking or starvation.	1-Simplicity. 2-support for priority.
R.R	preemptive	If the set time is too long, then the system may become unresponsive, time wasting and would emulate First Come First Served.	It is easy to implement in software

DeadLocks ch6

A Computer System consist of a finite number of resources to be distributed among a number of computing processes. The resources are partitioned into several types, each of which consists of some number of identical instances memory, CPU, cycles, files, and I/O devices are examples of resource types.

Under the normal of operation, a process may utilize a resource in only the following sequence:

a- **Request:** If the request cannot be granted immediately then the requesting process must wait until it can acquire the resource.
b- **Use:** The process can operate on the resource (for example, if the resource is a printer, the process can print on the printer).
c- **Release:** The process releases the resource.

Deadlock definition

A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set .
Example
Consider a C/S with two tape drives, suppose that there or two processes each holding one of these tape drives. If each process now requests another tape drive, the two processes will be in a deadlock state, see the fig bellow.

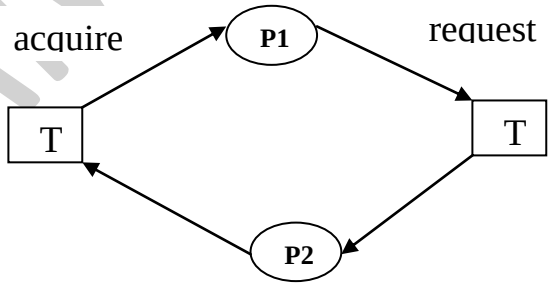


Fig1 : Deadlock

Deadlock necessary conditions

In a deadlock , processes never finish executing and system resources are tied up , preventing other processes from ever starting.
A deadlock situation can arise if and only if the following four conditions hold simultaneously in a system . these condition are :

a- Mutual Exclusion

At least one resource is held in a non –sharable mode that is only one process at a time can use the resource . if another process requests that resource the requesting process must be delayed until the resource has been released

b- Hold and Wait

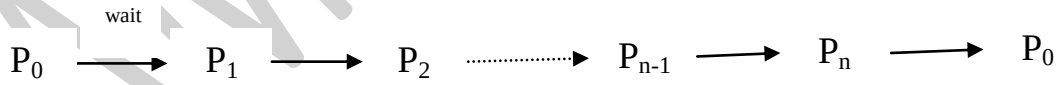
There must exist a process that is holding at least one resource and is waiting to acquire additional resources are currently being held by other processes.

c- No Preemption

Resources can not be preempted , that is a resource can be released only voluntarily by the process holding after that process has completed its task.

d- Circular Wait

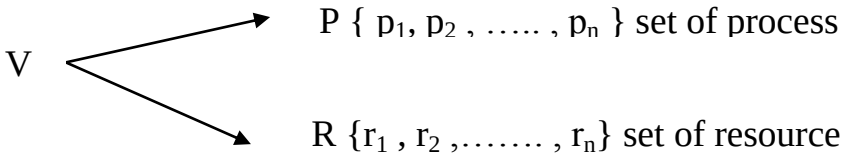
There must exist a set $\{p_0,p_1,\ldots,p_n\}$of waiting processes such that p_0 is waiting for resource that is held by p_1 , p_1 is waiting for a resource that is held by p_2, $\ldots p_{n-1}$ is waiting for a resource that is held by p_n , and p_n is waiting for a resource that is held by p_0 .



Resource – Allocation Graph(RAG)

Deadlocks can be described more precisely in terms of a directed graph called (RAG).

RAG = (V,E) where V is a set of vertices and E is a set of edges.



Each element in the set E of edges is an ordered pair (p_i,r_j)or (r_j, p_i) where pi is an a process ($p_i \in P$) and r_j is a resource type ($r_j \in R$)

If $(p_i , r_j) \in E$ then p_i —Request edge→ R_j and if $(r_j , p_i) \in E$ then r_j —Assignment edge→ p_i

Graphically we represent each process p_i as a circle and each resource type R_j as a square . Since resource type R_j may have more than one instance we represent each such instance as a dot within a square , see fig bellow .

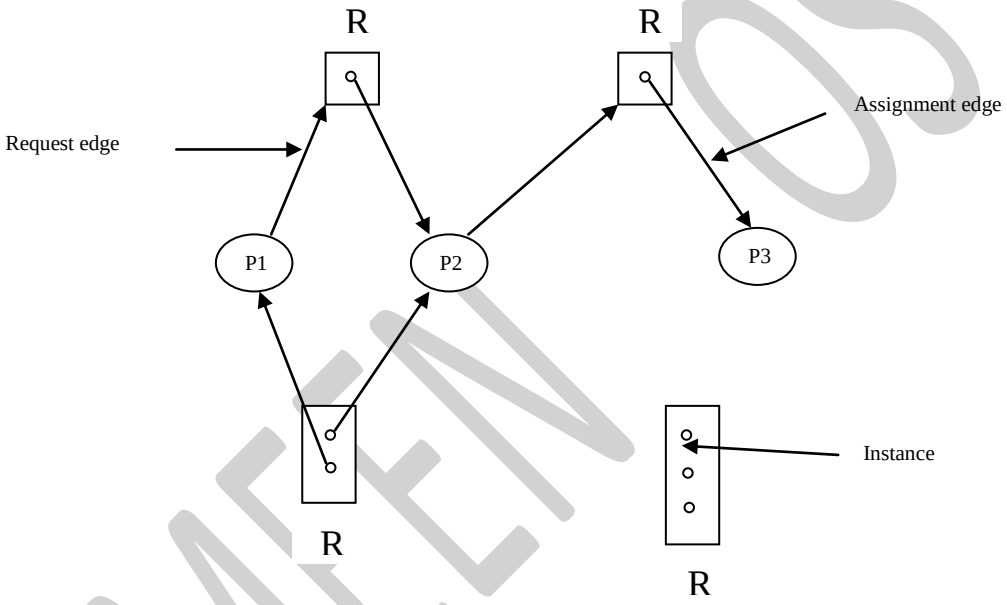


Fig2: Resource – allocation graph

The RAG in this fig depicts the following situation
The sets P, R and E :

- P = {p1 , p2 p3}
- R = {r1, r2 , r3 , r4 }
- E= { (p1,r1) , (p2,r3) , (r1,p2) , (r2, p2) ,(r2, p1) , (r3,p3)}

Resource states:
R1= 1, R2= 2, R3= 3 , and R4 = 4 instances.

Process states:
p3→P2 , r3→P1 , r2→P2 , r2→r3 , r1→r1, P2→P1

By using a RAG it can be easily shown that if the graph contain no cycles, than no process in the system is deadlocked. If on the other hand the graph contains a cycle than a deadlock may exist .

If each resource type has exactly one instance then a cycle implies that a deadlock has occurred. If the cycle involves only a set of resource types each of which has only a single instance then a deadlock has occurred.
Each process involved in the cycle is deadlock.
In this case a cycle in the graph is both a necessary and a sufficient condition for the existence of deadlock .
If each resource type has several instances then a cycle does not necessarily imply that a deadlock occurred . In this case a cycle in the graph is a necessary but not a sufficient condition for the existence of deadlock.
To illustrate this concept let us return to fig bellow suppose that p3 requests r2 .
r2 is added to → Since no resource instance is currently available a request edge p3 the graph fig. At this point two minimal cycles exist in the system :
p1→r2→ p3 → r3→ p2→r1→P2
p2→ r2→ p3→r3→P2
Process p1,p2 and p3 are deadlocked.

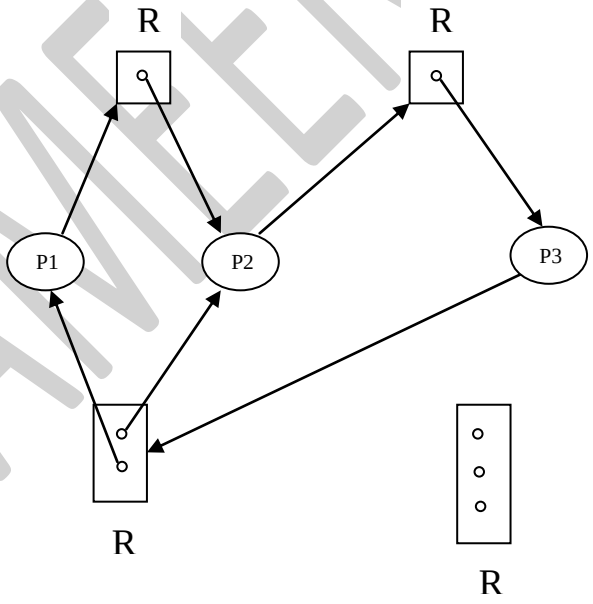


Fig3 : RAG with deadlock

In the fig bellow we have a cycle but there is no deadlock .
p1⟶r2⟶p3⟶r1⟶P1
Where p4 way release its instance of R2 that source can then be allocated to p3 breaking the cycle .

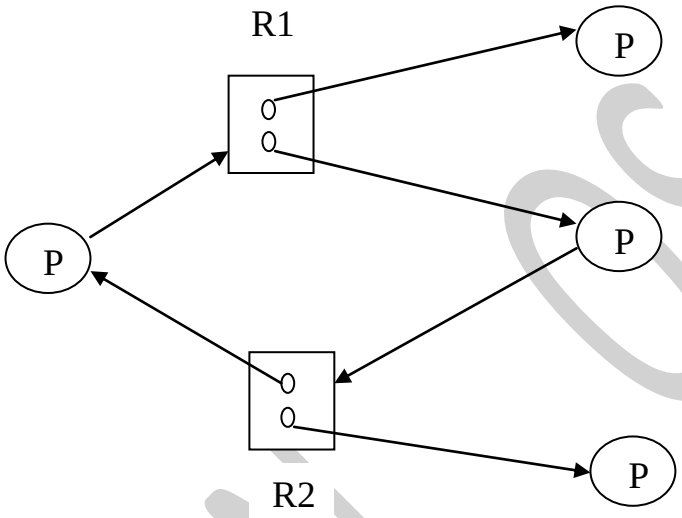


Fig4 : RAG with a cycle but no deadlock

Methods for Handling Deadlock

There are three methods for dealing with the deadlock problem:
a- We can use a protocol to ensure the system will never enter a deadlock state.
b- Allow the system to enter a deadlock state and then recover.
c- We can ignore the problem all together and pretend that deadlocks never occur in the system.

To ensure that a deadlocks never occur the system can use either a deadlock-_avoidance scheme. prevention or a deadlock

Deadlock prevention

It is a set of methods for ensuring that at least one of the necessary conditions can not hold. These methods prevent deadlocks by constraining how requests for sources can be made.

- The **mutual-exclusion** condition must hold for non –sharable resource, sharable resources on the other hand do not require mutually exclusive access.

- To ensure that **hold-and-wait** condition never occurs in the system must guarantee that whenever a process request a resource it does not hold any other resources. This can be implemented by :

1- Allocate all resources to process before its execution .

2- A process request a resource only allowed when it has none. The problems with this the low utilization of resource usage and starvation.

- **No –preemption**
  If a process that is holding some resources request another resources that can not allocated to it then all resources currently being held are preempted.
- **Circular wait**

Let R ={ R1 , R2 , R3 , ….. , R_n } be the set of resource types we can assign to each type a number which allow us to compare two resources . If we define a N where N is the set of numbers.⟶one – to – one function F : R
Example :
F(T/Drive)=1
F(Disk/Drive)=5
F(Printer)=12
Each process can request resources only in an increasing order of enumeration. That is process initially request any number of instances of R_i after that the process can request instances of resource type R_j if and only if

$F(R_j) > F(R_i)$

Deadlock Avoidance

For avoiding deadlock is to require additional information about how resources are to be requested.
For example in C/S with one tape and one printer we might be told that process P will request first M/T and later L/P before releasing both resources. Process Q on the other hand will request first the printer and then the M/T. With this knowledge of the complete sequence of request and releases for each process we can decide for each request whether or not the process should wait.
Each request requires that then consider the following :

1- The resources currently available .
2- The resources currently allocated to each process.
3- The future requests and releases of each process.

The above information used to decide whether the current request can be satisfied or must wait to avoid a possible future deadlock.
The various algorithms differ in the amount and type of information required. The simplest and most useful model requires that each process declare the maximum number of each type that it may need. A deadlock avoidance algorithm dynamically examines the resource- allocation state to ensure that these can never be a circular – wait condition.
The resource allocation state is defined by number of available and allocated resources and maximum demands of the processes.

- **Safe state**

A state is safe if the system can allocate resources to each process (up to maximum) in some order and still avoid a deadlock.
A safe state is not a deadlock state , and a deadlock state is an unsafe state , but not all unsafe states are deadlock. An unsafe state may lead to a deadlock.

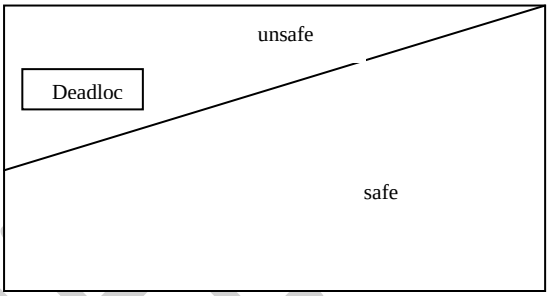


Fig : unsafe &deadlock state space

Example :- to illustrate consider a system with 12 M/T and 3 process :
P0 , p1 and p2 . The maximum needs and current needs for each process as indicated below

(Allocated)

	Maximum needs	Current needs	available
P0	10	5	3
P1	4	2	
P2	9	2	

At time the system is in a safe state. The sequence [p0 , p1 , p2]
Suppose that at time t1 process p2 requests and is allocated 1 move tape drive . the system is no longer in safe state .At this point only process p1 can be allocated all

its tape drives . When it returns the system have only if available and this number not satisfy the request for p0 or p2 therefore

Wait Wait
P0 ⟶ p2 ⟶ P0

If we had made p2 wait until either of the order process had finished and released its resources then we could have avoided the deadlock situation.
There are many deadlock avoidance algorithms , some of these are :

a- RAG Algorithm

If we have a RAG system with only one instance of each resource type . In addition to the request and assignment edges we introduce a new type of edge called a claim R_j indicates that process P_i may request resource R_j at some time in the → edge P_i future. It is as a request edge in direction but is represented by a dashed- line when R_j is converted to a request edge . When a → process p_i request R_j the claim edge P_i P_i is reconverted to a claim → resource R_j is released by p_i the assignment edge R_j R_j → edge P_i

To illustrate this algorithm consider the RAG in fig below suppose that p_2 request r_2 . Although r_2 is currently free we can not allocate it to p_2 since this action will create a cycle in the graph figure . A cycle indicates that the system is in an unsafe state . If P_1 then requests r_2 a deadlock will occur.

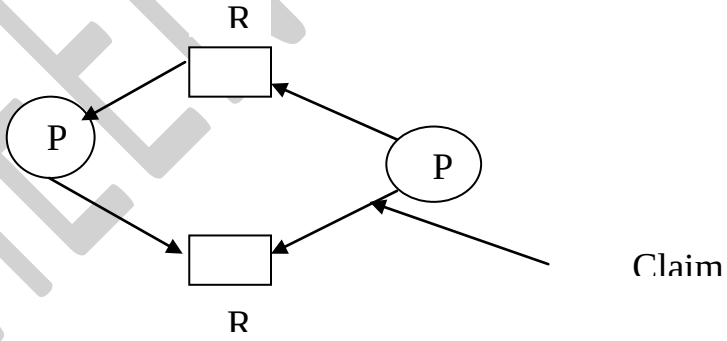


Fig : RAG for deadlock avoidance

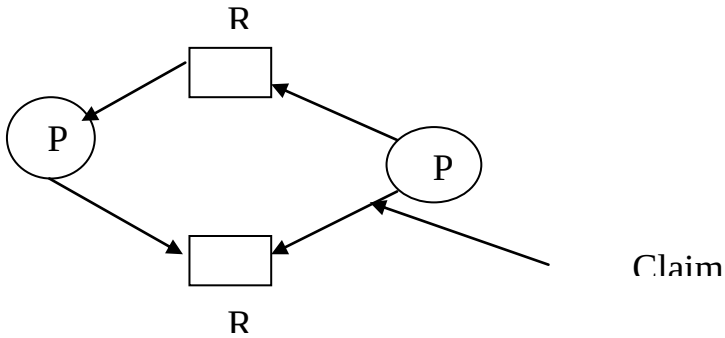


Fig : An unsafe state in a RAG

b- Banker's Algorithm

This Alg. Could be used in a banking system to ensure that the bank never allocates its available cash in such a way that it can no longer satisfy the needs of all its customers.
When a new a process enters the system it must declare the maximum number of instances of each resource type that it may need.

- The maximum must be ≤ total number of resources in the system.

-When a user requests a set of resources must be leave the system in a safe state if the resources are allocated otherwise the process must wait until some other process releases enough resources.

Several data structures must be maintained to implement banker's algorithm
Let n be the number of processes in the system and m be the number of resource types . We need the following data structures:

- Available : A vector of length m indicates the number of available resources of each type.

available . If available [j] = k these are k instances of resource type R_j

- Max : An n_xm matrix defines the maximum demand of each process . If max[i,j] = k , then process P_i may request at most k instances of resource type R_j
- Allocation : An n_xm the resources currently allocated to each process . If allocation [i,j] = k then process p_i is currently allocated process p_i is currently allocated 1 instances of resources of resource type R_j.
- Need : An n_xm indicates the remaining resource need of each process . If need[i,j] = k then process p_i may need k more instances of resource type R_j to complete its task .

Need [i,j] = Max [i,j] – Allocation [i,j]
1- If request i ≤ Need i go to step 2 otherwise raise an error since the process has exceeded its maximum claim.
2- If request i ≤ Available go to step 3 . Otherwise p_i must wait since the resources are not available .
3- The system pretends to have allocated the requested resources to process p_i by modifying the state as follows:
Available := Available – Requesti ,
Allocationi := Allocationi + Requesti ,

Needi:= Needi – Requesti,

If the resulting resource allocation state is safe the transaction is completed and process p_i is allocated its resources. If the new state is unsafe the p_i must wait for request i and the old resource allocation state is restored.

c- Safety Algorithm

The algorithm for finding out whether or out a system is in a safe state can be described as follows:

1- Let work and finish be vectors of length m and n respectively.
   Initialize work := Available and Finish [i] := False for I = 1,2, …, n
2- Find an i such that both
- Finish [i] = false
- Need i ≤ work

If no such i exits, go to step 4

3- Work := work +allocation I Finish[i] := true go to step2
4- If Finish [i] = true for all I then the system is in a safe state

This algorithm may require an order of $m_x n^2$ operations to decide whether a state is safe.

Example:

Consider a system with five processes $\{p_0,p_1,p_2,\ldots\}$ and three resource types {A,B,C} . Resource type A has 10 instances. Resource type B has 5 instances, and resource type c has 7 instance. Suppose that at time T_0 the following snapshot of the system has been taken.

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

The content of the matrix Need is defined to be max-Allocation and is:

	Need
	A B C
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 the safe state if the processes executed in the sequence (p_1, p_3 , p_4 ,p_2 , p_0).
Suppose now that process p_i requests one additional instance of resource type A and two instance of resources type C so request 1= (1 ,0 , 2)
To decide whether this request can be immediately granted we first check that Request 1 ≤ Available (that is , (1,0,2) ≤ (3,3,2)) which is true we then pretend that this request has been fulfilled and we arrive at the following new state.

Allocation	Max	Available	
A B C	A B C	A B C	
0 1 0	7 4 3	3 3 2	P_0
3 0 2	0 2 0		P_1
3 0 2	6 0 0		P_2
2 1 1	0 1 1		P_3
0 0 2	4 3 1		P_4

By execute the safety Alg. We find the sequence (p_1 , p_3 , p_4 , p_0 , p_2) satisfies our safety requirements . Hence we can immediately grant the request of process p_1
If p_4 request for (3 , 3 , 0) . The request can not granted since the resources are not available . Request 1> Available . If p_0 request (0 , 2 , 0) can not granted even though the resources are available since the resulting state is unsafe.

Deadlock Detection

If a system does not employ some protocol that ensures that no deadlock will never occur. Then a detection and recovery scheme must be implemented . The system can use an algorithm to examines the state of the system periodically to determine whether has occurred . If so the system must recover from the deadlock by providing :

a- Maintain information about the current allocation of resources to processes and outstanding request.
b- Provide an Alg. That use the above information to determine whether the system has entered the deadlock state.

The detection Alg. Employs several time – varying data structures that are very similar to those used in the Banker's Algorithm :

- **Available**
- **Allocation**
- **Request** . An n_xm matrix indicating the current request of each process.
  If Request [i,j] = k then p_i is requesting k more instances of resource type r_j .

The detection Alg. Simply investigates every possible allocation sequence for the processes that remain to be completed.

The detection Alg . as follows:

1- Let work and finish be vectors of length m and n respectively . Initialize
Work := Available , for i = 1 ,2 ,3, … , n. If allocation ≠ 0 the Finish [i] := false.
Otherwise , Finish[i] := false.

2- Find an index i such that :

- Finish [i] = false , and
- Request i ≤ work .

If no such I exits go to step 4.

3- Work := work +Allocation i
Finish [i] := true
Go to step2

4- If Finish [i] = false , for some i , 1≤ i ≤ n then the system is in a deadlock state. More over , if Finish [i] = false then process p_i is deadlocked.

Example

Consider a system with five processes {p_0 , p_1 ,… p_4} and three resources types { A=7 instance , B =2 , C=6 instance} suppose that at time T_0 we the following resource allocation state.

Allocation	Max	Available	
A B C	A B C	A B C	
0 1 0	0 0 0	0 0 0	P_0
2 0 0	2 0 2		P_1
3 0 3	0 0 0		P_2
2 1 1	1 0 0		P_3
0 0 2	0 0 2		P_4

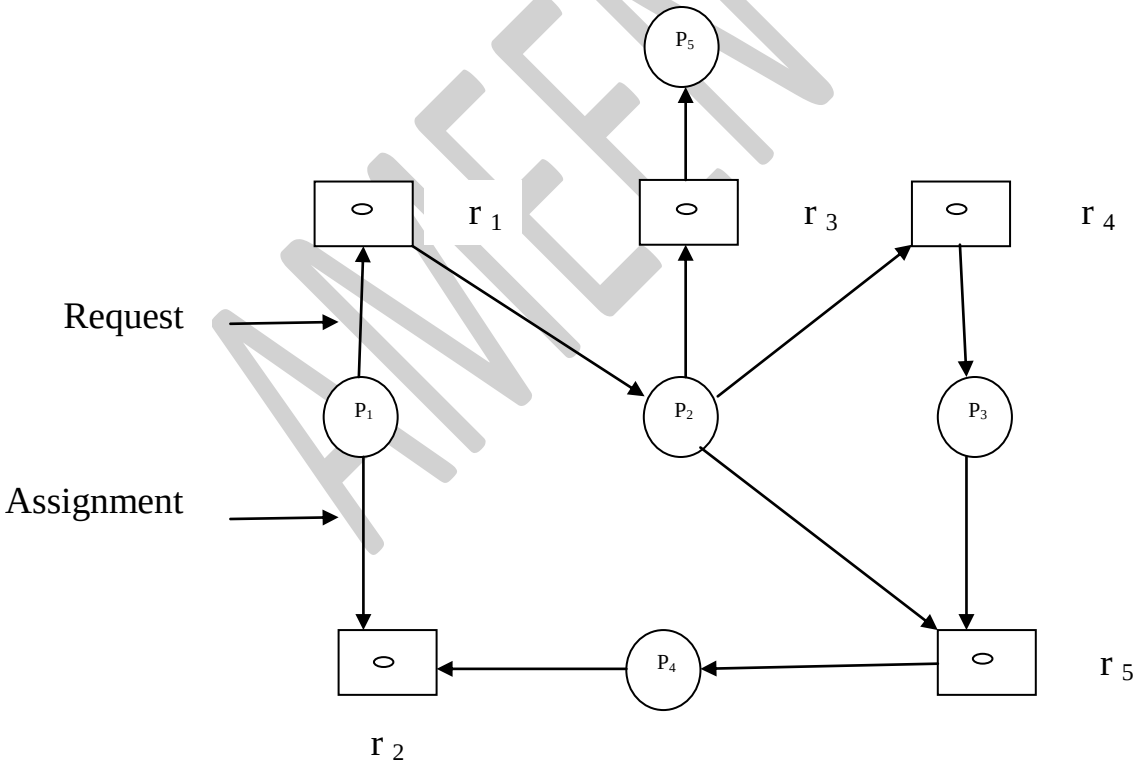
If we execute the detection Alg. We find the system is not in a deadlock state and the sequence < p_0 , p_2 , p_3 , p_1 ,p_4 > will result in finish [i] = true for all i .
Suppose now that process p2 makes one additional request for an instance of type C. the Request matrix is modified as follows :

	Need
	A B C
P_0	0 0 0
P_1	2 0 2
P_2	
P_3	
P_4	

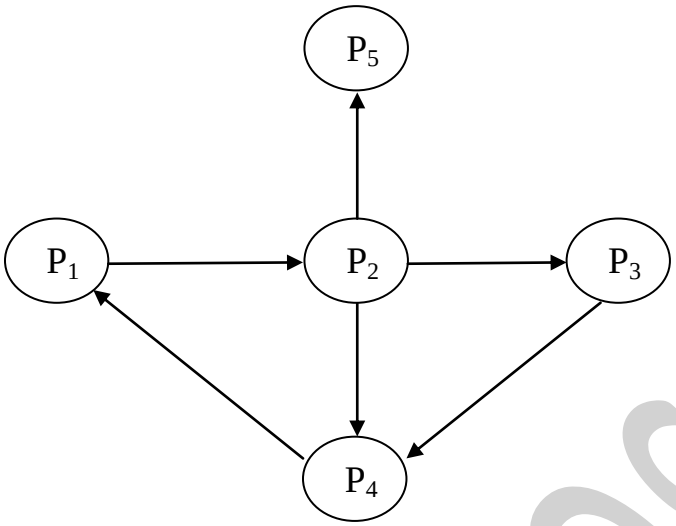
0 0 1
1 0 0
0 0 2

We claim that the system is now deadlocked . Although we can reclaim the resources held by process p_0 the number of available resources is not sufficient to fut fill the requests of the other processes . Thus a deadlock exist , consisting of processes < p_1 , p_2 , p_3 and p_4 >.

- Single Instance of each resource type
  The detection Alg. is of order $m_x n^2$. If all resources have only a single instance we can define a faster Alg. we will use a variant of the resource allocation graph called a wait – for graph . This graph is obtained from the resource allocation graph by removing the nodes of type resource and collapsing the appropriate edges. Where the edge from p_i is waiting for process p_j to release a resource that it needs.
  An edge (p_i ,p_j) exists in a wait – for graph if and only if the resource RAG contains two edges (p_i , r_q) and (r_q , p_i) for some resource , see fig bellow



(a)



(b)

Fig: RAG (a) and its wait – for graph (b)

A deadlock exists in the system if and only if the wait – for graph contains a cycle.

- **Recovery from deadlock**

When a detection Alg. determines that a deadlock exists the system must recover from the deadlock .

There are two options for breaking a deadlock

a- **Process termination** by killing a process , two methods:

- Kill all deadlocked processes.
- Kill one process at a time until the deadlock cycle is eliminated.

b- **Resource preemption**, to eliminate deadlocks using resource preemption we can preempt some resources from processes and give them to other processes until the deadlock cycle is broken .

If preemption is required in order to deal with deadlocks then three issues need to be addressed:

- Selecting a victim: which process and which resources .
- Rollback : if we preempt a resource from a process what should be done with that process?

- Starvation : How do we ensure that Starvation will not occur?

That is how can we guarantee that resources will not always be preempted from the some process?

Real Memory Organization ch7

Memory Management Strategies

-These strategies are designed to obtain the best possible use of main memory. They are divided into:

1. Fetch strategies: determine when to move the next piece of a program or data to main memory from secondary storage. We divide them into demand and anticipatory. In demand fetch strategy, the system places the next piece of program or data in main memory when a running program references it. Today, many systems have increased performance by employing anticipatory fetch strategies, which attempt to load a piece of program or data into memory before it is referenced.
2. Placement strategies: determine where in main memory the system should place incoming program or data pieces, first fit, best fit, and worst fit, memory placement strategies.
3. Replacement strategies: when memory is too full to accommodate a new program, the system must remove some (or all) of a program or data that currently resides in memory. The system's replacement strategy determines which piece to remove.

Contiguous Vs Noncontiguous Memory Allocation

-Contiguous Memory Allocation: to execute a program in early computer systems, the system operator or the operating system had to find enough contiguous main memory to accommodate the entire program. If the program was larger than the available memory then the system could not execute it.

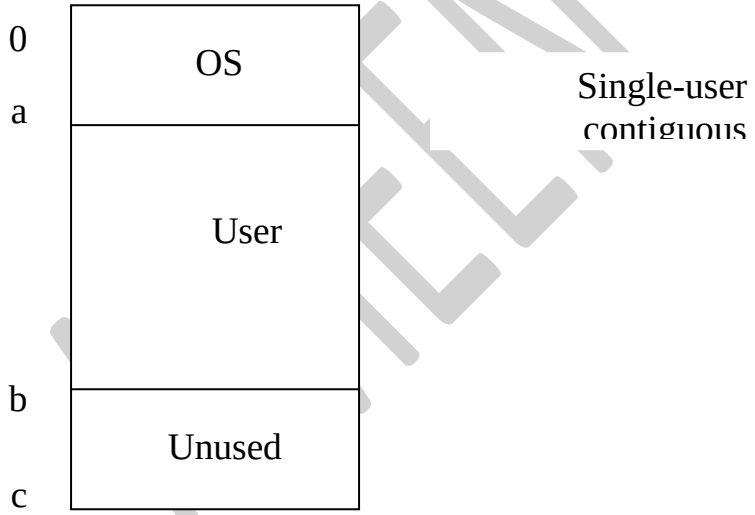
-In Non-Contiguous Memory Allocation: a program is divided into blocks or segments that the system may place in non adjacent slots in main memory. This

allows making use of holes (unused gaps) in memory that would be too small to hold whole programs.

Single-User Contiguous Memory Allocation

-Early computer systems allowed only one person at a time to use a machine. All the machine's resources were dedicated to that user and the user was charged for all the resources whether or not the user's job required them.

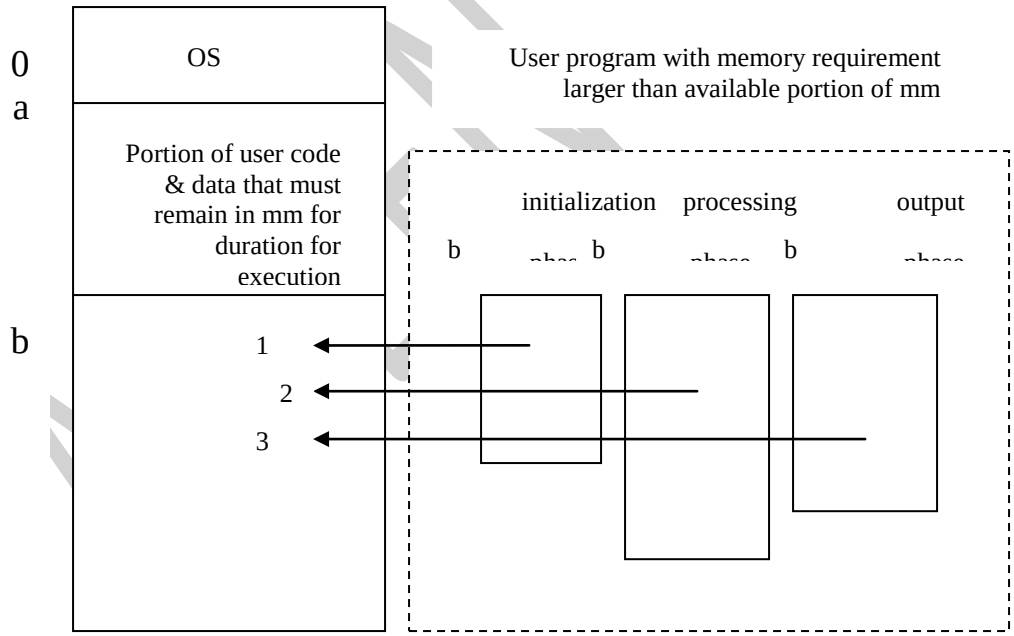
-The programmer wrote all the code necessary to implement a particular application including I/O instructions. The designers consolidated (combined) I/O coding that implemented basic functions into an I/O control system (IOCS). The programmer called IOCS routines to do the work.



Overlays

-One way in which a software designer could overcome the memory limitations was to create overlays, which allowed the system to execute programs larger than main memory.

-The programmer divides the program into logical sections. When the program does not need the memory for one section, the system can replace some or all of it with the memory for a needed section. Overlays enable the programmers to extend main memory.

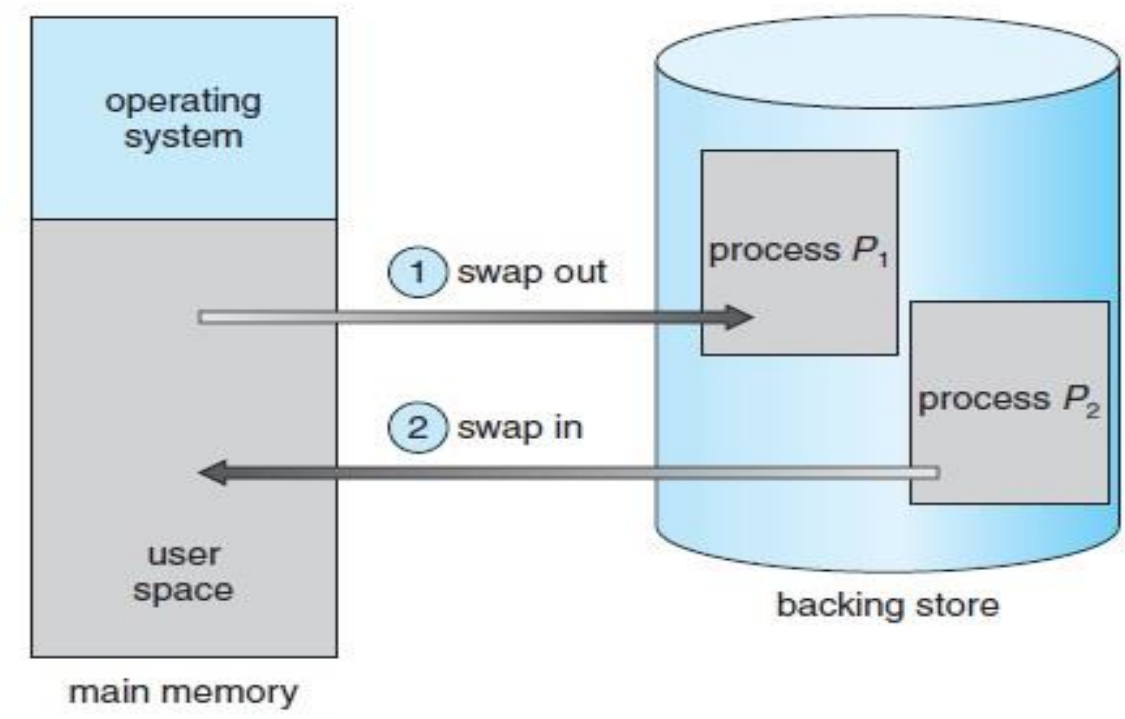


1 load initialization phase at b and run
2 then load processing phase at b and run
3 then load output phase at b and run

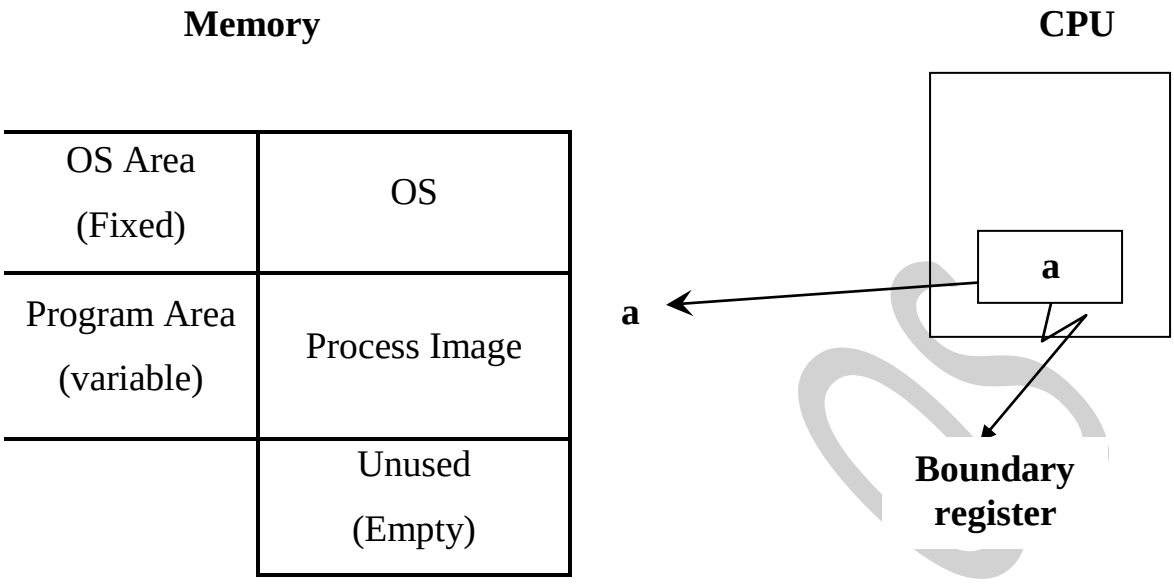
Swapping:

A process can be swapped temporarily out of memory to a backing store, and then brought back into memory for continued execution

- Backing store – fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images
- Roll out, roll in – swapping variant used for priority-based scheduling algorithms; lower-priority process is swapped out so higher-priority process can be loaded and executed
- Major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped
- Modified versions of swapping are found on many systems (i.e., UNIX, Linux, and Windows)
- System maintains a ready queue of ready-to-run processes which have memory images on disk



Protection in a Single-User System



-Without protection, the process may alter the operating system. The protection can be implemented with a single boundary register built into the processor which can be modified only by a privileged instruction.

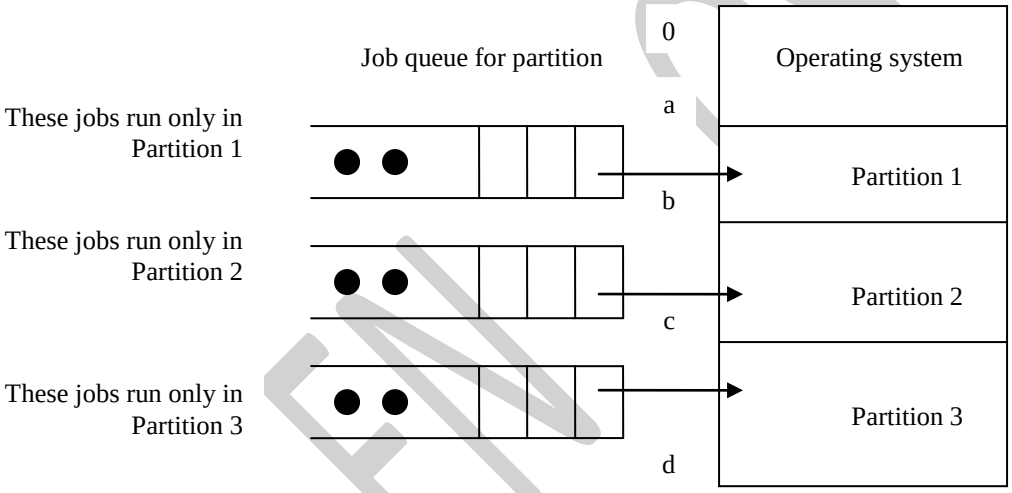
-The boundary register contains the memory address at which the user's program begins. Each time a process references a memory address, the system determines if the request is for an address greater than or equal to that stored in the boundary register.

- If so, the system services the request. If not, then the program is trying to access the operating system. The system intercepts the request and terminates the process with an appropriate error message.

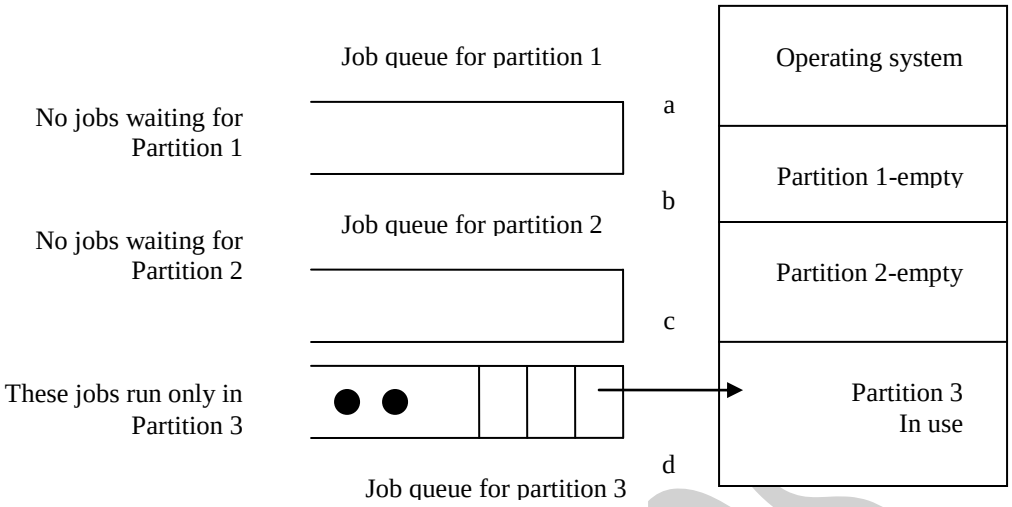
Fixed-Partition Multiprogramming

-The earliest multiprogramming systems used fixed partition multiprogramming. The system divides main memory into a number of fixed size partitions. Each partition holds a single job.

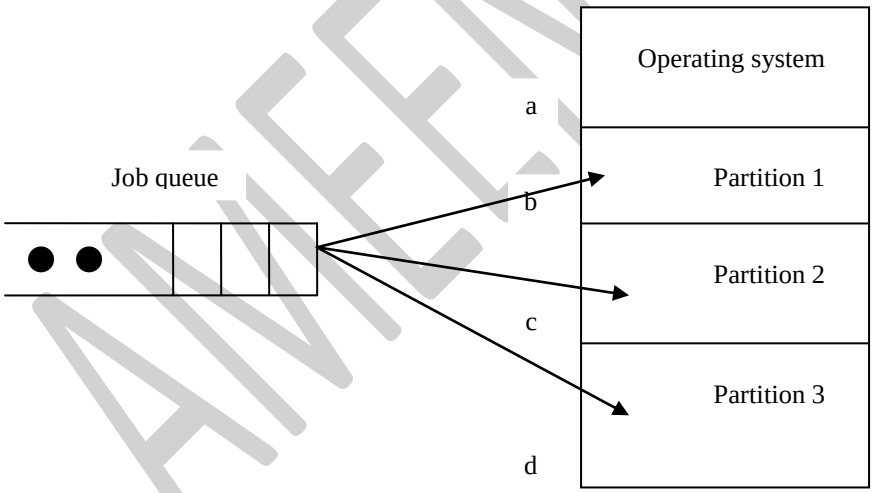
-In the earliest multiprogramming systems, the programmer translated a job using an absolute assembler or compiler. It meant that a job had its precise location in memory determined before it was launched and could run only in a specific partition. If the programs partition was occupied then that job had to wait even if other partitions were available.



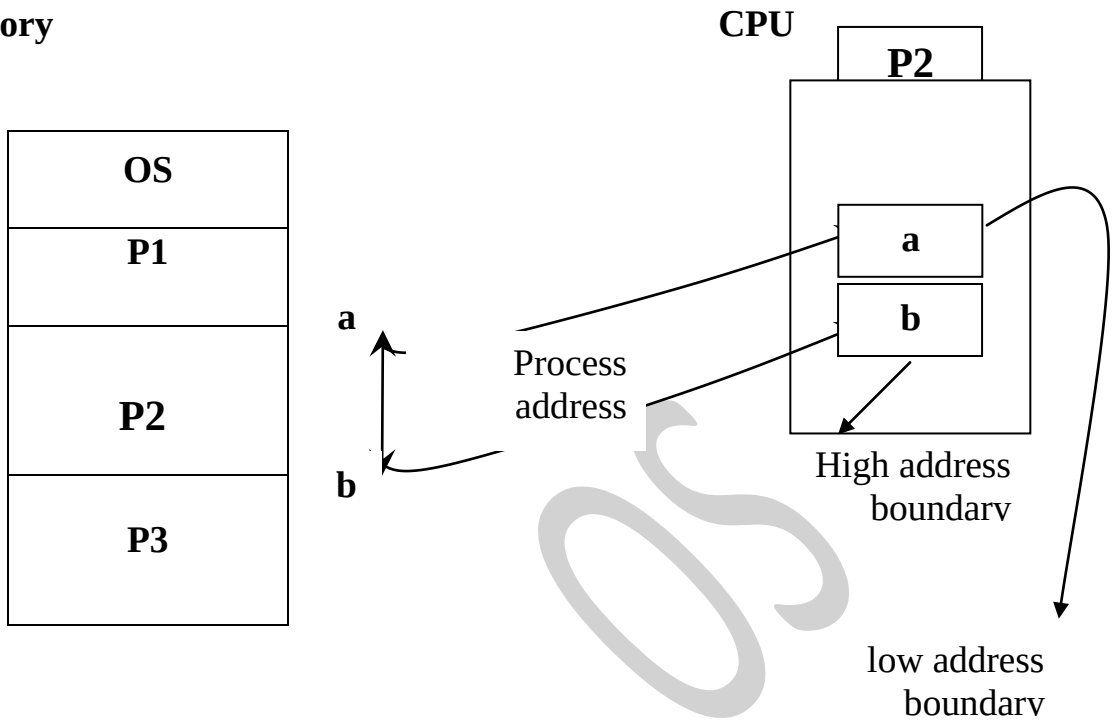
-In the following figure, all the jobs in the system must run in partition 3. Because this partition currently is in use, all other jobs are forced to wait, even though the system has two other partitions in which the jobs could run.



-To overcome the problem, the developers created relocating compilers, assemblers and loaders. These tools produce a Relocatable program that can run in any available partition that is large enough to hold that program.

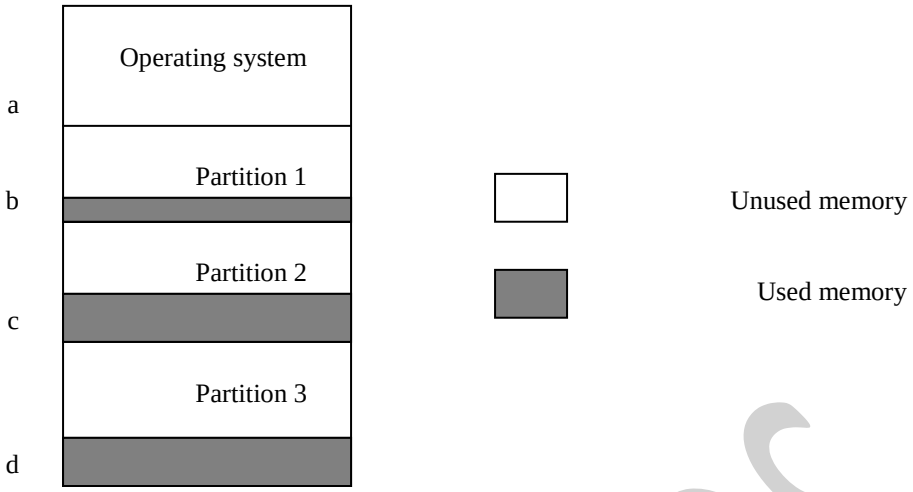


-Protection often is implemented with multiple boundary register. The system can delimit each partition with two boundary registers low and high, also called base and limit registers.



-When a process issues a memory request, the system checks whether the requested address is greater than or equal to the process's low boundary register value and less than the process's high boundary register value. If so, the system honors the request, otherwise, the system terminates the program with an error message.

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



-The system's three user partitions are occupied but each program is smaller than its corresponding partition. Consequently, the system may have enough main space in which to run another program but has no remaining partitions in which to run the program. Thus, some of the system's memory are wasted.

Variable-Partition Multiprogramming

Variable-Partition Characteristics

-The queue at the top of the figure contains available jobs and information about their memory requirements. The operating system makes no assumption about the size of a job except that it does not exceed the size of available main memory.

-The system progresses through the queue and places each job in memory, where there is available space, at which point it becomes a process. This organization does not suffer from internal fragmentation, because a process's partition is exactly the size of the process.

-The waste does not become obvious until processes finish and leave holes in main memory. The system can continue to place new processes in these holes. Every hole eventually becomes too small to hold a new process. This is called external

fragmentation, where the sum of the holes is enough to accommodate another process.

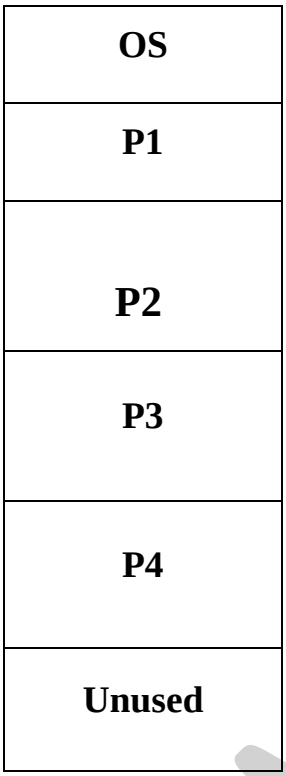
-The system can determine whether the newly freed memory area is adjacent to other free memory areas. The system then records in a free memory list either (1) that the system now has an additional hole or (2) that an existing hole has been enlarged.

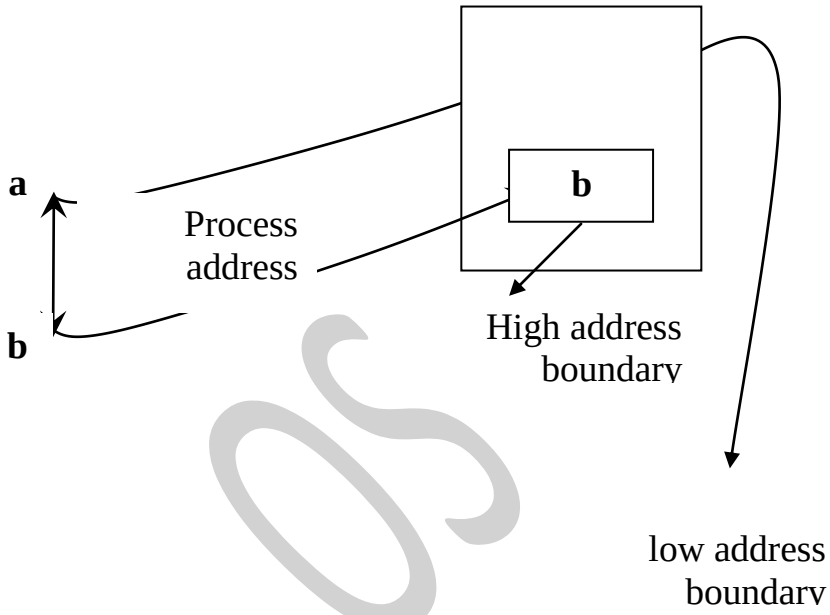
-The process of merging adjacent holes to form a single, larger hole is called coalescing (merge things). The system reclaims the largest possible contiguous blocks of memory.

-Another technique for reducing external fragmentation is called memory compaction (burping the memory or garbage collection). This relocates all occupied areas of memory to one end or the other of main memory. Now all of the available free memory is contiguous. The drawbacks are:

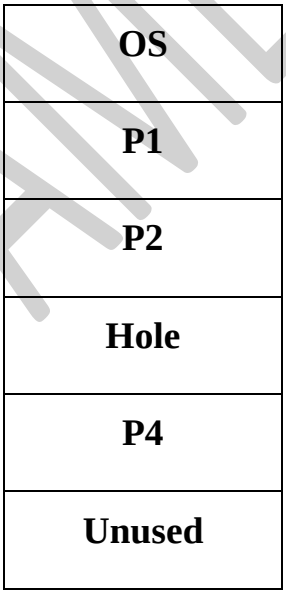
- Overhead consumes system resources.
- The system must cease (stop something) all other computation during compaction which results in erratic response times for interactive users.
- Compaction must relocate the process.

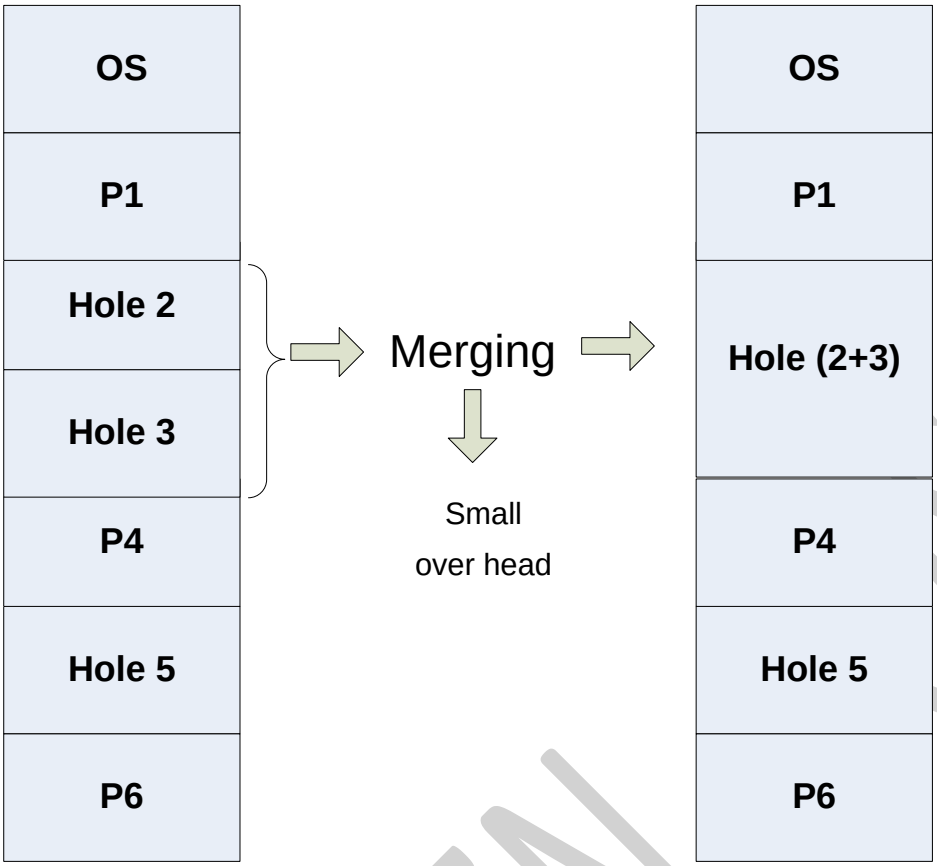
Memory **CPU**

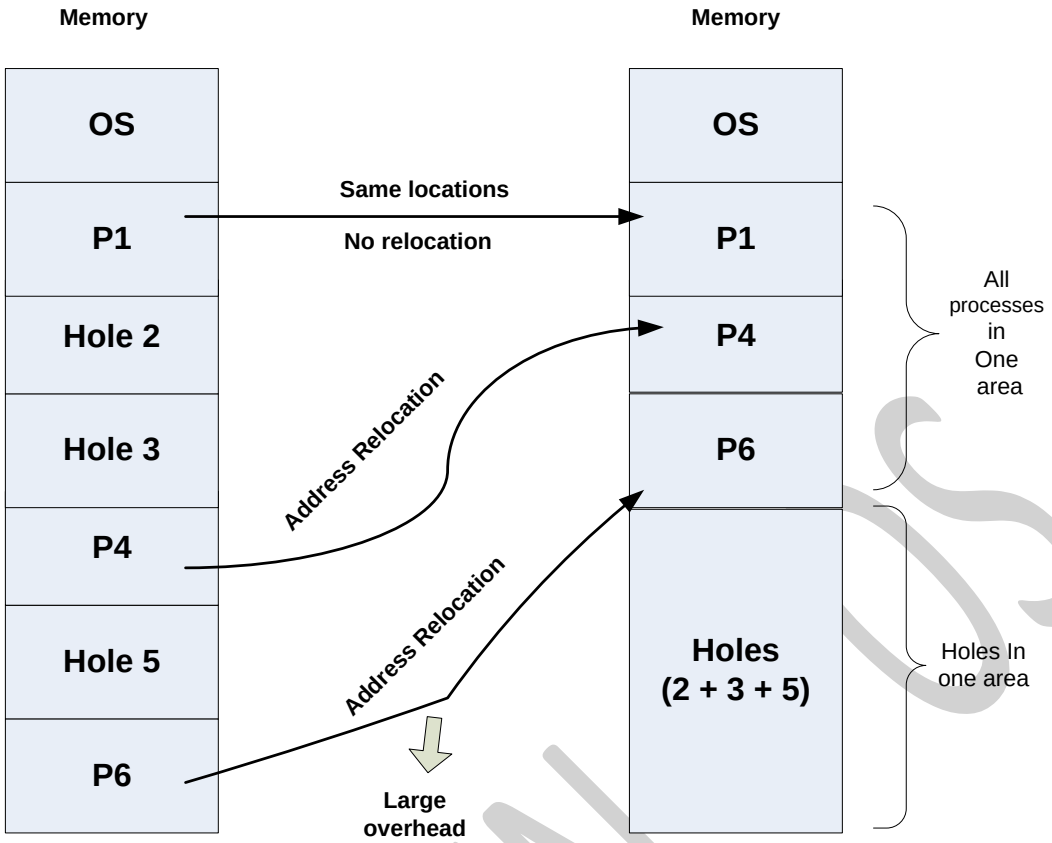




After some time (P3 is completed)



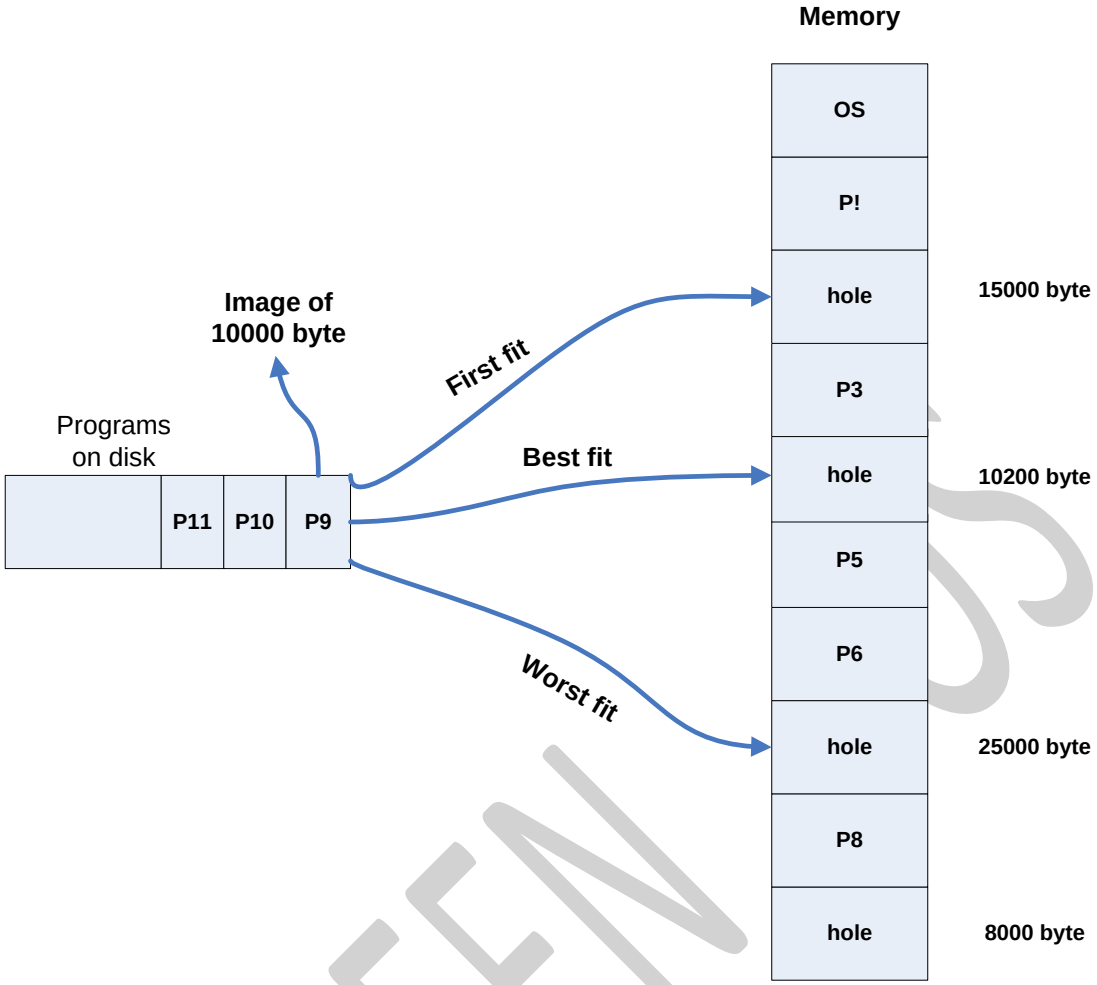

OS
P1
Hole 2
Hole 3
P4
Hole 5
P6
Merging
Small
over head
OS
P1
Hole (2+3)
P4
Hole 5
P6



Memory Placement Strategies

-Determines where in main memory to place incoming programs and data. The main strategies are:

- **Best fit:** place the job in the smallest possible hole. The disadvantage is that the rest of hole will not be enough for new job.
- **First fit:** place the job in the first suitable hole. The advantage is low overhead i.e. small CPU wasted time in implementing the strategy.
- **Worst fit:** place the job in the largest available hole. The rest of hole may be still enough for new job

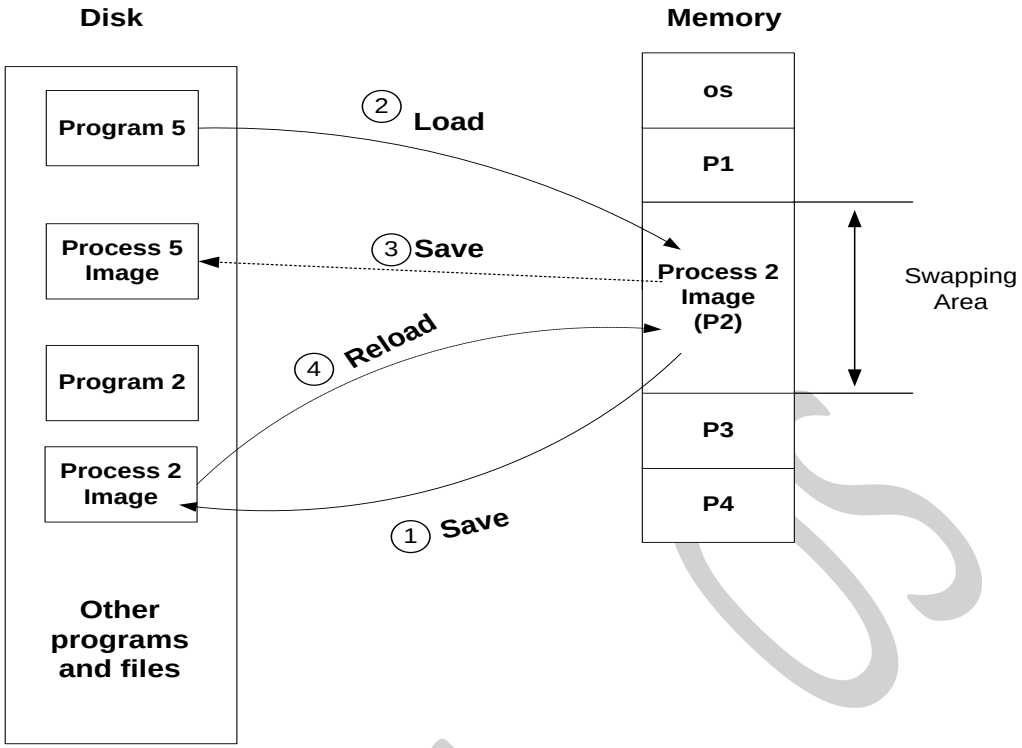


Multiprogramming with Memory Swapping

-When memory is full and new program is required to be executed then OS has to carry out a swapping process as follows:

1. Save resident process image to disk. The process should be in ready or blocked state (or suspended state which is preferred as will be shown later). The memory space of that image becomes free (empty) and may be used for new process. This free space is called “Swapping Area".
2. Load the new program from disk to swapping area and create a PCB for it (create new process).

-When the old process is needed to run again, it has to be reloaded to its original space i.e. to the swapping area, however, after saving the new process to disk.



Fragmentation:

- External Fragmentation: total memory space exists to satisfy a request, but it is not contiguous
- Reduce external fragmentation by compaction

- shffle memory contents to place all free memory together in one large block
- Compaction is possible only if relocation is dynamic, and is done at Execution time.

- I/O problem

@ Latch job in memory while it is involved in I/O

@ Do I/O only into OS buffers

- Internal Fragmentation: allocated memory may be slightly larger than requestedmemory, this size difference is memory internal to a partition, but not being used

Virtual Memory

Virtual Memory: Basic Concepts

Virtual memory is a technique that allows the execution of processes that may not be completely in memory. One major advantage of this scheme is that programs can be larger than physical memory.

There are two types of addresses in virtual memory systems: those referenced by processes (virtual addresses) and those available in main memory (physical or real addresses).

Virtual memory systems contain special-purpose hardware called the memory management unit (MMU) that quickly maps virtual addresses to physical addresses. A key to implementing virtual memory systems is how to map virtual addresses to physical addresses as process execute. Dynamic address translation (DAT) mechanisms convert virtual addresses to physical addresses during execution.

Paging

Paging is a memory management scheme that eliminates the need for contiguous allocation of physical memory. This scheme permits the physical address space of a process to be non – contiguous.

The physical memory is broken into fixed-sized blocks called frames. Logical memory is also broken into blocks of the same size called pages. When a process is to be executed, its pages are loaded into any available memory frames from the backing store. The backing store is divided into fixed-sized blocks that are of the same size as the memory frames.

The hardware support for paging is illustrated in Figure (1). Every address generated by the CPU is divided into two parts:

- **Page number (p):** Number of bits required to represent the pages in Logical Address Space.
- **Page offset (d):** Number of bits required to represent particular word in a page or page size of Logical Address Space.

The page number is used as an index into a page table. The page table contains the base address of each page in physical memory. This base address is combined with the page offset to define the physical memory address that is sent to the memory unit.

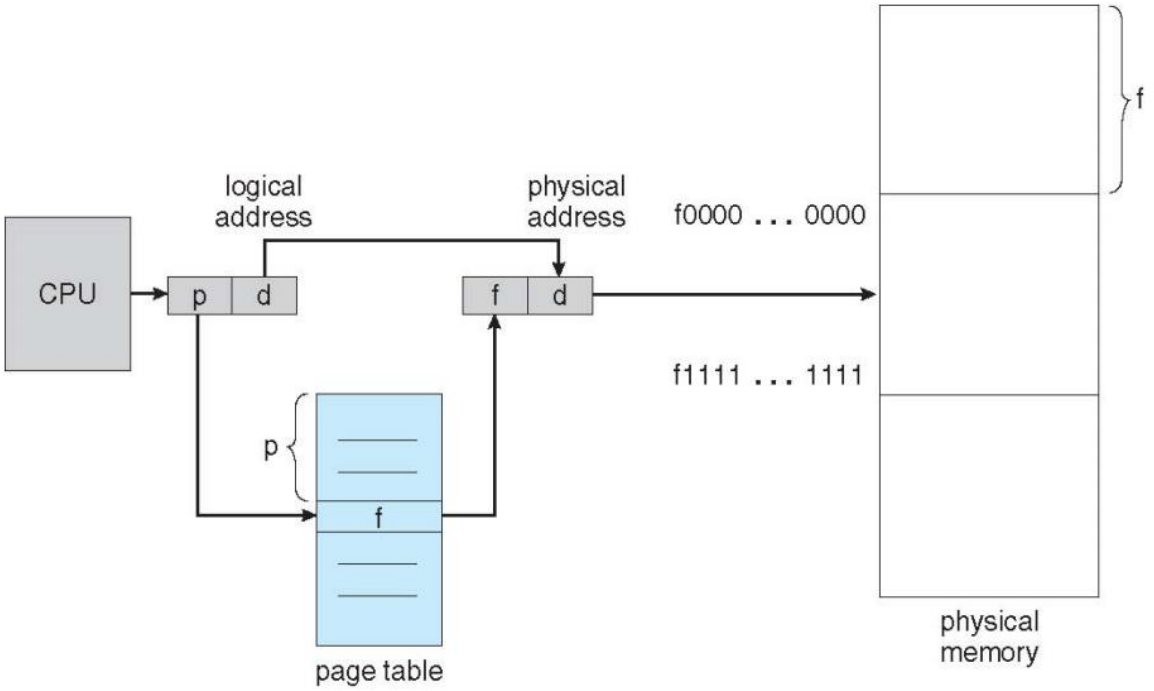


Figure 1: Paging hardware

When we use a paging scheme, we have no external fragmentation: Any free frame can be allocated to a process that needs it. However, we may have some internal fragmentation.

The hardware implementation of page table can be done by using dedicated registers. But the usage of register for the page table is satisfactory only if page table is small. If page table contain large number of entries then we can use TLB (translation Look-aside buffer), a special, small, fast look up hardware cache. The TLB is used with page tables in the following way.

The TLB contains only a few of the page-table entries. When a logical address is generated by the CPU, its page number is presented to the TLB. If the page number is found, its frame number is immediately available and is used to access memory. The whole task may take less than 10 percent longer than it would if an unmapped memory reference were used.

If the page number is not in the TLB (known as a TLB miss), a memory reference to the page table must be made. When the frame number is obtained, we can use it

to access memory. In addition, we add the page number and frame number to the TLB, so that they will be found quickly on the next reference.

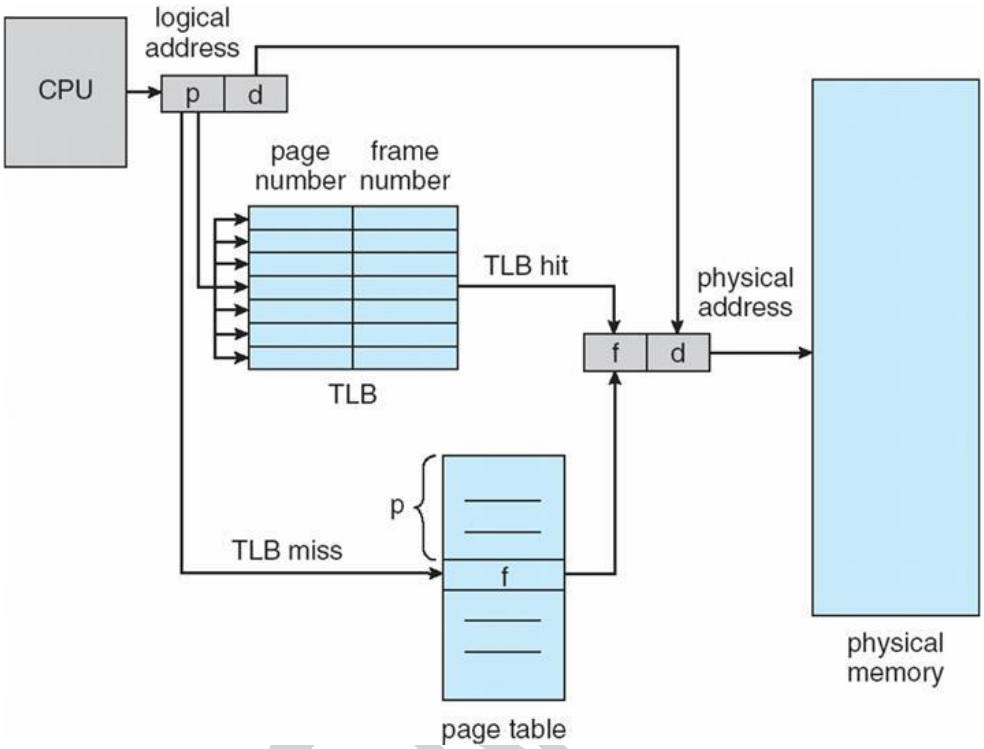


Figure 2: Paging hardware with TLB

Protection

Memory protection in a paged environment is accomplished by protection bits that are associated with each frame. Normally, these bits are kept in the page table. One bit can define a page to be read-write or read-only. Every reference to memory goes through the page table to find the correct frame number. At the same time that the physical address is being computed, the protection bits can be checked to verify that no writes are being made to a read-only page. An attempt to write to a read-only page causes a hardware trap to the operating system (or memory-protection violation). We can easily expand this approach to provide a finer level of protection. We can create hardware to provide read-only, read-write, or execute-only protection. Or, by providing separate protection bits for each kind of access, we can allow any combination of these accesses; illegal attempts will be trapped to the operating system.

One more bit is generally attached to each entry in the page table: a valid-invalid bit. When this bit is set to "valid," this value indicates that the associated page is in

the process' logical-address space, and is thus a legal (or valid) page. If the bit is set to "invalid", this value indicates that the page is not in the process' logical-address space. Illegal addresses are trapped by using the valid-invalid bit. The operating system sets this bit for each page to allow or disallow accesses to that page.

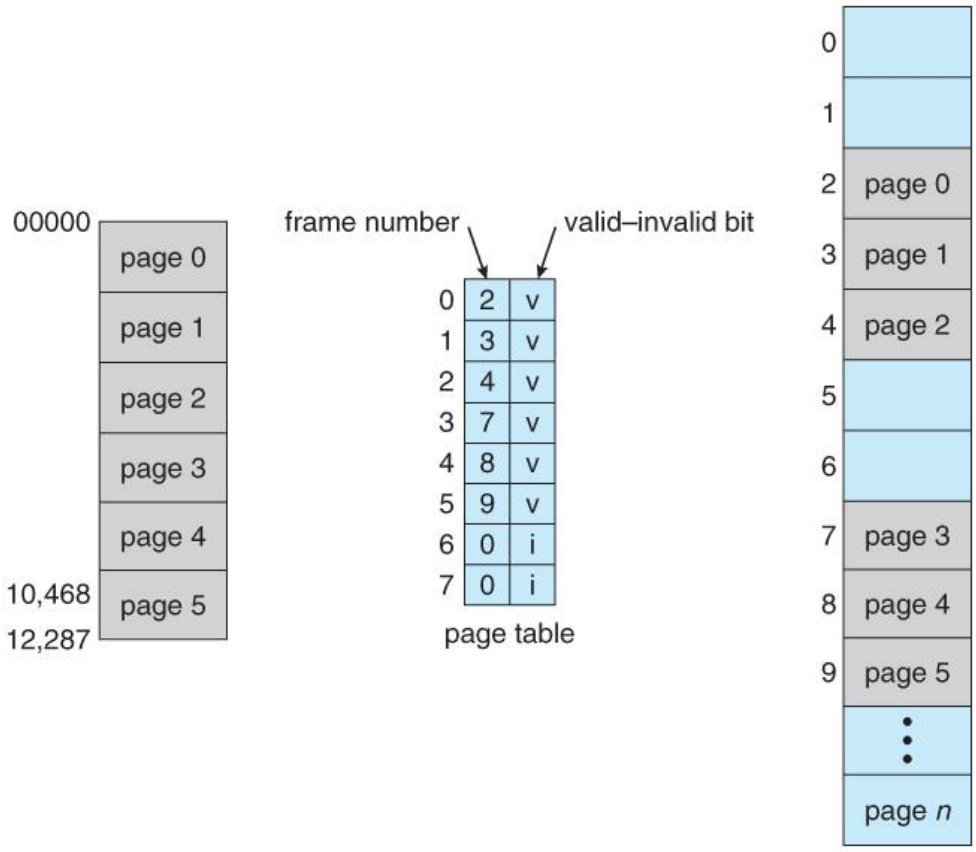


Figure 3: Valid (v) or invalid (i) bit in a page table.

Segmentation

A Memory Management technique in which memory is divided into variable sized chunks which can be allocated to processes. Each chunk is called a **Segment**. Each segment consists of contiguous locations. The segments need not be the same size nor must be placed adjacent to one another in main memory.

A table stores the information about all such segments and is called segment table. It maps two dimensional Logical address into one dimensional Physical address. Each entry of the segment table has a segment base and a segment limit. The segment base contains the starting physical address where the segment resides in memory, whereas the segment limit specifies the length of the segment.

The use of a segment table is illustrated in Figure (4). Every address generated by the CPU is divided into two parts:

- **Segment number (s):** Number of bits required to represent the segment.
- **Segment offset (d):** Number of bits required to represent the size of the segment.

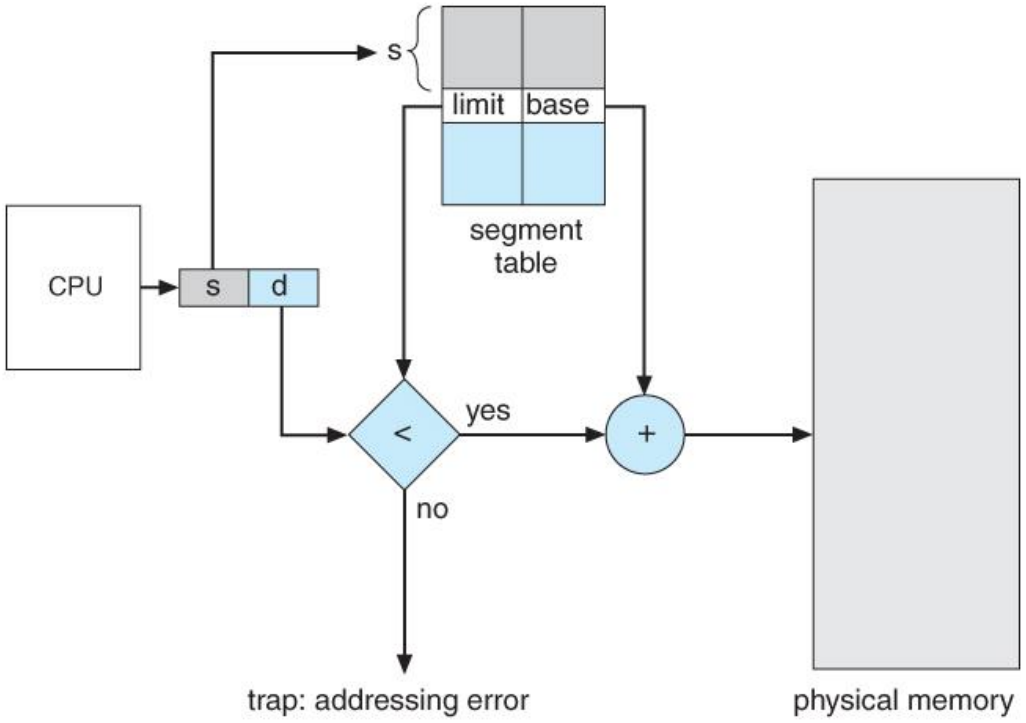


Figure 4: Segmentation hardware

As an example, consider the situation shown in Figure (5). We have five segments numbered from 0 through 4. The segments are stored in physical memory as shown. The segment table has a separate entry for each segment, giving the beginning address of the segment in physical memory (or base) and the length of that segment (or limit). For example, segment 2 is 400 bytes long and begins at location 4300.

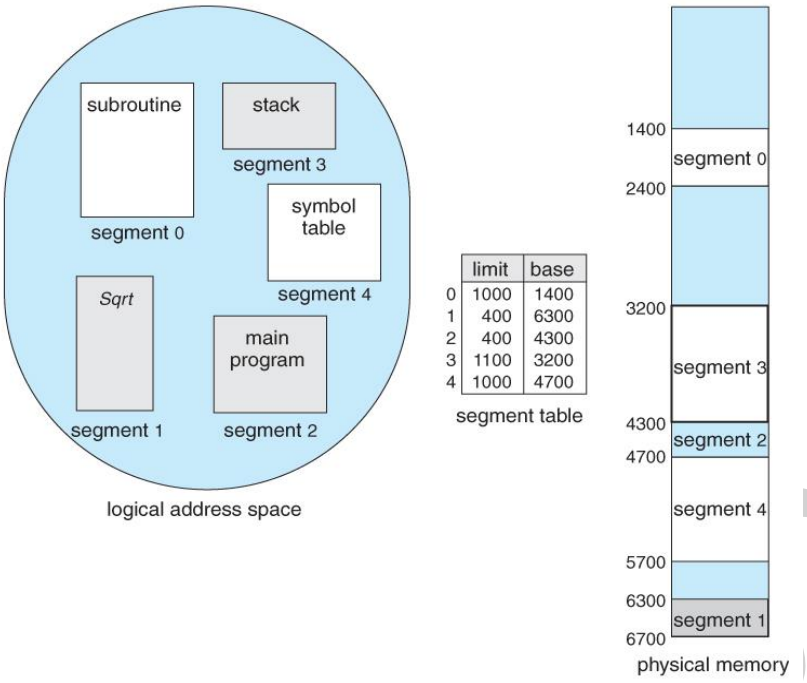


Figure 5: Example of segmentation.

Demand Paging

A demand-paging system is similar to a paging system with swapping (Figure 6). Processes reside on secondary memory (which is usually a disk). When we want to execute a process, we swap it into memory.

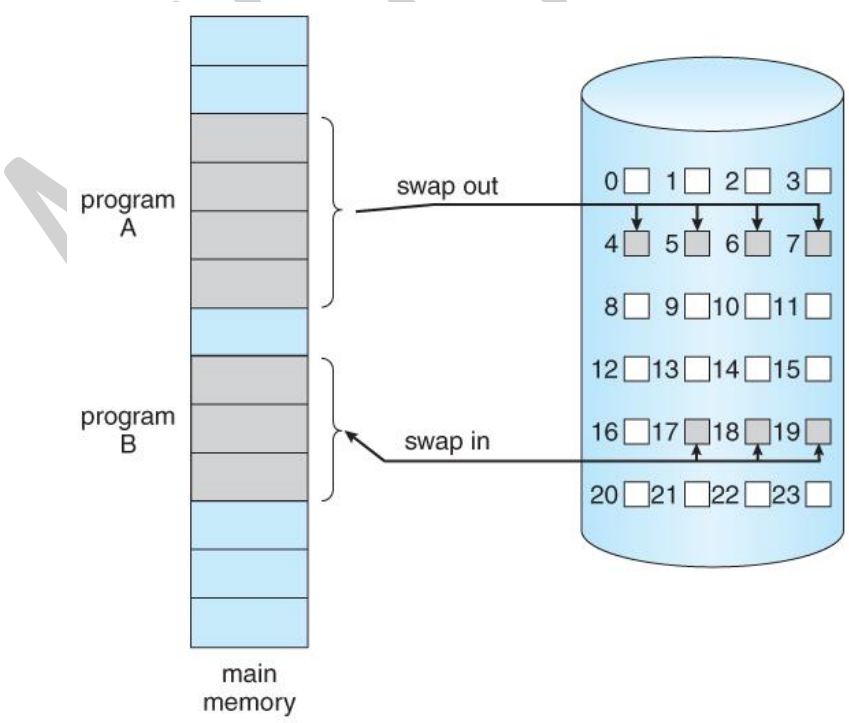


Figure 6: Transfer of a paged memory to contiguous disk space.

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 anyway, decreasing the swap time and the amount of physical memory needed.

With this scheme, we need some form of hardware support to distinguish between those pages that are in memory and those pages that are on the disk. The valid invalid bit scheme can be used for this purpose. When this bit is set to "valid," this value indicates that the associated page is both legal and in memory. If the bit is set to "invalid," this value indicates that the page either is not valid (that is, not in the logical address space of the process), or is valid but is currently on the disk. The page-table entry for a page that is brought into memory is set as usual, but the page-table entry for a page that is not currently in memory is simply marked invalid, or contains the address of the page on disk. This situation is depicted in Figure 7.

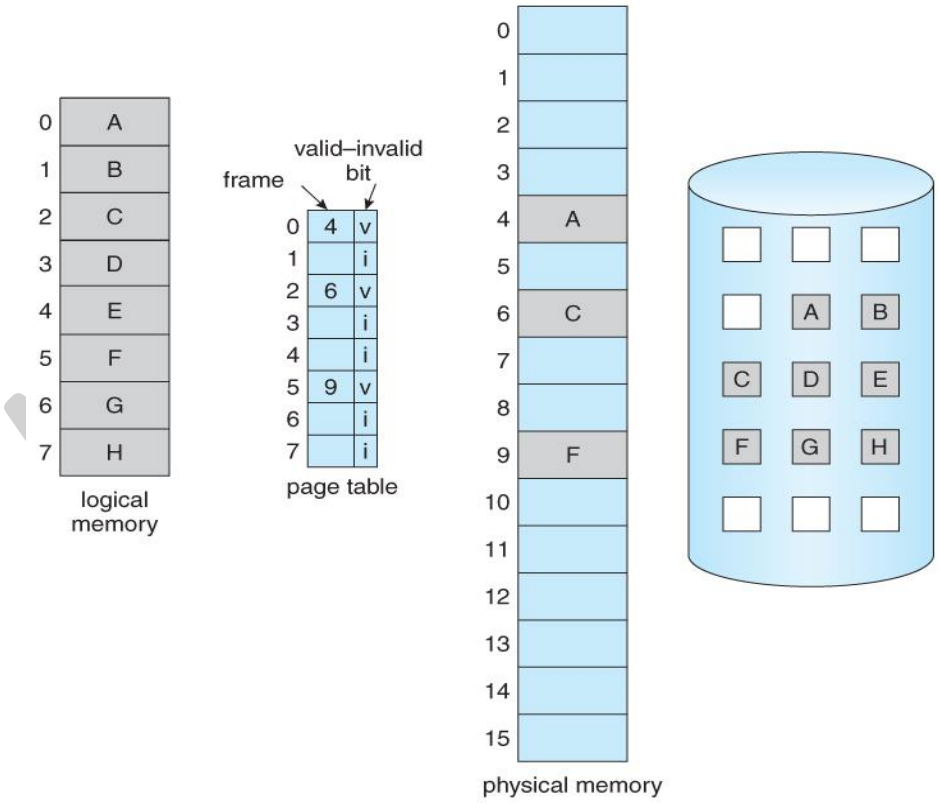


Figure 7: Page table when some pages are not in main memory.

Page Fault

When the process tries to access a page that was not brought into memory (access to a page marked invalid) causes a page-fault trap.

The paging hardware, in translating the address through the page table, will notice that the invalid bit is set, causing a trap to the operating system. This trap is the result of the operating system's failure to bring the desired page into memory, rather than an invalid address error as a result of an attempt to use an illegal memory address. The procedure for handling this page fault is illustrates in Figure (8).

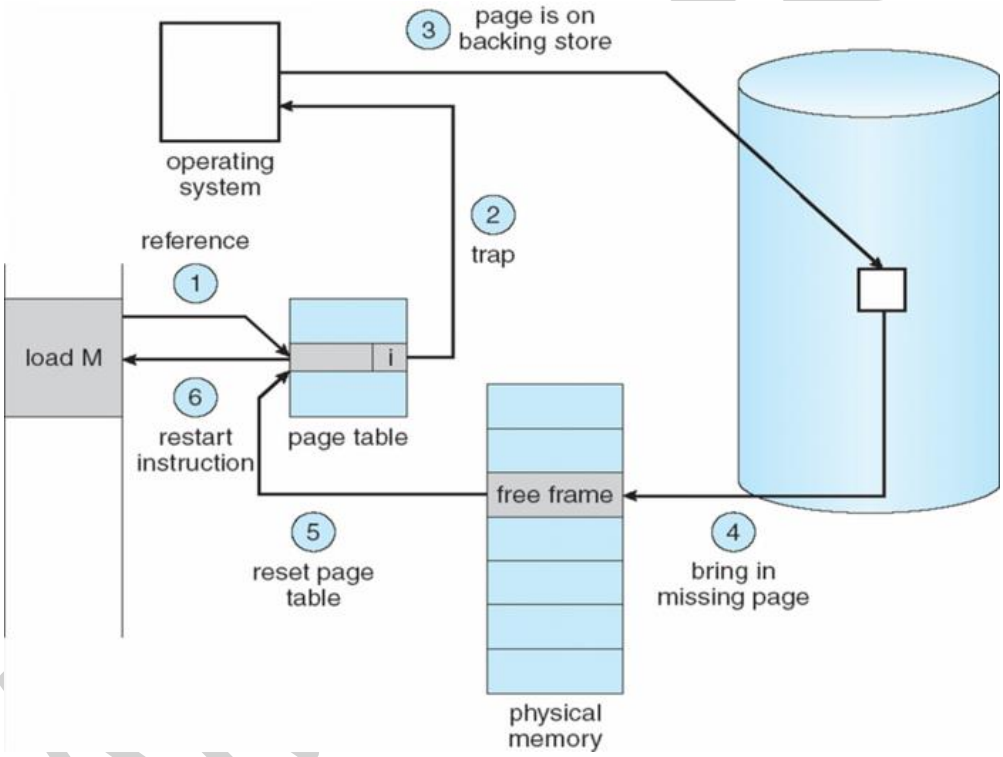


Figure 8: Steps in handling a page fault.

1. Check the page table (usually kept with the process control block) for this process, to determine whether the reference was a valid or invalid memory access.
2. If the reference was invalid, a page fault exception is raised.
3. The operating system must locate logical page in secondary memory.
4. Schedule a disk operation to read the desired page and swap into memory into a free frame.

5. When the disk read is complete, we modify the page table to indicate that the page is now in memory (set valid bit).
6. Restart the instruction that was interrupted by the illegal address trap.

Page Replacement

Page replacement takes the following approach. If no frame is free, we find one that is not currently being used and free it. We can free a frame by writing its contents to swap space, and changing the page table to indicate that the page is no longer in memory (Figure 9). We can now use the free frame to hold the page for which the process faulted. We modify the page-fault service routine to include page replacement:

1. Find the location of the desired page on the disk.
2. Find a free frame:
   a. If there is a free frame, use it.
   b. If there is no free frame, use a page-replacement algorithm to select a victim frame.
   c. Write the victim page to the disk; change the page and frame tables accordingly.
3. Read the desired page into the (newly) free frame; change the page and frame tables.
4. Restart the user process.

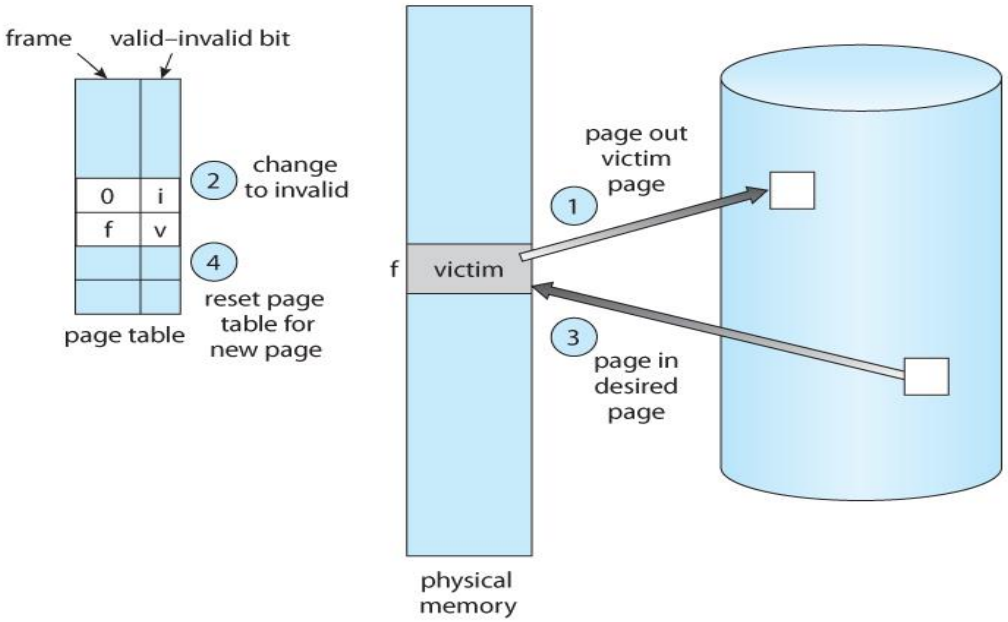


Figure 9: Page replacement.

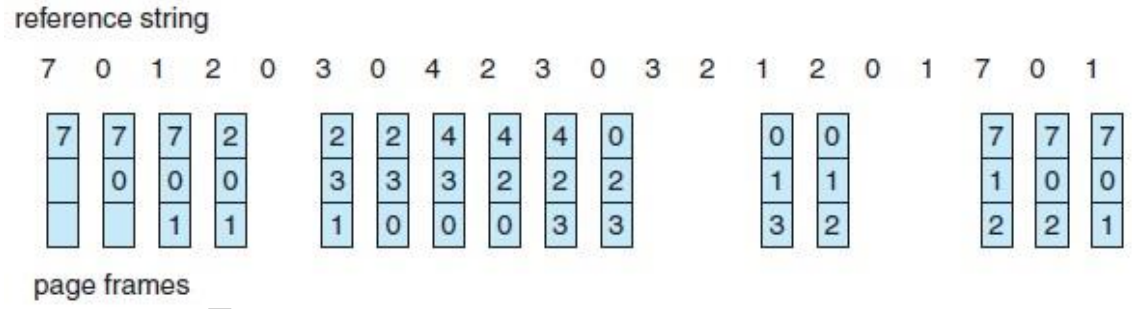
Page Replacement Algorithms

In an operating systems that use paging for memory management, page replacement algorithm are needed to decide which page needed to be replaced when new page comes in. Whenever a new page is referred and not present in memory, page fault occurs and Operating System replaces one of the existing pages with newly needed page. Different page replacement algorithms suggest different ways to decide which page to replace. The target for all algorithms is to reduce number of page faults.

1. FIFO Page Replacement

The simplest page-replacement algorithm is a FIFO algorithm. A FIFO replacement algorithm associates with each page the time when that page was brought into memory. When a page must be replaced, the oldest page is chosen. Notice that it is not strictly necessary to record the time when a page is brought in. We can create a FIFO queue to hold all pages in memory. We replace the page at the head of the queue. When a page is brought into memory, we insert it at the tail of the queue.

Example: Suppose three pages can be in memory at a time per process. Process references pages: 7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1 , what is the number of page fault?



The number of page fault = 15

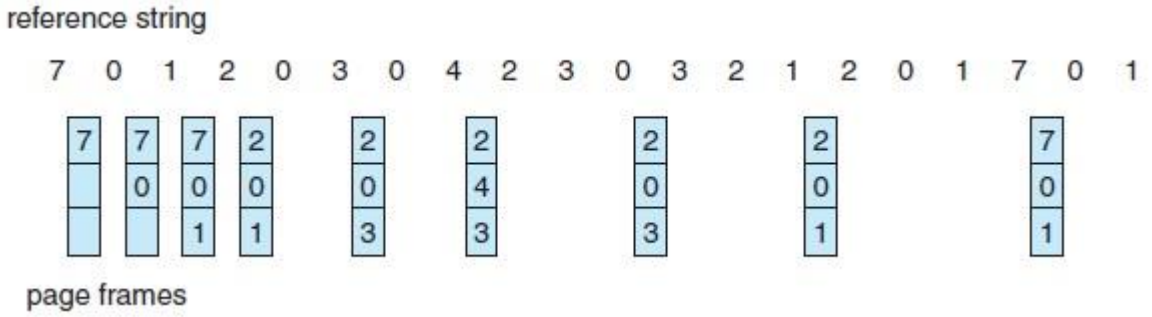
2. Optimal Page Replacement

In this algorithm, pages are replaced which are not used for the longest duration of time in the future. Use of this page-replacement algorithm guarantees the lowest possible page fault rate for a fixed number of frames.

Example: Suppose we have the following process references pages:

7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1

What is the number of page fault if we use three page frames?

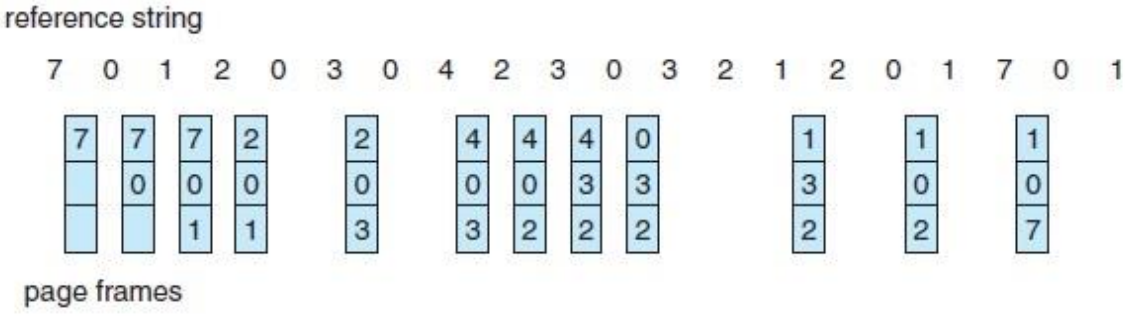


The number of page fault = 9

3. LRU Page Replacement

In least recently used (LRU) replacement algorithm page will be replaced which is least recently used. When a page must be replaced, LRU chooses that page that has not been used for the longest period of time.

Example: Suppose three pages can be in memory at a time per process. Process references pages: 7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1 , what is the number of page fault?



The number of page fault = 12

Disk Performance Optimization ch.8

Introduction

In recent years, processors & memory speeds have increased more rapidly than those of hard disk. As a result, processes requesting data from disk tend to experience long service delay. In this chapter, we discuss how to optimize disk performance by recording disk requests to increase throughput, decrease response time & reduce the variance of response times. We also discuss how OSs reorganize data on disk & exploit buffers & caches to boost performance.
Finally, we discuss Redundant Arrays of Independent Disks (RAIDs), which improve disk access times & fault tolerance by servicing requests using multiple disks at once.

Characteristics of Moving-Head Disk Storage

The general structure of hard disk is shown in fig below. In this figure, we notice the followings:

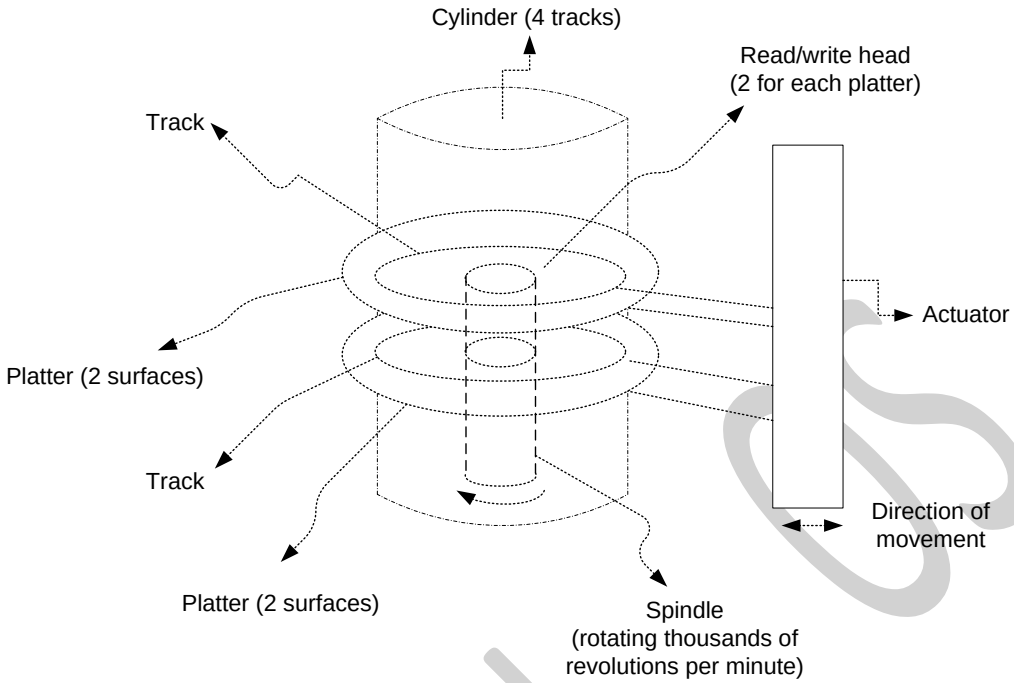


Fig Disk Structure (Schematic Side View)

- The disk storage may consist of several platters & each has a separate read/write moving-head. All heads are fixed to the same actuator & hence move together to select certain cylinder. The cylinder is a set of tracks on all surfaces. Usually, at one time, only one head is active & deals with one track of the whole cylinder. This means that OS has to select the proper head to read/write (r/w) data.
- Each track is divided to several sectors as shown in fig 10.2 each sector is of 512 byte size.

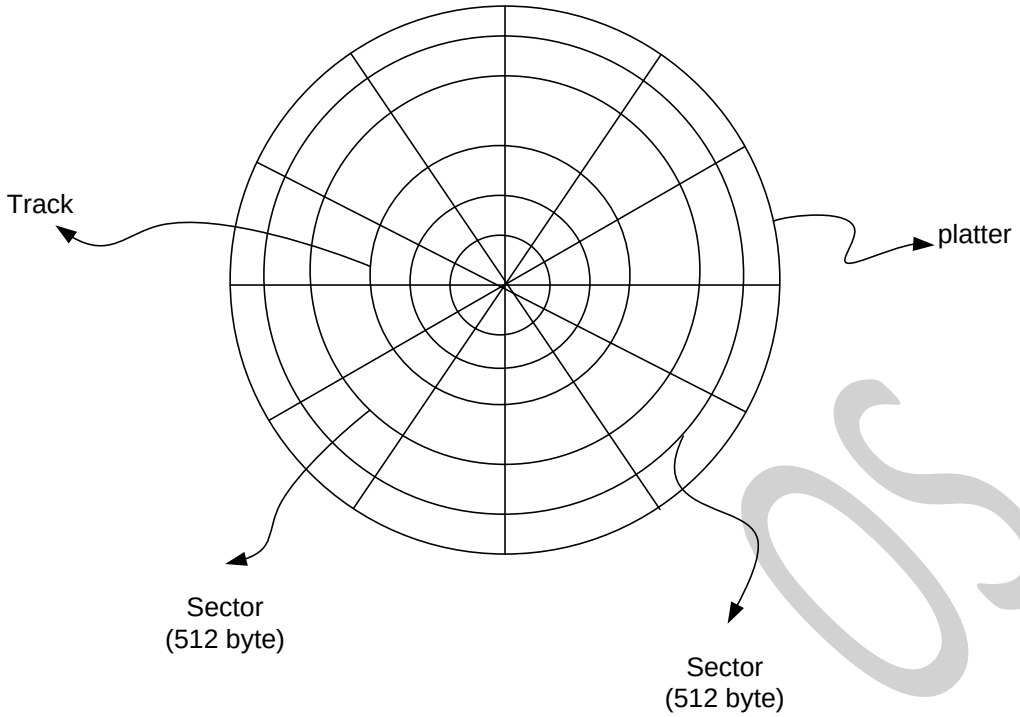


Fig Tracks & Sectors of Disk

From the above, it is clear that for the OS to R/W data from disk, it needs to:

1- Specify the proper surface containing the data & hence the proper moving head.

2- Specify the track & sectors containing the data on that surface.

3- Instruct actuator to move head to the proper track. This movement takes time which is called "Seek Time" & its average value is in the range of few milliseconds (e.g. 7 msec).

4- The platter has to be rotating & the head should wait for the proper sector in the track to get the data.
This time depends on revolution speed & its average value is half of one revolution period & is usually of few milliseconds value (e.g. 4 msec). This time is called "Latency Time".

5- When the head is on the proper sector, it starts reading/writing data & also this process takes time depending on number of sectors to be read.
This time is called "Transmission Time" as shown in Figure.

From the above, It is clear that a few milliseconds are necessary to R/W data from the disk while the CPU can execute millions of instructions in thatv time.

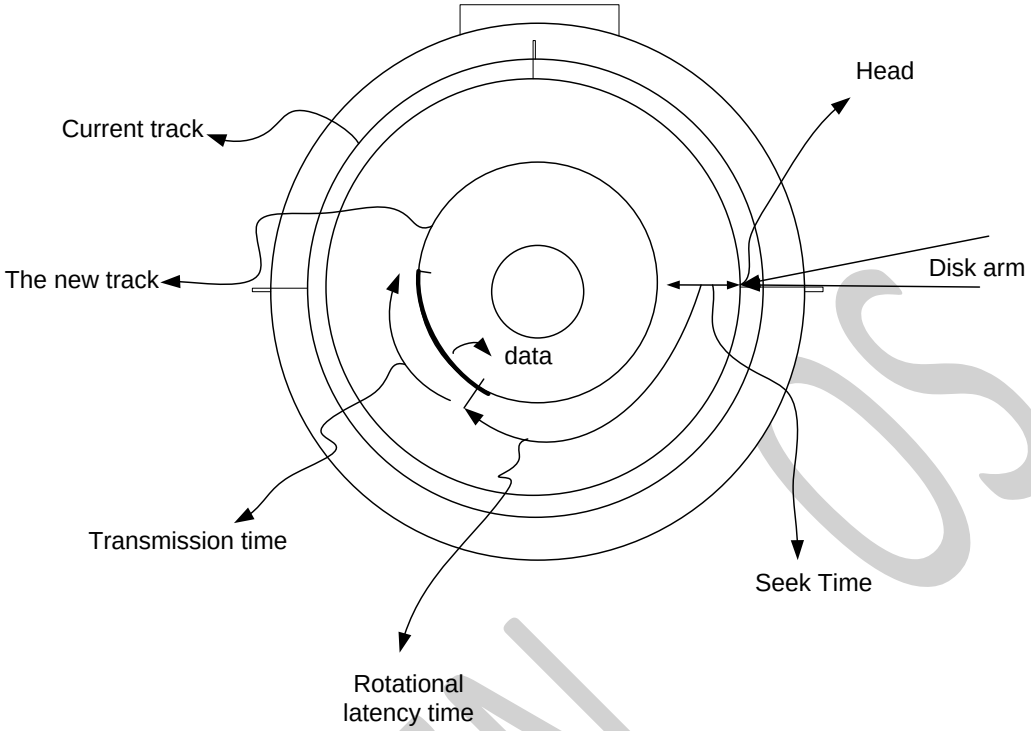


Fig components of disk access (Total time of few milliseconds) e.g. 10 msec

Why Disk Scheduling is Necessary

Many processes can generate requests for reading & writing data on a disk simultaneously. Because these processes sometimes makes requests faster than they can be serviced by the disk, waiting lines or queues build up to hold disk requests. Some early computing systems simply serviced these request on a "First Come First Served FCFS" basis, in which the earliest arriving request is serviced first. FCFS exhibits a random seek pattern in which successive requests can cause time consuming seeks from the innermost to the outermost cylinders (tracks). To reduce the time spent seeking records, it seems reasonable to reorder the request queue in some manners other than FCFS. This process, called disk scheduling, can significantly improve throughput.

The two most common types of scheduling are "Seek optimizing" & "Rotational Optimizing". Because seek times are usually greater than latency times, most scheduling algorithms concentrate on minimizing total seek time for a set of requests.

Disk Scheduling Strategies

The strategies are evaluated by the following criteria:
- Throughput: The number of requests serviced per unit time. The maximum number is the better.
- Mean response time: The average time spent waiting for a request to be serviced. The minimum time is the better.
- Variance of response time: The difference between the request waiting time andthe mean response time. The minimum variance is the better.

Here, it is necessary to check the possibility of a request indefinite postponement.

There are many strategies & we shall discuss some of them as follows:

1 First Come First Served (FCFS) Disk Scheduling

This has been already discussed & it suffers from long seek time & hence low throughput especially under heavy loads.

2 Shortest Seek Time First (SSTF)

In this strategy, the next request to be serviced is the one that is closest to the R/W head & thus incurs the shortest seek time.
The main problem is the possibility of indefinite postponement for the innermost & outermost tracks especially under heavy loads i.e. many requests are coming all the time.

3 Scan Disk Scheduling

Here, the disk head moves from the outer track to the inner & then in the opposite direction. The request to be serviced is the one that its track is ahead of the head in the motion direction.
This means that the requests coming in front of the head in the motion direction are serviced first.
The scheduling may suffer indefinite postponement or long waits for requests of innermost and outermost tracks under heavy load.

4 C-Scan Disk Scheduling

C-Scan mean circular scan & it is similar to SCAN but the head doesn't service requests when moving in the opposite direction i.e. it service requests in only one direction & hence decrease the possibility of indefinite postponement of outerside tracks.

5 Other scheduling strategies

There are also other strategies such as:

- Fscan
- N-Step Scan
- Look Scan
- C-Look Scan
- Shortest Latency Time First (SLTF)
- Shortest Positioning Time First (SPTF)
- Shortest Access Time First (SATF)

Caching & Buffering

Many systems maintain a "disk cache buffer", which is a region of main memory that the OS reserves for disk data. In one context, the reserved memory acts as cache, allowing processes quick access to data that would otherwise need to be fetched from disk. The reserved memory also acts as a buffer, allowing the OS to delay writing modified data until the disk experiences a light load or until the disk head is in a favorable position to improve I/O performance.

The disk cache buffer presents several challenges to OS designers such as:
- Size of cache buffer
- Replacement strategy
- Inconsistency of data when power or system fail.

Many of today's hard disk drives maintain an independent high-speed buffer cache (on board cache) of several megabytes it's not related to main memory i.e. not part of it (i.e. can't be addressed by CPU directly).
Also, some hard disk controllers (e.g. SCSI, RAID) maintain their own buffer cache (normal RAM) separate from main memory.

ALL buffers are used to enhance the disk performance i.e. increase the speed of data retrieval.

Redundant Arrays of Independent Disks (RAID)

Previously, we have been discussing non RAID disks that have the following features:

- The disk includes several platters. Each platter has two R/W heads. ALL the heads are mounted on one actuator & hence move together.

- Usually, the OS determines the location of data on which surface of which platter & instructs the proper head to R/W.

- At any one time, only one head is used for reading or writing i.e. it is not possible to make multiple accesses with several heads.

In other words, the disk has multiple heads but only one of them is used at any one time.

- The file is usually stored on one surface of one platter only unless it is very large.
- The only objective of this disk structure is to get large storage volume.

In the RAID structure, the philosophy is completely different from the nonRAID as it has the following features:

- The disk includes several platters & heads as before but each head here has its own actuator and hence can move independently of the other heads. This will enable multiple reads & writes to be carried out at the same time & hence faster disk response.

- The file may be stored on one platter or on several ones & hence it is possible to read/write several parts of the same file at the same time & this means fast R/W.

- Reliability issue is handled here and hence we find out that an error correcting code (ECC) is being used & hence more storage is needed and this is reason for the term "Redundancy". The redundancy will help in correcting errors & hence the RAID system will be "fault tolerant" in this case.

From the above, we conclude that the RAID structure is equivalent to the use of multiple "Independent" disks & therefore we are going to use the word "disk" instead of platter when describing the RAID technology (RAID structure).

There are several methods for using the disks in the RAID system & these methods are identified by the following names: level0, level1, level2, etc.
We are going to discuss some of these levels in a brief way as showing later.

In RAID systems we use the following terms:

- Data Striping: entail dividing data into fixed size blocks called "strips". Contiguous strips of a file are typically placed on separate disks so that request for file data can be serviced using multiple disks at once, which improves access times.

- Stripe: consists of the set of strips at the same location on each disk of the array.

- Fine grained strips: small size strips & this tend to spread file data across several disks & hence reduce access time.

- Coarse grained strips: large size strips & this enable some files, to fit entirely on one strip & hence the access time is as in the Non-RAID system, for that file, however, several requests for several files can be serviced together (Simultaneously).

Notes: The RAID systems (levels) take into consideration the following factors: Access time (Multiple access for one file), fault tolerance, Multiple accesses (for several files on several disks)

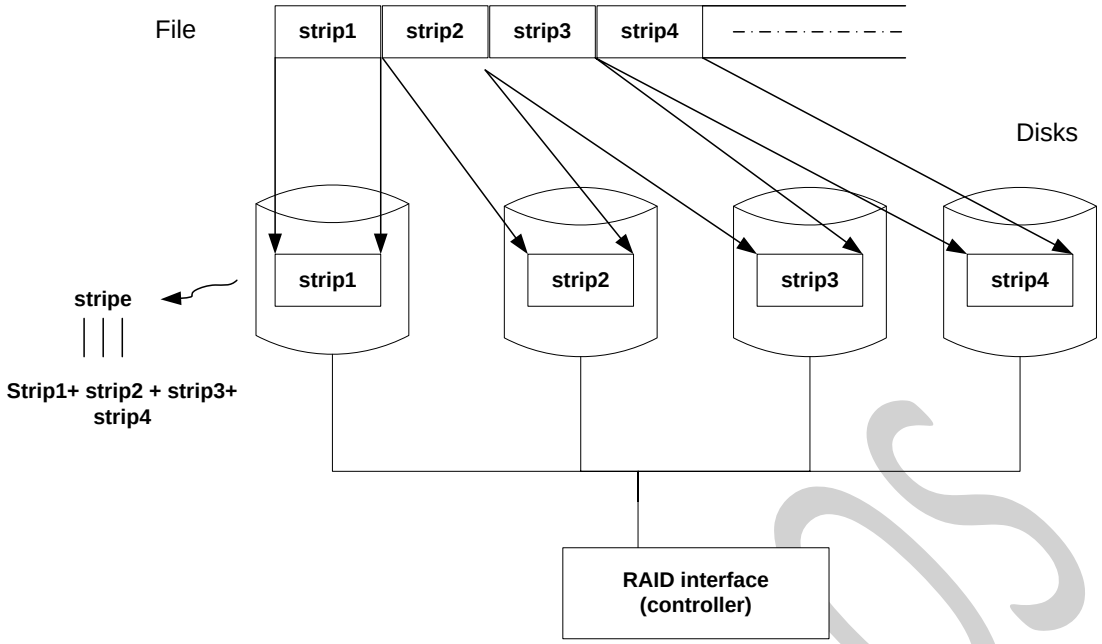


Fig Strips & stripe in RAID systems

RAID Level :

RAID level ○ uses a striped disk array with no fault tolerance and hence has no redundancy. The disk contains data only & there is no ECC data.
In level ○, multiple reads & writes are possible.
The striping level (size) is a block.
Level ○ is sometimes not considered as RAID as it has no redundancy (No ECC).

2 RAID Level1 (Mirroring)

This level employs disk mirroring (shadowing) to provide redundancy, so each disk in the array is duplicated. Stripes are not implemented in level1 & hence multiple access for the same file is only possible on reading but not on writing. On writing, the same data has to be written on both disks (the original & the mirror) but on reading, 2 different parts of the same file can be read at the same time from the original & mirror disks. The mirror technology enhances reliability & restricted multiple access but doubling the cost.

3 RAID Level2

RAID level2 arrays are striped at the bit level, so each strip stores one bit. This means that adjacent bits of file are stored on different disks. Level2 arrays are not mirrored, which reduces the storage overhead incurred by Level1.

The fault tolerance is achieved here by using hamming error correcting codes (hamming ECCs). The error code bits are stored on separate disks (parity disks).
Of course, each stripe of one bit size on data disks has a stripe of one bit size on parity disks. This means that each group of data bits has a corresponding group of ECC bits.
The clear problem here is that if the OS wants to write few bits of the group (stripe), it has to read all data stripes first & then modify the necessary data bits & then calculate the ECC bits & at last store the new data & ECC bits. This is called "read-modify-write" cycle.
From the above, we notice that the storage overhead is decreased compared to mirror system but the multiple access for several files is not possible as all disks will be occupied for one file request (remember that the adjacent file bits are distributed among the disks, also, it is necessary to read the parity disks).

✓ ***Note:*** In hamming, we can correct one data bit error. The number of ECC bits are as follows:

Number of data bits	**number of ECC bits**
3	
11	4
26	5
27	

It is clear that the larger data stripe, the better & hence the more data disks in the array (stripe) are the better.

4 RAID Level3

RAID level3 stripes data at the bit or byte level but use parity checks for fault tolerance instead of Hamming. In parity check, we use only one bit (even or odd parity) & this bit does not locate the place of error (as incase of Hamming) but indicates only its existence. When the error occurs, the OS will inform the user immediately who has to find out the erroneous disk and replace it. The data on the faulty disk can be regenerated automatically with the help of other data disks & the parity disk. The advantages of such system:

1- Large storage.
2- Fault tolerance with one extra disk (one parity disk).
3- Multiple access for one file is possible and hence fast access time.
Of course, multiple access for several files is not possible as any file request will occupy all of the disks.

5 RAID Level4

RAID Level4 systems are striped using fixed size blocks (typically much larger than a byte) & use one disk for parity (even or odd parity). The difference with level3 is that the file may occupy fraction of disks & not all of them (remember the coarse grained stripes) & hence multiple requests for multiple files may be possible at the same time. Here, we should remember that when reading data from disks, it is not always necessary to read parity bits as these bits are stored not for error detection but mainly for error correction (of one bit –usually one disk may be faulty-).

✓ ***Note:*** Multiple writes is not possible because the parity disk will be occupied for one write.

6 RAID Level5

RAID Level5 arrays are striped at the block level & a parity check (even or odd) is used like in level4. The difference with level4 is that the parity bits are not located on one disk but distributed throughout the arrays of disks. This means that disk1 carries parity bit1, disk2 carries parity bit2, & so on. The advantage of this level is that multiple writes (writes for several files) are possible because the parity bits are not stored on one disk as the case of level4.

7 Other RAID Levels

There are other levels such as:

- RAID Level6
- RAID Level10+1
- RAID Level10
- RAID Level10+3, 0+5, 50, 1+5, etc.

8 comparison of RAID Levels

The properties of different RAID levels can be summarized as follows:

From multiple files

RAID Level	Read concurrency	Write concurrency	Redundancy	Striping Level
0	Yes	Yes	None	Block
1	Yes	No	Mirroring	None
2	No	No	Hamming ECC	Bit
3	No	No	Even or odd parity	Bit/Byte
4	Yes	No	Even or odd parity	Block
5	Yes	Yes	Distributed even or odd parity	Block

✓ ***Notes:***

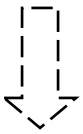
1- When striping level is bit or byte. This means that the file data are distributed on all the disk arrays & hence it is not possible to service multiple requests for several files, however, the single file will be read/write very quickly i.e. fast access (fast data transmission).

When block level is used then the file may occupy few disks of the array (stripe) & the others may be for other file, & hence multiple files may be serviced.

2- The main purpose of redundancy is not to detect errors but to correct it & hence, it is not always necessary to read parity disks during read cycle & this allows multiple read for several files when block (coarse grained striping) striping is used.

3- Hamming ECC is not necessary as the faulty disk may be discovered by other means & hence one parity (even or odd) bit is enough for error detection & correction. In other words, the parity bit will inform us about the existence of error and then by other means (electrical, mechanical) we can find out the faulty disk & hence regenerates its data from knowing the other data bits on the data disks & the parity bit from the parity disk.

✓ ***Note:*** in RAID, we discover that there are errors by using parity check. Then by some means we find the faulty disk & then we regenerate the data on the faulty disk by making parity of all other data disks (except the faulty one) & of the parity disk itself.



Conclusions;

The parity check can be used to correct data if the erroneous bit location is known.
Once we know this location, we can find the missing bit by making parity of all other data bits & the parity bit.
In RAID, we know location by knowing the faulty disk by different means & this starts by finding parity check error.

From the above we notice that RAID systems features are:

- Large storage volume as it uses arrays of disk.
- Fast data transmission as many heads work together (Independent Disks).
- Fault tolerance with low overhead storage by using parity check (redundancy).