1 Introduction To Operating System

1.1 OPERATING SYSTEM FUNDAMENTALS

Introduction:

Definition

"An operating system is a software which acts as a mediator/interface in between the user and computer hardware." In other words, an OS is a software that performs work such as memory management, process management, file management, etc.

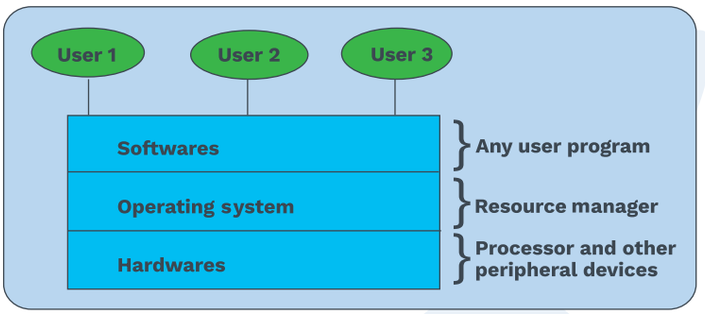


Fig. 1.1 Basic Overlay of Computer System

Operating System acts as Resource Manager/Allocator because it takes care of the responsibility to allocate the CPU to some process and I/O to some other process.
Software/Applications communicate with the hardware through Operating System (O.S.) using system calls. For different kinds of services, there exist different system calls.

Objectives of an operating system:
Mainly two objectives: Primary and Secondary.

1) **Primary objective: Convenience**
   The main aim of an operating system is to give convenience to the user. Users can perform work like process scheduling, and conversion of user code to machine code easily and conveniently with the help of OS.
2) **Secondary objective: Efficiency**
   It is a task of the operating system to manage the resources in such a way that resources are not kept to be idle and can be utilized efficiently.

Note:

The primary and secondary objective of an operating system is not the same for all operating systems.

- It can vary depending on the purpose of the operating system.

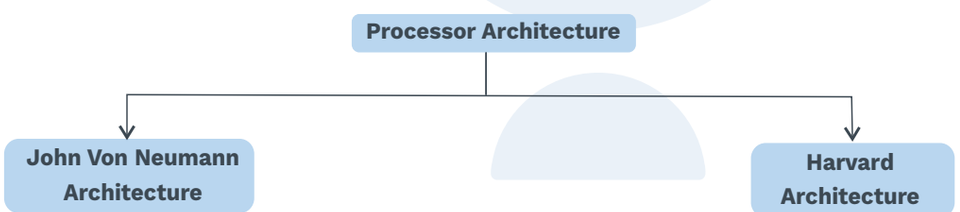
For example:

- The primary goal of WINDOWS → Convenience
- The primary goal of LINUX → Efficiency

Note:

Design and principles of operating systems directly depend on the computer architecture.

Types of Architecture



John von neumann architecture:

- John Von Neumann is considered one of the greatest mathematicians and computer scientists who came up with a simple and elegant architecture of what a computer is. Neumann architecture is also known as stored-program concept. The concept of a stored program states that any program we want to execute is going to be stored in RAM.
- A program is a sequence of instructions that are there in machine level language. The CPU reads these instructions from memory and executes them one by one in a sequence.
- In this architecture, physical memory is used to store instructions and data both.
- There is a common bus for data and instruction transfer.
- Instructions usually take 4 or more clock cycles to execute.
- Cost-wise cheaper.
- Instructions, as well as read/write, cannot be accessed by the CPU at the same time.
- In personal and small computers, this structure is used.

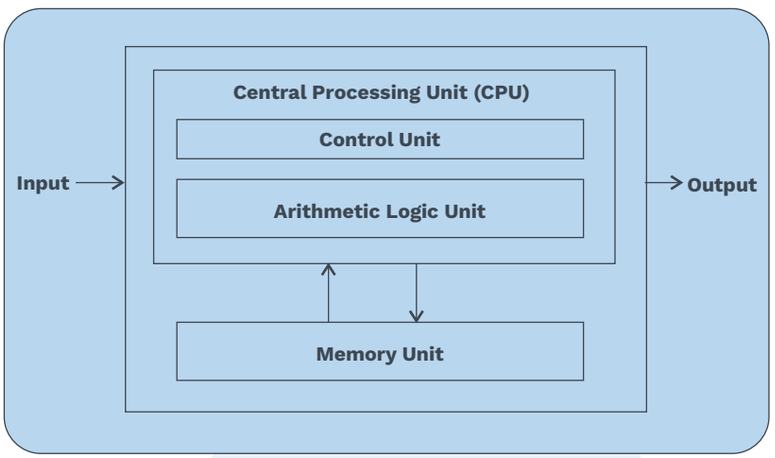
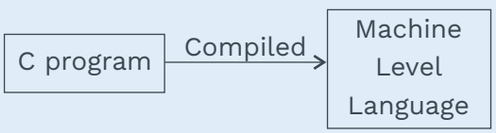


Fig. 1.2 Central Processing Unit (CPU)

Note:



Machine Level Language: The sequence of instructions specific to a microprocessor is stored in RAM.

Harvard Architecture:

In Von Neumann architecture, same memory is used to store instructions and data and a common bus is multiplexed to read/write instructions and data. The CPU is bound to read the instruction and read/write data alternatively in Von Neumann architecture. In Harvard Architecture separate buses are used for instruction and data. It is considered an advancement of Von Neumann Architecture. The CPU can read the instruction and read/write data in parallel in Harvard architecture.

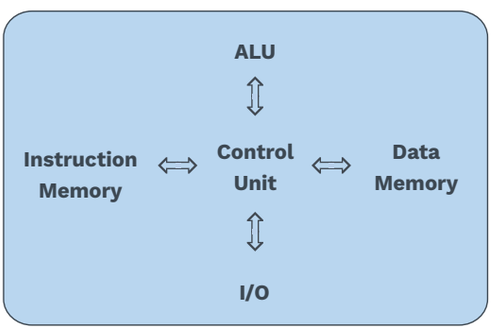
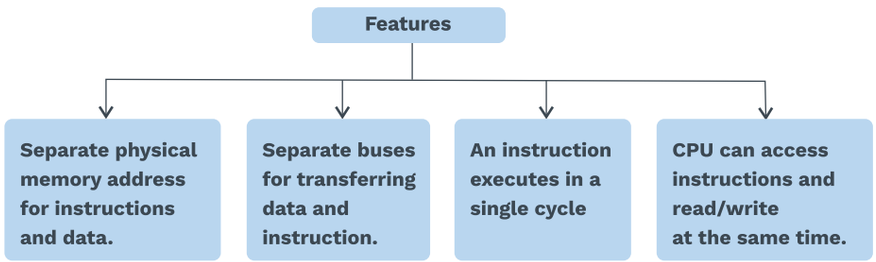


Fig. 1.3 Harvard Model



Note:

- Harvard architecture is more costly than John Von Neumann architecture
- Harvard architecture is used in microcontrollers and signal processing.

History of operating system:

1) First Generation (1940's -1950's):

In that era, punch cards were used. There was no operating system. Computers were of big size.

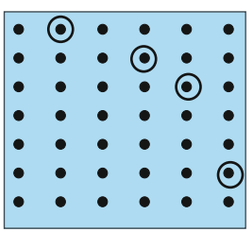


Fig. 1.4 Punch Card

Each line would represent some set of instructions. They would punch a hole and use electron techniques to read these codes from punch cards and convert them into a program. The magnetic Drum was the form of main memory.

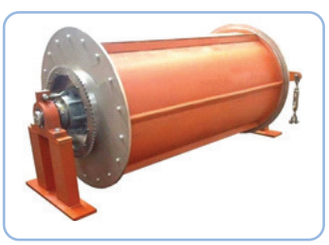


Fig. 1.5 Magnetic Drum

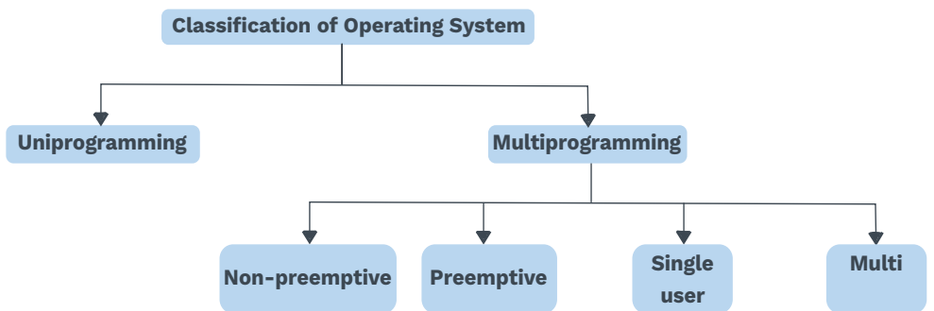
2) **Second generation (1950 – 1970):**
Still no operating system, people started using magnetic tapes as the permanent way to store data, e.g: Cassette tapes etc.
Data is moved from these tapes to RAM and then processed. So it can store more programs compared to First Generation Punch Cards. This era was called Batch Processing Era.
3) **Third generation (1980 – 1990):**
Operating Systems started to boom, and also disk technology became popular. These were early stages of First Hand Operating Systems like MS-DOS, Unix etc., which were built in this generation. Ram size has drastically increased and hard disk was introduced Multiprogramming also became popular.
4) **Fourth generation (2000 – present):**
New function specific Operating Systems were built like Network Operating Systems. Also called distributed Operating Systems, where we have hundreds of computers connected in the network. Real Time Operating Systems etc. were made and became popular for their specific purposes.

Functions of an operating system

1) **Process management:**
- A situation when the ready queue has multiple processes, but the processor can process only one process at a time.
- In such a situation, the CPU has to schedule processes one at a time such that every process gets a fair chance to execute.
- It means the CPU should not give priority to one particular process.
Some well-known CPU scheduling algorithms:
- FCFS(First Come First Serve), RR(Round Robin), etc.

2) **Memory management:**
For any process execution, it needs to be placed in memory first. The memory gets free after the completion of process execution, and other processes can use that memory. OS allocates and deallocates the memory for the process resulting in efficient use of memory.
3) **I/O Device management:**
- While executing a process, the process needs access to different I/O devices.
- But a process can not access any of the I/O devices directly. It requires to take the help of OS to access these devices.
4) **File management:**
A file system residing within the operating system manages different kinds of files, folders(directories) within the computer.

Classification of operating system:



Uniprogramming:

- In uniprogramming system can execute only one program at once.
- A lot of Wastage of CPU resources occurs here.
- We were using uniprogramming in old computers and mobiles.

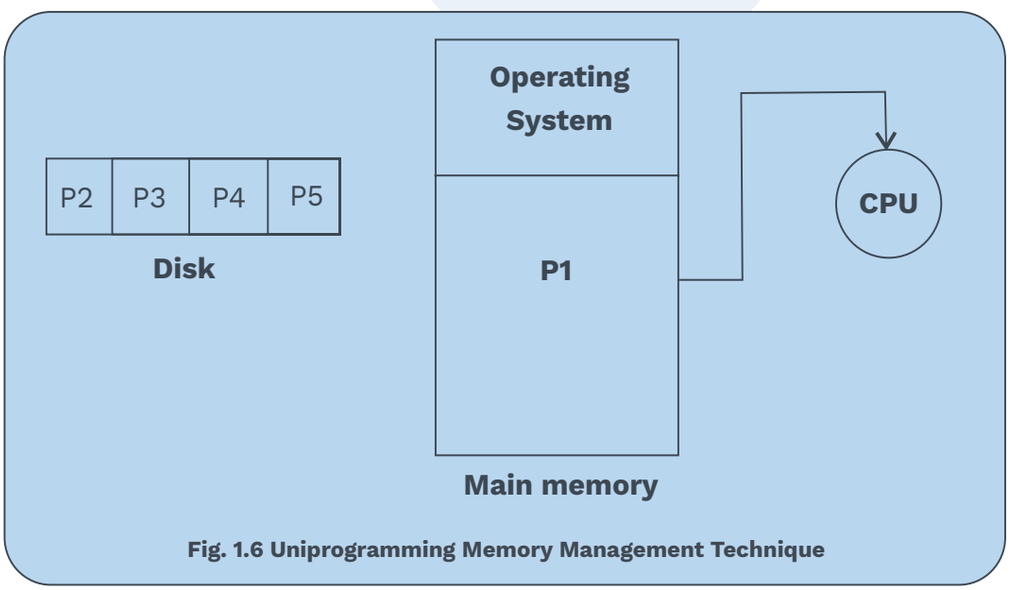


Fig. 1.6 Uniprogramming Memory Management Technique

Example:

1) Batch handling in old computers and mobiles.
2) Old portable working system.
3) Some operating systems which is Uniprogramming are Batch operating system, MS-DOS etc.

Multiprogramming:

- Multiprogramming came into the picture to overcome the shortcomings of Uniprogramming.
- Operating system is able to manage multiple ready to run programs in memory at once.

Note:

Context Switching:

It is the procedure of storing the state of a process or thread to be restored and resume execution later, so all programs are given a suitable amount of time.

In a multiprogramming environment processor needs to run various process, so processes need context switching from CPU to memory frequently, because only one process can run on the CPU at a time.

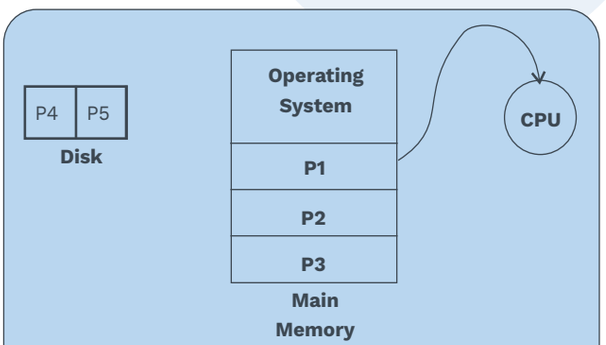


Fig. 1.7 Multiprogramming Memory Management Technique

Example:

Multiprogramming, Multitasking, Real-Time and Modern Operating Systems like Windows 7, 8, 10.

Types of multiprogramming:

1) **Non-Preemptive based:**
   In this case, the process currently running releases the CPU voluntarily:
   The process releases CPU:
   i) When they require I/O
   ii) When the process invokes the system call
   iii) When the process gets completed.

Drawbacks:

a) Starvation **b)** Less Response time overall

e.g., First Version of Windows, i.e. Windows 3.0, 3.11 etc.

2) Preemptive based:

It allows forceful reallocation of the CPU from the current running process to another process from the process pool.

When the running process can be preempted?

i) When I/O required

ii) Time quantum exhausts (RR Scheduling)

ii) High priority process comes

3) Single user based:

A Single user does all the programs submitted to the main memory for an execution. E.g., Windows XP, MS-DOS, Palm OS, etc.

4) Multi user based:

Programs from multiple users are submitted to the main memory for an execution.

E.g., Unix, Linux, Web-server based etc.

Architecture support for multiprogramming:

1) Address translation (memory):

The process of translating Logical Address(Disk Address) to Physical Address (Main Memory Address)

2) DMA (direct memory access)/IO device:

Independent to the CPU

Simultaneous working with CPU.

3) Two modes of CPU execution:

User Mode

Kernel Mode

Operating system types:

Various operating systems exist for various purposes.

Batch operating system:

In a batch operating system, similar jobs get grouped into batches with the help of some operator, and these batches get executed one by one. For example, let us assume that we have two programs that need to be executed and need Input-Output devices. While running, when a process leaves the CPU for the Input-Output device. It cannot load another program into the CPU until the currently running process terminates successfully.

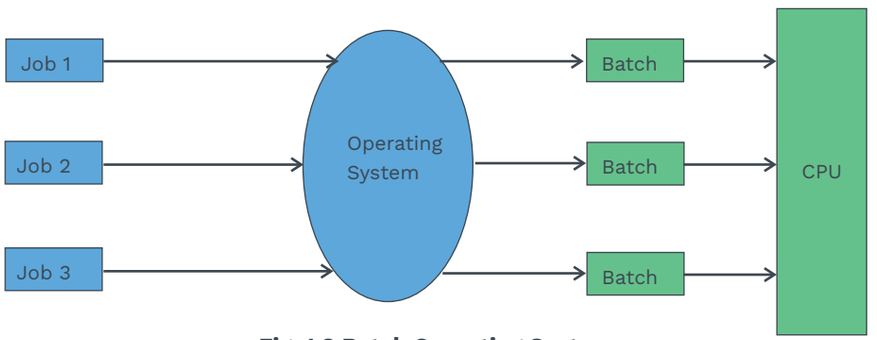


Fig. 1.8 Batch Operating System

- Batch OS is not good for interactive applications.
- In this OS, there is an operator that takes similar jobs together and groups them into batches.

Examples of batch os:
Payroll System, Bank Statement, etc.

Advantages:
i) Repeated jobs are done fast in batch systems without user interaction.
Disadvantages:
i) Increased CPU idleness.
ii) Decreased throughput of the system.

Note:
Throughput: Number of jobs completed per unit time.

Multiprogramming Operating System:
In today's scenario, a system has to execute various applications simultaneously. The operating system has to provide the required resources to all the applications for efficient and effective execution. The main memory accommodates several ready to execute jobs, and the operating system schedules them one after another on the CPU for execution.
In this multiprogramming environment, when an executing process needs I/O, it gets blocked and waits in I/O queue, by the time the operating system schedules other processes from the ready queue to execute on the CPU. The operating system does the scheduling processes repeatedly.

Advantages:
i) Throughput of system increases.
ii) Processor idle time is less. Despite a process waiting for the I/O event to happen; meanwhile, it keeps busy with another process.

Disadvantages:

i) Requires efficient memory management.

ii) Requires CPU Scheduling

iii) Tracking of all processes is sometimes complex.

Multitasking operating system:

Multitasking is considered an advancement to multiprogramming. A multitasking operating system simultaneously executes multiple tasks residing in the main memory. When a system has multiple processors, and they can be used to execute multiple instructions parallelly then it is known as multiprocessing. Hence multiple processes are required for multitasking while a single processor is needed to do multitasking.

In a modern computer, many tasks like text editor, web browser, mp3 player, etc can be executed simultaneously by a user. The multitasking operating system switches these tasks very fast, which users can't notice and get a feel of parallel execution of these tasks.

The only difference between multitasking and multiprogramming is that the later works on the concept of context switching, whereas the former works on time-sharing alongside context switching.

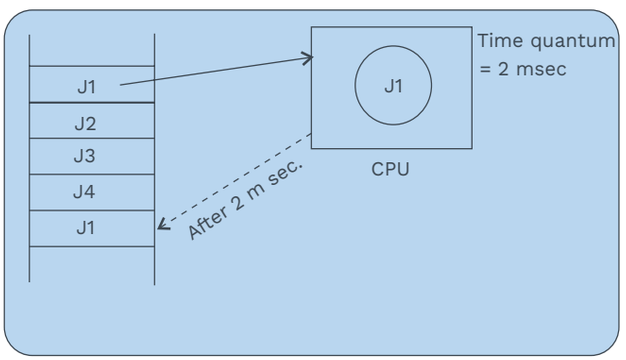


Fig. 1.9 Multitasking Operating System

Each job has a time constraint with it only inside which job can remain inside CPU. So, each job gets its turn so fast it looks as if every process is executing simultaneously.

Real-time operating system

Definition

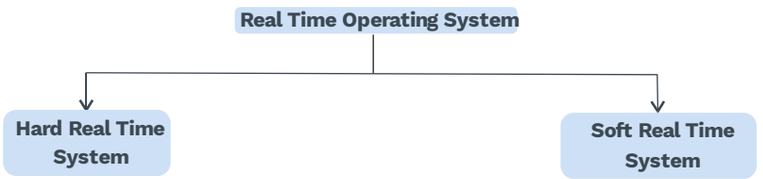
An OS that processes data as it comes in without any buffer delays.

It works on strict deadline. Response must be given within the deadline.

Examples: Traffic Control Systems, Missile Systems, etc.

Note:

The time interval required to process and respond to inputs is known as response time and it needs to be minimum.



Hard real time system:
An OS that works for those applications whose deadlines are very strict.
Example: Satellite control

Soft real time system:
These OS works for applications whose deadlines are narrow(less strict).
Real Time OS works mainly on primary memory only.
Example: Banking

Advantages of real time os:
Error free
Gives more focus on running applications.
Utilization of devices occurs in best possible way.

Disadvantages/Restrictions of Real Time OS:
1) Limited task
2) Use heavy system resources
3) Complex Algorithm
4) Thread priority

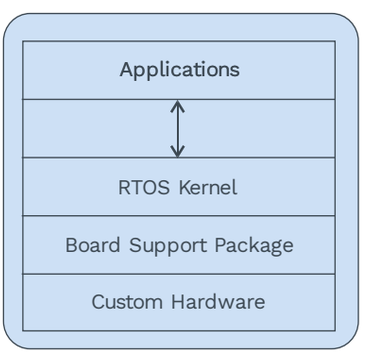


Fig. 1.10 Block Diagram of RTOS

Rack Your Brain

Can we use multitasking OS in airplane control systems?

Distributed OS:

Definition

"A distributed operating system is system software over a collection of independent, networked, communicating, and physically separate computational nodes. The puspose of this OS is to tackle the jobs that are executed by many CPUs."

In the distributed system(loosely coupled systems), there is a shared communication-network through which different interconnected computers can communicate.
Where all the individual systems have their own CPU and memory unit.
The function and size of the processor of these systems are different.
Benefit: Any user can access the files that are present in any other connected system in the network but absent in users own system.

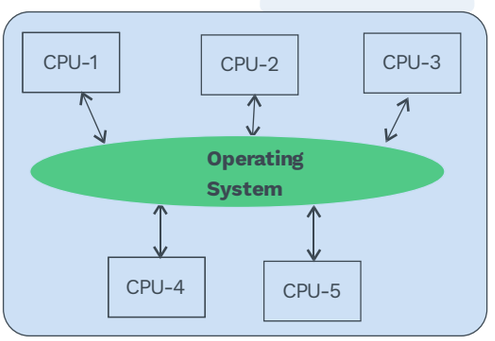


Fig. 1.11 Block Diagram of Distributed Operating System

Advantages:

1) No single point of failure
2) Efficiency and Interoperability
3) More room for Innovation

Disadvantages:

1) More things to secure
2) Complex System Design
3) Monitoring / Security

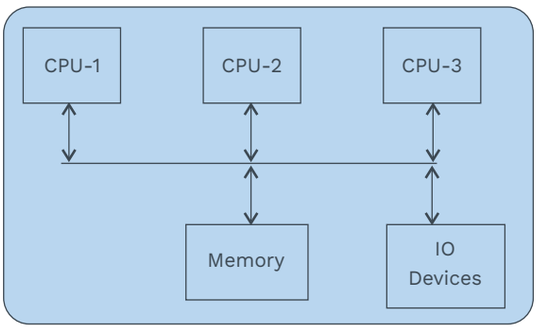


Fig. 1.12 Block Diagram of Multi Processor Systems

Multi processor systems

A single computer system with more than one processor uses an operating system called multiprocessor operating system. In this system the multiple processors share common computer bus, memory, peripheral devices, etc.
eg.: Unix

Advantages:

1) Throughput of the system will enhance.
2) Reliability of the system becomes high

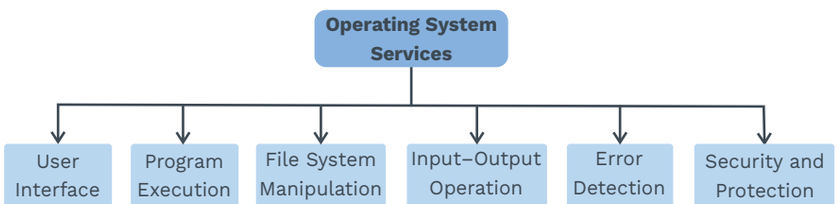
Disadvantages:

1) Increased Expense
2) Complicated Operating System code
3) Large Main Memory Required

Services of OS:

OS works as an interface between user and hardware that provides some services to the user to implement the program.
OS services is not same for every OS, it changes from one OS to another.
All these services are given by OS to make the programming easy.

These services are as follows:



1) **User interface:**
   The operating system is designed in a such a way that it provides a user interface which is easy to use by a novice user.
   Example: Batch Interface, GUI(Graphical User Interface)
2) **Program execution:**
   OS provides the feature to load the program into a memory and to execute it.
3) **File-system manipulation:**
   OS provides feature like read/write a file,creation and deletion of the files, etc to the programmer.
4) **Input-output operations:**
   In order to maintain the protection, the I/O devices cannot be controlled by the users directly, OS helps user to access these devices.
5) **Error detection:**
   OS is responsible to handle any kind of errors whether it is present in hardware, I/O devices, within network.
   For all kind of error, a proper action needs to be taken by an OS.
6) **Protection and security:**
   Sometimes in a system where multiusers are present, OS plays an important role to ensure security and protection by controlling the user or process to access only those resources and information that are required for them.

Booting of Operating System:

All the components of a hardware as well as OS have to work properly for a computer system to start(boot).
It will result into a failed boot sequence if any of these elements fail.

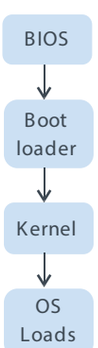


Fig. 1.13 Steps involved in a Boot Process

Booting is a startup sequence that starts the Operating System of a computer when it is turned on.

- When the computer is switched on, the boot program is the first to get executed, and it starts the OS in the computer.
- Boot sequence is available in every computer.
- Kernel is located by the bootstrap loader which loads the kernal into the main memory, and then the execution starts.

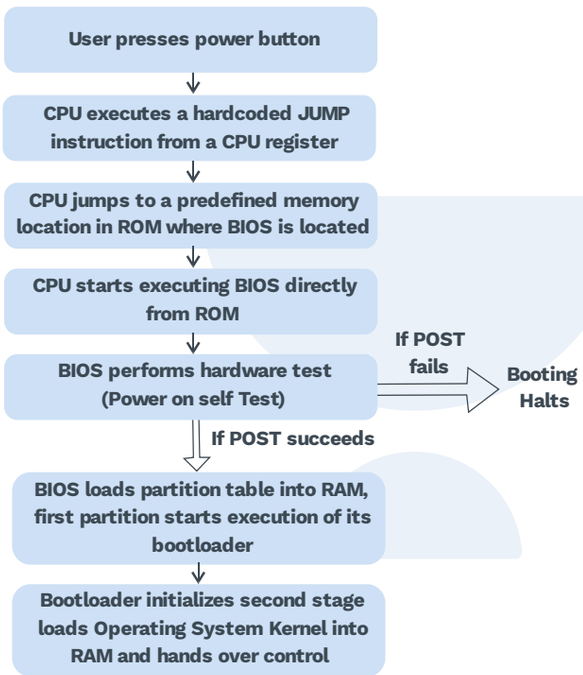


Fig. 1.14 Computer Booting Sequence

Rack Your Brain

How booting process will change if there are more than one OS installed in computer system?

System Calls:

Definition

System call is a request for the kernel to access a resource.
It gives an interface to the services provided by the OS.
These are typically written in language C, C++.

Example:

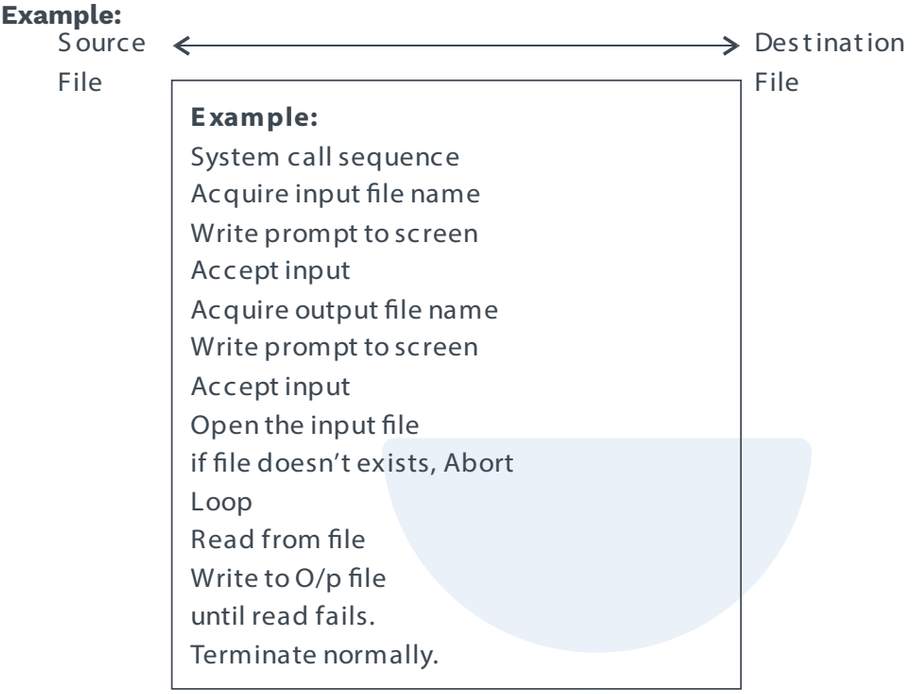


Fig. 1.15 Example of How System Calls are Used

In the above figure, even a small operation of copying a file into another file takes many sequences of system calls to the hardware. Most users never see this level of detail because application developers design programs according to an API (Application Programming Interface).

Application programming interface(API):

Definition

"The API specifies a set of functions that are available to an application programmer, including the parameteres that are passed to each function and the return values the programmer can expect."

Handling of system call:

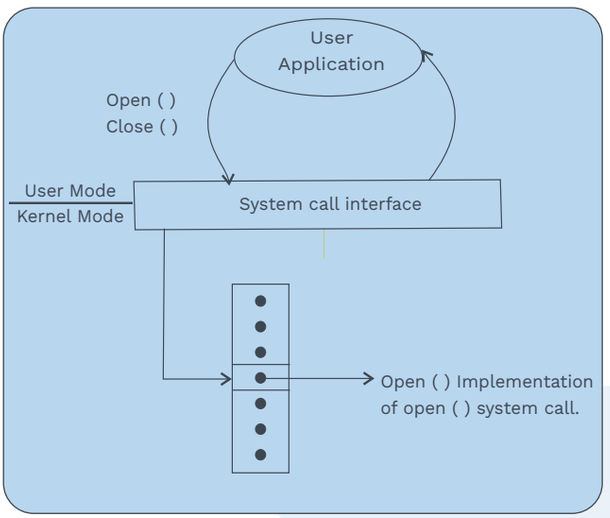


Fig. 1.16 Open ( ) System Call Handlings

1) Usually, for any programming language, an interface to system call is given by the runtime support system.

2) Each system call is assigned a number and an indexed table is managed by the system call-interface.
A system call is then invoked in the kernel and the status of the system call is returned by the interface

3) User need not worry about how the system call is working, the caller of the system call just has to follow the API and see the result after the execution of the system call.

4) Methods for passing parameters to the OS.

- **i)** Registers — less size parameter
- **ii)** Block or table — address of the block is passed
- **iii)** Stack method — This does not limit the number of parameters being passed.

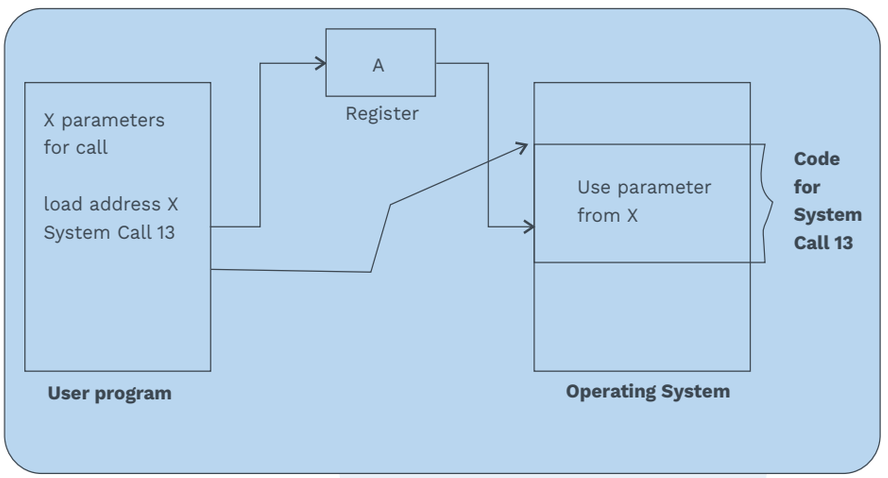


Fig. 1.17 Passing of Parameters as a Table

Types of System Calls:

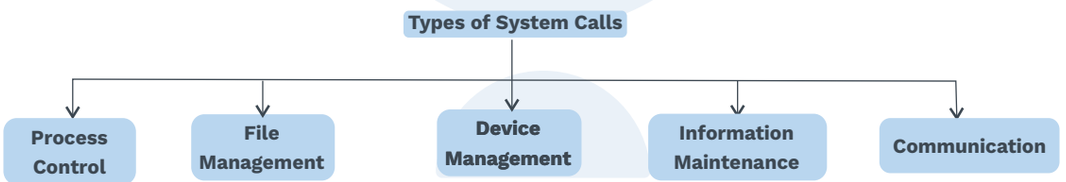


Fig. 1.18 : Types of System Calls

Process Control:

Process Control			
1)	end, abort	**2)**	load, execute
3)	create process, terminate process	**4)**	wait for time
5)	wait event, signal event	**6)**	allocate and free memory

Table 1.1

File management:

File Management	
1)	create file, delete file
2)	open, close
3)	read, write, reposition

Table 1.2

Device management:

Device Management	
1)	request device, release device
2)	get device attributes, set device attributes

Table 1.3

Information maintenance:

Information Maintenance	
1)	get time or date, set time or date
2)	get system data, set system data

Table 1.4

Communication:

Communication	
1)	send messages, receive messages
2)	transfer status information
3)	create, delete communication connections

Table 1.5

System call examples:

1) **wait ( )**
   Some processes are interdependent i.e. execution of one process depends on the other process execution.
   For instance, during fork() if the execution of parent process is dependent on the execution of child process then during the time child process completes its execution, parent process gets suspended and this suspension is done by wait().
2) **exec ( )**
   The purpose of exec() is to execute a .exe file in place of the file which was executing previously.
3) **fork ( )**
   The purpose of fork() is to create a copy of a process itself.
   Execution of fork () at runtime will create a child process i.e a new process.
   Child process code is exact same as the parent one.
   The resources used by child process like cpu registers, PC are same as parent process.
4) **exit ( )**
   The purpose of exit() is to terminate the execution of the ongoing program.
   After using exit() system call, OS can claim again for the resources used earlier by the process.

Rack Your Brain

What would happen if user process can directly access hardware resources without system calls?

1.2 DUAL-MODE IN OPERATING SYSTEM

User mode (Non-Privileged mode) and Kernel mode (Privileged Mode):

- Every CPU operates in the above two modes.
- All operating system services execute in the Kernel mode, with atomic execution i.e. non-preemptively.
- All user applications runs in user mode pre-emptively.
- There is a bit known as mode bit that indicates in which mode processor is currently working.
- Shifting of mode occurs according to the requirement.

e.g.: When a service is requested by a process via OS, mode shift changes from user mode to kernel mode.

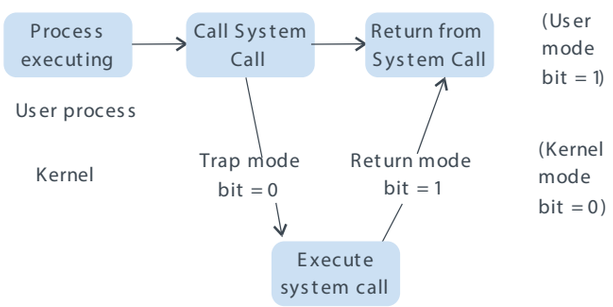


Fig. 1.19 Transition from User to Kernel Mode

- Mode bit is added to the hardware that represents the current mode.
- When mode bit = 0 means representing kernel mode.
- When mode bit = 1 means representing user mode.
- All those instructions that are privileged one must have to run in kernel-mode. They cannot run in user mode otherwise these instructions will be considered as an illegal one.
- Input-Output Control, management of interrupts are the example of privileged instruction.
- When a system starts, the hardware operates in the kernel mode.
- The operating system is then loaded and starts user applications in user mode.
- When an interrupt or system call occurs, it leads to a supervisory call which handles ISR(Interrupt Service Routine), when a supervisory call is generated, ISR changes mode bit in PSW(Program Status Word) from 1 to 0 i.e from user to kernel mode.
- After completion of all the functions in the kernel mode,again supervisory call is generated and mode bit changes from 0 to 1 i.e. kernel to user mode.
- It means OS implements its functionality only when it is in kernel mode.

Rack Your Brain

What would happen if system has only one mode for program execution?

1.3 INTERRUPTS

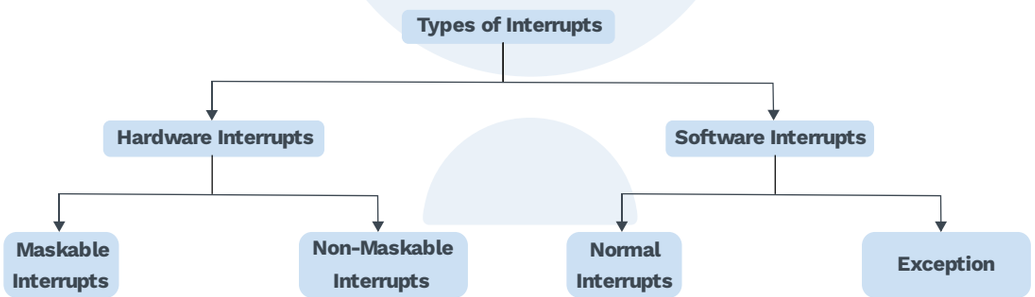
It is a signal initiated by a software or a hardware when any of the process needs sudden attention.
External interrupt occurs due to hardware.
Internal interrupt occurs due to software instructions.

It is also called as software interrupt.
Divide by 0, protection violation are the examples of internal interrupts.

What are the steps to handle a interrupt?
Steps:

- Suppose an interrupt is initiated by a device and by that time process P1 was executing.
- Processor will first complete its execution.
- The address of interrupted instruction is saved to a location that is temporary one.
- After that it will load the PC(program counter) with the address of the 1st instruction of ISR(Interrupt service routine).
- After completion of new request (Interrupted request), processor will start excuting next process P_1 +1.

Types of Interrupts:



1) **Hardware Interrupts:**
   When interrupt occurs due to the external devices.
   i) **Maskable Interrupts:**
      Those interrupts can be disabled if high priority interrupt comes. **eg.** RST 6.5, RST 7.5, RST 5.5 of 8085 microprocessor.
   ii) **Non-Maskable Interrupts:**
      Those interrupts that needs to process immediately. **eg.** Trap of 8085 microprocessor.
2) **Software Interrupts:**
   i) **Normal Interrupts:**
      Those interrupts that occur due to software-instruction.
   ii) **Exception:**
      Some unexpected events occur while a program is executing.
   eg. Trap, Divide by 0 etc.

PRACTICE QUESTIONS

Q1 **Operating systems manage _____.**
a) Only hardware
b) Only user programs
c) Both hardware and user programs
d) Hardware and OS modules only

Sol: **c)**
Apart from managing hardware components, operating systems manage applications programs and other software abstractions like virtual machines.

Q2 **Kernel is _____.**
a) Designed by processor manufacturer
b) Designed by the OS designer
c) Part of processor
d) A central control point of an OS

Sol: **b), d).**
Operating system is a collection of several programs; kernel is the program which controls other programs and services of the OS. Kernel is designed by the OS designer.

Q3 **Can operating systems be portable?**
(Note: Answer 1 for TRUE and 0 for FALSE)

Sol: **1**
Yes, few operating systems are portable which can be directly booted from flash or USB drives. Portable OS is mainly used to recover data on a disk after system crashes.

Q4 **A user is allowed to read/write _____.**
a) In any section of the disk
b) In user specific sections of the disk only
c) In volatile memory only
d) Nowhere in the disk

Sol: **b)**

A user should not be allowed to read/write in any section of the disk because knowingly or unknowingly it can overwrite operating system files or other sensitive data which can cause in crashing the system.

Q5 **Which one of the following statements is correct?**

a) A multi-tasking operating system must be installed on a multiprocessor system to execute multiple processes simultaneously

b) A multi-tasking operating system executes multiple processes simultaneously on a uniprocessor system

c) A and B both

d) None

Sol: **b)**

A multi-tasking operating system executes multiple processes simultaneously on a uniprocessor system by switching the processes on CPU in time sharing fashion.

Chapter Summary

- **Operating System:** It is an environment, resource allocator, interface between hardware and computer user.
- **Goals of an Operating System** : Convenience
  Efficiency
- **Function of Operating System** : Process Management
  Memory Management
  I/O Device Management
  File Management
- **Types of Operating System** : Batch Operating System
  Multiprogramming Operating System
  Multi-tasking Operating System
  Real-Time Operating System
  Multiprocessor System
- **History Operating System** : 1st Generation → [1940 – 1950] Punch Cards
  2nd Generation → [1950 – 1970] Magnetic Tapes
  3rd Generation → [1980 – 1990] Multiprogramming
  4th Generation → [2000– Present] Modern OS
- **OS Services** : GUI (User Interface)
  Program Execution
  File System Manipulation
  I/O Operations
  Error Detection
  Protection and Security
- **OS Operations** : Dual mode
  (i) Kernel mode
  (ii) User mode

S.No.	Kernel Mode	User Mode
1)	Unrestricted and full access to computer hardware	Limited access to system's hardware.
2)	Only core functionality can be allowed to operate	User application are allowed to operate.
3)	System Crashes are fatal and increases the complexity.	System crashes are recoverable.

Table 1.6

- **Booting of an OS:**

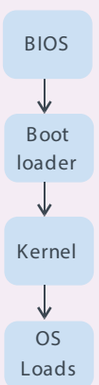


Fig. 1.18 Steps Involved in a Boot Process

2 Process Management

2.1 PROCESSES

Basics of process:

- A program under execution is known as a process.
- A process is different from a program/text section as a process also includes the current activity as depicted by the program counter and contents of the CPU.
- A process also contains a process stack, which holds temporary data like function parameters, return address and local variables, and heap memory allocated to the process dynamically during run time.
- It contains a data section/global section which has global variables.

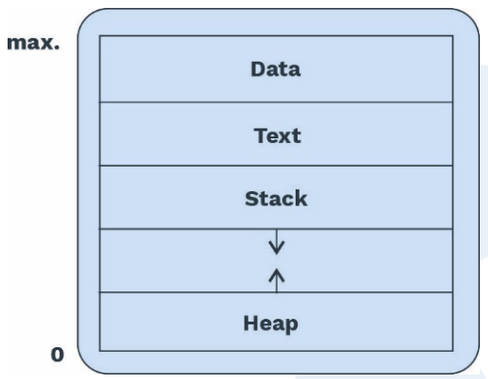


Fig. 2.1 Process Image in Memory

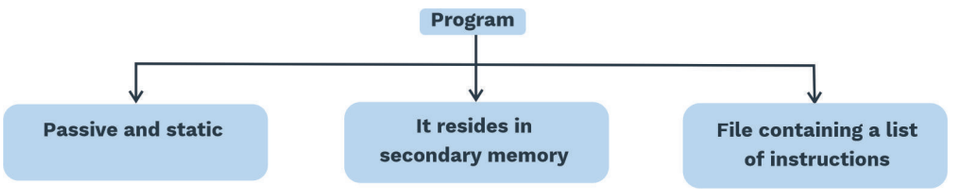


Fig. 2.2 Program

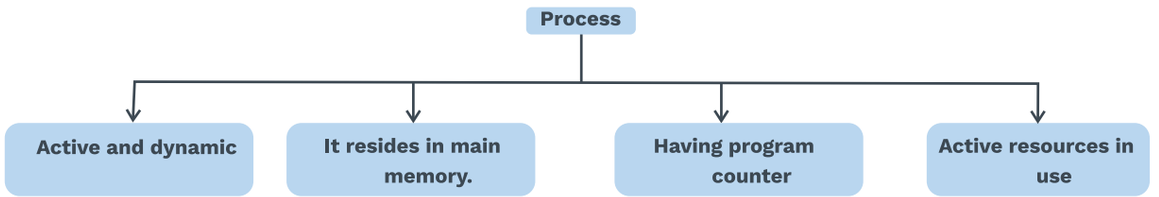


Fig. 2.3 Process

Relationship between process and program:

- **One-one:**

A single execution of a sequential program.

- **Many to one:**

Simultaneous execution of processes of a program.
The process is visualised as Abstract Data Structure (ADT).

Abstract data structure has:

- Definition
- Structure/Implementation
- Operations
- Attributes

Operations on process:

Process execution is a complex function which involves the following operations:

1) Creation of process:

A new process is constructed by the parent process or system for execution.

- When we start the computer system, the user can demand for the creation of a new process.
- This new process can be created by a process itself while executing.

2) Scheduling:

In this event, the state of a process is changed from ready to running or from new to ready or from ready to blocked, etc.

3) Blocking:

Whenever an input/output system call gets invoked by a process, the operating system blocks the process. In block mode, the process completes I/O operations or waits for an event.

4) Preemption:

A process executing in its allotted time is preempted by other processes waiting for execution.

5) Termination:

The event of completing the process is termed process termination.
In other words, it is the deallocation of computer resources allocated to it.
Some scenarios where process termination happens:

- When the execution of the process is completed, indicating that it has finished.
- When there are service errors present in the process, the operating system will terminate the process.
- Any problem in hardware will also lead to the termination of the process.
- Sometimes a process may get terminated by some other process.

Process attributes:
Identification related: Process ID, group ID, parent process ID, etc.
CPU related: Program counter, general-purpose registers, priority, states, etc.
Memory related: Size, memory limit, etc.
File related: List of open files, etc.
Device related: List of connected devices.
Protection: Access rights.

Note:

- All attributes of the process are kept in PCB (Process Control Block).
- Every process has its own PCB.
- PCB is used to represent a process in the Operating System.
- PCB is a kernel data structure.

Process control block (PCB):

- It is also called as task control block.
- It is a representation of the process in the Operating System.

Structure of PCB:

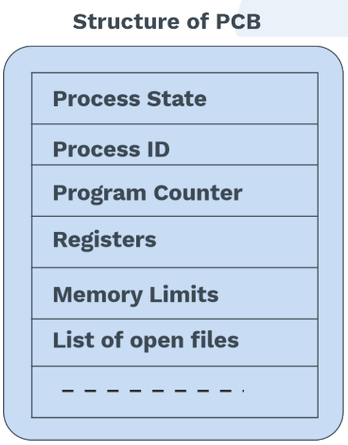


Fig. 2.4 Process Control Block

Process state:
The state of a process can be new, ready, running, waiting, halted, etc.

Program counter:
It holds the address of the next instruction to be executed.

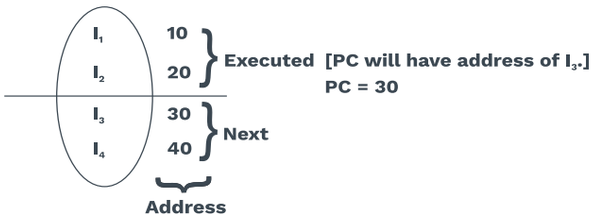


Fig. 2.5 Instruction Sequence of a Process

CPU registers:
They depend on computer architecture, i.e. accumulator, index register, stack pointers, etc.

Process ID:
It is a unique ID which is assigned by the Operating System at the time of process creation.

Memory management information:
It includes values of base and limit registers, page tables, etc.
PCB of the processes will be stored in the main memory. PCBs are implemented in the memory using a **doubly linked list.**

Rack Your Brain

Can the ready queue be implemented with a data structure other than a linked list?
If yes, why do we use a linked list popularly?

Context switch:

Definition:

Switching the CPU to another process requires performing a state save of the current process and restoring the other process

- Interrupts cause the OS to change the CPU from one task to run a kernel routine.
- Such operations frequently happen in the general-purpose system, and when that happens, the system needs to save the current process state so that it can restore later when continued.
- The context here is the PCB of the process. It contains CPU register values and other information.

- Context switching is purely overhead because the system does not perform any useful work while switching.
- Its speed varies from system to system, memory speed and the number of registers being copied.

Process state models:

State of the process changes during its execution. Process state tells the current activity of that process, i.e. whether the process is running on CPU or using I/O device, etc. A process can be in one of the following states:

New: Creation of a new process.

Ready: Waiting to get scheduled by the processor.

Running: Executed by the processor.

Waiting: Waiting for some event to occur (such as an I/O completion or reception of a signal).

Terminated: Execution completed.

Suspend ready: A process in the ready state, which is moved to secondary memory from the main memory due to lack of the resources (RAM).

Suspend wait: Instead of removing the process from the ready queue, it is better to remove the blocked process, which is waiting for some resources in the main memory. Since it is already waiting for some resources to get free, it is better to wait in the disk and give space to the high priority process.

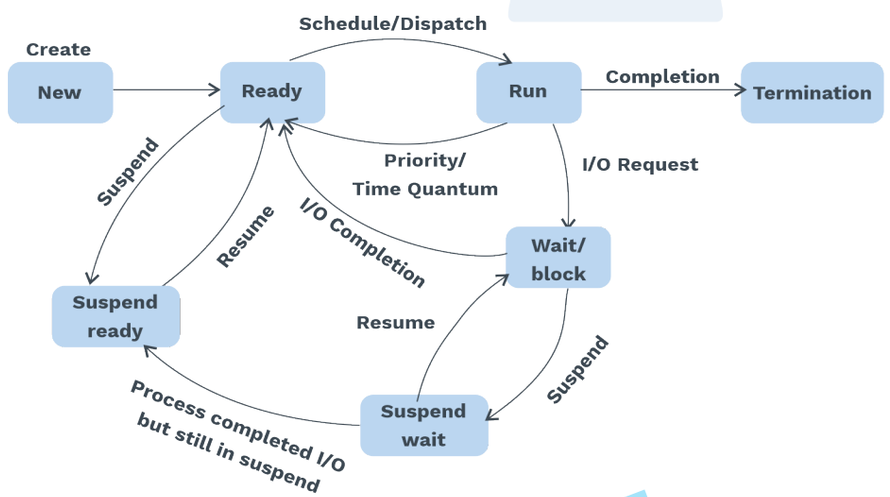


Fig. 2.6 Process State Model

SOLVED EXAMPLES

Q1 **A running process gets into blocked state if it _____**
Which of the following is/are TRUE?
a) needs data stored on a hard disk.
b) needs a resource which is held by other process.
c) is waiting for an event to occur such as scrolling the mouse.
d) completes its execution.

Sol: **Options: a), b), c)**

A running process gets into blocked state:
- if it needs to read/write data from secondary storage.
- if it needs a resource which is currently held by other process.
- if it is waiting for an event to happen either by user or other processes.
- completion of a process leads to its termination.

Important commands:
Fork () system call:
A system call that creates a new process is known as a child process. It is identical to the calling one, i.e. parent process.

- Makes a copy of text, data, stack and heap.
- Starts executing on that new copy.
- When the child process is created using fork(), a new memory location with a copy of all the parent's data will be allocated to the child process.
- Both parent and child processes will have the same virtual address but different physical address.
- The next instruction after the fork() will be executed by both child and parent processes.
- The program counter, CPU registers and open files, which are present in the parent process, are also used by the child process.
- It does not take any parameter and returns an integer value containing the newly created child process ID.
- If fork() returns:

Negative value: Child process creation unsuccessful.
Zero: Returned to the newly created child process.
Positive value: Returned to parent or caller.

Whether the parent process will execute first or the child process will be decided by the operating system.

Fork example:

CODE:

```
# include <stdio.h>
  int main ()
  {
    fork ();
    printf("Hello
world\n");
    return 0;
  }
```

O/P

```
Helloworld
Helloworld
```

Inefficient to physically copy memory from parent to child.

- Code (text section) remains identical after fork ().
- Even portions of the data section, heap and stack may remain identical after fork ().

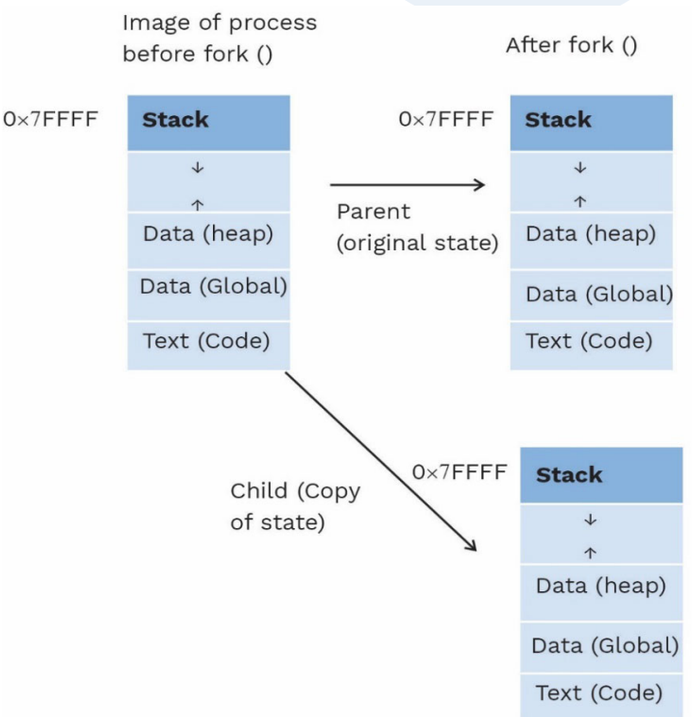


Fig. 2.7 Before and After Fork() Process Image in Physical Memory

Copy-on-write:

- OS memory management policy lazily copy pages only when they are modified.
- Initially map same physical page to child virtual memory space (but in read mode).
- Write to child virtual page triggers page protection violation (exception).
- OS handles exception by making physical copy of page and remapping child virtual page to that page.

SOLVED EXAMPLES

Q2 **Calculate number of times hello is printed?**

CODE:

```
int main ()
{
fork();
fork();
fork();
printf("Hello");
return 0;
 }
```

Sol: Range: 8 to 8

The total number of times 'hello' is printed is equal to the number of processes created and the total number of the processes when forks are in series = 2^n, where 'n' is the number of fork system calls, so $2^3=8$. Execution of parent and child process happens in which order is not decided by application, OS decides which process will get control first, parent or child.

Note:

- Parent and child processes run in parallel by running on different processors on a multi-processor system, and if it is a uniprocessor system, then by context switching.
- When n forks are in series, number of child processes created is 2^n-1, and the total number of processes is 2^n.

Previous Years' Question

Consider the following code fragment.

```
if (fork()==0)
{
    a=a+5;
    printf("%d,%d\n",a,&a);
}
else
{
    a=a-5;
    printf ("%d,%d\n",a,&a);
}
```

Let u,v be the values printed by the parent process and x,y be the values printed by the child process. Which one of the following is true?

a) u=x+10 and v=y **b)** u=x+10 and v!=y

c) u+10= x and v=y **d)** u+10= x and v!=y **(GATE CS - 2005)**

2.2 SCHEDULING

Scheduling queues and state queuing diagram:

OS maintains two types of queue in memory

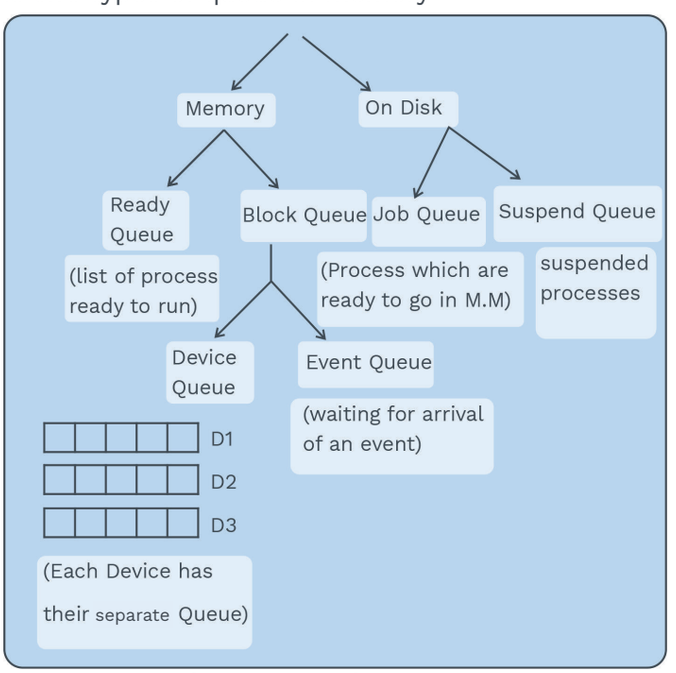


Fig. 2.8 Types of Queues

- **Job queue:** When a process is created, it is placed into job queue, which is maintained in secondary memory.
- **Ready queue:** When the process is brought in the main memory, it is kept inside the ready queue.
- Ready queue uses linked list as the data structure, and the header of the ready queue contains the pointers to the starting and ending PCBs of the processes in the list.
- **I/O queue:** In order to complete its execution, a process may require to do any other event, such as I/O operation. Till the I/O operation gets completed, the state of the process changes from running to block state. In the block state, there are multiple processes present, and the context of the PCBs of all the processes are stored in a I/O Queue.

Queuing diagram of process scheduling:

Queuing diagram is the common representation used for process scheduling discussion.

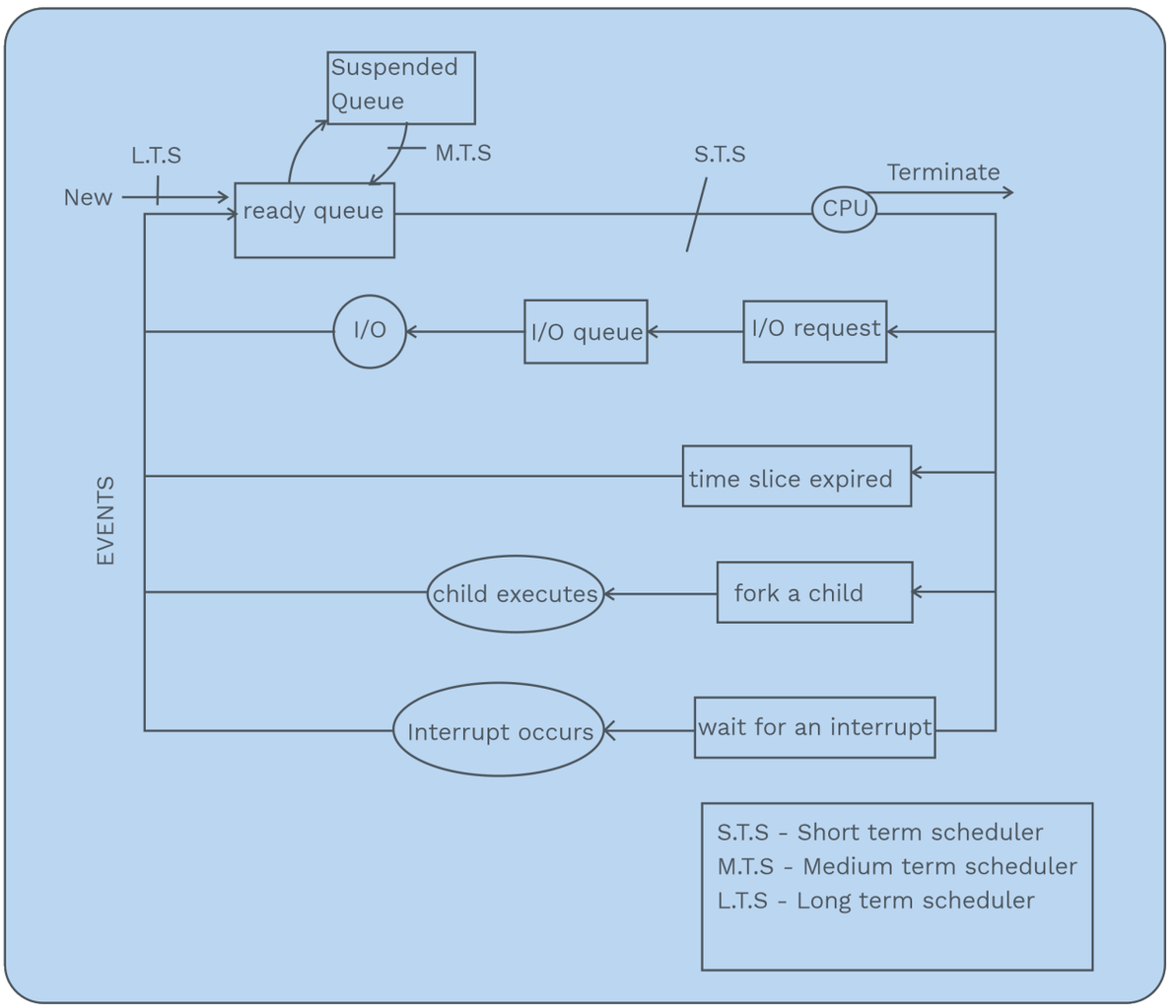
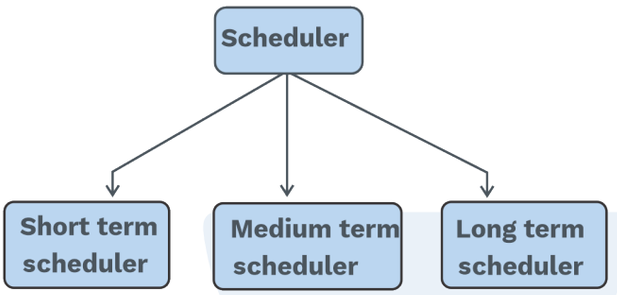


Fig. 2.9 Queuing Diagram of Process Scheduling

- When a process is created, it is brought into the main memory by Long-Term Scheduler (LTS) and placed in the ready queue, where it waits to get scheduled.
- The ready queue contains many processes; Short-Term Scheduler (STS) selects one process and schedule it on the CPU.
- The process can be removed from the CPU in the following ways:
- Process may require I/O operations, so it is placed in I/O queue.
- Process can create a new subprocess by calling fork() system call and then waits for the subprocess completion.
- If an interrupt occurs, the process can be preempted forcefully from the CPU.

- In preemptive scheduling, the time slice of the process may expire.
- In all the above situations, the process will switch from the waiting state to the ready state, where it is again placed onto the ready queue.
- This cycle is continued until the process gets terminated.

Types of scheduler:



1) STS (short-term scheduler)

- It selects one process from among the processes in the ready queue for allocation of CPU.
- Its frequency of execution is more compared to the other schedulers.
- It is invoked each time the CPU requires a new process for execution.

2) MTS (medium-term scheduler)

- It is used to decrease the number of processes in the main memory, so as to complete the remaining processes faster by reducing the CPU contention, giving better performance times.
- It is used to decrease the load on CPU.
- It is used in swapping out the processes from the main memory to the secondary memory.

3) LTS (long-term scheduler)

- It is responsible for choosing processes from the job queue to put in the ready queue.
- It controls degree of multiprogramming.
- It should choose a good mix of CPU-bound and IO-bound processes to keep its system's throughput high.

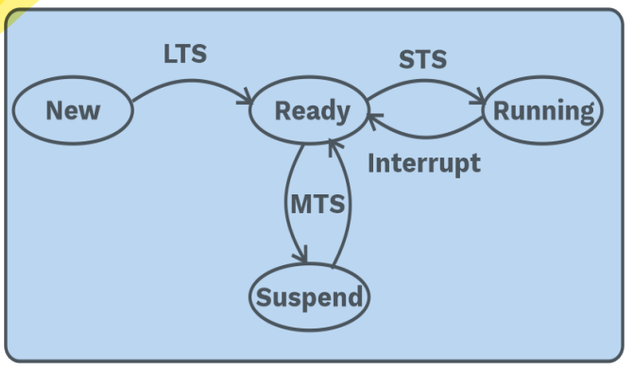


Fig. 2.10 Types of Scheduler

Dispatcher:

- It is a module/component of the CPU scheduling function.
- Short-term scheduler is responsible for selecting a job from the ready queue. The selected job is loaded on the CPU by the dispatcher.
- It switches the context of the already running process with the selected process, which is to be loaded on the CPU.

Dispatch latency:

Definition:

It is the time taken by the dispatcher to preempt the already running process and load the selected process on the CPU.

2.3 CPU SCHEDULING

Introduction:

- Single processor system can execute one process at a time; other processes in the ready queue must wait until the processor gets freed from the current running process.
- In a uni-programming system, the CPU has to remain idle till the current process completes its I/O.
- In a multiprogramming system, environment CPU cannot be left idle because many processes are in a ready queue waiting to get the CPU when the current executing Process waits for I/O.
- Scheduling is done before hand to orchestrate all the above tasks 'on' all computer resources. CPU is one of the primary computer resources; hence CPU scheduling is required.

CPU-I/O burst cycle:
Execution of process includes CPU cycle and I/O cycle, and these cycles repeat until the process gets completed.

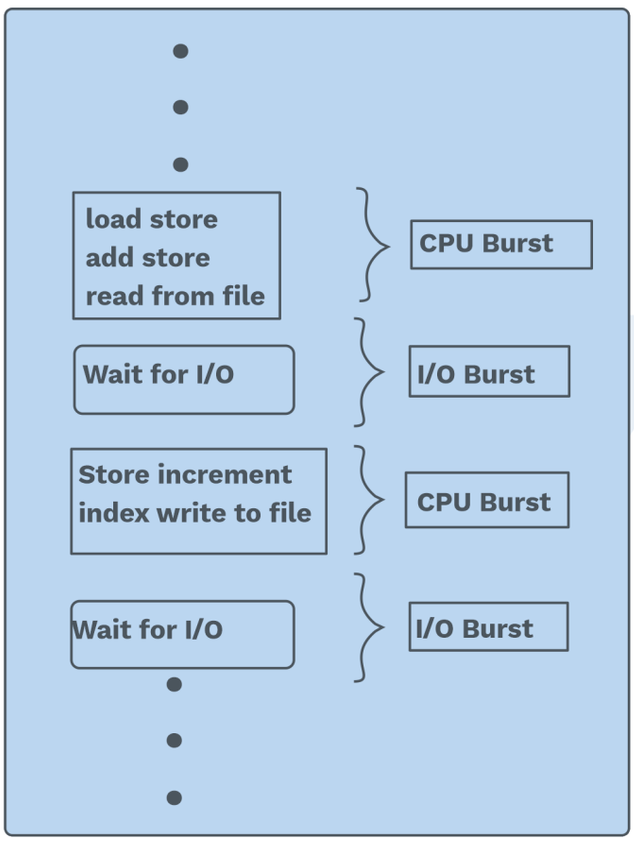


Fig. 2.11 Alternating Sequence of CPU and I/O

CPU-bound processes have:
- Long CPU bursts
- Short I/O bursts

I/O-bound processes have:
- Short CPU bursts
- Long I/O bursts

Goals of CPU scheduling:
CPU scheduling is required to choose one among the many processes present inside the ready queue for execution.
- **Fairness:** CPU must be allocated to every process fairly.

- **Efficiency:** Ideally, it should be 100% for CPU as we want to keep CPU busy.
- **Response time:** Response time for the interactive users must be reduced.
- **Throughput:** Maximum number of jobs must be executed per unit of time.
- **Waiting time:** Waiting time of the processes in the ready queue must be reduced.
- **Turnaround time:** Time between the termination of the process and the submission of the process must be reduced.

Preemptive and non-preemptive algorithms:

Preemptive scheduling: In preemptive scheduling, a process scheduled onto the CPU can be replaced by some other process based on priority, time, etc.

E.g.: Shortest Remaining Time First (SRTF), Round Robin Scheduling Algorithm (RR), etc.

Non-preemptive scheduling: In non-preemtive scheduling, once a process is scheduled onto the CPU, it cannot be replaced by any other process unless it completes its execution or is waiting for some event to occur.

E.g.: (First Come First Serve (FCFS)), Shortest Job First (SJF), etc.

Preemptive Scheduling	Non-preemptive Scheduling
1) A processor can be preempted to execute the different processes in the middle of any current process execution.	**1)** Once the processor starts the execution of a process, it must finish it before executing the other. It cannot be paused in the middle.
2) CPU utilisation is more efficient compared to non-preemptive.	**2)** CPU utilisation is less effie i e nt compared to preemptive.
3) Waiting and response time of preemptive scheduling is less.	**3)** Waiting and response time of the nonpreemptive is more.
4) 1t is flexible	**4)** It is rigid.
5) **E.g.:** Shortest remaining time first, Round Robin, etc.	**5)** **E.g.:** FCFS, SJF, highest response ratio next, etc.

Concept of starvation:

In a preemptive environment, low-priority processes have to wait for high-priority processes to complete their execution.

Sometimes these low-priority processes wait for an indefinite amount of time which leads to the problem of starvation.
One solution for starvation is aging, i.e. temporarily increasing the priority of long-waiting processes so that they can execute after a short wait time and do not go in an indefinite wait.

Scheduling criteria:

Goal of CPU Scheduling

1) To maximise the throughput and CPU utilisation

2) To minimise the average turnaround time (TAT), average waiting time and average response time of processes

Where to apply: Ready state

Who will apply: Short-term scheduler

When to apply:

Running → Termination

Running → Wait

Running → Ready

Wait → Ready

- Choosing a particular CPU scheduling algorithm among different scheduling algorithms may favour a particular group of processes over others.
- Many CPU scheduling algorithms have been considered, and characteristics used for comparison can make a considerable difference in which algorithms are found to be the best.

These characteristics are:

1) CPU utilisation:
- CPU has to be kept busy for a maximum amount of time.
- In real, the utilisation of CPU may lie between 40% to 90%.

2) Throughput:
- Number of processes that finished their execution in per unit time.
- Completion rate depends on the size of the processes. If the process size is long, then the completion rate can be a few processes per unit of time; else, for short processes, more processes can be completed per unit of time.

3) Turnaround time:

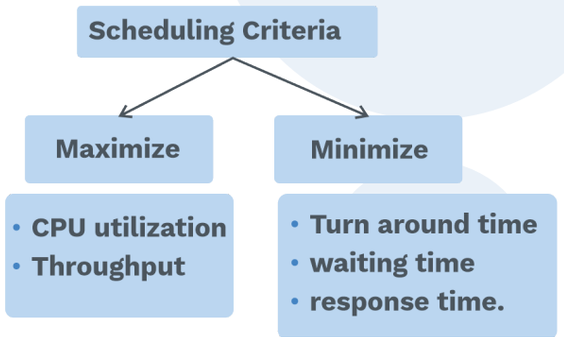
- It is the total time for which the process remains in the system, from the time it is submitted to the time it is completed.
- Turnaround time = Completion time of the process - Arrival time of the process

4) Waiting time:

- CPU scheduling does not affect the process burst time but affects how much time a process spends waiting in the ready queue.
- It is the sum of time for which the process was ready for execution, waiting for the CPU.

5) Response time:

- Time between the first response from the CPU and the first submission to the ready queue.



Let a process p_i from 'n' processes(P_1,P_2,P_3,.....,P_n)

- T.A.T = (C.T)–(A.T)
- WT = TAT– (BT + IO)
- WT = (CT – AT) – (BT + IO)

Abbreviation
W.T – Waiting time
B.T – Burst Time
RQ – Ready queue
A.T – Arrival time
C.T. – Completion time
TAT – Turn around time
IO – Input/outputtime

Let the following notation for process be Pi

1) $A.T(P_i)=A_i$

2) $B.T(P_i)=X_i$

3) $C.T(P_i)=C_i$

4) $TAT\ (P_i)=C_i-A_i$

5) $W.T(P_i)=TAT_i-X_i$

6) Average TAT $(P_i)=\frac{1}{n}\sum_{i=1}^{n}(C_i-A_i)$

7) Average WT $(P_i)=\frac{1}{n}\sum_{i=1}^{n}((C_i-A_i)-X_i)$

8) Total time for all process (schedule length) = $\max(C_i)-\min(A_i)$

9) $\text{Throughput}=\frac{n(\text{total processes})}{(\text{total time for all process})}$

10) Deadline $(P_i)=D_i$

Scheduling algorithms:

1) First come-first serve scheduling (FCFS)

Concept: Process that requested the CPU First will be allocated the CPU first

Criteria: Based on arrival time

Mode: Non-preemptive

SOLVED EXAMPLES

Q3 **Find the average turnaround time and average waiting time using FCFS algorithm?**

Process	Arrival time	Burst time
P1	0	24
P2	2	3
P3	3	3

CT | TAT | WAT
24 | 24 | 0
27 | 25 | 22
30 | 27 | 24

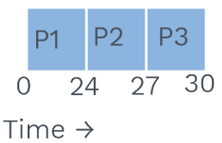
CT − AT → TAT
TAT − BT → WAT

a) **Avg. TAT = 25.3 and Avg. WT = 15.3**
b) **Avg. TAT = 24.3 and Avg. WT = 15.3**
c) **Avg. TAT = 30 and Avg. WT = 14.3**
d) **Avg. TAT = 27.3 and Avg. WT = 15.3**

Sol: **Option: a)**

Process	Arrival time	Burst time	TAT	WT
P1	0	24	24	0
P2	2	3	25	22
P3	3	3	27	24
		Avg.	25.3	15.3

Gantt chart:



$$\text{Avg TAT} = \frac{24+25+27}{3} = 25.3 \text{ units}$$

$$\text{Avg W.T} = \frac{0+22+24}{3} = 15.3 \text{ units}$$

Q4

Process ID	A.T.	B.T.	1/0	B.T
P1	0	3	2	2
P2	0	2	4	1
P3	2	1	3	2
P4	5	2	2	1

Consider the FCFS algorithm with non-zero context switch overhead of 111 unit. I/O operations can overlap any number of processes. Find the percentage of CPU

a) CPU context overhead = 33%, percentage of CPU idleness = 3%, CPU efficiency = 64%

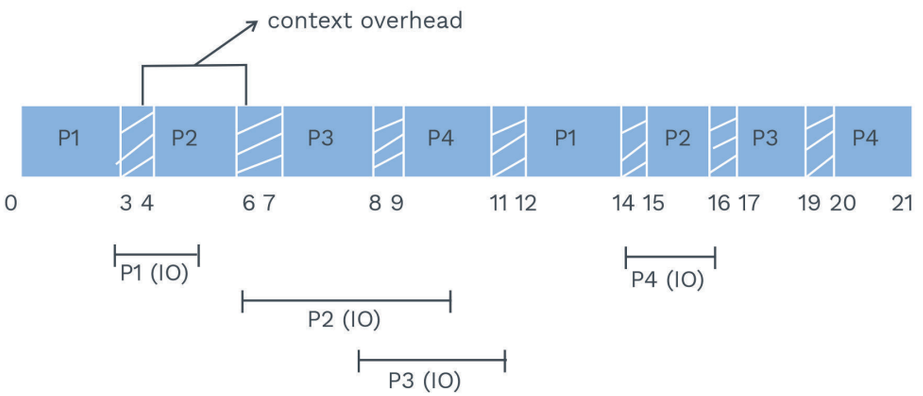
b) CPU context overhead = 30% , percentage of CPU idleness = 10%, CPU efficiency = 60%

c) CPU context overhead = 33.3%, percentage of CPU idleness = 5%, CPU efficiency = 61.67%

d) CPU context overhead = 33.3%, percentage of CPU idleness = 0%, CPU efficiency = 66.67%

Sol: **Option: c)**

Let's make Gantt chart using the given information.



So let total context switch overhead be x = 7 units
Total CPU idleness be y = 0 units
So %CPU idleness (y) = 0%

$$\%\text{CPU overhead due to context switch (x)} = \frac{7}{21} \times 100 = 33.3\%$$

So %CPU utilisation = 100 – (x + y)
= 100 – (33.3 + 0)
= 66.67%

Observations:

1) FCFS is non-preemptive.
2) Average waiting time in FCFS is generally not minimal.
3) In a dynamic situation, FCFS may experience a convoy effect, which causes low CPU utilisation.

Convoy effect:

It is a phenomenon associated with the FCFS algorithm in which the whole OS slows down due to a few slow processes.

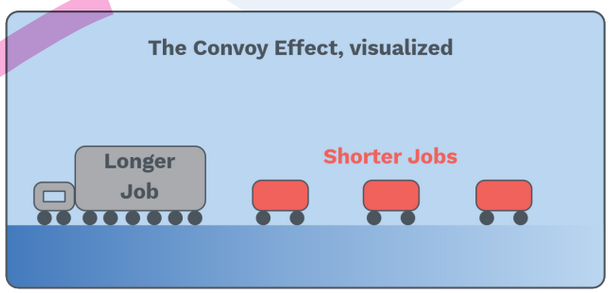


Fig. 2.13 Convoy Effect

- FCFS is non-preemptive in nature when a process is allocated CPU; other processes can get CPU only after the current process leaves CPU voluntarily.
- One long CPU extensive process takes a long time to execute while other short I/O intensive waits more for their turn, so whole system utilisation decreases.
- Preemptive scheduling algorithm like Shortest Remaining Time First can be used to avoid the convoy effect because processes having small size will not have to wait for more CPU time.

Note:

In any CPU scheduling algorithm, unless given in the question, if arrival times of the process are the same, then schedule the process which has the lowest process id.

2) Shortest job first scheduling (SJF)

Criteria: Burst time

Mode: Non-preemptive scheduling

Concept: Allocate the CPU to the process with smallest next CPU burst. If two or more processes are having the same burst time, the FCFS is used to break the tie.

SOLVED EXAMPLES

Q5 **Find the average turnaround time and average waiting time using SJF algorithm ?**

Pid	A.T	B.T
P1	0	4
P2	1	3
P3	2	5
P4	3	2
P5	4	1
P6	5	6

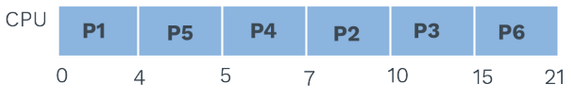
a) Avg. TAT = 7.5 and Avg. WT = 4.1
b) Avg. TAT = 7 and Avg. WT = 4.3
c) Avg. TAT = 7.8 and Avg. WT = 5.3
d) Avg. TAT = 7.8 and Avg. WT = 4.3

Sol: **Option: d)**

Pid	A.T	B.T	T.A.T	W.T
P1	0	4	4	0
P2	1	3	9	6
P3	2	5	13	8
P4	3	2	4	2
P5	4	1	1	0
P6	5	6	16	10

Ready Queue: P1 P2 P3 P4 P5 P6

Gantt chart



$$\text{Avg. TAT}\left(\text{completion time-arrival time}\right) = \frac{\sum \text{TAT}(P_i)}{n}$$
$$= \frac{4+9+13+4+1+16}{6}$$
$$= 7.8 \text{ units}$$

$$\text{Avg. waiting time}\left(\text{TAT-B.T}\right) = \frac{\sum \text{Wt}(P_i)}{n}$$
$$= \frac{0+6+8+2+0+10}{3}$$
$$= 4.3 \text{ units}$$

Q6 **Consider the following table :**

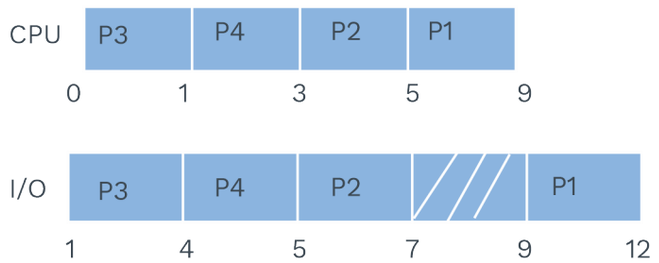
Process	CPU Burst time	I/O service time
P_1	4	3
P_2	2	2
P_3	1	3
P_4	2	1

Assume that all processes are arrived at time 0. The first process starts executed from '0' unit time. Every process completes its CPU request then gets I/O service. If any two process need same amount of CPU service time then prefer the process which has less I/O service request. Find the time at which process P_4 completes both CPU and I/O requests. Processes are scheduled using SJF for CPU services and I/O scheduling is done using FCFS scheduling.

Sol: **Range: 5 - 5**

Let us make Gantt chart

Gantt chart:



So P_4 completed its CPU cycle at '3' unit time and waited '1' time units for I/0. As P_3 is doing I/0 and then executes I/0 for 1 unit time and completes at '5' unit time.

3) Shortest remaining time first (SRTF):

Concept: It is the same as the shortest job first. Scheduling but with preemptive mode.

Criteria: Based on burst time.

Mode: Preemption.

Q7 **Find the average turnaround time and average waiting time using SRTF algorithm?**

Process	Arrival time	Burst time
P1	0	6
P2	1	3
P3	2	1
P4	3	4

Which of the following is correct?

a) Avg. TAT = 6 and Avg. WT = 2.5

b) Avg. TAT = 6.25 and Avg. WT = 2.75

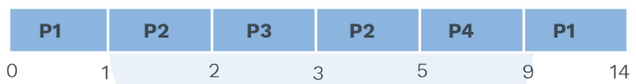
c) Avg. TAT = 7 and Avg. WT = 2.5

d) Avg. TAT = 6.3 and Avg. WT = 2.8

Sol: **Option: b)**

Process	Arrival time	Burst time	TAT	WT
P1	0	6	14	8
P2	1	3	4	1
P3	2	1	1	0
P4	3	4	6	2

Gantt chart:



Average waiting time = $WT_{AVG} = \frac{8+1+0+2}{4} = \frac{11}{4} = 2.75$

Avg. turnaround time = $TAT_{(AVG)} = \frac{14+4+1+6}{4} = 6.25$

Note:

- P SRTF gives overall less waiting time compared to other scheduling algorithms.
- It can be implemented with predicted burst time.

Prediction technique for burst time of processes:

For CPU scheduling algorithms like SJF or SRTF, which is based on burst times of all processes, it is not possible to get burst time data of processes at the start of the execution, but we can predict burst time based on previous data.

Few prediction techniques are:

1) Size of process:

Let P_i be a process, t_i is the actual burst time and τ_i is the predicted burst time of process.

Previous Data	Current Prediction
P_{old} = 198 KB	P_{new} = 200 KB
t_{old} = 20 ms	T_{new} = 20 ms

So burst time is predicted based on the size of the process.

2) Process types:

Burst times are predicted based on the type of processes going to be scheduled.

Various types of burst time:

a) OS processes – $[4–6]_{ms}$

b) User interactive process – $[10–12]_{ms}$

c) Foreground process – $[20–25]_{ms}$

d) Background process – $[50–60]_{ms}$

3) Averaging previous CPU bursts:

Next CPU burst of the process is the average of its previous CPU burst. It is dynamic in nature. If t_i is the actual burst time of process P_i and τ_i is the predicted burst time of process P_i

$$\tau_{n+1} = \frac{1}{n}\sum_{i=1}^{n} t_i$$

4) Exponential averaging:

$$\tau_{n+1} = \alpha t_n + (1-\alpha)\tau_n$$

SOLVED EXAMPLES

Q8 Calculate the exponential averaging with τ_1 = 10, a = 0.5, and the algorithm is SRTF with previous runs as 8, 7, 4, 16.

Sol: Range: 7.5 to 7.5

τ_1=10, α=0.5. putting in formula

[∴ As SRTF is applied, so process will be served 4, 7, 8, 16]

$\tau_{n+1} = \alpha t_n + (1-\alpha)\tau_n$

$\Rightarrow \tau_2$=0.5*4+0.5*10 $[t_1=4 \ \& \ \tau_1=10]$

$\Rightarrow \tau_2$=7

$\Rightarrow \tau_3$=0.5 * 7+0.5 * 7 $[t_2=7 \ \& \ \tau_2.7]$ $\Rightarrow \tau_4$=0.5*8+0.5*7 $[t_3=8, \tau_3=7]$

$\Rightarrow \tau_3 = 7$ $\Rightarrow \tau_4 = 7.5$

Previous Years' Question

Consider three processes, all arriving at time 0 with total execution time of 10, 20 and 30 units, respectively. Each process spends the first 20% of execution time doing 1/0, the next 70% of time doing computation and the last 10% of time doing 1/0 again. The operating system uses the shortest remaining time first scheduling algorithm and schedules a new process, either when the running process gets blocked on 1/0 or when the running process finishes its compute burst, assuming that all i/O operations can be overlapped as much as possible. For what time of percentage does the CPU remain IDLE?

a) 0% **b)** 10.6%

c) 30.0% **d)** 89.4%

Sol: b) **(GATE CS-2006)**

4) Longest remaining time first (LRTF):

Definition:

LRTF, which stands for Longest Remaining Time First, is a scheduling algorithm used by the operating system to schedule the incoming processes so that they can be executed in a systematic way. This algorithm schedules those processes first which have the longest processing time remaining for completion. This algorithm can also be called as the preemptive version of the LJF scheduling algorithm.

Criteria: Burst time (largest first)
Mode: Preemptive

SOLVED EXAMPLES

Q9 **Find the average turnaround time and average waiting time using LRTF algorithm?**

PNo	A.T	B.T
P1	0	2
P2	0	4
P3	0	8

Which of the following is correct?

a) Avg. TAT = 13.25 and Avg. WT = 8.3
b) Avg. TAT = 12.7 and Avg. WT = 8.2
c) Avg. TAT = 13 and Avg. WT = 8.3
d) Avg. TAT = 13 and Avg. WT = 8

Sol: Option: **c)**

PNo	A.T	B.T	TAT	WT
P1	0	2	12	10
P2	0	4	13	9
P3	0	8	14	6

Gantt chart:

CPU	P3	P2	P3	P2	P3	P1	P2	P3	P1	P2	P3
0	4	5	6	7	8	9	10	11	12	13	14

$$\text{Avg TAT} = \frac{13+12+14}{3} = \frac{39}{3} = 13$$

$$\text{Avg W.T} = \frac{10+9+6}{3} = \frac{25}{3} = 8.3$$

5) Round-robin scheduling (RR):

Definition:

Round-robin (RR) is one of the algorithms employed by operating systems and network schedulers in computing. As the term is generally used, time slices (also known as time quanta) are assigned to each process in equal portions and in circular order, handling all processes without priority (also known as cyclic executive). Round-robin scheduling is simple, easy to implement and starvation-free.

Criteria: Order of arrival
Mode: Preemptive
Concept:

1) It is FCFS (preemptive) based on the time out interval.
2) Ready queue is maintained for scheduling.
3) Small time quantum leads to more context switches (overhead).
4) Large T.Q makes RR act as FCFS.

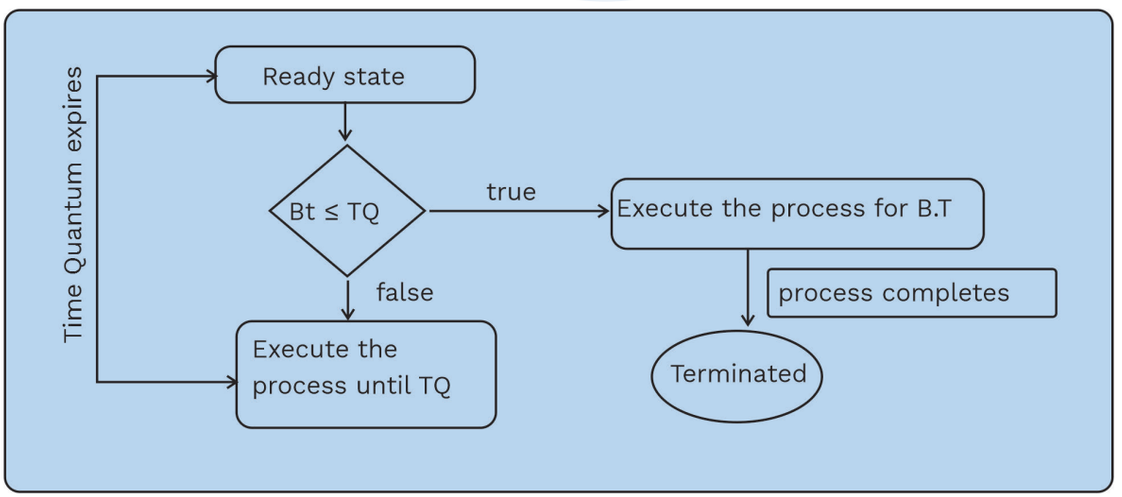


Fig. 2.14 RR Scheduling Flow Chart

SOLVED EXAMPLES

Q10 **What is the average turnaround time and average waiting time using round robin scheduling algorithm, when time quantum is 5 units?**

Process	AT	B.T
P1	0	5
P2	1	7
P3	3	3
P4	4	6

Which of the following is correct?

a) Avg. TAT = 12.75 and Avg. WT = 7.5
b) Avg. TAT = 12 and Avg. WT = 7
c) Avg. TAT = 12.75 and Avg. WT = 7
d) Avg. TAT = 12 and Avg. WT = 7.5

Sol: **Option: a)**

Process	AT	B.T	TAT	W.T
P1	0	5	5	0
P2	1	7	19	12
P3	3	3	10	7
P4	4	6	17	11

Time Quantum (TQ) = 5 unit

Gantt chart:

CPU	P1	P2	P3	P4	P2	P4
0	5	10	13	18	20	21

$$\text{Avg TAT} = \frac{5+19+10+17}{4} = \frac{51}{4} = 12.75$$

$$\text{Avg WT} = \frac{0+12+7+11}{4} = \frac{30}{4} = 7.5$$

Q11 **A system implementing Round Robin Process Scheduler, with a context switch delay of 3 time units. Which of the following is/are the values of the time quantum, for which the context switch overhead stays below 50%? (Calculations should be rounded off upto 2 decimal points)**

a) 7 time units
b) 3 time units
c) 4 time units
d) 6 time units

Sol: **Options: a), c), d)**

Let context switch delay be expressed as TD = 3 time units and time quantum be expressed as TQ.

Context switching overhead is given as, TO = TD / (TQ + TD)

- A) TQ = 7 time units. TO = 3 / (7 + 3) = 0.3 < 0.5
- B) TQ = 3 time units. TO = 3 / (3 + 3) = 0.5 == 0.5
- C) TQ = 4 time units. TO = 3 / (4 + 3) = 0.43 < 0.5
- D) TQ = 6 time units. TO = 3 / (6 + 3) = 0.33 < 0.5

Previous Years' Question

Consider a process sharing the CPU in a round-robin fashion. Assuming that each process switch takes's' seconds, what must be the quantum size 'q' such that he overhead resulting from process switching is minimised, but at the same time each process is guaranteed to get its turn at the CPU at least every 't' seconds?

a) $q \leq \frac{t\text{-}ns}{n\text{-}1}$ **b)** $q \geq \frac{t\text{-}ns}{n\text{-}1}$

c) $q \leq \frac{t\text{-}ns}{n+1}$ **d)** $q \geq \frac{t\text{-}ns}{n+1}$

Sol: a) **(GATE CS-2066)**

Advantages	Disadvantages
1) This works best for shorter jobs **2)** Starvation is not possible	**1)** It works poor if all jobs have the same length **2)** Very large quantum will make it behave s FCFS

Rack Your Brain

How do you say whether a time quantum in Round Robin Scheduling Algorithm is fairfor all processes or not?

6) Highest response ratio next (HRRN):

Criteria: Response ratio.

Mode: Non-preemptive.

$$\text{Response Ratio } (RR) = \frac{\text{Turn around time}}{\text{Burst time}}$$

$$= \frac{\text{Waiting time} + \text{Burst time}}{\text{Burst time}}$$

Concept:

1) HRRN selects a process which has the highest response ratio to schedule as next process into CPU.
2) It minimises the average turnaround time.
3) It favours both longer and shorter processes.
4) Starvation is not possible.
5) Non-preemptive in nature.

SOLVED EXAMPLES

Q12 **HRRN scheduling is used; compute the average turnaround time and average waiting time?**

Process	A.T.	B.T
P1	0	4
P2	2	1
P3	5	2
P4	4	2

Which of the following is correct?

a) Avg. TAT = 3.5 and Avg. WT = 1.5
b) Avg. TAT = 3.5 and Avg. WT = 1.25
c) Avg. TAT = 3 and Avg. WT = 1.25
d) Avg. TAT = 3.25 and Avg. WT = 1.5

Sol: **Option: b)**

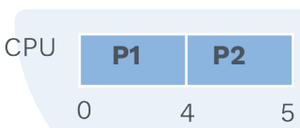
Process	A.T	B.T	W.T	TAT
P1	0	4	0	0
P2	2	1	2	3
P3	5	2	2	4
P4	4	2	1	3

Ans: At t = 0, only P1 is in the ready queue

At t = 4, P2 and P4 are in the ready queue.
So to assign CPU, we calculate their response ratio.

$$RR(\text{process } 2) = \frac{WT_2 + BT_2}{BT_2}$$
$$= \frac{2+1}{1} = 3$$
$$RR(\text{process } 4) = \frac{WT_4 + BT_4}{BT_4}$$
$$= \frac{0+2}{2} = 1$$

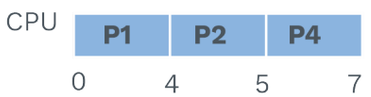
P2 has a large response ratio.



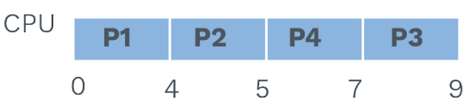
Then again at t=5, $RR(\text{process } 4) = \frac{1+2}{2} = 1.5$

$RR(\text{process } 3) = \frac{0+5}{5} = 1$

P4 has the highest response ratio.



Now only P3 is left, so we directly push it into the CPU after P4.



Avg. waiting time $= \frac{0+2+2+1}{4} = \frac{5}{4} = 1.25$

Avg. Turnaround time $= \frac{4+3+4+3}{4} = 3.5$

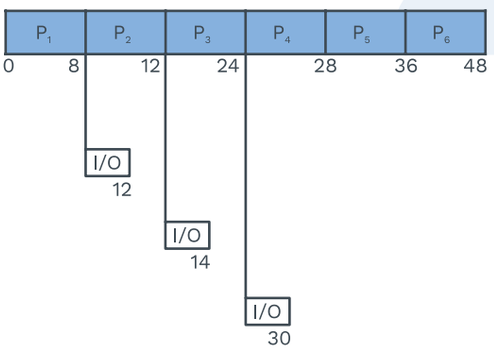
Q13 **Three processes with their execution time are given below:**

Process	Service Time		
	CPU	I/O	CPU
P_1	8	4	8
P_2	4	2	4
P_3	12	6	12

All times in milliseconds, and all processes arrive at time 0. Processes start executing on CPU first, then go for 1/0 and at last finish execution with CPU. 1/0 computations can be done parallelly and no overhead for context switch. What is the average waiting time (in milliseconds) of these processes when scheduled using highest response ratio next (HRRN) algorithm?

Sol: **Range: 10.66 - 10.67**

At time 0 response ratio for all the processes are 1. So process with the lowest number gets scheduled.



$$\text{Response Relation}\left(P_i\right) = \frac{WT\left(P_i\right) + ST\left(P_i\right)}{ST\left(P_i\right)}$$

Response Ratio at time 8:	Response Ratio at time 12	Response Ratio at time 24
$P_2 = \frac{8+10}{10} = 1.8$	$P_1 = \frac{0+12}{12} = 1$	$P_1 = \frac{12+8}{12} = 1.67$
$P_3 = \frac{8+30}{30} = 1.26$	$P_3 = \frac{12+30}{30} = 1.4$	$P_2 = \frac{10+6}{6} = 2.6$
P_2 is scheduled at time 8	P_3 is scheduled at time 12	P_2 is scheduled at 24

WT(P1) = 16, WT(P2) = 10, WT(P3) = 6

$$\text{Average waiting time} = \frac{16 + 10 + 6}{3} = \frac{32}{3} = 10.66$$

7. Priority-based scheduling algorithm:

Concept: 1) Based on the priority of the processes, they are kept in the ready queue.
2) CPU allocated to highest priority first, and ties are broken by arrival time.

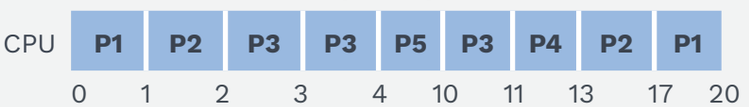
Criteria: Priority assigned

Mode: Non-preemptive/preemptive

Grey Matter Alert!

	Priority	Pid	A.T	B.T
(low)	4	P1	0	4
	6	P2	1	5
	8	P3	2	3
	7	P4	3	2
(high)	9	P5	4	6

(Non-preemptive)



(Preemptive)

Disadvantages:

1) Once priority is assigned to the process, it will not change during the execution of the process.

2) Low-priority process may get infinite blocking/starvation.

3) To prevent starvation problem, "Aging" mechanism is used.

4) Aging is the technique in which the priority of the process increases with the waiting time, i.e. if a process is waiting for the CPU for a longer duration of time, then its priority increases gradually.

Previous Years' Question

The arrival time, priority and duration of the CPU and 1/0 bursts for each of the three processes P_1, P_2, and P, are given in the table below. Each process has a CPU burst followed by an 1/0 burst followed by another CPU burst. Assume that each process has its own 1/0 resource.

Priority	Arrival time	Priority	Burst Duration (CPU)	Burst Duration (I/0)	Burst Duration (CPU)
P1	0	2	1	5	3
P2	2	3 (lowest)	3	3	1
P3	3	2 (highest)	2	3	1

The multiprogrammed OS uses preemptive priority scheduling. What are the finish times of the process P_1, P_2 and P_3?

a) 11, 15, 9 **b)** 10, 15, 9

c) 11, 16, 10 **d)** 12, 17, 11

Sol: d) **(GATE CS-2006)**

8) Multilevel queue scheduling:

Concept:

a) Ready queue is divided into several separate queues for different kinds of processes.

b) Processes are allocated permanently to one queue based on some property.

c) Different scheduling algorithms are applied for different queues.

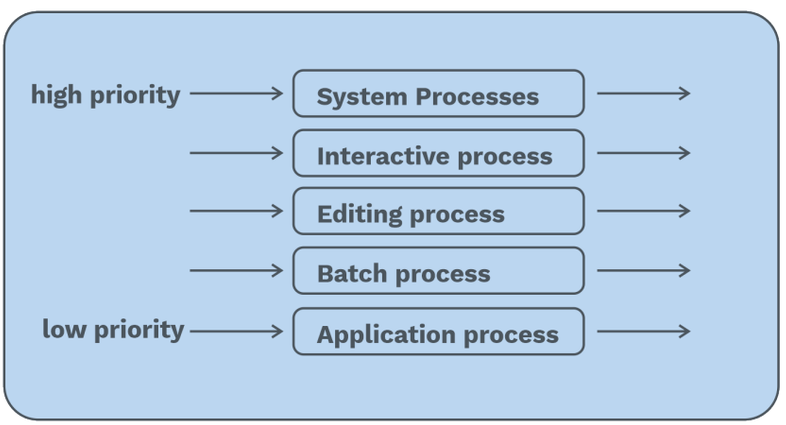


Fig. 2.15 Multilevel Queues

SOLVED EXAMPLES

Q14 **Consider a uniprocessor system which uses multilevel queue scheduling for processes. For example, given a set of four processes in the following table.**

Process	Queue No.	Arrival_Time	Burst_Time
P_1	1	0	5
P_2	1	2	3
P_3	2	4	6
P_4	1	9	4

All times are in milliseconds.

Queue No. 1 has the higher priority than Queue No. 2. Processes in Queue No. 1 et scheduled using Round Robin Scheduling with Time Quantum 2 ms and processes in Queue No. 2 are scheduled using First Come First Serve Scheduling. What is the average turnaround time (in milliseconds) of these processes?

Sol: **Range: 7.75 - 7.75**

As Queue No. 1 has the higher priority, processes in it also has a higher priority than processes in Queue No. 2.

Gantt chart:

Turnaround time (TAT) calculations:

TAT (P1) = 8–0 = 8

TAT (P2) = 7–2 = 5

TAT (P3) = 18–4 = 14

TAT (P4) = 13–9 = 4

$$\text{Average TAT} = \frac{8+5+14+4}{4} = \frac{31}{4} = 7.75\,\text{ms}$$

Note:

- Too much CPU time (low priority).
- 1/0 bound and interactive process (higher priority).
- Aging can be used to present starvation.

9) Multi-level feedback queue scheduling:

a) In this, every new process is scheduled to Queue 1 and based on the scheduling algorithm, if the process completes its execution, then it leaves the system, else it gets preempted and goes to the next queue.

b) Level i^{th} process preempted goes level $(i+1)^{th}$ ready queue.

c) MLQ and MLFQ both favour short burst time jobs.

d) Aging can be used to prevent starvation.

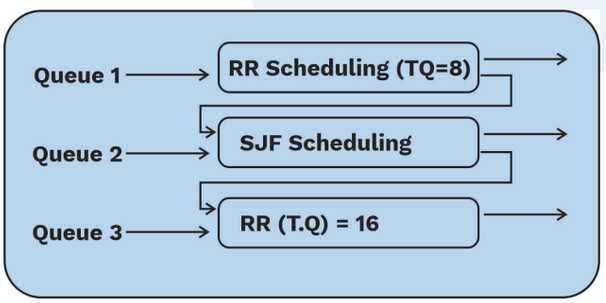


Fig. 2.16 Multi-level Feedback Queues

A MLFQ scheduler is defined by the following parameters:

i) Number of queues.

ii) Scheduling techniques required for each queue.

iii) Rules required to determine when to move a process to a higher priority queue or vice versa.

Q15 Consider a uniprocessor system that implements Multilevel Feedback Queue Scheduling algorithm for processes. There are 6 levels (1 to 6) of queue present In the system. The level 1 queue has a time quantum of 2 milliseconds and in each of the further level queues, time quantum Increases by 6 milliseconds. Currently, the system has only one CPU bound process requiring a CPU burst of 48 milliseconds.
Let 'X' be the number of times the process gets interrupted before completion and 'Y' be the queue level at which the process lastly moves before termination. What is the absolute difference between X and Y?

Sol: **Range: 1 - 1**

		Time Quantum	Process Execution	(48 ms)
L_1	Queue	2	2	
L_2	Queue	8(2+6)	8	
L_3	Queue	14(8+6)	14	
L_4	Queue	20(14+6)	20	
L_5	Queue	26(20+6)	4	Process executes 48 ms at level 5 Queue
L_6	Queue	32(26+6)	–	

→ Process gets interrupted four times at levels (1,2,3 and 4). So X = 4.
→ Process terminates at level 5 queue, so Y = 5.
→ Absolute difference between X and Y is 1.

2.4 THREADS

1) Thread is considered a lightweight process. As process, it is a unit of CPU utilisation.
2) A thread has its own program counter, a register set and a stack.
3) Code section, data section and other operating system resources are shared among all the threads of a process.
4) A heavyweight process is simply a single thread process.

Fig. 2.17 Single Thread Process

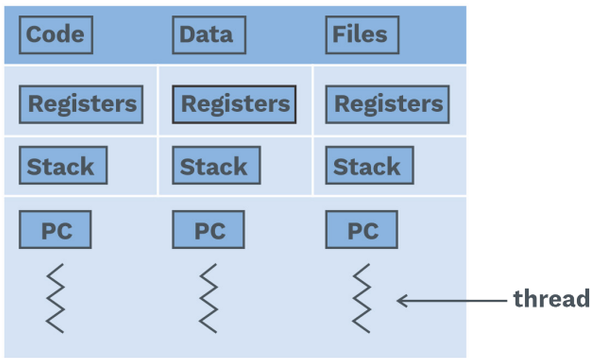


Fig. 2.18 Multithreaded Process

Modern PCs consist of many multithreaded software applications.
For example:

1) Web browser:

A web browser having multiple threads may contain one thread that shows the text or images and other threads may retrieve data from the web.

2) Word processor:

A word processor might contain one thread to show graphics, one thread can be used to respond to the keystrokes, while other threads can be used to check spellings and grammatical errors. Would they have been a single-threaded process, different services have to wait for their turn to get executed, and the waiting time would be enormous.

Benefits of multithreaded process:

1) **Responsiveness:**
   Multithreading in an interactive application that allows the process to continue its execution even if a part of it gets blocked or takes a longer time. Multithreading decreases the responsiveness of the system.

2) **Resource sharing:**
   Code section, data section and files are shared among all the threads of a process.
   A single application can have many threads present in the same address space using resource sharing.

3) **Economy:**
   It is more economical to use threads as they share the process resources.

4) **Utilisation of multiprocessor systems:**
If a process has several threads, then each thread can be scheduled on a separate processor, thus multiple processors can be effectively utilised.

Types of threads:

User level threads (ULT)

- These threads are created and implemented at the user level without the knowledge of the operating system, i.e. the operating system does not know the existence of these threads.
- Implementation of ULT is easy.
- Context switching time in ULT (user level threads) is less compared to KLT (kernel level thread).
- If any one of the user level threads performs a blocking operation, then the entire process gets blocked as OS does not see ULT but considers it as a single process.
- Context switching requires no hardware support.
- ULT favours non-blocking processes.

Kernel level threads (KLT):

- Kernel level threads are implemented by operating system and they are executed in kernel space.
- Kernel level threads are recognised by the operating system.
- Implementation of kernel thread is complicated.
- Context switch time in KLT is more.
- If one kernel thread performs blocking operation, then another thread can continue execution.
- Hardware support is needed.
- KLT favours blocking processes.

SOLVED EXAMPLES

Q16 **Find the TRUE statement(s) from the following.**

a) Threads of a user process communicate without invoking the kernel.

b) With improved software design, multithreaded applications execute faster.

c) A thread can be terminated when it completes its task or when another thread terminates it.

d) Process creation typically requires more CPU cycles than thread creation.

Sol: **Options: a), b), c), d)**

Threads communicate via their shared space.

An application can have multiple independent segments when each segment can execute simultaneously on a multiprocessor system.

Threads can be terminated on the completion of task or when another thread kills it.

Process needs more information than a thread.

Multithreading models in operating system:

Many–to–one model:

- In this model, many user level threads are mapped to a single kernel thread.
- If a single user level thread performs blocking operation, then the whole process will get blocked.
- Only one kernel thread can operate on a single processor at a time.
- Since, in this model, kernel level thread can be accessed by only one user level thread, so this model does not allow one process to be distributed among many CPUs.

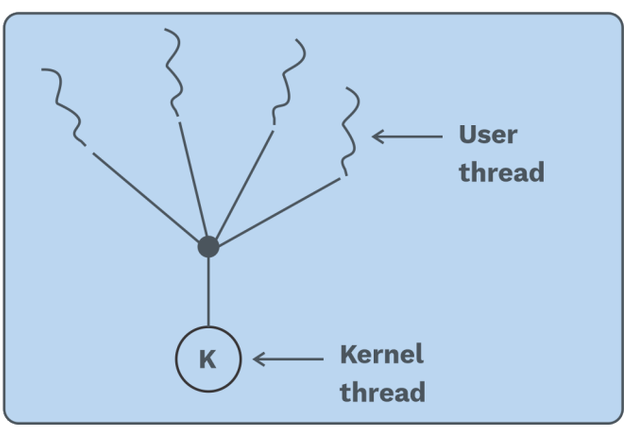


Fig. 2.19 Many to One Model

One–to–one model:

- In this model, each user thread is handled by a separate kernel thread.
- This model overcomes the problems involving blocking system calls and dividing the process across multiple CPUs.
- There is a significant overhead in managing one-to-one model.
- Most implementations of this model place a limit on how many threads can be created.

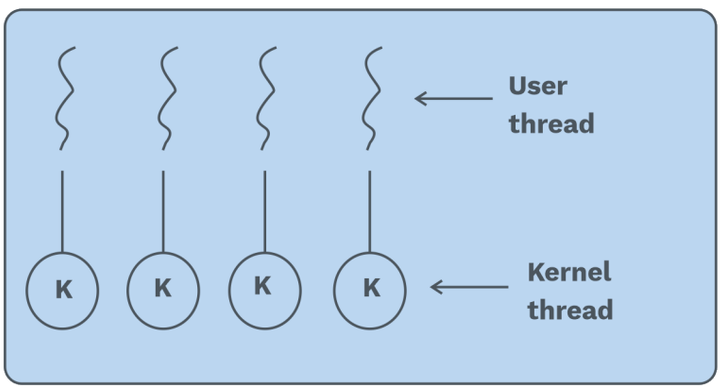


Fig. 2.20 One-to-One Model

Many–to–many model:

- In this model, any number of user level threads can be mapped to an equal or less number of kernel level threads.
- This model implements the best features of both one-to-one model and many-to-one model.
- There is no restriction on the number of threads created by the user.
- If any kernel level thread is blocked, then the entire process is not blocked.
- A process can be split among many processors.

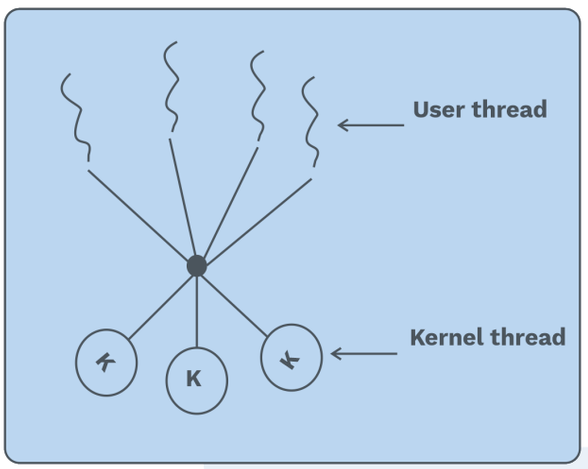


Fig. 2.21 Many-to-Many Model

SOLVED EXAMPLES

Q17 **Consider a multiprocessor system with two processors. Currently, there are two processes, P and Q, executing on separate processors. Let P have five user level hreads and Q have four kernel level threads. If one of the threads of each process gets blocked, then which one of the following is TRUE?**

a) All threads in process Q get blocked, but there is only one thread in process P that gets blocked.

b) All threads in process P get blocked, but there is only one thread in process Q that gets blocked

c) All threads in process P and process Q get blocked.

d) Only one thread in process P gets blocked and only one thread in process Q gets blocked.

Sol: **Option: b)**

All user level threads of a process get blocked, even single thread blocks; this is also a major drawback with user level threads. But in the case of kernel level threads of a process, if one thread gets blocked, other threads continue to execute.

Difference between multitasking and multithreading:

Multitasking

In the case of multitasking, these multiple jobs are executed on a single CPU in time-sharing mode. CPU is switched among different jobs.

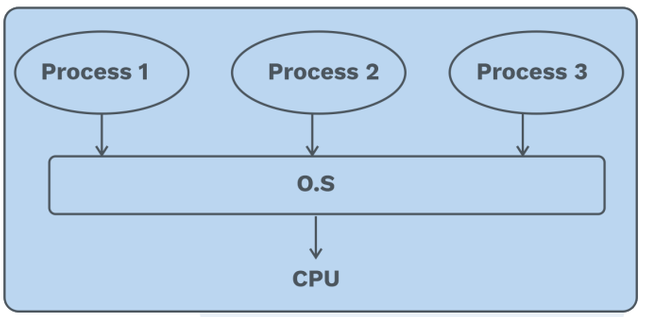


Fig. 2.22 Multitasking

Multithreading

In the case of multithreading, a process is divided into multiple threads, which can execute concurrently, thus increasing the power of the system.

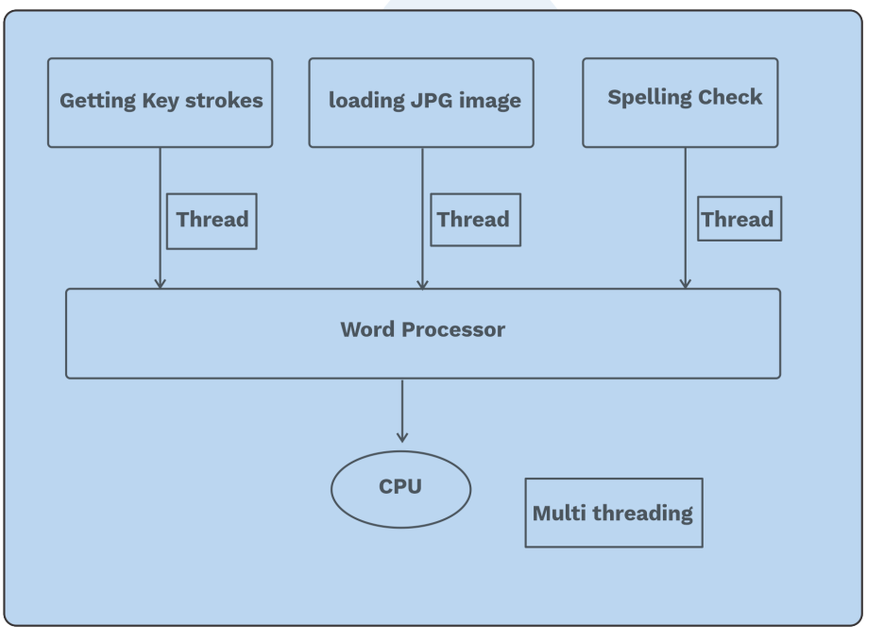


Fig. 2.23 Multithreading

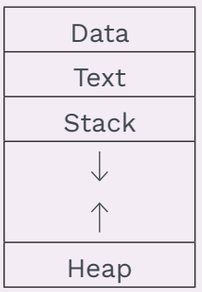
Multitasking	Multithreading
1) Multiple jobs are executed on a single CPU in time-sharing mode	**1)** Many threads are created from a process through which system's throughput is increased
2) It involves CPU switching between jobs	**2)** It involves switching between threads
3) Separate memory space is allocated to each job	**3)** Threads of a process share the same address space
4) There is no resource sharing among different jobs, each job will have its own resources	**4)** Threads share resources
5) It is slow	**5)** It is faster than multitasking
6) Termination of the process takes more time	**6)** Termination of thread takes less time

Rack Your Brain

Will multithreading give any benefits to a unicore processor?

Chapter Summary

- Process: **1)** Process is a program in execution
  2)



Process image

- Process operations:
  1) Create
  2) Schedule
  3) Run
  4) Block /Suspend
  5) Resume
  6) Terminate

- Process control block: Resumption of the process in OS.
- Types of process states:
  1) New
  2) Running
  3) Waiting
  4) Ready
  5) Terminated
  6) Suspend

- Dispatcher: It is a module/component for context switch operation in OS.
- Types of scheduler:
  1) Short-term scheduler
  2) Middle-term scheduler
  3) Long-term scheduler

- Context switch: Operation of storing the current process and loading another

- Goals of CPU sheduling:
  1) Maximise throughput and CPU utilisation
  2) Minimise turnaround time and waiting time.

- Scheduling algorithms:
  1) FCFS
  2) SJF/SRTF
  LJF/LRTF
  4) RR
  5) Priority
  6) HRRN
  7) MLQ
  8) MLFQ
- Threads: It is a basic unit of CPU utilisation
- Multithreading benefits:
  1) Responsiveness
  2) Resource sharing
  3) Economy
  4) Utilisation of multiprocessor architectures
- Types of threads:
  1) User level threads (ULT)
  2) Kernel level threads (KLT)
- Multithreading models:
  1) Many-to-one
  2) Many-to-many
  3) One-to-one

3 IPC and Synchronization

3.1 BASICS OF INTERPROCESS COMMUNICATION

Co–operative process:

Definition

A process is co-operating if it can affect or be affected by the other processes executing in the system.

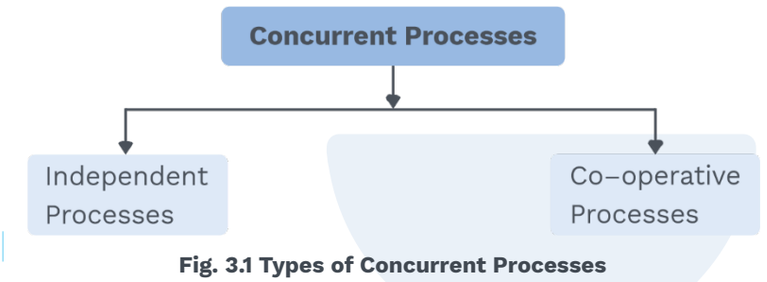


Fig. 3.1 Types of Concurrent Processes

Independent process:

Definition

A process is independent if it cannot affect or be affected by other processes. Clearly, any process that shares data with other processes is a co-operating process.

Need for co–operative process:

There are plenty of reasons to have the co-operating processes in a system co-operation:

1) **Information sharing:** involves concurrent access to the resources of the system.
2) **Modularity:** With modularity, a complex task be partitioned into several easy tasks.
3) **Computation speedup:** Involves breaking a task into multiple subtasks for faster concurrent execution.

To achieve above said benefits processes of the system should be co-operating with each other.

a) Shared memory:

Definition

A region of memory that is shared by co-operating processes is established. The process can then exchange information by reading and writing data to the shared region.

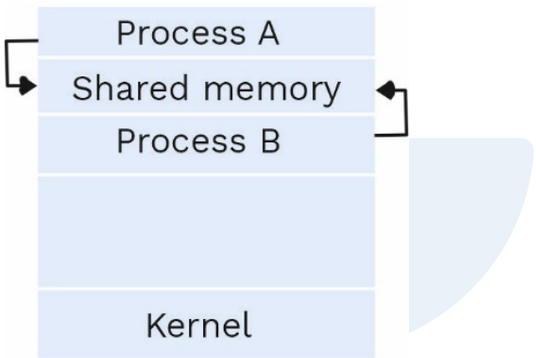
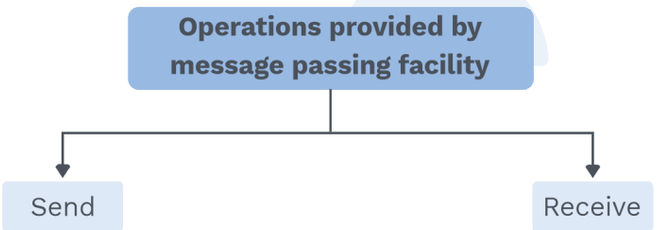


Fig. 3.2 Shared Memory



b) Message passing system:

1) Message passing is a mechanism that lets two processes communicate and synchronize provided communication link should be there between two processes.

2) Messages sent by a process can be either fixed-sized messages or variable-sized messages. Fixed-sized messages can be sent with less complex mechanisms, while variable-sized messages require a more complex system-level implementation.

3) But more important, how these processes logically communicate with each other.

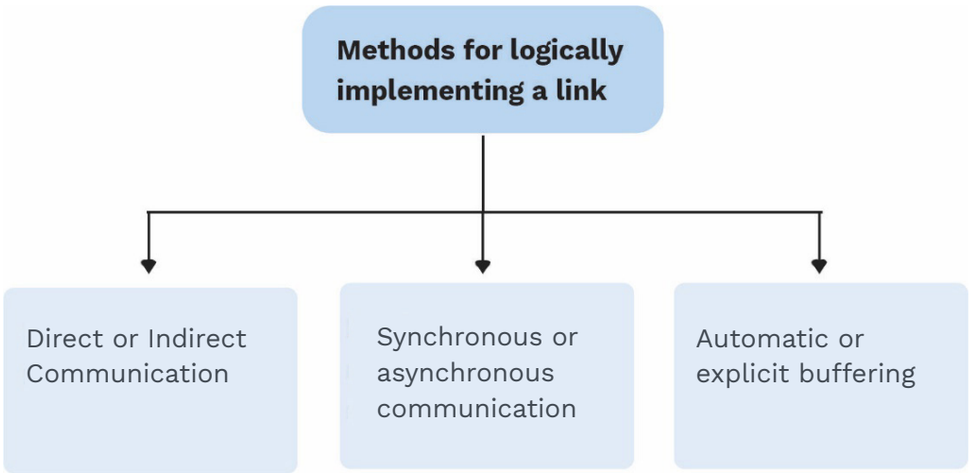


Fig. 3.3 Different Methods to Implement a Link

a) Direct communication:

"Each process that wants to communicate must explicitly name the recipient or sender of the communication."

Send(Process_name, message), this will send message to mentioned process and Receive(Process_name, message) will receive the message from the mentioned process.

Properties of communication link:

The link is automatically established between exactly two processes providing the process knows each other's id.

b) Indirect communication:

1) When messages can be placed or removed by the process or received in ports.

2) Send(Process_name, message) will send the message to port A and Receive(Process_name, message) will receive the message from port A.

Properties of communication link:

Link will be established if and only if both participants have the same port and it can involve two or more processes.

Blocking send:

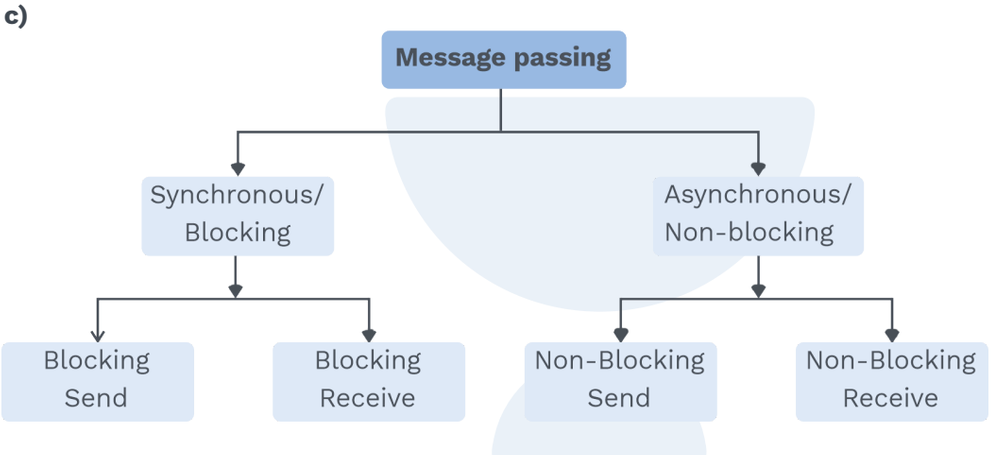
Here the sending process is halted until the receiving process receives a message from the mailbox.

Non-blocking send:
The sending procedure sends the message and returns the system to normal functioning.

Blocking receive:
Until the message is available, the process is halted.

Non-blocking receive:
Either the message received by the recipient is legitimate or it is null.

c)



d) Automatic or explicit buffering:
Messages exchanged by communicating processes, whether direct or indirect, are stored in a temporary queue.

Zero capacity:
No messages can be retained in the queue because it can only have a maximum length of zero. In this case, the communication initiator process has to wait until the recipient receives the message.

Bounded capacity:
The queue can only hold n messages because it has a finite length of n. When a new message is sent and the queue is not yet full, the message is added to the queue, and the sender is free to continue without waiting. The capacity of the link is limited; nevertheless, if the link is full, the sender will have to wait until a spot in the queue becomes available.

Unbounded capacity:
There is no bound on the limit of messages that can wait in the queue because the length of the queue is infinite; therefore, it never gets blocked.

SOLVED EXAMPLES

Q1 **Consider the following statements regarding "Shared memory" and "Message passing" (the two fundamental modes of IPC).**

I. Message–passing involves more kernel intervention than shared memory.
II. Shared–memory is easier to implement than message passing.
III. Shared–memory allows maximum speed than message–passing.
Which of the above statements are TRUE?

a) Only I and III **b) II and III** **c) Only II** **d) All are correct**

Sol: **Option: a)**

Message passing requires kernel support everytime it sends message.
Shared memory is more complex to implement because of security concerns.
Shared memory is faster than message passing.

Q2 **Consider the following statements:**

S1. In a blocking send communication, the process waits for the acknowledgement from the receiving process.
S2. A blocking send is asynchronous communication
S3. A non-blocking send is asynchronous communication
Which one of the following is TRUE?
a) S1 and S2
b) S2 and S3
c) S1, S2 and S3
d) S1 and S3

Sol: **Option: d)**

In a blocking send communication, the receiver sends an acknowledgement which is a notification to the sending process so that the sending process can continue.

1) A blocking send is synchronous communication as sender and receiver processes are synchronized with each other

2) A non-blocking send is asynchronous communication. In this communication the sending process can continue execution without getting an acknowledgment from the receiver.

3.2 BASICS OF PROCESS SYNCHRONIZATION

Race condition:

Definition

A race condition in an operating system is a situation where two or more processes/threads manipulate a shared resource, and the final outcome depends on the order in which processes/threads complete their execution.

Consider the following scenario: two processes are required to conduct a bit flip at a certain memory address.

P1	P2	Value
Read		0
Flip		1
	Read	1
	Flip	0

Process 1 conducts a bit flip from 0 to 1 in this scenario. After that, process 2 conducts a bit flip from 1 to 0.
If there was a race condition, two processes would overlap.

P1	P2	Value
Read		0
	Read	0
Flip		1
	Flip	1

The bit in this situation is set to 1 when it should be set to 0. This occurred as a result of processes being unaware of one another.

- A race condition occurs when two or more threads attempt to read, write, or make decisions depending on the contents of memory they are accessing at the same time.
- To avoid a race condition, we must ensure that only one process has access to the shared data at any given time. Mutual exclusion is the term for this.
- Critical sections are areas of a program that attempt to access shared resources and trigger race conditions.

3.3 THE CRITICAL SECTION PROBLEM

Definition

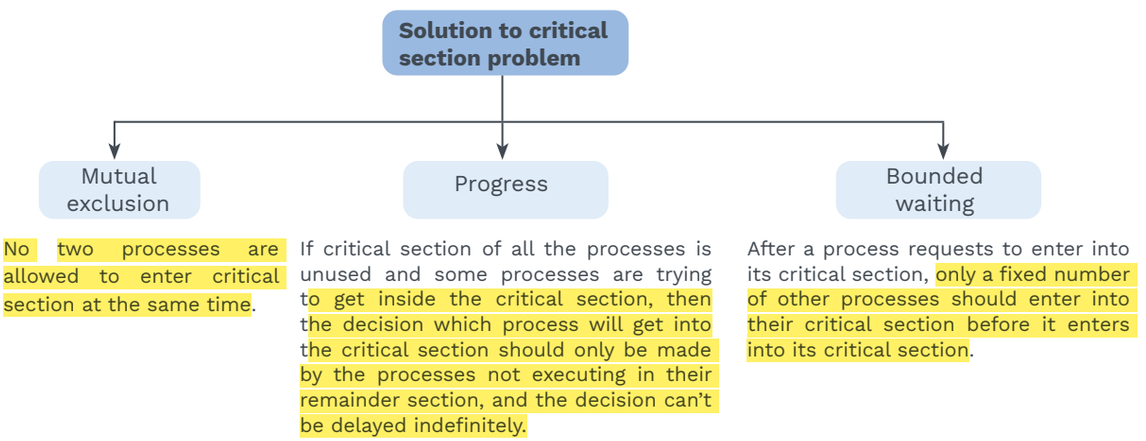
The critical section problem is to design a protocol that the processes can use to co-operate. The protocol must ensure that when one process is executing in its critical section, no other process is allowed to execute in it's critical section.

- The critical section is the area in the code section of a process where it manipulates shared resources.
- Each task should first ask for approval to get into its critical region.
- This request is made in the entry section. The critical region may be followed by an exit segment. The remaining code is included in the remainder section.
- The complexity in the critical region is to develop a protocol that tasks can use to ensure that their actions are independent of the order in which they are carried out.

```
do{
      /*Entry Section*/
//       Critical Section
       /*Exit Section*/
//     Remainder section
} while (1);
```

Fig. 3.4 Critical Section Program Structure

Three requirements given below must met by a solution to the critical section problem.



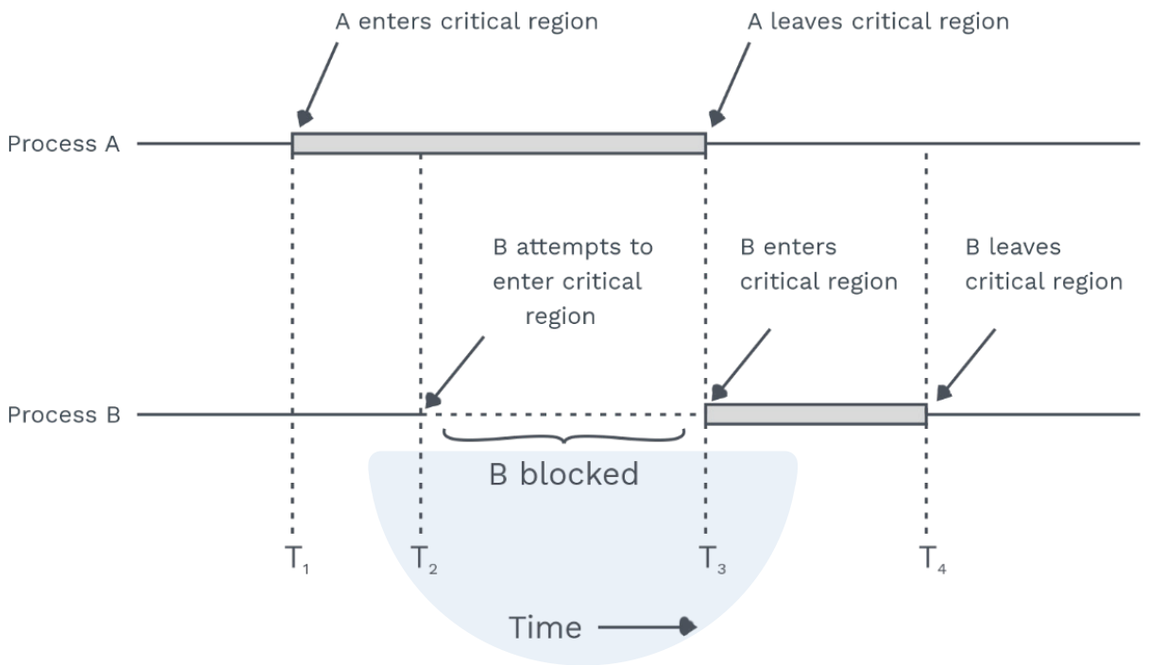
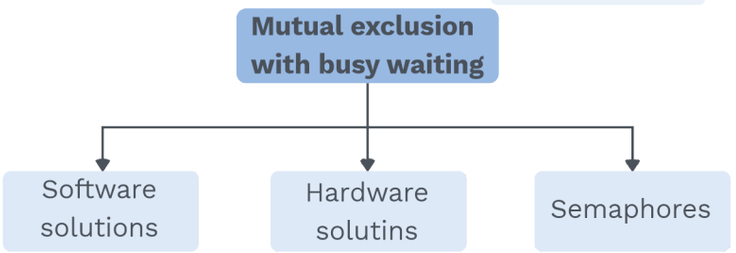


Fig. 3.5 Mutual Exclusion Using Critical Regions

Mutual exclusion with busy waiting:



Software solutions:

Lock variables:

Consider a single shared lock variable that has a value of 0 at the start. When a process wants to enter its critical zone, it first checks the lock and, if it's 0, sets it to 1 before proceeding to the critical zone. If the lock is already 1, the other process simply waits for it to reach 0, indicating that no processes are running in the critical region at the moment.

Woefully, this method has a flaw. Assume that one process reads the lock and finds it to be 0 done by other processes. Before it sets the lock to 1, another process gets scheduled, which runs and sets the lock to 1, and when the first process begins again, it will set the lock to 1; therefore, mutual exclusion fails here.

Strict alternation:

Process 0	Process 1
while (True)	while (True)

{
while (turn ! = 0); //loop
critical_region ( );
turn = 1;
non critical region ( );
}
	{
	while (turn ! = 1); //loop
	critical_region ( );
	turn = 0;
	non critical region ( );
	}

The integer variable turn, which starts at 0, keeps track of who is in charge of entering the critical region and inspecting or updating the shared memory. Process 0 inspects turn at first, finds it to be zero, and then enters its critical zone. Process 1 also thinks it's 1, so it's stuck in a tight loop, testing each turn to see when it becomes 1. Busy waiting is the process of continuously testing a variable until a needed value occurs.
It should be avoided because it consumes CPU resources.

Note:

A lock that uses busy waiting is called a spin lock.

Grey Matter Alert!

- Preemption is just a temporary stop, and the process will come back and continue the remaining execution.
- If there is any possibility of a solution becoming wrong by taking the preemption, then consider the preemption.
- If any solution has a deadlock, progress is not satisfied, but if a deadlock is not there, progress may/may not be satisfied.

Mutual exclusion and bounded waiting are both satisfied in strict alternation, but progress is not.

As when process 1 finishes executing its critical section, it changes the turn variable to 0 and then starts executing in remainder section. Now, if again Process 1 wants to start, it cannot execute as the turn variable is having value 0, so only process 0 can allow process 1 to execute after it has done executing. So here, progress is not satisfied.

Previous Years' Question

Consider the following two-process synchronization solution

Process 0	Process 1
Entry: loop while (turn == 1); (critical section) Exit : turn = 1;	Entry: loop while (turn == O); (critical section) Exit turn = O;

The shared variable turn is initialized to zero. Which one of the following is TRUE?

a) This is a correct two-process synchronization solution

b) This solution violates mutual exclusion requirement

c) This solution violates progress requirement

d) This solution violates bounded wait requirement

Sol: c) **(GATE CS-2016)**

Peterson's solution:

Algorithm:

```
    # define FALSE      0
    # define TRUE       1
    # define N          2              /* number of processes */
    int turn;                          /* whose turn is it */
    int interested [N];                /* all values initially 0 */
void entry_section (int process)/* process is 0 or 1 */
{
    int other;      /* other process */
    other = 1–process ;
    interested [process] = TRUE ;
    turn = process ;      /* set flag */
    while (turn = = process & & interested [other] = = TRUE);
}
    /* Critical Section */
void exit_section (int process)        /* process leaving */
{
    interested [process] = FALSE;      /* process exited */
```

}

Peterson's solution for achieving mutual exclusion

Peterson's solution for mutual exclusion

turn is a common variable that is used by both the P_0 and P_1 processes. Each of the two processes calls the entry_ section with its individual process number 0 or 1 as the parameter before entering the critical zone. This causes processes to wait until it is safe to enter, if necessary.

When the process is finished with the critical section, it calls the exit section to signify that it is finished and to allow the other process to enter if it so chooses.

Working:

- Neither process is in its critical zone at the start. Process 0 now invokes entry section. It signals interest by setting the turn to 0 and the array element to 1. The entry section returns instantly because process 1 isn't interested. If process 1 now calls the entry section, it will remain in that section until interested [0] is set to FALSE, which happens only when process 0 calls the exit section to exit the critical zone.
- Consider the case where both processes call the entry section at nearly the same time. In turn, each will save their process number. The process that stores the most recent data is the one that counts.
- Suppose process 1 is the last to store; therefore, turn = 1, process 0 will now enter its critical section while process 1 waits.

Advantages:

1) It satisfies mutual exclusion.
2) It satisfies progress.
3) It satisfies bounded waiting.

Disadvantages:

1) It suffers from busy waiting.
2) It suffers from priority inversion.

Previous Years' Question

Consider Peterson's algorithm for mutual exclusion between two concurrent processes i and j. The program executed by the process is shown below. repeat

```
flag[i] = true;
turn =j;
while (P) do no-op;
Enter critical section, perform actions, then
exit critical section
Flag[i] = false;
Perform other non-critical section actions.
```

Until false;

For the program to guarantee mutual exclusion, the predicate P in the while loop should be

a) flag[j] = true and turn = i **b)** –flag[j] = true and turn = j

c) flag[i] = true and turn = j **d)** –flag[i] = true and turn = I

Sol: b) **(GATE CS-2001)**

Mutex locks:

- Solutions to the critical section problem using hardware are complex and restricted to be used only by system processes.
- For reasons, OS designers came up with the solutions to the critical section problem using software methods which can be used by the application programs.
- Mutex Lock is one such tool (short for Mutual exclusion lock)
- A mutex lock is used to protect critical regions, preventing race conditions.
- Entry into the critical section is possible only after the process obtains the lock; the lock is obtained using acquire() function.
- When it exits the critical region, the process releases the lock using the release() function in the exit section so that another process can obtain it to get into its critical section.

```
acquire ( ) {
    while (!available) ; //busy wait
    available = FALSE ;
}
    release ( ) {
    available = TRUE ;
}
```

Fig. 3.6 Definition of Release () and Acquire ()

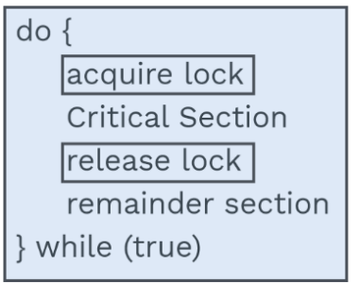


Fig. 3.7 Solution to Critical Section Using Mutex Lock

1) Calls to acquire() and release() must be made alternatively.
2) The main flaw in this implementation is that it necessitates processes to do busy waiting. Because the process "Spins" while waiting for the lock to become available, this sort of mutex lock is also known as a spin lock.

Hardware solution:
Disabling interrupts:
The simplest solution is for each process to disable all interrupts when it reaches its critical section and then re–enables them when it exits.
If interrupts are disabled, there will be no clock interrupts. Because the CPU is only transferred between processes as a consequence of clock or other interrupts, once interruptions are disabled, the CPU will not be shifted to another process, and the current process will be allowed to update the shared memory without risk of interference from other processes.

The TSL Instruction (test and set lock):
Let's have a look at a concept that requires some hardware assistance.
An instruction is found on many computers, particularly those designed with many processors in mind.

Test and set lock works as follows:
- We'll utilise a shared variable called LOCK to coordinate access to shared memory while using the TSL instruction.
- Register RX is filled with the contents of the memory word. This is an Atomic or Indivisible process, which implies no other process can access the memory word until the instruction is finished.
- If LOCK is 0, any process can use the TSL instruction to set it to 1 and access or write the shared memory. When it's finished, the operation uses an ordinary move command to reset LOCK to 0.

How the TSL instruction satisfies mutual exclusion?

```
Enter section:
1) TSL REGISTER, LOCK        |Copy LOCK to register and set LOCK to 1
2) CMP REGISTER, # 0         |Was LOCK zero?
3) JNE ENTER_SECTION         |If it was non zero, LOCK
4) RET                       |was set, so loop return to caller ; critical
                              section entered.
Critical Section:
Exit Section:
1) MOVE LOCK, # 0            |Store a 0 in LOCK
2) RET                       |return to caller
```

Fig. 3.8 Entering and Leaving a Critical Region Using TSL Instruction

- Although the mutual exclusion and progress are guaranteed it also suffers from busy waiting.

Grey Matter Alert!

Priority Inversion Problem:
Consider a computer with two processes, H, with high priority and L, with Low priority, which shares a critical region. The scheduling rules are such that H will run whenever it is in a ready state. At a certain moment, with L in its critical region. H becomes ready to run (e.g., a 1/0 operation completes). H now begins waiting but since Lis never scheduled while H is running, L never gets the change to leave its critical region, so H loops forever.

Let's take a look at some interprocess communication primitives that, when denied access to their critical regions, block rather than squander CPU time. One of the most basic pairs is sleep and wake up. A sleep call puts the caller to sleep, whereas a wake-up call just has one parameter: the process to be awakened.

Compare and swap

- Following is the implementation of Compare_and_Swap():

```
int Compare_and_Swap (int *value, int expected, int new value)
{
    int temp = *value ;
    if (*value == expected)
        *value = new_value ;
    return temp ;
}
```

Fig. 3.9 The Definition of the Compare_and_Swap( ) Instruction

- The operand 'value' is set to 'new value' if the expression (*value == expected) is TRUE.
- Compare and Swap() always returns the variable's original value in any situation.

```
do {
    while (Compare_and_Swap (&lock,0,1)! = 0)
    i /* do nothing */
     /* critical section */
    lock = 0 ;
       /* remainder section */
    } while (TRUE);
```

Fig. 3.10 Mutual–exclusion Implementation with Compare and Swap

Compare_and_Swap( ) instruction:

- Test and Set() and Compare_and_Swap() are both atomic operations ( ).
- When Compare_and_Swap() is called for the first time, 'lock' is set to 1. It will then enter into the critical area because the original value of 'lock' was equal to the expected value of 0. Following calls to Compare_and_Swap() will fail since the anticipated value and lock value are different.
- When a process quits its critical section, the lock is reset to 0, allowing another process to enter the critical area.

Note:

Solution	Mutual Exclusion	Progress	Bounded Waitiong
Lock Variable	X	✓	X
Strict Alternation	✓	X	✓
Peterson's Algorithm	✓	✓	✓
Test and Set( )	✓	✓	X

SOLVED EXAMPLES

Q3 **Consider the following synchronization mechanism for two process**

```
Process 0                          Process 1
// Start                           // Start
while (id == 1) ;                    while (id == 0) ;
   /* critical section */                /* critical section */
// end                             // end
      id = 1 ;                          id = 0 ;
```

Shared variable id is initialized to 0, which of the following is true?

a) Mutual Exclusion is satisfied but not progress
b) Progress is satisfied but not Mutual Exclusion
c) Both satisfied
d) None satisfied

Sol: **Option: a)**

The value of the shared variable could be 0 or 1 at a time, and only one process can enter into the critical section at a time, so Mutual Exclusion is satisfied, but there is a strict alternation algorithm applied here, so progress is not satisfied.

Sleep and wake (Producer consumer problem)

- Assume there are two system calls, one for sleeping and the other for waking. The process that requests sleep will be turned off, while the process that requests wake-up will be turned on.
- A well-known example is a producer-consumer problem, which is the most common problem for modeling the sleep-wake cycle.
- Problem statement – There is a fixed-size buffer. The producer can produce an item and write it into the buffer, and the consumer can consume an item from the buffer. These two events should not take place simultaneously. Buffer is a critical section here.

```
        # define N 100
        int count = 0 ;
        Void producer (void)
{
        int item ;
        while (TRUE) {
        item = produce_item ( ) ;
        if (count == n) sleep ( ) ;
        insert_item (item) ;
        count ++ ;
        if (count == 1) wake up (consumer) ;
}

Void Consumer (void)
{
        int item ;
        while (TRUE) {
        if (count == 0) sleep ( ) ;
        item = remove_item ( ) ;
        count –– ;
        if (count == N–1) wake up (producer) ;
        consume_item (item) ;
}
}
```

Fig. 3.11 The Producer–Consumer Problem with Fatal Race Condition

- A variable, count, will be used to keep track of the number of objects in the buffer. If the buffer's maximum number of items is N, the producer code will check if the count is N or not so that the producer can go to sleep or add an item; the consumer code will check if there are items present in the buffer (count == 0) if no items are present it will go to sleep or consume an item otherwise.
- It also wakes up the producer process if the count is equal to N-1, i.e., space is available in the buffer.

Grey Matter Alert!

How race condition occurs in producer-consumer problem?
It occurs because access to count is unconstrained. The buffer is empty, and the consumer has just read the count to see if it is 0. Now consumer preempts and producer schedules and enters an item to buffer, increments count to 1, and calls wake up system call to wake the consumer up.
But, the consumer is not yet sleeping. When the consumer next runs, it knows the count value to be zero and goes to sleep; sooner or later producer fills up the buffer and goes to sleep. Both producer and consumer sleep forever. Deadlock occurs.

Tracing:

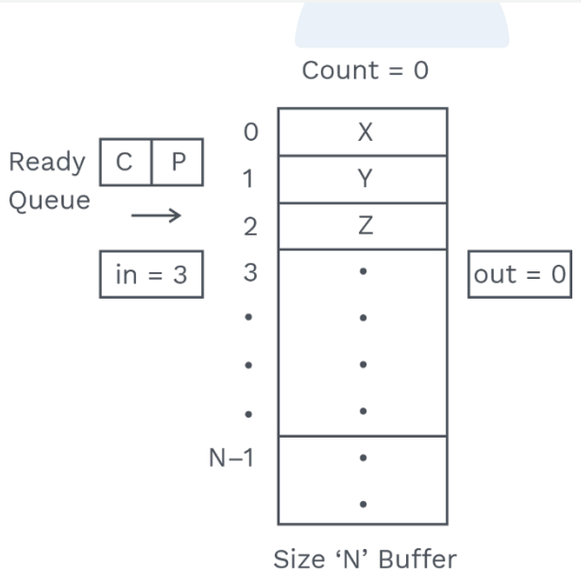


Fig. 3.12 Tracing of Bounded Buffer

Assume first Consumer is Scheduled to execute

Time	Process	Action
t_0 :	C	Checks the count and preempt
t_5 :	P	Add one item and increment count to 1 and wake up call to consumer [∴ wakeup call lost]
t_{10} :	C	Consumer rescheduled, checks count to 0, goes to sleep
t_{15} :	P	Producer fills up the buffer and goes to sleep
t_N :	–	Deadlock Occurs

Note:

The essence of the problem here is that a wake up sent to a process that is not yet sleeping is lost. If it were not lost, everything would work. So one solution to this problem is resolved by semaphores or mutexes.

Semaphores:

Semaphore is an integer value having wait and signal operation.

```
Wait (S) {
    While (S < = 0) ; //Busy wait
    S –– ;
}
```

Fig. 3.13 Wait Definition

```
    Signal (S) {
    S ++ ;
}
```

Fig. 3.14 Signal Definition

All adjustments to the integer value of the semaphore must be made indivisibly in the wait() and signal() actions, exactly as in the case of wait (S), the testing of the integer value of S (S $\leq$ 0), as well as its possible modification (S––), must be accomplished without interruption.

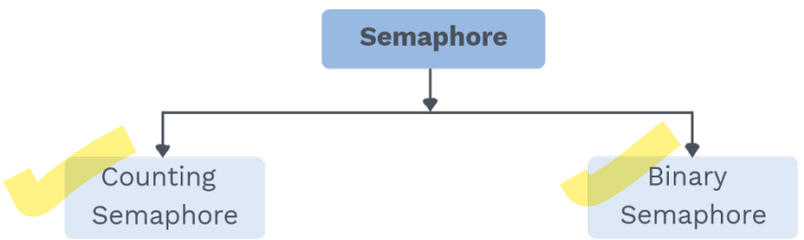


Fig. 3.15 Types of Semaphores

Counting semaphore:

- A counting semaphore's value can span an unrestricted range.
- They can be used to limit which process has access to a resource with a limited number of instances.

Binary semaphore:

- Binary semaphore can hold a value either 0 or 1.
- Binary semaphores can be utilised to ensure mutual exclusion on systems that lack mutex locks.

Semaphore usage:

In the semaphore, the quantity of resources accessible is initialized. When a process uses a resource, it uses the semaphore to execute a wait() activity, and when it releases a resource, it uses the semaphore to perform a signal() operation. When the semaphore count reaches 0, it means that all resources have been occupied. Then, depending on how the implementation is done, programs that wish to access a resource will either be blocked or forced to wait until the count exceeds zero. Semaphores can also be used to solve a variety of synchronization problems.

Example: Consider two processes that are executing at the same time: P_1 with statement S_1 and P_2 with statement S_2. Let's assume we only want S_2 to run if S_1 has finished. Allowing P_1 and P_2 to share a single semaphore "Synch" with a value of 0 makes this method simple to implement. We insert the statements in the process P_1.

S_1;
Signal (Synch);

In process P_2,

Wait (Synch);
S_2;

Because the synch semaphore is initially 0, P_2 will only execute S_2 after P_1 has invoked signal (synch), which is after statement S_1.

Semaphore implementation:

```
Wait (Semaphore *S) {
      S → Value −− ;
      if (S → Value < 0) {
                        add this process to S→list
                        block ( );
      }
}
```

Fig. 3.16 Modified Wait ( ) Definition

```
Signal (Semaphore *S) {
     S→Value ++ ;
     if (S →Value < = 0) {
                        remove a process P from S→list ;
                        Wake up (P) ;
     }
}
```

Fig. 3.17 Modified Signal( ) Definition

- Note that, unlike the prior definitions of wait() and signal() with busy waiting, semaphore values in this implementation can be negative.
- The magnitude of a semaphore is the number of processes that are waiting on it if its value is negative. A link field in PCB can easily implement a list of pending processes. An integer and a pointer to a list of PCBs are stored in each semaphore.
- Bounded waiting is ensured by using a FIFO queue (strong semaphore) or a LIFO queue (weak semaphore) to add and delete processes from the list. Both the head and tail pointers to the queue are included in the semaphore.

Note:

- Positive values of semaphore indicates number of successful down operation.
- Negative value of semaphore indicates number of processes in the suspended list/blocked list.

SOLVED EXAMPLES

Q 4 **Consider a system where the initial value of the counting semaphore S = +17. The various semaphore operations like 23P, 19V, 7P, 15V, 2P, 5V are performed. Then what is the final value of counting semaphore?**

Sol: **Range: 24–24**

After 23P – S = –6
After 19V – S = +13
After 7P – S = +6
After 15V – S = +21
After 2P – S = +19
After 5V – S = +24

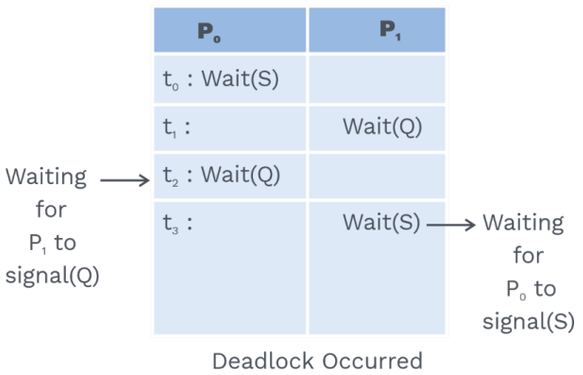
Deadlocks and starvation:

When a semaphore and a waiting queue are combined, two or more processes may end up waiting indefinitely for an event that can only be caused by one of the waiting processes. Signal() operation is the event. These processes are said to be stalled when they reach this state.

Consider the following scenario: Consider a system with two processes, P_0 and P_1, each of which has access to two semaphores S and Q, all of which are set to the value 1.

P_0	P_1
Wait(S);	Wait(Q);
Wait(Q);	Wait(S);
.	.
.	.
.	.
.	.
Signal(S);	Signal(Q);
Signal(Q);	Signal(S);

Tracing:



Indefinite blocking or starvation, a circumstance in which processes wait eternally within the semaphore, is another difficulty related to deadlock.

Grey Matter Alert!

- Every semaphore variable will have its own suspended list.
- Down and up operations are atomic.
- If more than one process is in the suspended list, then every time we perform one up operation, one process will wake up and this will be based on FCFS.
- If two or more processes are in the suspended list and there is no other process to wake these processes. Then deadlock occurs.

SOLVED EXAMPLES

Q5 **Consider the following three processes with the semaphore variable S_1, S_2 and S_3.**

P_1	P_2	P_3
while (1) **{** **P(S_3) ;** **Print ("a") ;** **V(S_2) ;** **}**	**while (1)** **{** **P(S_1) ;** **Print ("b") ;** **V(S_3) ;** **}**	**while (1)** **{** **P(S_2) ;** **Print ("c") ;** **V(S_1) ;** **}**

Sol: **Option: d)**

If $S_1 = 0$, $S_2 = 1$ and $S_3 = 0$ the pattern cbacba.... will be printed.

Previous Years' Question

Consider two processes P, and P2 accessing the shared variables X and Yprotected by two binary semaphores Sx and Sv respectively, both initialized to 1. P and V denote the usual semaphore operators, where P decrements the semaphore value, and V increments the semaphore value. The pseudo-code of P, and P2 is as follows:

P1:	P2 :
L1:	L3: ....
L2 :	L4 : ..... .
X=X+1;	Y=Y+1;
Y=Y-1;	X=Y-1;
V(Sx) ;	V(Sv) ;
V(Sv) ;	V(Sx) ;
}	}

In order to avoid deadlock, the correct operators at L,, L2 , L3 and L. are respectively.

a) P(Sv), P(Sx) ; P(Sx), P(Sv) **b)** P(Sx), P(Sv) ; P(Sv), P(Sx)

c) P(S,), P(Sx) ; P(S,), P(S,) **d)** P(Sx), P(Sv) ; P(Sx), P(Sv)

Sol: d) **(GATE CS: 2004)**

3.4 CLASSICAL PROBLEMS OF SYNCHRONIZATION

The bounded–buffer problem:

- Here we present a solution to producer–consumer processes with the use of Semaphore.
- Producer and Consumer share the following data structure:
  int n; //Size of buffer
  Semaphore mutex = 1; //Binary Semaphore for mutual exclusion
  Semaphore empty = n;
  Semaphore full = 0;
- Semaphore empty and full represent the state of the buffer of how much empty and full is buffer at the current moment.

```
do
{    ...
     // produce item
     ...
     wait (empty) ;
     wait (mutex) ;
     ...
     // insert item to buffer.
     ...
     Signal (mutex) ;
     Signal (full) ;
}    while (TRUE)
```

Fig. 3.18 The Structure of the Producer Process

```
do
{
     wait (full) ;
     wait (mutex) ;
     ...
     // Remove an item from buffer
     ...
     Signal (mutex) ;
     Signal (empty) ;
     ...
     // Consume the item
}    while (TRUE)
```

Fig. 3.19 The Structure of the Consumer Process

Note:

The symmetry between the producer and the consumer we can interpret this code as the produce producing full buffers for the consumer or as the consumer producing empty buffers for the producer.

Rack Your Brain

What if we interchange wait (empty) and wait (mutex) in the producer process, at what will happen?

Rack Your Brain

What happens if we interchange signal (mutex), and signal (full), in the producer process code?

Previous Years' Question

Consider the solution to the bounded buffer producer/consumer problem by using general semaphores S, F and E. The semaphore S is the mutual exclusion semaphore initialized to 1. The semaphore F corresponds to the number of free slots in the buffer and is initialized to N. The semaphore E corresponds to the number of elements in the buffer and is initialized to 0.

Producer Process	Consumer Process
Produce an item; Wait(F); Wait(S) ; Append the item to the buOffer ; Signal(S); Signal(E) ;	Wait(E); Wait(S) ; Remove an item from the buffer ; Signal(S) ; Signal(F) ; Consume the item ;

Which of the following interchange operations may result in a deadlock?

I) Interchanging Wait (F) and Wait (S) in the Producer process

II) Interchanging Signal (S) and Signal (F) in the Consumer process

a) I) only **b) 11)** only

c) Neither **I)** nor **11)** **d)** Both **I)** and **11)**

Sol: a) **(GATE CS: 2006)**

The reader's–writer's problem:

- An area in memory is shared by multiple processes running concurrently. Some of these processes may merely wish to read data from the shared

area of memory, while others may want to update it. These two kinds of processes are referred to as readers and writers, respectively.

- There will be no detrimental effects if several readers access the shared data at the same time, but if some process (reader or writer) accesses the data with a writer, chaos may ensue.
- So, the solution could be to give writers exclusive access to the shared area in memory.
- The readers–writers process shares the following data structures:

Semaphore rw_mutex = 1;
Semaphore mutex = 1;
int read_count = 0;

```
do
    {
    wait (mutex) ;
    read count ++ ;
    if (read_count == 1)
        wait (rw_mutex) ;
    Signal (mutex) ;
        ...
    // reading is performed
        ...
    wait (mutex) ;
    read_count –– ;
    if (read_count == 0)
       Signal (rw_mutex) ;
    Signal (mutex) ;
}   while (true)
```

Fig. 3.20 The Structure of a Reader Process

```
do
    {
    wait (rw_mutex) ;
          ...
    // writing is performed
          ...
    Signal (rw_mutex) ;
}   while (true)
```

Fig. 3.21 The Structure of a Writer Process

Reader process:

- The entry to the critical portion is requested by the reader.
- If this option is enabled, it will increase the number of readers in the critical portion. If this is the first reader to enter, the "rw mutex" semaphore is locked, preventing writers from entering into the critical portion.
- The mutex is then signalled, allowing any additional reader to enter while the others are reading.
- It exits the critical section after conducting a reading. It checks if there are any more readers within before exiting, and if there aren't, it signals the semaphore "rw mutex", indicating that the writer can now reach the critical area.

Writer process:

- The entry to the critical part is requested by the writers.
- It enters and performs the write if authorised, i.e., wait () returns a true value. It continues to wait if it is not permitted.
- It is no longer in the critical section.
- As a result, the semaphore 'rw mutex' is queued on both readers and writers in such a way that readers are given priority if writers are also present.
- As a result, no reader will have to wait simply because a writer has requested access to a critical section.

Note:

- Reader ⟶ Writer (x): Will lead to inconsistency
- Reader ⟶ Reader (v): Multiple readers are allowed
- Writer ⟶ Reader (x): Inconsistent read operation
- Writer ⟶ Writer (x): Inconsistent data

Rack Your Brain

What happens if we interchange wait (mutex) and read count ++ in the reader process?

Hint: Both reader and writer will enter into database.

Rack Your Brain

What happens if we interchange wait (mutex) and read_count -- in reader process?

The dining philosopher problem:

- According to the dining philosopher problem, there are K philosophers seated around a circular table, each with one fork between them. If a philosopher can pick up the two forks next to him, he can eat.
- One of the adjacent followers may take up one of the forks, but not both.
- It is an example of a broad category of concurrency–control. It's a basic illustration of the necessity to distribute multiple resources among many processes in a way that avoids deadlock and starvation.

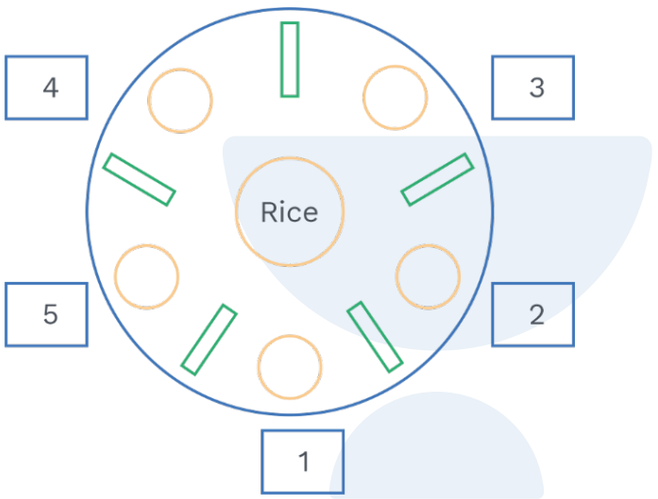
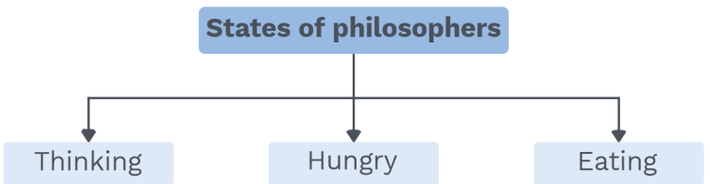


Fig. 3.22 The Situation of the Dining Philosophers.



An easy option is to use a semaphore to symbolize each fork. A philosopher attempts to obtain a fork by executing the wait () instruction on the semaphore. She releases her forks by using the signal() operation to send the relevant semaphores.

```
Semaphore chopstick [5] ;
do
    {
    Wait (chopstick [i]) ;
    Wait (chopstick [i+1] % 5) ;
      ...
    // eat for a while
      ...
    Signal (chopstick [i]) ;
    Signal (chopstick [(i+1) % 5]) ;
      ...
    // think for a while
      ...
    } while (TRUE)
```

Fig. 3.23 The Structure of Philosopher 'i'

Despite the fact that this method ensures that no two neighbors consume at the same time, it must be rejected due to the possibility of a deadlock.

Assume that each of the five philosophers is hungry at the same time. Each of them takes out her left chopstick. Chopsticks will now have all of their components. The same as 0. When each philosopher has exhausted her options for picking up her right chopstick, she will be postponed indefinitely (deadlock).

There are several potential remedies/solutions to the deadlock issue. in place of:

- At the table, no more than four philosophers should be present at the same time.
- Only let a philosopher pick up her chopsticks if both of her hands are free (to do this, she must pick them up in a critical section).
- Use an asymmetric solution: all odd-numbered philosopher picks up her left chopstick first, then her right, while all even-numbered philosopher picks up her right chopstick first, then her left chopstick.

Semaphore based solution:

```
#   define N 5
#   define Thinking 0
#   define Hungry 1
#   define Eating 2
#   define left (i + N – 1) % N;
#   define Right (i + 1) % N;
int State [N] = {Thinking };
Semaphore mutex = 1;
Semaphore S[N] = {1};
void philosopher (int i)
{
    while (1)
     {
       think (i) ;
       take_chopsticks( );
       eat (i);
       put_chopsticks (i);
     }
    }
void take_chopsticks (int i)
    {
       down (mutex);
       State [i] = Hungry;
       Test(i);
       up (mutex);
       down (S[i]);
    }
void put_chopsticks (int i)
{
    down (mutex);
    State [i] = Thinking;
    Test (Left);
    Test (Right);
    Up (mutex);
}
void test (i)
{
    if (State [i] == Hungry && State [Left] ! = Eating && State [Right] ! = Eating)
        {
             State [i] = Eating;
             Up (S[i]);
        }
}
```

SOLVED EXAMPLES

Q 6 **Each process P_i, i = 1 to 9 execute the following code**

```
while (true)
{
    P(mutex);
    C. S.   // Critical Section
    V(mutex);
}
```

The process P_{10} executes the following code.

```
while (true)
{
    V(mutex);
    C.S.
    V(mutex);
}
```

What is the maximum number of processes present inside the critical section at any point of time? (Assume initial value of Binary Semaphore mutex = 1)

a) 2 **b) 3** **c) 1** **d) 10**

Sol: **Option: d)**

First, P_1 will enter C.S. after the down operation, then P_{10} will enter up operation on the mutex.

The P_2 will go down the mutex and enter C.S, and P_{10} will get come out of C.S. and up on mutex, so P_3 will enter and so on.

C. S $\boxed{P_1\ P_{10}\ P_2\ P_3\ P_4\ P_5\ P_6\ P_7\ P_8\ P_9}$

So total 10 processes can enter at max.

Previous Years' Question

The following two functions P1 and P2 that share a variable B with an initial value of 2 execute concurrently.

P1 () {	P2 () {
C = B - 1;	D = 2 * B;
B = 2 * C;	B = D - 1;
}	}

The number of distinct values that B can possibly take after the execution is .....

Sol: 3 **(GATE CS: 2015)**

Monitors:

Although semaphores are a straightforward and effective technique for process synchronization, they can cause a variety of defects and delays that are difficult to notice if they are used incorrectly.

Example: 1)
```
Wait (mutex)
    ...
Critical section [Deadlock]
  ...
Wait (mutex)
```

Example: 2)
```
Signal (mutex)
  ...
Critical Section      [Violating Mutual Exclusion]
  ...
Wait (mutex)
```

When programmers employ semaphores poorly to address critical–section problems, as shown in the examples above, a variety of faults can be easily caused. High-level language structures called monitors – have been designed to deal with such problems.

```
Monitor monitor name
{
    /* shared variable declaration */
    function P1 (...) {
    }
    function P2 (...) {
    {
        .
        .
        .
    function Pn (...) {
    }
    Initialization_code (...) {
      ...
    }
}
```

Fig. 3.24: Syntax of a Monitor

- One method for achieving process synchronization is to use a monitor.
- Programming languages help the monitor accomplish mutual exclusion between processes.
- A monitor is an ADT. An Abstract Data Type (ADT) encapsulates data and provides a set of functions to interact with it that are independent of the ADT's implementation.
- It's a specific type of module or package that contains a collection of condition variables and procedures.
- Processes running outside the monitor can't access the monitor's internal variables, but they can call the monitor's procedures.
- Code can only be executed on monitors by one process at a time.

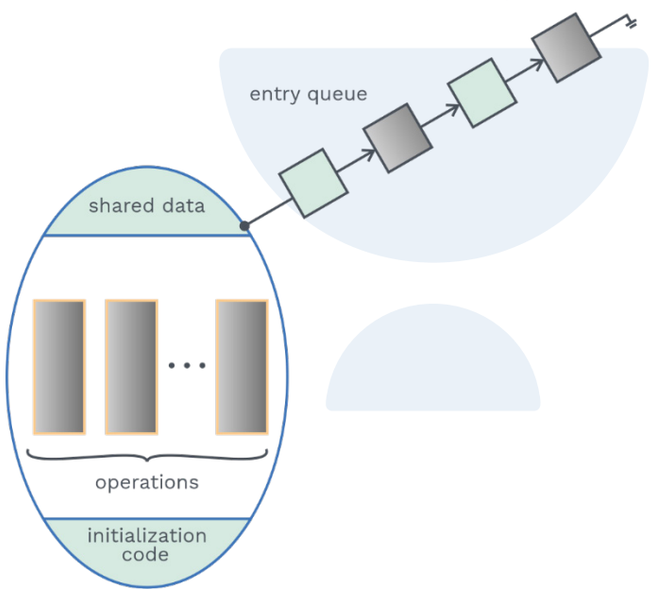


Fig. 3.25 Schematic View of a Monitor

- This synchronization restriction does not need to be explicitly coded by the programmer. We must employ mechanisms like the condition construct.
- Two operations can be performed on a condition variable: wait() and signal().
- Consider x as a condition variable, then x.wait() — This method suspends the process that is doing the action until it is called by another process.
- With x.signal, it resumes exactly one paused process. If there is no suspended process, Signal() has no effect.
- Assume that x.signal() is called by a process P for a suspended process Q that is associated with condition x. As a result, if 'Q' is allowed to continue execution, the signaling process 'P' must wait; otherwise, both 'P' and 'Q' will be active within the monitor.
- There are two options: 1) signal and wait and 2) signal and continue.

- P either waits for Q to exit the monitor or for another condition to occur.

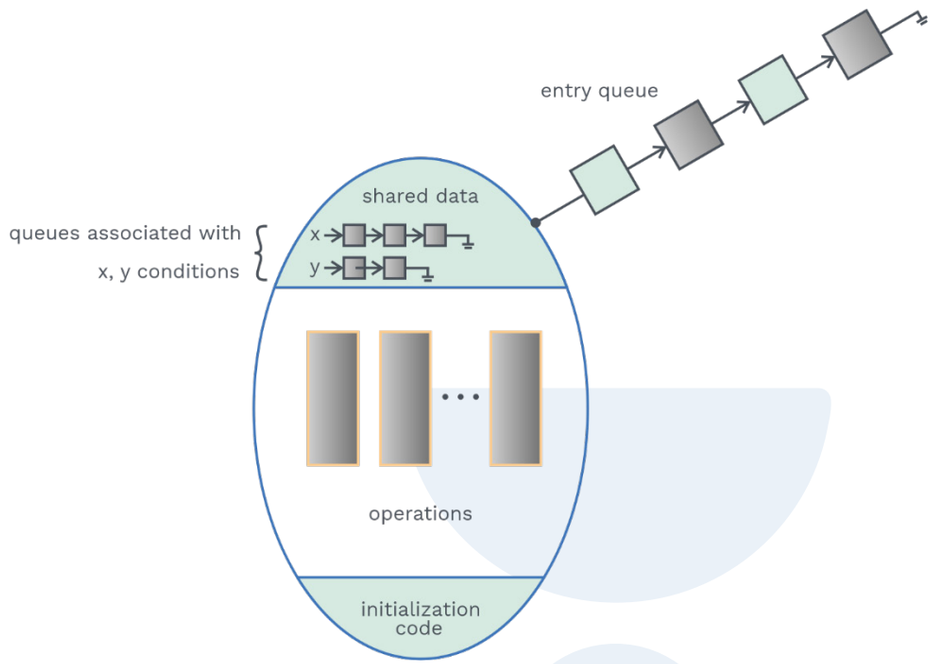


Fig. 3.26 Monitor with Condition Variables

- When P is already running in the monitor, the signal and continue method seems more sensible. If we let thread P continue, the logical condition for which Q was waiting may no longer be valid by the time Q is resumed.
- Using monitors as a dining philosophers solution
- The monitor mechanism solves the dining–philosopher's dilemma without causing a stalemate. The solution stipulates that a philosopher may only pick up her chopsticks if both of them are there.

```
Monitor Dining Philosophers.
{
      enum {Thinking, Hungry, eating} state[5] ;
      condition self [5] ;
      void pickUp(int i)
           State [i] = Hungry ;
           test (i) ;
           if (State [i] ! = Eating)
               Self [i]. Wait ( ) ;
      }
void putDown (int i) {
      State [i] = Thinking ;
      test ((i + 4) % 5) ;
      test ((i + 1) % 5) ;
}.
Void test (int i) {
      if ((State [(i + 4) %5] ! = Eating)
        && (State [i] = = Hungry) &&
            (State [(i + 1) %5] ! = Eating))
      {
            State [i] = Eating ;
            Self [i]. Signal ( ) ;
        }
     }
Initialization_code ( ) {
    for (int i = 0 ; i < 5 ; i++)
            State [i] = Thinking ;
    }
}
```

Fig. 3.27 Monitor Solution to Dining–Philosopher

3) "Condition Self [5]" must be declared in order for philosopher I to postpone herself when she is hungry but unable to obtain the chopsticks she needs.

4) The monitor dining philosopher in the above figure is in charge of chopstick distribution. Before beginning to eat, each philosopher must perform the procedure pick up ( ). The philosopher process may be suspended as a result of this action.

5) The philosopher may eat when the operation is completed successfully.

6) After then, the philosopher uses the put down ( ) command.

```
Dining Philosophers. pick up (i);
...
eat
...
Dining Philosophers . put down (i);
```

7) This solution assures that no two neighbours eat at the same time, preventing a deadlock, but a philosopher can die of starvation.

SOLVED EXAMPLES

Q7 **Consider the following code for producer-consumer problems with semaphores; producer and consumer can run in parallel.**

```
Semaphore A = 1;
Semaphore B = 0;
int item;
void producer()
{
 while 1)
 {
 wait A);
 item = produce();
 signal B);
 }
}
void consumer()
}
 while 1)
 {
 wait B);
 consume(item);
 signal A);
}
```

Which of the following statement(s) is/are true?

a) Starvation can be possible for either of producer and consumer

b) Deadlock can occur to the process

c) Progress is not ensured

d) An item produced by producer gets consumed by consumer before a new item gets produced by producer

Sol: **Options: c), d)**

a) There is no starvation to either the producer or the consumer, because bounded waiting is satisfied

b) Producer and consumer are operating on different semaphores, so there is no change of deadlock, so no hold and wait

c) True, if producer gets preempted before signal(B), then consumer can't enter its critical section

d) Producer and consumer can execute in parallel. Within infinite while loop, both the producer and the consumer has three instructions to execute. Let's start with the producer.

Producer:

1) wait A) changes A = 0 and goes to next instruction.

2) item is produced.

3) signal B) makes semaphore B as 1

Producer blocked at instruction 1 because of busy waiting. (A = 0).

Next consumer gets scheduled.

Consumer:

1) wait B) changes B = 0

2) consume (item)

3) signal A) changes A to 1, so producer now can execute instruction 2, i.e., can produce next item.

Q8 **Consider the following two-process synchronization solution:**

```
#define FALSE O
#define TRUE 1
#define N2
int turn;
int interested[N]={FALSE, FALSE);
void enter_region (int process);
{
  1) int other;
  2) other = 1-process;
  3) interested [process] = TRUE;
  4) turn = process;
  5) while (turn == other && interested process] == TRUE);
}
```

Critical_Region

```
void leave_region(int process)
}
  6) interested[process] = FALSE;
}
```

Which one of the following statement is correct?

a) Mutual exclusion is guaranteed, but progress fails

b) Deadlock occurs

c) Both mutual exclusion and progress are guaranteed

d) Mutual exclusion is not guaranteed

Sol: **Option: d)**

N = 2 //number of processes

Process P_o	Process P_1
process = 0	process = 1
other = 1	other = 0
interested[0]=TRUE	interested[1]=TRUE

Both processes can enter into their critical section simultaneously.
Assume P_0 starts executing first,
When P_0 executes statement no. 5.
while (turn == other && interested[process] == TRUE);
turn is 0, other is 1 so turn == other becomes FALSE, P_0 enters into critical region.

When P1 executes statement no. 5.
while (turn == other && interested[process] == TRUE);
turn is 1, other is 0 so turn == other becomes FALSE, P1 also enters into critical region.
while P_0 is still in its critical region, as a result mutual exclusion fails to hold.

Q9 **Following is the code for reader-writer algorithm where read_count = 0, mutex m = 1 = mutex db, counting semaphore S = 166.**

Reader	Writer

void reader()
{
while 1) {
P(m);
read_count++;
if (read_count == 1)
P(db);
X:_____
Y:_____
<critical section>
P(db);
read_count--;
if (read_count == 0)
V(db):
W:_____
Z:_____
}
}
	void writer()
	{
	while 1) {
	P(db);
	<critical section>
	v(db):
	}
	}

Choose the correct option for X, Y, W and Z, so that reader-writer works in synchronization:

a) X: V(s), Y: V(m), W: V(m), Z: P(s)
b) X: V(s), Y: V(m), W: P(m), Z: P(s)
c) X: V(m), Y: P(s), W: V(m), Z: V(s)
d) None of the above

Sol: **Option: c)**

X: V(m): provide entry for other readers
Y: P(s): value will reduce to 1 after entering one reader in critical section.
W: V(m): will give chance to other reader when It will come out of critical section.
Z: V(s): will increase number of readers come out from critical section.

Q10 **Consider two processes P1, P2 that access shared binary semaphore variables S1, S2, S3. Given below is the concurrent execution of these processes:**

P_1	P_2
P(S1)	P(S2)
P(S2)	P(S3)
P(S3)	V(S2)
<Critical Section>	<Critical Section>
V(S3)	V(S3)
V(S2)	P(S1)
V(S1)	V(S1)

Assume, initially S1 = 1, S2 = 1 and S3 = 1.
Which of the following statement is true?
a) Mutual exclusion is satisfied and no deadlock.
b) Mutual exclusion is dissatisfied and deadlock.
c) Mutual exclusion is satisfied and deadlock.
d) Mutual exclusion is dissatisfied and no deadlock.

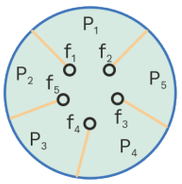
Sol: **Option: c)**
V(S3) occurs after P2 exits critical section.
Then, P1 can perform P(S3) successfully and enter the critical section.
So, mutual exclusion is satisfied.
Deadlock is present as P1 and P2 are waiting on each other to wake up/signal S3.

Q11 **Which of the following is/are a solution to the standard dining philosopher problem that avoids deadlock?**
a) Allow only 4 philosophers to dine-in
b) Odd philosopher picks up left fork first and even one picks right fork first
c) Ensure that all philosophers pick up left fork first
d) None

Sol: **Options: a), b)**

Standard dining philosophers problem have five philosophers.

a)



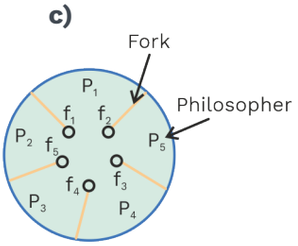
At t = 0
$P_1 \leftarrow f_2$
$P_2 \leftarrow f_1$
$P_3 \leftarrow f_5$
$P_4 \leftarrow f_4$
At t = 1,
$P_4 \leftarrow f_3$ (completes execution) releases f_3 and f_4
So, no deadlock possible.

b)

P_1 <- f_1
P_2 waits [f_5 is not available]
P_3 <- f_5
P_4 waits [f_3 is not available]
P_5 <- f_3
Now, P_5 can finish execution using f_3 and f_2 and releases f_3 and f_2 after execution.

P_3 <- f_5 and f_4
P_1 <- f_1 and f_2
P_2 <- f_5 and f_1
P_4 <- f_3 and f_4
Hence no deadlock.

c)



If all philosophers pick the left fork first, then it will lead to infinite waiting (deadlock).

Q12 **Consider the following pseudo-code for process X and Process Y. Assume the initial value of temp to be 0. Also, consider testing of temp, and it's assignment to be atomic.**

Process X:	Process Y:

while (TRUE)
{
if (temp == 1) then
sleep();
<Critical_Section>
temp = 1;
wakeup (Y);
}
	while (TRUE)
	{
	if (temp == 0) then
	sleep();
	<Critical_Section>
	temp = 0;
	wakeup (Y);
	}

Which of the following is/are TRUE?
a) Mutual exclusion is not guaranteed
b) Progress is not satisfied
c) Bounded waiting is satisfied
d) Both the processes may sleep at the same time

Sol: **Options: b), c), d)**

a) False, mutual exclusion is guaranteed.

b) True,
temp = 0 (Initially)
Let's process Y shows interest to enter CS but it cannot enter <Critical_Section> until process X sets the temp value to 1.

c) True,
However, deadlock is possible in the given code.
** Misconception: Deadlock implies no bounded wait.

Definition of bounded wait :
"There exists a bound or limit on number of times that other processes are allowed to enter <CS> after a process has made a request to enter < Critical_Section >".

d) True,
Process Y → if(temp==0) ↓
Preempt

Process X → Completely executes the code ↓
Preempt

Process Y goes to sleep

Process X → if (temp| == 1) is true and it goes to sleep

Chapter Summary

- Concurrent processes :
  - **a)** Co-operative process
  - **b)** Independent process
- Need of co-operative process :
  - **a)** Information sharing
  - **b)** Computation speed up
- Fundamental models of IPC :
  - **a)** Shared memory
  - **b)** Message passing
- Methods for implementing a link :
  - **a)** Direct or Indirect communication
  - **b)** Synchronous or asynchronous communication
  - **c)** Automatic or explicit buffering
- Critical section problem Requirements for solution to critical section problem :
  - **a)** Progress
  - **b)** Mutual exclusion
  - **c)** Bounded waiting
- Software solutions :
  - **a)** Lock variables
  - **b)** Strict alternation
  - **c)** Peterson's solution
  - **d)** Mutex locks
- Hardware solution :
  - **a)** The TSL instruction
  - **b)** Compare and swap Instruction
- Semaphores :
  - **a)** Counting semaphore
  - **b)** Binary semaphore

- Classical problems of synchronization : **a)** The bounded buffer problem
  b) Readers–writer problem
  c) The dining philosopher problem
- Monitors : **a)** One of the ways to achieve process synchronization where semaphore fails sometimes

4 Deadlock

4.1 BASICS OF DEADLOCK

In an environment of multi-programming, many processes compete for a finite number of resources. A process gets into a waiting state if the resource the process requested or needs is not available. A deadlock happens when such a process can never change its state to a ready state since the resource is held by other processes that are in a waiting state.

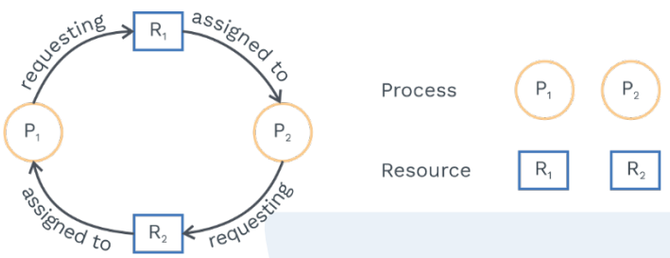


Fig. 4.1 Deadlock State

System model:

1) A system consists of a number of resources and processes.

2) The number of resources is finite and is to be shared between the processes.

3) Some resources are further divided into instances that are identical.

4) If a process requests for some type of resource any one of the instances of the same type of resource can be allocated.

Examples– CPU cycles, memory, files and I/O devices.

5) If the request is not granted, there is a problem with the definition of the resource type, and the instances are also not identical in the resource type.

Example– If there are two printers on the same floor of an office, suppose the 12th floor, employees working on the 12th floor can use any of the two printers.

6) Consider another case, if one printer is on the 12th floor and another is on the ground floor, both can't be treated equivalent and have to be defined with different resource classes.

7) A process needs to request if a resource/resources are needed to complete the assigned task. The process needs to release the same resources after its use. The number of resources a process requests is always equal or less than the total number of resources in the system.

Example– A process cannot ask for three printers if only two printers are available in the system.

8) A process under normal mode of operation utilizes a resource in the below sequence-

a) Request:
The process asks for a resource, if the resource asked for is not available at that time (maybe allocated to any other process), then the process gets into waiting state. It waits until it acquires the resource asked for.

b) Use:
The process makes use of resources to complete the task.
Example – A process uses the printer to print.

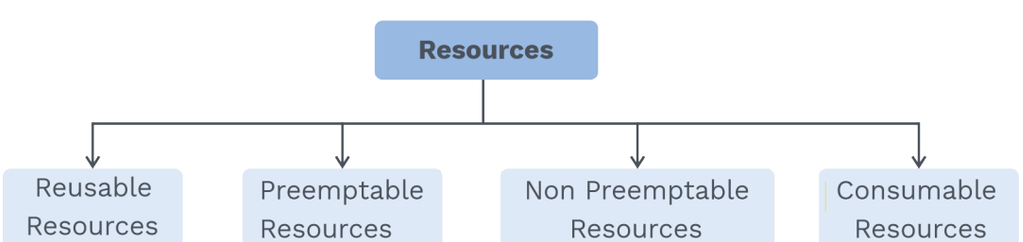
c) Release:
The process frees the resource.
The request() and release() function calls of the resource are system calls like open(), close(), allocate() and free() memory system calls.

Note:

Request and release of resources that are not managed by the operating system can be accomplished through wait() and signal() operations on semaphore or through acquisition and release of a mutex lock.

9) A set of processes are said to be in a deadlock state if every process is waiting for happening for an event which would never happen.

Types of resources:



a) Reusable resources:
It is used only by one process at a time and a does not get depleted by that use.
eg.: CPU, I/O, memory (physical/secondary).

b) Consumable resources:
It is created (produced) and destroyed (consumed).
eg.: Interrupts, signals, I/O buffers, etc.

c) Preemptable resources:
These are the resources which can be taken away from the process owning it, with no ill effect.
eg.: Memory, buffers, CPU, etc.

d) Non-preemptable resources:
These are the resources which cannot be taken away from the current owning process without causing the computation to fail.
e.g.: Tape drives, printers and optical scanners.

Deadlock characterization:
In a deadlock, a process never finishes execution and releases the resources required by another process which is in a deadlock state, thus preventing the execution of other processes.

Necessary conditions for deadlock:
Deadlock occurs if all the below four conditions occur together.

1) Mutual exclusion
No two processes share the only instance of a resource. If the resource a process asked for is with another process, asking process has to wait till the resource gets freed.

2) Hold and wait
A process holds atleast one resource before it requests for another resource which is already held by another process.

3) No pre-emption
A resource cannot be preempted from a process all of a sudden until the process itself releases it.

4) Circular wait
A set $\{P_0, P_1, \ldots\ldots\ldots, P_n\}$ of waiting processes must exist such that P_0 is waiting for a resource held by P_1, P_1 is waiting for a resource held by P_2, $\ldots\ldots\ldots\ldots$, P_{n-1} is waiting for a resource held by P_n, and P_n is waiting for a resource held by P_0.
All the four conditions must hold for a deadlock to a occur.

Note:
The circular-wait condition implies the hold and wait condition, which means four conditions are not completely independent.

Grey Matter Alert!

Suppose all the four necessary conditions be represented as (1, 2, 3 and 4). So for a deadlock to happen, The propositional Logic (1 ∧ 2 ∧ 3 ∧ 4) must be true and if ¬ (1 1 ∧ 2 1 ∧ 3 1 ∧ 4) is true, then deadlock does not exist.

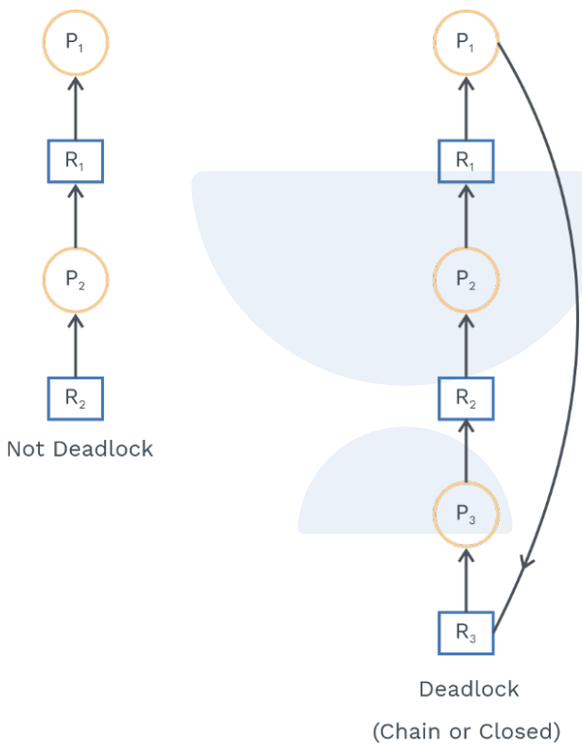


Fig. 4.2 Representation of Deadlock

4.2 RESOURCE ALLOCATION GRAPH (RAG)

1) Deadlocks can be represented more precisely using a resource allocation graph. The resource allocation graph contains two types of vertices, the active processes in the system and the resource type.

2) The graph used to represent the resource allocation graph is a directed graph. The arrow from process to resource denotes the request edge, and the arrow from resource to process denotes the assignment edge.

Fig. 4.3 Request Edge and Assignment Edge

Note:

In resource allocation graph, the process node is denoted by a circle and the resource node is denoted by a rectangle.

3) When a process P_i requests for a resource R_j, a request edge is added between the P_i and R_j; this edge is made into an assignment edge after checking if the resource R_j is free (checked by OS). The process P_i has to wait until the resource is released by the assigned process and their assignment edge is deleted.

Example:

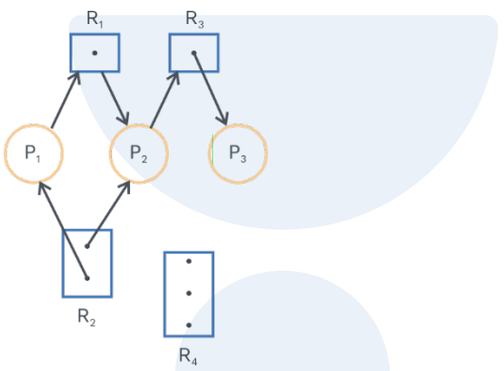


Fig. 4.4 Resource Allocation Graph

Graph elements:

1) P (Processes) = $\{P_1, P_2, P_3\}$

2) R (Resources) = $\{R_1, R_2, R_3, R_4\}$

3) E (Edges) = $\{P_1 \rightarrow R_1, P_2 \rightarrow R_3$
$R_1 \rightarrow P_2, R_2 \rightarrow P_2$
$R_2 \rightarrow P_1, R_3 \rightarrow P_3\}$

Instances of resources:

1) R_1 has one instance

2) R_2 has two instances

3) R_3 has one instance

4) R_4 has three instances

Note:

The instances of a resource are denoted by dots inside the rectangular box (resource).

Process states:

1) Process P_1 is having an instance of resource type R_2 and waiting on R_1.
2) Process P_2 is having an instance of resource type R_1 and R_2, and waiting on R_3.
3) Process P_3 is having an instance of R_3.
4) It can be shown that if a graph has no cycles, then no process is in a deadlock state, but if a graph has a cycle, then a deadlock may or may not exist.

Note:

1) Cycle in a resource allocation graph (RAG) is a necessary but not sufficient condition for the existence of deadlock in the case of multi-instance resources.
2) Cycle in a resource allocation graph (RAG) with only single instance resources is necessary as well as sufficient condition for the existence of deadlock.

Grey Matter Alert!

Sufficient and necessary:
Suppose we have two events A and B. Now "If A then B" condition is TRUE then the following two statements can be concluded.

1) A is a sufficient condition for B whenever the occurrence of A is all that is needed for the occurrence of B.
2) B is a necessary condition for A whenever A cannot occur without the occurrence of B.

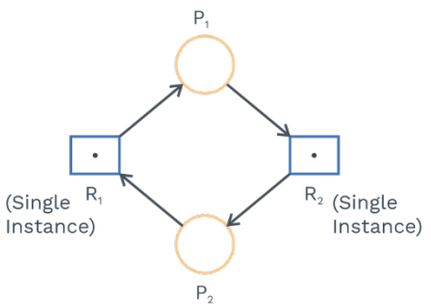


Fig. 4.5 Cycle in RAG with Single Instances

In above figure, cycle exist

$$P_1 \rightarrow R_2 \rightarrow P_2 \rightarrow R_1 \rightarrow P_1$$

So in RAG with single instance resources, we know the cycle is necessary as well as sufficient, so above RAG has a deadlock.

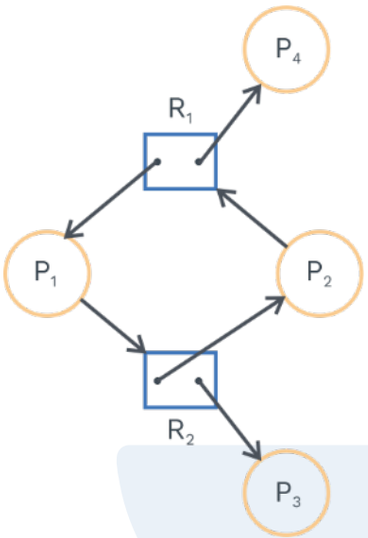


Fig. 4.6 Cycle in RAG of Multi Instance Resources (Cycle but no Deadlock).

In the above figure of resource allocation graph with multiple instances resources,

$$\text{Cycle exist : } P_1 \rightarrow R_2 \rightarrow P_2 \rightarrow R_1 \rightarrow P_1$$

So, the necessary condition is met, so deadlock may or may not occur but to confirm the existence of deadlock, we need to check whether this cycle is sufficient in the above graph or not as cycle is not sufficient in multi-instance RAG. In the figure, we can see one instance of R_1 and one instance of R_2 is allocated to P_4 and P_3, respectively. So when P_3 and P_4 are done with their work, they will release their instances, and other processes will acquire them, thus breaking the cycle.

Grey Matter Alert!

If a resource-allocation graph does not have a cycle, then the system is not in a deadlocked state. If there is a cycle, then the system may or may not be in a deadlocked state. This observation is crucial when we deal with the deadlock problem.

SOLVED EXAMPLES

Q1 **Suppose a system with only one type of resource and three processes and each process request a maximum of two instances of resource. Find, maximum number of resource instances so that system goes into deadlock. Also, find the number of resource instances so that system doesn't go into deadlock. What is the sum of both the numbers found?**

Sol: **Range: 7–7**

i) Let processes be P_1, P_2, P_3 each acquired one copy of resource each, so every process needs one copy more, but if the system has only three copies of resource, then deadlock happens.

P1 → One copy of resource
P2 → One copy of resource
P3 → One copy of resource
Three copies of resource
So maximum three copies for deadlock.

ii) Now for minimum resources for deadlock prevention, we can provide just one more copy to the system so that one process satisfies with its need and, after execution, deallocate its resource, and some other process acquires. So four copies of resources for deadlock prevention.
So, the sum of both the numbers is 3 + 4 = 7.

Q2 **Consider the system with 'N' processes and '10' tape drives (resources). If each process requires '2' tape drives to complete their execution, then what is the maximum value of 'N' which ensures deadlock-free operation.**
a) 3
b) 6
c) 9
d) 5

Sol: **Option: c)**
There are total '10' tape drives. Each process requires = '2' tape drives. First, we will allocate one resource to each process, leaving one resource aside.

P_1	P_2	P_3	P_4	P_5	P_6	P_7	P_8	P_9
1	1	1	1	1	1	1	1	1

⟶ All processes are waiting for second resource

Now add that last resource to any of the processes then it will guarantee deadlock-

free operation.

P_1	P_2	P_3	P_4	P_5	P_6	P_7	P_8	P_9
1	1	1	1	1	1	1	1	1
1								

P_1 will execute and deallocate resources after completion.

Q3 **Consider a system with five processes. Where each process requires two tape drives to complete their execution then what is the minimum number of resources required to ensure deadlock-free operation.**

a) 2

b) 4

c) 6

d) 10

Sol: **Option: c)**

Number of resources for 'n' processes for deadlock-free operation where each process requires 'k' instances of resources of resource z

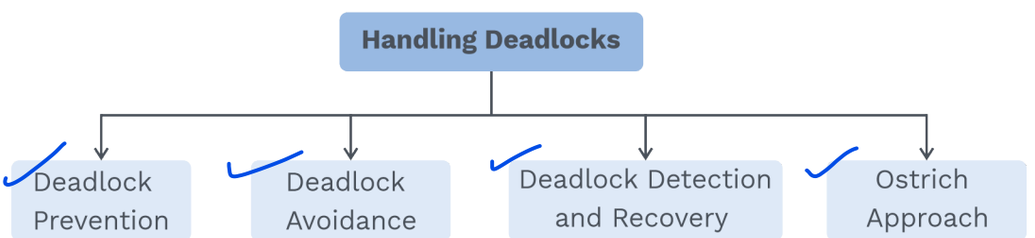
$n (k - 1) + 1$

$5 (2 - 1) + 1 = 6$ [n = 5, k = 2]

4.3 METHODS FOR HANDLING DEADLOCKS

There are many ways to solve deadlocks in your system.

1) One approach involves assuming that deadlock will never happen; this is known as ignorance (Ostrich method).
2) One approach puts attention well before occurring, which is known as prevention.
3) One approach allows the system to get into deadlock and then solves it; this is known as deadlock detection and recovery.
4) One other approach involves checking whether the system gets into deadlock a step ahead and avoiding it. This approach is named as deadlock avoidance.



If a deadlock has to occur, there are four necessary conditions that have to be satisfied. Deadlock can be removed by making sure that atleast one of these conditions doesn't hold. The four necessary conditions are:

a) Mutual exclusion:

1) Sharable resources do not get into deadlock since they need not be accessed mutually exclusive. We need atleast one resource that must be non–sharable so that mutual exclusiveness can apply.

2) Read–only files are a good example of a sharable resource, i.e., if several processes try to read the read–only file, permission is always granted.

Note:

We cannot always prevent deadlock by denying the mutual-exclusion condition because some resources are intrinsically non-sharable. For example, mutex lock.

b) Hold and wait:

The problem arises when a process is holding resources and waiting for the resources currently held by other processes.

This can be violated in three ways :

1) **Conservative approach:** The process has to acquire all the resources before starting its task. But, it is very less efficient and cannot be implemented because the resources needed cannot be known well ahead.

2) **Violating hold:** If a process holds some resources, it has to release the resources held to make a fresh request.

3) **Violating wait:** After a particular waiting period, if the process is not getting alloted the requesting resources, it has to release the currently held resources.

Examples:

1) Let's consider a process that copies data from CD–drive to a file on disk and then prints the file. If all resources must be requested at the beginning of the process, then CD–drive, disk file and printer will be allocated to the process, so it will hold the printer for the entire duration of its execution, even though it needs the printer in the end. Here, wait is not satisfying.

2) Let's consider another process, which does the same job as the above process but in a different manner as it requests initially only the CD–drive and disk file. It copies from the drive to file and then releases both the resources. Then the process request the disk file and printer to print, and when done, it again releases both resources and terminates.

c) No preemption

No resource can be preempted from the process it is alloted. To violate this, a process will release all the resources if it is not alloted the resources it has requested for. The process is restarted only when it can access both old and new resources.

Grey Matter Alert!

1) Another method could also be that if the process requests some resources, we first check whether they are available; if yes, they are allocated.

2) If resources are not available, then we check whether they are allocated to some other process that is also waiting for some resources. If yes, we preempt the resources from the waiting process to requesting process.

3) If the resources are neither held by the waiting process nor available, the requesting process must wait and while it is waiting, some of the allocated processes could also be preempted explained in the above point, if another process requests them. The process will only restart when it regains all the previously allocated resources and the ones it is waiting for, from the other process.

d) Circular wait

If process and resources dependency makes a circle resulting in deadlock, it is known as a circular wait. This can be violated by allowing a process to take resources in increasing enumeration order strictly.

1) All the resources are identified uniquely.

2) Never allow a process to request a lower number of resources than the last one requested and allocated.

Example: Let's suppose we have eight resources with resource ID as follows.

Resource	Resource ID
R_1	10
R_2	5
R_3	3
R_4	8
R_5	9
R_6	12

Resource	Resource ID
R_7	15
R_8	11

Now a process 'P' requests resources in the following sequences of resource ID – 3, 9, 10, 11, 12, 15, 8, 3. So, if circular wait does not hold, then process can request the resources in increasingly linear order. So, resource ID –

When process request resource with ID 3 again, It will not allow us to break the cycle (circular wait).

Previous Years' Question (GATE-2018)

Consider a system with three processes that share four instances of the same resource type. Each process can request a maximum of K instances. Resources can be requested and released only one at a time. The largest value of K that will always avoid deadlock is .....

Sol: 2.0

4.4 DEADLOCK AVOIDANCE

All the above-discussed approaches are used to avoid deadlock, but they bring disadvantages with them. These disadvantages include low device utilization and reduced overall system throughput.

Another approach to overcome these disadvantages needs more information about how resources are requested.

Example:

Let's consider there are two resources R1 and R2 in the system. The system needs to know the order in which the resources are held before releasing them. With this knowledge, the system can decide whether a process should wait for its resources (if they are currently available or not) at a particular time.

Grey Matter Alert!

Various algorithms that use the above approach differ in the amount and type of information needed. The most algorithm requires that each process declare the maximum number of resources of each type that it might need. Given this information, algorithm ensures that the system never enters a deadlocked state. A deadlock avoidance algorithm dynamically examines the RAG state to ensure that circular wait conditions can never exist.

Note:

The resource allocation state is defined by the number of available and allocated resources and the maximum demands of the processes.

4.5 SAFE AND UNSAFE STATES

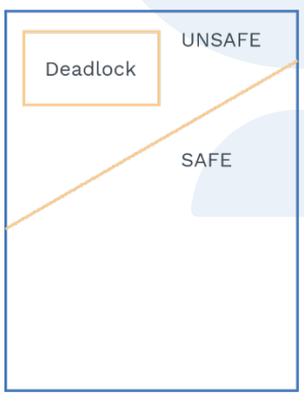


Fig. 4.7 Safe, unsafe and Deadlock State

Safe state:

Definition

There is atleast one sequence of processes that does not result in deadlock.

1) No process is deadlocked, and there exists no possible sequence of future requests in which deadlock could occur.
2) No process is deadlocked, and the current state will not lead to a deadlock state.
3) A system is in a safe state only if there exists a safe sequence.

Definition

Safe sequence: A sequence of processes <P_1,P_2,P_3, ... P"> is a safe s quence for the current allocation state if, for each P_1, the resource that P; can still request, can be satisfied by the currently available resources.

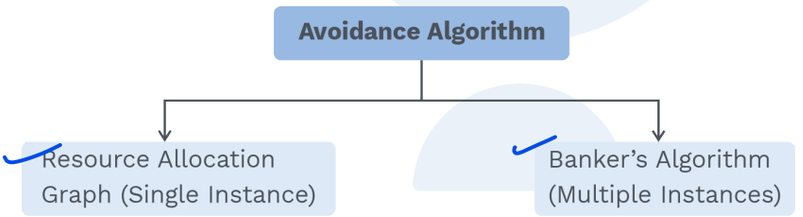
Unsafe state:

It is not a safe state. There is a possibility of deadlock with some sequence.

Note:

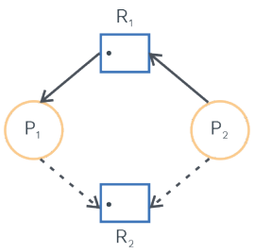
1) If a system is in a safe state, then the system has no deadlock.
2) If a system is in a unsafe state, then there is a possibility of deadlock.
3) Do not start a process if its maximum requirement might lead to deadlock.
4) Deadlock avoidance ensures that a system will never enter

Deadlock avoidance algorithm:



Resource allocation graph avoidance:

1) This algorithm can be used if we have only one instance of each resource type.
2) **Claim edge:** An edge P---->R represents that the process may request the resource R in the near future.
3) **Assignment edge:** When the process requests the resource then claim edge is changed to assignment edge i.e., P->R
   The assignment edge is changed to the claim edge right after the process releases the resource.
4) Resources have to be claimed well before the process starts executing, and the respective claim edges have to appear in the resource allocation graph.
5) When a process P_i requests a resource R_j, the request should only be granted if the conversion of the request edge $P_i \rightarrow R_j$ to an assignment edge wouldn't result in a cycle in RAG.
   For that, when a cycle detection algorithm is used on RAG if no cycle exists, then the allocation of the requested resource will leave the system in a safe state. If a cycle is found, then the process P_i will have to wait.



Resource-allocation graph for deadlock avoidance.

$R_1 \rightarrow P_1$ [Assignment edge]
$P_2 \rightarrow R_1$ [Request edge]

$P_1 \rightarrow R_2$
$P_2 \rightarrow R_2$ [Claim edges]

Banker's algorithm:

1) It is used for a system with multiple instances of resources.
2) Each process has to claim the resources required.
3) If a process is allocated all the resources it requested, it has to complete its task and release them in a finite time.

Note:

Banker's algorithm runs each time

- A process requests resource: Is it safe?
- A process terminates: Can I allocate released resources to a suspended process waiting for them.

A new state is safe if and only if every process can complete after allocation is made. Make allocation and then check system state and deallocate if unsafe.

Banker's algorithm implementation:

1) Let 'n' is the number of processes and 'm' is the number of resource types.
2) Available: If available [j] = K, it means there are K instances of resource type R_j available.
3) Max: If Max [i, j] = K means that process P_i may request atmost K instances of R_j.
4) Allocation: If allocation [i, j] = K then P_i is currently allocated K instances of R_j.
5) **Need:** If need [i, j] = K, then it depicts that P_i may need K more instances of R_j to complete its task.
6) Need [i, j] = Max [i, j] – Allocation [i, j].

Resource request algorithm working (banker's algorithm):

Assume $request_i$ be the list of resources a process P_i requested and $request_i[j]=k$ means process P_i needs k instances of R_j.

1) Check if the sum of values in $Request_i \leq$ the sum of values in $Need_i$, continue if true, else an error is raised since the requested number of resources exceeded the number of needed (claimed before).

2) Check if the sum of values in $request_i \leq$ the sum of values in available, continue if true, else the process must wait as the resources are not available.

3) Algorithm must calculate the below process states by updating values according to
Available = Available – $Request_i$;
$Allocation_i = Allocation_i + Request_i$;
$Need_i = Need_i – Request_i$;
If the resulting state is safe, i.e., if a process can be allocated with resources required, then the process is allocated resources in that order.
If the resulting state is unsafe, the process has to wait until the resources in the old state are restored.

Previous Years' Question (GATE-2007)

A single processor system has three resource types X, Y and Z, which are shared by three processes. There are five units of each resource type. Consider the following scenario, where the column "alloc" denotes the number of units of each resource type allocated to each process, and the column "request" denotes the number of units of each resource type requested by a process in order to complete execution. Which of these processes will finish LAST?

	alloc			request		
	X	Y	Z	X	Y	Z
P0	1	2	1	1	0	3
P1	2	0	1	0	1	2
P2	2	2	1	1	2	0

a) P0
b) P1
c) P3
d) None of the above, since the system is in a deadlock

Sol: c)

SOLVED EXAMPLES

Q4 **The following system state given defines how four (A,B,C,D) types of resources are allocated to five running processes.**

$$\text{Available} = \begin{matrix} A & B & C & D \\ \{2, & 1, & 0, & 0\} \end{matrix}$$

$$\text{Allocation} = \begin{matrix} & A & B & C & D \\ P_0 & 0 & 0 & 1 & 2 \\ P_1 & 2 & 0 & 0 & 0 \\ P_2 & 0 & 0 & 3 & 4 \\ P_3 & 2 & 3 & 5 & 4 \\ P_4 & 0 & 3 & 3 & 2 \end{matrix}$$

$$\text{Max} = \begin{matrix} & A & B & C & D \\ P_0 & 0 & 0 & 1 & 2 \\ P_1 & 2 & 7 & 5 & 0 \\ P_2 & 6 & 6 & 5 & 6 \\ P_3 & 4 & 3 & 5 & 6 \\ P_4 & 0 & 6 & 5 & 2 \end{matrix}$$

Compute the need matrix and determine whether the system is in a safe state? Answer 1 if any safe state exists, else answer 0.

Sol: **Range: 1–1**

Need = Max – Allocation.

$$\text{Need} = \begin{matrix} & A & B & C & D \\ P_0 & 0 & 0 & 0 & 0 \\ P_1 & 0 & 7 & 5 & 0 \\ P_2 & 6 & 6 & 2 & 2 \\ P_3 & 2 & 0 & 0 & 2 \\ P_4 & 0 & 3 & 2 & 0 \end{matrix}$$

As we can see, P_0 doesn't need additional resources in the above need matrix, so after P_0 done its execution.

Available = {2, 1, 1, 2}

So now, P_3 can acquire A and D resources available to complete its execution.

Similarly P_4, P_1 and P_2 also get completed in that sequence.

Safe sequence: $<P_0, P_3, P_4, P_1, P_2>$

So, answer is 1.

Previous Years' Question (GATE-2010)

A system has n resources R_0, ... Rn_{n-1}, and k processes P_0, ..., Pk_{k-1},. The
implementation of the resource request logic of each process P_i is as follows:

if(i%2 == 0) {
if(i < n) request R_i;
if(i + 2 < n) request R_{i+2};}
else{
if(i < n) request R_{n-1};
if(i + 2 < n) request R_{n-i-2};}

In which of the following situations is a deadlock possible?

a) n = 40, k = 26
b) n = 21, k = 12
c) n = 20, k = 10
d) n = 41, k = 19

Sol: b)

Deadlock detection and recovery

If the system did not employ both the deadlock prevention and avoidance algorithms, then a deadlock may happen. Therefore the system should provide the following:

1) An algorithm to detect the deadlock.
2) An algorithm to recover it.

Note:

The detection and recovery scheme requires overhead that includes not only the run-time costs of maintaining the necessary information and executing the detection algorithm but also losses from recovering from a deadlock.

SOLVED EXAMPLES

Q5 **There are four processes and three types of resources. The number of copies of each resource type are : (r1, r2, r3) = (3, 2, 2).**
Given table shows the process ids, number of allotted resources of each type for every process and the resource requests for every process.

Process id	Allotted			Request		
	r1	r2	r3	r1	r2	r3
1	1	0	0	0	1	2
2	0	0	1	1	2	1
3	0	1	0	1	0	1
4	1	0	0	0	2	2

Which of the following is a safe sequence?
a) P4, P2, P3, P1
b) P1, P2, P4, P3
c) P2, P1, P4, P3
d) P3, P2, P1, P4

Sol: **Option: d)**

Process id	Allotted r1 r2 r3	Request r1 r2 r3	Available r1 r2 r3
1	1 0 0	0 1 2	1 1 1 1 2 1 1 2 2 2 2 2 3 2 2
2	0 0 1	1 2 1	
3	0 1 0	1 0 1	
4	1 0 0	0 2 2	

Therefore the following is a safe sequence:
P3 –> P2 –> P1 –> P4

Q6 **The system has five process and three resources (A B C). The maximum count of resources are (10 5 7). Consider the following table of resources allocation.**

	MAX			Allocated		
	(A	B	C)	(A	B	C)
P_0	7	5	3	0	1	0
P_1	3	2	2	2	0	0
P_2	9	0	2	3	0	2
P_3	2	2	2	2	1	1
P_4	4	3	3	0	0	2

What will be a safe sequence?

a) P_1, P_2, P_3, P_4, P_o

b) P_1, P_o, P_3, P_4, P_2

c) P_1, P_3, P_4, P_0, P_2

d) Unsafe Sequence

Sol: **Option: c)**

	MAX Allocation			Allocated			Current Need		
	A	B	C	A	B	C	A	B	C
P_0	7	5	3	0	1	0	7	4	3
P_1	3	2	2	2	0	0	1	2	2
P_2	9	0	2	3	0	2	6	0	0
P_3	2	2	2	2	1	1	0	1	1
P_4	4	3	3	0	0	2	4	3	1

After P_1 available resources are (5, 3, 2).
Now, P_3 and P_4 both can be served. After serving both available resources (7,4,5).
Now, similarly, P_0 and P_2 can be served.
So possible safe sequence is an option (C) which is P_1, P_3, P_4, P_0, P_2

Deadlock detection:

If all resources are with only one instance, then for deadlock detection, a variation of resource–allocation graph named as wait–for graph can be obtained from the RAG by removing the resource nodes and corresponding edges.

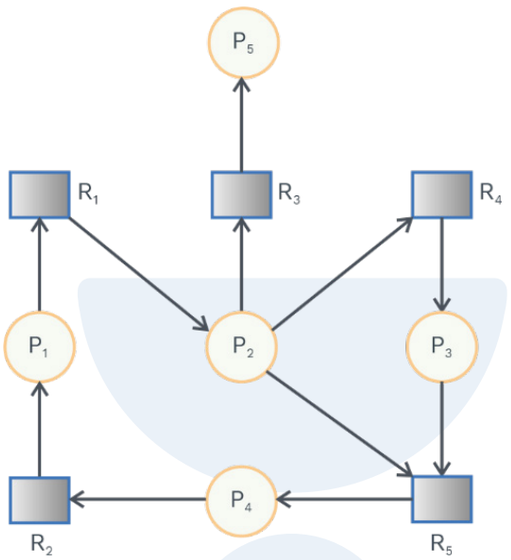


Fig. 4.8 Resource Allocation Graph

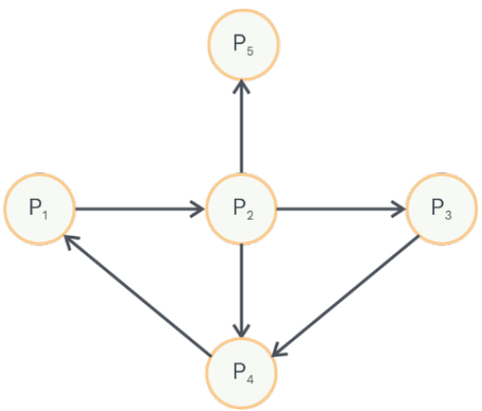


Fig. 4.9 Corresponding Wait–For Graph

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

Note:

Wait-for graph is a graph sharing the dependencies between processes waiting for one another.

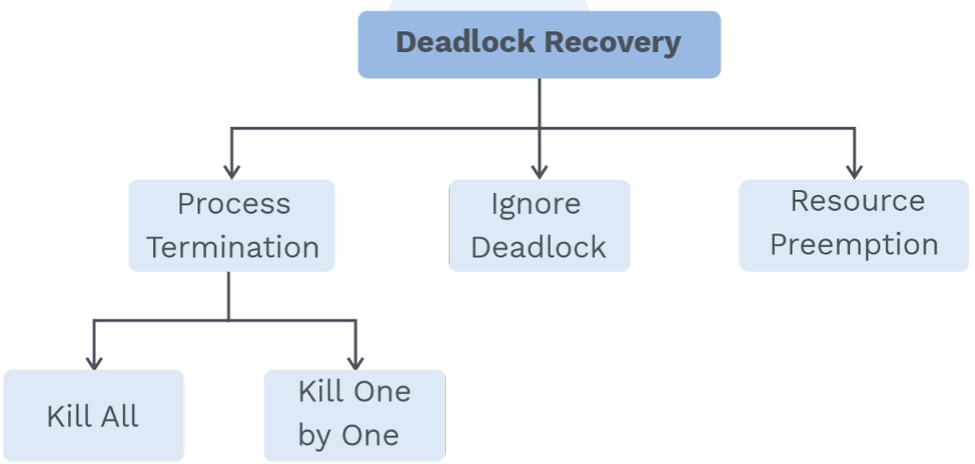
The system needs to maintain the wait–for graph and periodically calls an algorithm that searches for a cycle.

Rack Your Brain

- What would be the time complexity of banker's algorithm?
- What would be the time complexity to detect a cycle in a graph of 'n' vertices?

If resources have multiple instances, then for deadlock detection, variation of banker's algorithm is used; we satisfy the process request of resource one by one if it leads to deadlock, we roll back else find a safe sequence.

Deadlock recovery:



a) Process termination:

To eliminate deadlock by killing a process, we use two methods.

1) **Abort all deadlock processes:**
   All the process gets killed, and this method will clearly break from the deadlock cycle.
2) **Abort one process at a time:** until the deadlock cycle is eliminated.

Note:

Aborting a process may not be easy if the process was in a middle of an important task like printing or updating a database.

Killing processes one by one is more useful than killing every process in the system. Various factors which decide which process is chosen to be killed are:

1) What is the current priority of the process?
2) How long the process has already been executed?
3) How many resources are still required for the process to get completed?
4) What is the nature of the process, i.e., whether it is an interactive or batch process?

b) Resource preemption

Definition

To eliminate deadlocks using resource preemption, we successively preempt some resources from the process and given these to other processes until the deadlock is broken.

Various issues while preempting resources to deal with deadlocks,

1) Victim selection:
Which resources and processes are to be preempted? So that cost will be minimum, such as a number of resources a particular process is holding.

2) Rollback:
Now after preemption, we have to determine whether to continue the preempted process execution or rollback it to some safe state.

3) Starvation:
In a system where a selection of victim is based on the cost factor, it may very well happen that the same process get victimized everytime, so starvation occurs for some process in the system, so we have to put an upper bound for a process for preemption.

c) Ignore the deadlock:
It is also known as ostrich algorithm where the system ignores the deadlock and acts like the deadlock never happened. It's widely regarded as the most effective way to deal with a deadlock.

SOLVED EXAMPLES

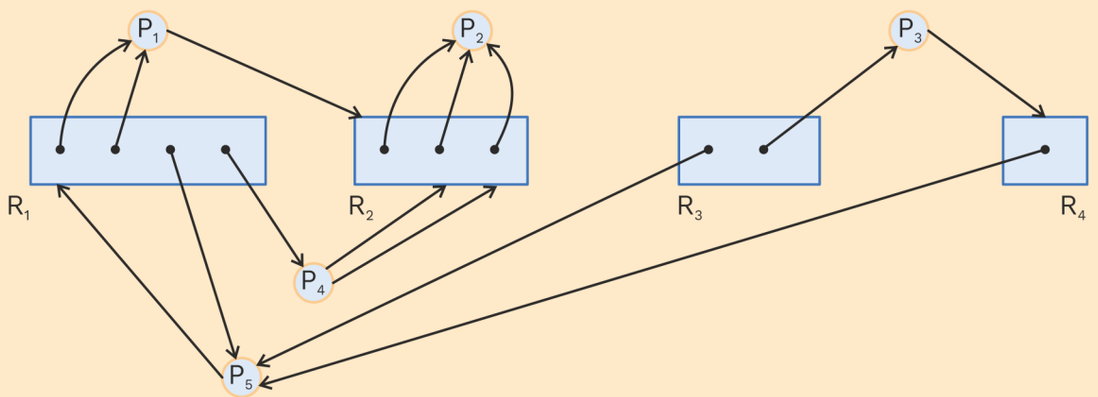
Q7 **Consider the following statements**
S1: Deadlock avoidance is seldom used as a practical solution to the deadlock problem
S2: Banker's algorithm is sufficient for a practical system.
a) S1 is true, S2 is false
b) S1 is false, S2 are true
c) Both S1 and S2 are false
d) Both S1 and S2 are true

Sol: **Option: a)**
S1 is true: Deadlock avoidance needs lots of overhead and prior information on resources need a program.
S2 is false: Banker's algorithm has certain limitations in its implementation. It should know the number of instances of each resource a process will request. In most of the current systems, it is impossible to have such information prior.

Q8 **Which of the following statements is/are incorrect?**
a) Processes may lead to deadlock as a result of contending for the processor.
b) Only hardware resources can be non-preemptible.
c) An unsafe state is a deadlocked state.
d) A system cannot transition from an unsafe state to a safe state.

Sol: **Options: b), c), d)**
a) TRUE. The processor is a preemptible resource
b) FALSE. Some software resources are also non-preemptible like monitors.
C,D) Both FALSE. When processes release resources, the number of available resources can be enough for the state of the system to make a transition from unsafe to safe.

Q9 **Consider the following resource allocation graph with resources R_1, R_2z, R_3, R_4 with 4, 3, 2, 1 instances of the same resource, respectively.**



Which of the following execution sequences of processes is/are safe?

a) <P2$_2$, P_1, P_4, P_5, P_3>

b) <P_2, P_4, P_1, P_5, P_3>

c) <P_2, P_1, P_5, P_3, P_4>

d) <P_2, P_4, P_5, P_1, P_3>

Sol: **Options: a), b), c), d)**

From the given resource allocation graph, current allocation and need can be seen as follows:

Resources (Instances)	Allocated					Need				
	P_1	P_2	P_3	P_4	P_5	P_1	P_2	P_3	P_4	P_5
R_1(4)	2	0	0	1	1	0	0	0	0	1
R_2(3)	0	3	0	0	0	1	0	0	2	0
R_3(2)	0	0	1	0	1	0	0	0	0	0
R_4(1)	0	0	0	0	1	0	0	1	0	0

Currently available resource instances can be seen as <R_1, R_2, R_3, R_4> = <0, 0, 0, 0>, i.e., no resource is free. From the above table, it is clear that only P_2 can complete its execution as it doesn't need any resources.

Post P_2 completion resources available are <R_1, R_2, R_3, R_4> = <0, 3, 0, 0>, with 3 instances of R_2, either P_1 or P_4 can complete. Among P_3 and P_5, P_5 can complete its execution first as resource needs of P_5 are available post P_1 or P_4 completion. P_3 is waiting for one instance of resource R_4 which P_5 is holding, so P_3 will have to wait till P_5 completes and frees one instance of resource R_4.
So, all sequences are safe.

Q10 **Consider an operating system which implements Banker's Algorithm to avoid deadlock. Currently allocation table with three resources A, B and C to five processes P_0, P_1, P_2, P_3, P_4 can be seen below:**

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

REQ 1: Process P_1 asks for one additional instance of resource A and two instances of resource C.

REQ 2: Process P_2 asks for two additional instance of resource A.

a) Only REQ 1 can be permitted

b) Only REQ 2 can be permitted

c) REQ 1 and REQ 2 both can be permitted

d) Neither REQ 1 nor REQ 2 can be permitted

Sol: **Option: c)**
REQ 1 (P1) <1, 0, 2> <= Available <3, 3, 2>
This request can be fulfilled, and the new state is as following:

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

Now, by using safety algorithm

1) Need (P1) <0, 2, 0> is less than available <2, 3, 0> P1 gets the resources and after completion free the resources, updates available <5, 3, 2>

2) Need (P3) <01, 1> is less than available (5, 3, 2) P3 gets the resources and after completion free the resources, updates available (7, 4, 3)

3) Need (P4) <4, 3, 1> is less than available <7, 4, 3> P4 gets the resources and after completion free the resources, updates available <7, 4, 5>

4) Need (P2) <6, 0, 0> is less than available <7, 4, 5> P2 gets the resources and after completion free the resources, updates available <10, 4, 7>

5) Need (P0) <7, 4, 3> is less than available <10, 4, 7> P0 gets the resources and completes execution

6) All processes get completed, so REQ1 can be permitted. Safe sequence is <P1, P3, P4, P2, P0>

REQ 2 is in continuation after grant of REQ 1.
REQ2 (P2) <2, 0, 0> <= Available <2, 3, 0>.
This request can be fulfilled, and the new state is as following:

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

Now, by using safety algorithm,

1) Need (P1) <0, 2, 0> is less than available <0, 3, 0> P1 gets the resources completes execution and frees resources, updates available <3, 3, 2>
2) Need (P3) <0, 1, 1> is less than available <3, 3, 2> P3 gets the resources completes execution and frees resources, updates available <5, 4, 3>
3) Need (P4) <4, 3, 1> is less than available <5, 4, 3> P4 gets the resources completes execution and frees resources, updates available <5, 4, 5>
4) Need (P2) <4, 0, 0> is less than available <5, 4, 5> frees resources after completion and updates available <10, 4, 7>
5) Need (P0) <7, 4, 3> is less than available <10, 4, 7> P0 gets resources and completes its execution
6) REQ 2 can be permitted, safe sequence is <P1, P3, P4, P2, P0>

Q11 **CSuppose a system that uses banker's algorithm to deal with deadlocks. Currently allocation table with three resources A, B and C to five processes PO, P1, P2, P3, P4 can be seen below:**

Processes	Maximum			Allocated			Available		
	A	B	C	A	B	C	A	B	C
P_1	6	5	4	0	3	4	4	3	1
P_2	3	4	2	2	1	2			
P_3	1	0	4	0	0	2			
P_4	3	2	5	1	2	1			

Which of the following is invalid safe sequence(s)?

a) P_2, P_1, P_3, P_4 **b) P_2, P_4, P_1, P_3** **c) P_2, P_3, P_4, P_1** **d) P_2, P_1, P_4, P_3**

Sol: **Option: b)**

Need(P_2) = MAXIMUM(P_2) – ALLOCATED (P_2)
= (3 4 2) – (2 1 2) = (1 3 0)
CURRENT AVAILABLE >= Need(P_2)
[(4 3 1) >= (1 3 0)], so, P2 can execute.

After P2's execution current available = (4 3 1) + (2 1 2) = (6 4 3)

Need(P_1) = MAXIMUM (P_1) – ALLOCATED (P_1) = (6 5 4) – (0 3 4) = (6 2 0)

CURRENT AVAILABLE >= Need (P_1)
[(6 4 3) >= (6 2 0)], so, P1 can execute.

After, P_1's execution, Current Available = (6 4 3) + (0 3 4) = (6 7 7)
(6 7 7) is sufficient to process P3{NEED: (1 0 2)} and P4{NEED: (2 0 4)} in any order.
So, (A)[P2 P1 P3 P4] and (D) [P2 P1 P4 P3] are valid safe sequences.

(B) P_2 P_4 P_1 P_3
Similarly, Need(P_2) = (1 3 0) which can be satisfied by current available (4 3 1).
After P_2's execution:
Current Available = (4 3 1) + (2 1 2) = (6 4 3)
Need(P_4) = (2 0 4) which cannot be satisfied by current available (6 4 3)
Hence, (B) is an invalid safe sequence

(C) P_2 P_3 P_4 P_1
Need (P_2) = (1 3 0) which can be satisfied by current available (4 3 1)
After P_2's execution:
Current available = (6 4 3).
Need (P_3) = (1 0 2) can be satisfied by current available
After P_2's execution:
Current available = (6 4 5)
Need (P_4) = (2 0 4) can be satisfied by current available.
After P_4's sufficient to satisfy need of P_1 (6 2 0)
So, (C) is valid safe sequence.

Chapter Summary

1) Basics of deadlock – Process waiting for a resource which it won't get immediately and have a circular dependency of several processes to each other.

2) Types of resources –
   1) Reusable resources
   2) Preemptable resources
   3) Non-preemptable resources
   4) Consumable resources

3) Necessary condition for deadlock –
   1) Mutual exclusion
   2) Hold and wait
   3) No preemption
   4) Circular wait

4) Resource allocation graph – Graphical representation of a relationship between process and resources.

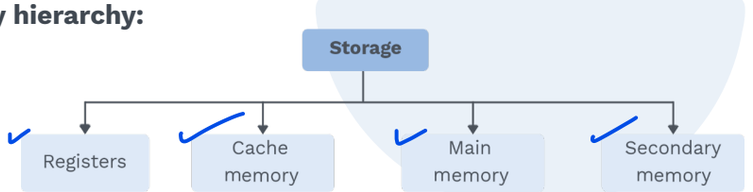
5) Handling of deadlock –
   1) Deadlock prevention
   2) Deadlock avoidance
   3) Deadlock detection and recovery
   4) Ostrich approach

5 Memory Management

5.1 BASICS OF MEMORY MANAGEMENT

- The main purpose of a computer system is to execute programs. These programs, along with their data, must at least be partially present in the main memory during execution.
- To greatly enhance the utilisation of the CPU and its response time to users, many programs exist in the main memory. Thus various memory management methods are present to manage memory in different situations.
- Memory Management provides the functionality of allocating and de-allocating the main memory to the processes. It helps the operating system to keep a record of all the memory locations which are allocated to any process or are freely available.

Memory hierarchy:



Different places at which data is stored in a computer system in a hierarchical manner.

1) Registers:

- A CPU register is one of a small number of data storage areas found within the computer processor.
- Registers are a type of computer memory that is used to execute programs quickly and efficiently by storing data that is often used. The sole function of a register is to allow for quick retrieval of data for processing.

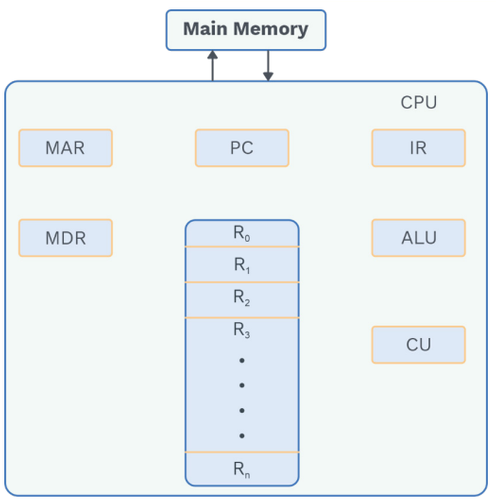
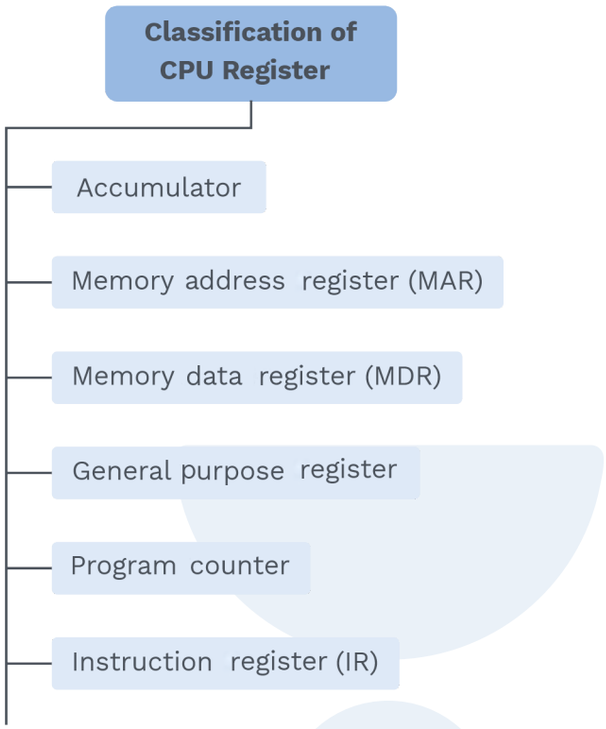


Fig. 5.1 Different Types of Register



- The CPU has direct access to registers; it can retrieve data from registers in a single clock cycle.

2) Cache memory:

- Cache memory is a volatile computer memory that saves frequently used applications and enables high-speed data access to a CPU.
- Cache memory is more expensive than main memory or disc memory, but it is less expensive than registers. It serves as a buffer between the processor and the main memory.
- Cache memory is used to lower the average time to access data from the main memory by storing copies of data from frequently accessed main memory locations. It is smaller and faster than the main memory.

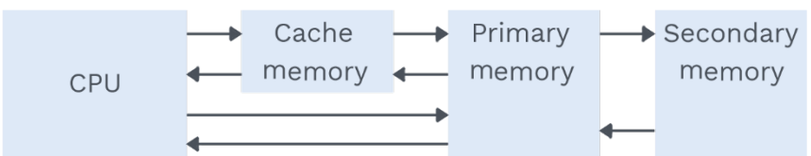


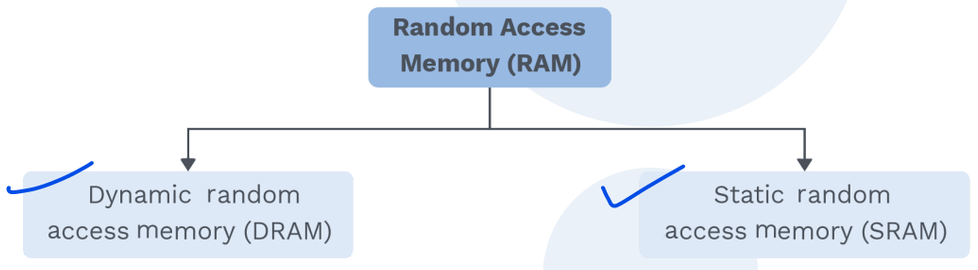
Fig. 5.2 Levels of memory hierarchy

Note:
Data transfer is in words between CPU and cache memory and in block between cache and main memory.

3) Main memory:

It's also known as read–write memory, primary memory, or main memory.

- Because the data is lost when the power is switched off, it is a volatile memory.
- Random Access Memory is another name for it (RAM). It is the portion of the computer that contains the operating system, software applications, and other data for the processor. The term "random access" refers to the fact that the CPU can go to any portion of the main memory without having to proceed in sequence.



Note:
Data transfer between main memory and secondary memory is done by the operating system.

i) DRAM:

The most prevalent type of main memory in a computer is DRAM. In PCs, it is a common memory source. DRAM is continually recovering whatever data is currently stored in memory. It sends millions of pulses per second to the memory cells to refresh the data.

ii) SRAM:

It's a popular choice for embedded devices. SRAM data does not need to be refreshed on a regular basis. When the power is turned off, the information in this RAM stays as a static image until it is overwritten or destroyed. When not in use, it is less dense and more energy efficient.

4) Secondary memory:

- It is non–volatile and persistent computer memory that cannot be accessed directly by a computer.
- Primary memory has limited storage capacity and is volatile; this limitation is overcome by secondary memory.
- It is slower in data accessing. Typically main memory is at least six times faster than the secondary memory.

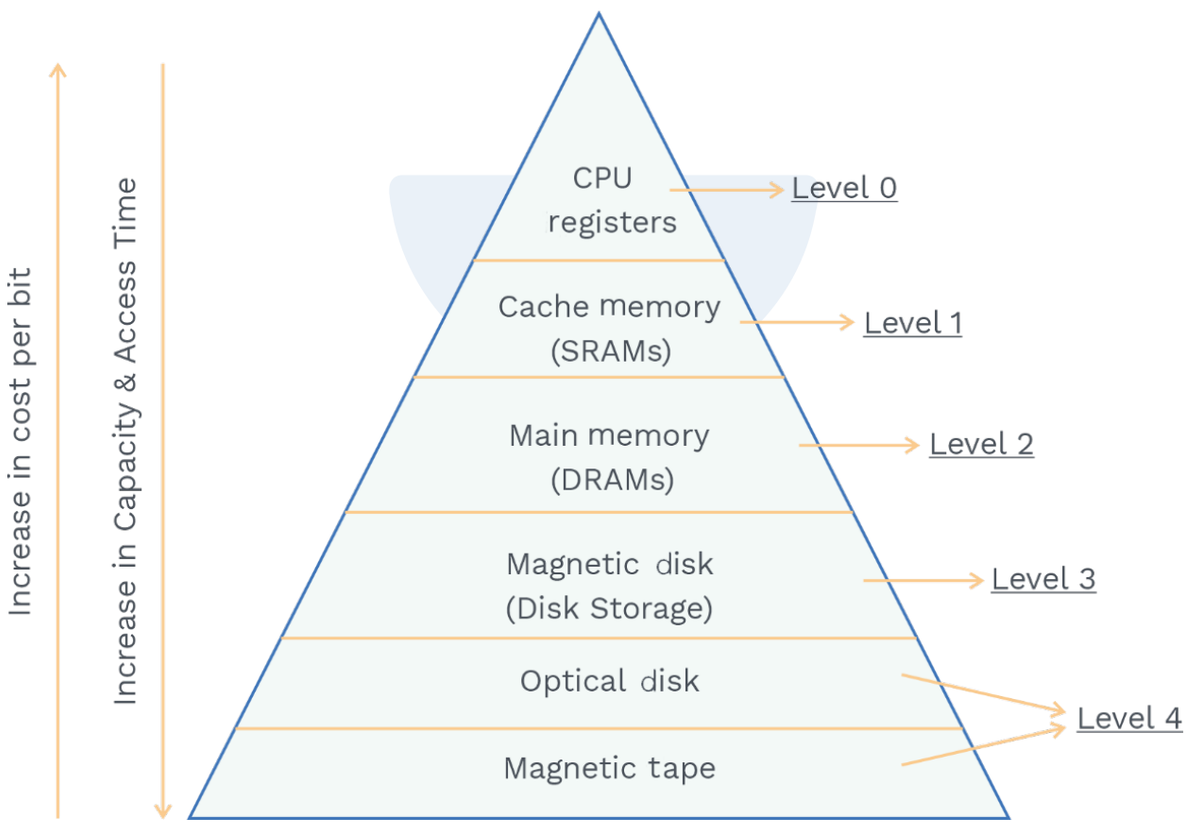


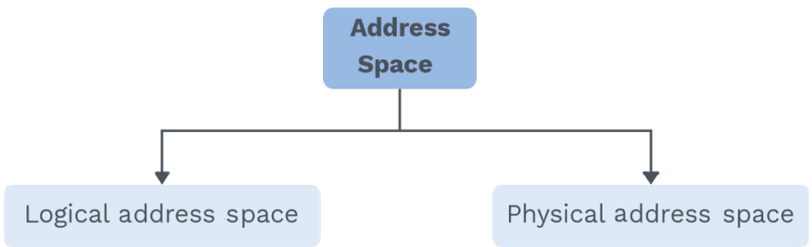
Fig. 5.3 Memory Hierarchy Design

Address space:

- An address space is the collection of all the addresses that are allocated to the program for its storage and execution. The memory locations in the address space can be accessed by the programs or the processes.

Classification of address space:

- Address space can be logical address space for storing the program or can be physical address space for executing the program.

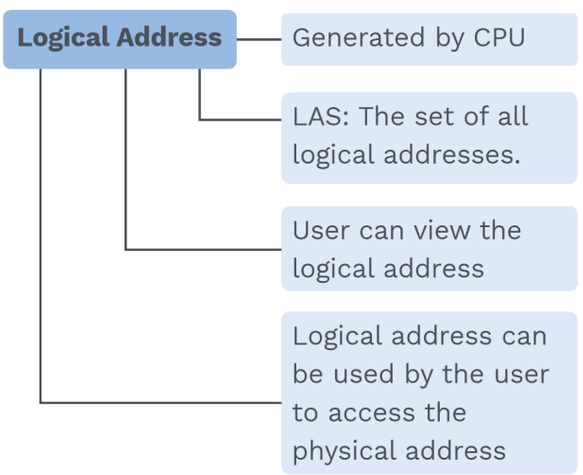


1) Logical address space:

- CPU generates logical addresses.
- It is used by the CPU to access the physical memory location as a reference.
- The set of all logical addresses generated from a program's point of view is known as the logical address space.

2) Physical address space:

- In memory, a physical address represents the physical location of requested data. The user never interacts with the physical address directly but can access it via its logical address.
- The logical address is generated by the user software.
- The hardware mechanism that converts a logical address to its physical address is the Memory-Management Unit (MMU).
- MMU must first map the logical address to the physical address.



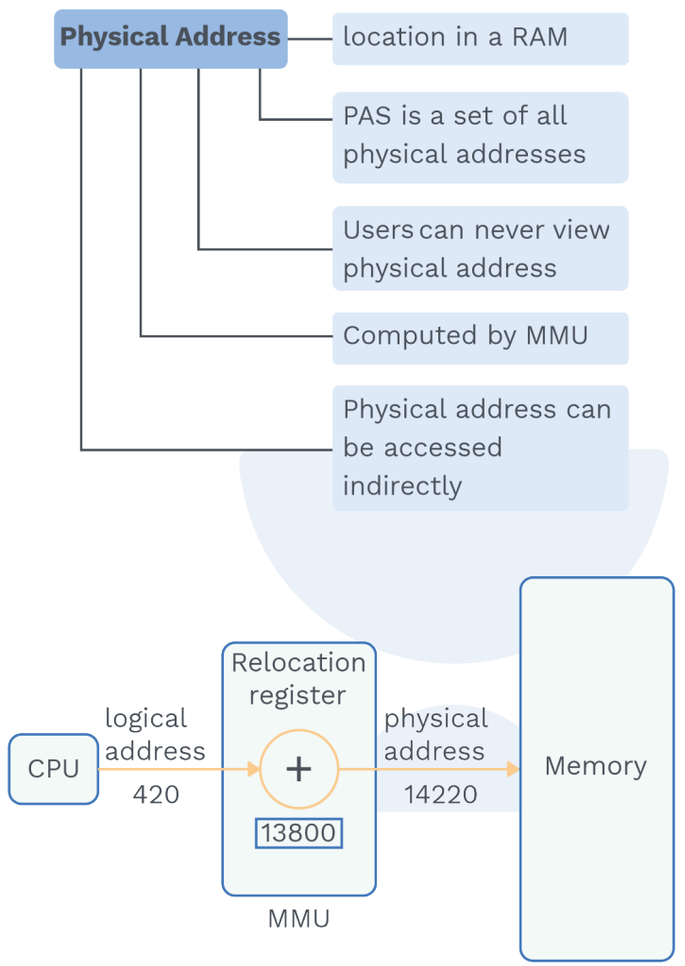


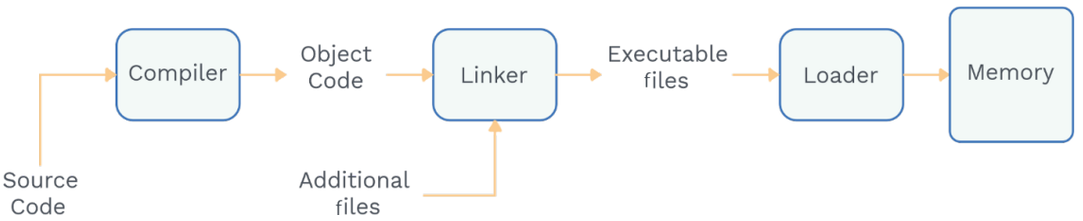
Fig. 5.4 Translation of Logical Address into Physical Address

Linker and loader:

Definition

The utility programs Linker and Loader play an important part in the execution of a program. Before being executed, a program's source code passes via the compiler, assembler, linker, and loader in that order.

Linking and loading:



1) Linking:

A program has multiple modules in it, which we compile together in order to generate a single executable code that can be loaded into the main memory for its execution. There are some codes which are reused, again and again; they are known as library functions. All different object files and library functions are combined into a single executable code; this process is known as linking. It can be done in two ways:

i) Static linking:

- Static linking is done during the compile time. In static linking, all the object modules and the library functions are combined in an executable file.
- Static linking is performed by special programs called linkers.
- Linker appends, all the library functions needed to execute the object file, which is generated by the compiler/assembler.

Advantage:

Programs in which all the modules and library functions are linked statically are faster than the programs in which all the modules and library functions are linked during run time.

Disadvantage:

- Executable file becomes very large, so loading time becomes more.
- If any library is replaced by a new version, all the programs must be re-compiled and re-linked to the new library version.

ii) Dynamic linking:

- Dynamic linking is a mechanism for linking a library into memory at runtime, retrieving variable and function addresses, executing the functions, and unloading the program from memory.
- It's frequently used to create software plugins. This method is used by Apache Web Server to load a dynamic shared object (*.dso) files at runtime.

Advantage:

- If any library is replaced by a new version, all the programs that reference the library will automatically use the new version without the need for recompilation.

Disadvantage:

- Program startup time is slower because during the execution, it checks all linked files if they are in main memory or not.

2) Loading:

- Initially executable code of every program is present in the secondary storage. But for the execution of the executable file, it must be present in the main memory.
- Bringing the code from secondary memory and storing it in the main memory is known as loading.
- It is done by the loader.
- Loader is a special program that takes the executable code present in the secondary memory, which is generated by the linker, and loads it into main memory. It can be done in two ways:

i) Static loading:

- Static loading refers to loading the entire executable file in the main memory before its execution starts.

Advantage:

- Program execution will be fast.

Disadvantage:

- Inefficient utilisation of main memory because even if the entire code need not be executed at once, loader still loads the whole code in main memory, thus occupying the main memory unnecessarily.
- Also, the size of the process which can be loaded into the main memory is limited by the size of the main memory. Thus, any process whose size is more than the size of the main memory cannot be executed.

ii) Dynamic loading:

- A routine in dynamic loading is not loaded until it is called.
- The main program is loaded into the memory and gets executed; other routines are kept on the disk in a relocatable format.
- When a routine needs to call another routine, the calling routine first checks whether the other routine has been loaded.

- If it is not, the relocatable linking loader is called to load the desired routine into the memory.
- This type of loading is useful when the large number of codes are needed to handle infrequently occurring cases, such as error routines.

Advantage:

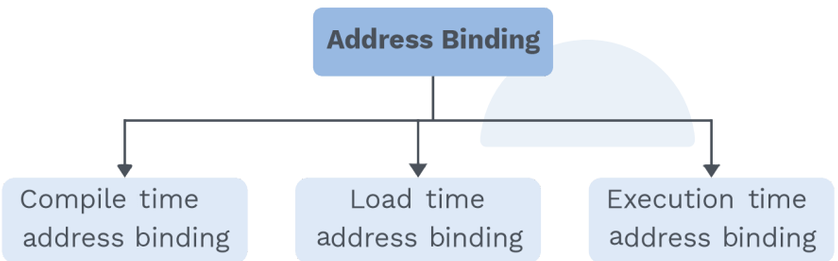
- Unused routine is never loaded into the main memory; thus main memory can be efficiently used.

Disadvantage:

- Program execution will be slower.

Address binding:

- Address Binding is the process of connecting application program instructions and data into physical memory locations.
- It refers to the process of converting one address space to another.
- The physical address relates to the location in the memory unit, but the logical address is generated by the CPU during execution.



1) Compile time address binding:

- If you know where the process will live in memory during compile time, an absolute address is generated, i.e. the physical address is given to the program's executable fire during compilation.
- Address binding is the responsibility of the compiler.
- It will be completed before the application is loaded into memory.
- To achieve compile–time address binding, the compiler must communicate with an OS memory management.

2) Load time address binding:

- If the location of the process is unknown at compile-time, a relocatable address will be produced. The relocatable address is converted to an absolute address by the loader.
- The loader generates an absolute address by adding the process's base address in the main memory to all logical addresses.
- It will be completed after the application has been loaded into memory.
- The OS memory manager, or loader, will handle this form of address binding.

Note:

If the process's base address changes during load time, address binding, the process must be reloaded.

3) Execution time/dynamic address binding:

- The CPU executes instructions of a process which are stored in memory.
- If a process can be relocated from one memory location to another during execution, this is employed.
- Even after the program has been loaded into memory, the address binding will be delayed. Until the program is finished, the application program will keep modifying the memory locations.
- The CPU does this type of address binding during program execution.
- It's compatible with dynamic absolute addresses.

Note:

Hardware support for address mappings, such as base and limit registers are required for dynamic address binding.

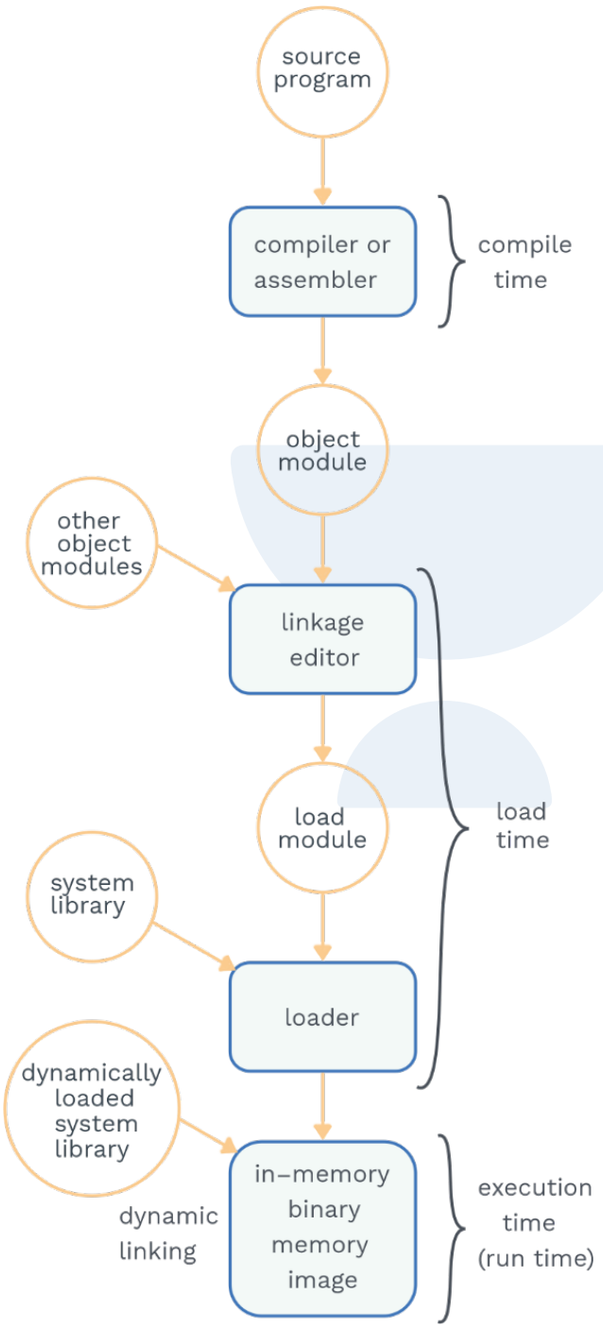


Fig. 5.5 Multistep Processing of a User Program

Swapping:

- A process must be in memory to be executed however, it can be temporarily moved to a backing store/secondary memory/nonvolatile storage, and then returned to main memory for further execution.
- Swapping allows the entire physical address space of all processes to surpass the system's actual physical (RAM) memory.
- Swapping is another technique for increasing the degree of multiprogramming in a system.

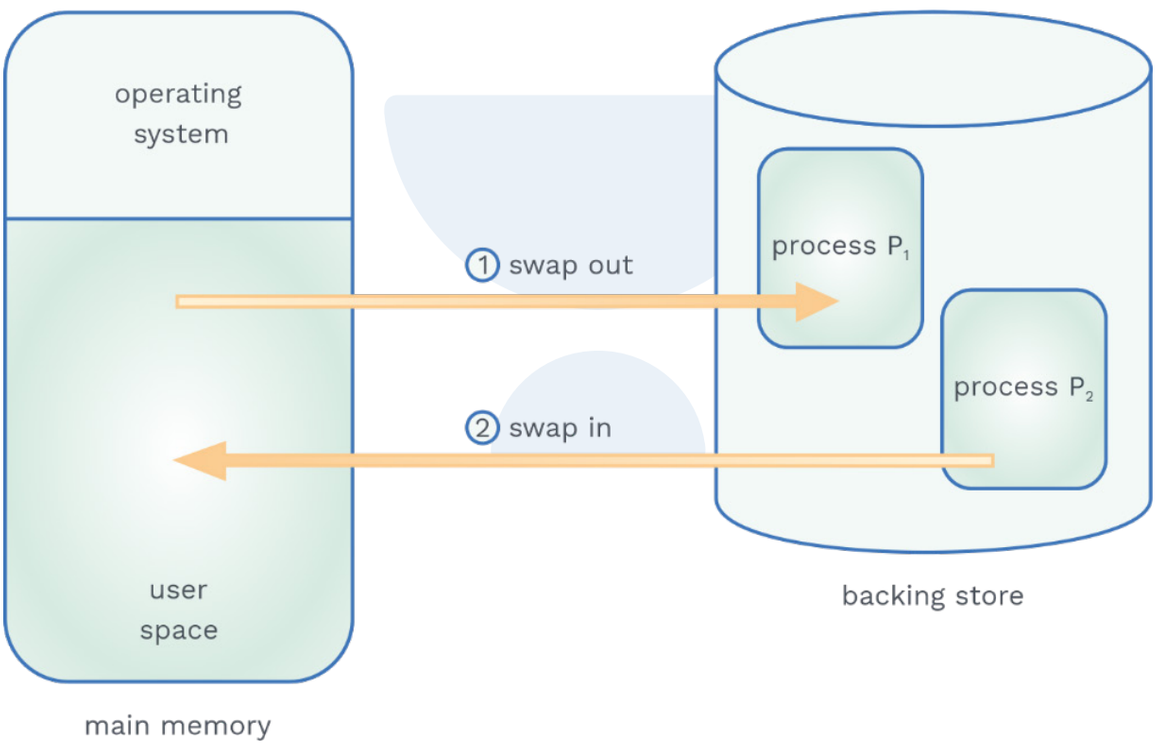
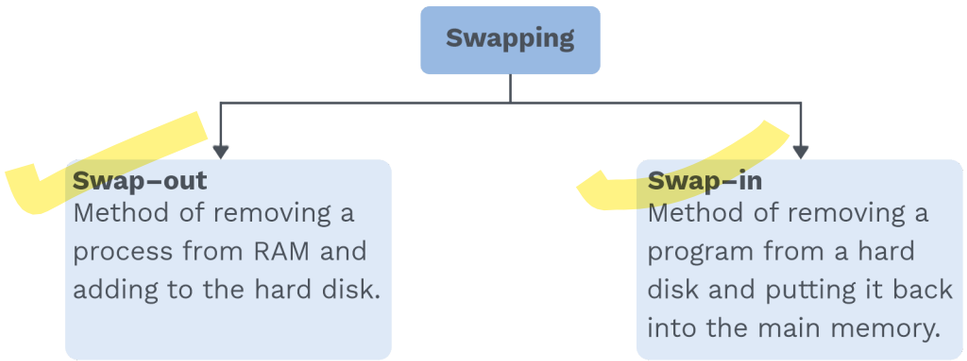


Fig. 5.6 Swapping of Two Processes Using a Disk as a Backing Store

Definition

Swapping is a memory management technique that allows any the process to be temporarily switched from main memory to secondary memory to free up main memory for other processes.

The concept of swapping has divided into two parts:



SOLVED EXAMPLES

Q1 **Suppose a user process of size 4096 KB is stored on a standard HDD, where swapping has a transfer rate of 1 MBPS. Calculate how long (in milliseconds) will it take to transfer the process from RAM to HDD?**

Sol: **Range: 4000-4000**

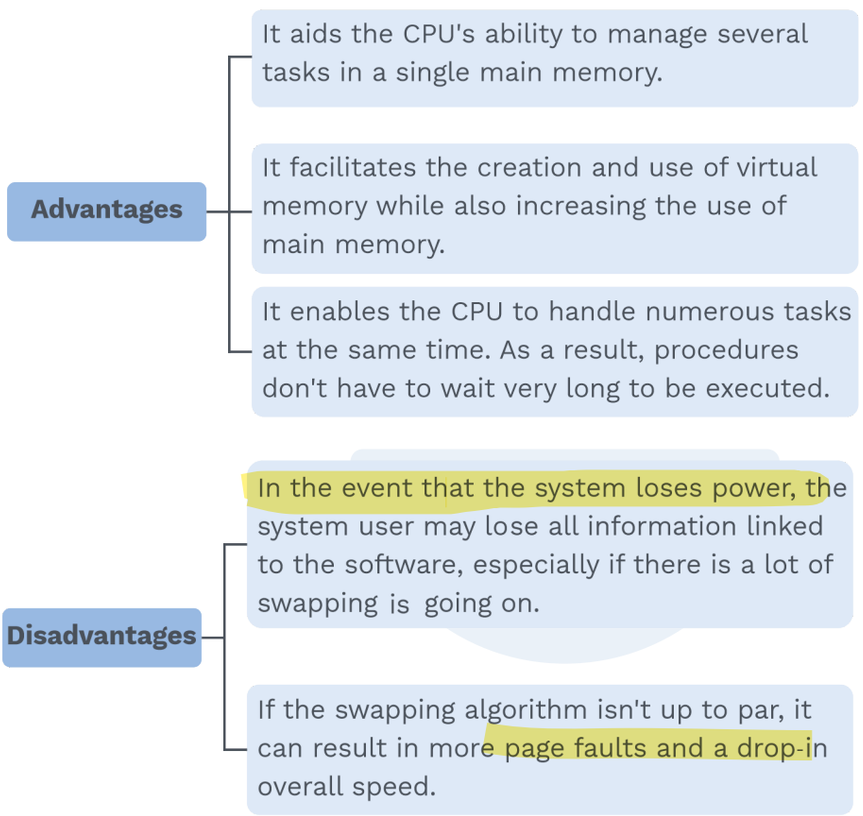
User process size = 4096 KB

Data Transfer Rate = 1 MBPS = 1024 KBPS

$$\text{Time} = \frac{\text{Process Size}}{\text{Transfer Rate}} = \frac{4096\,\text{KB}}{1024\,\text{KBPS}}$$

Time = 4 seconds

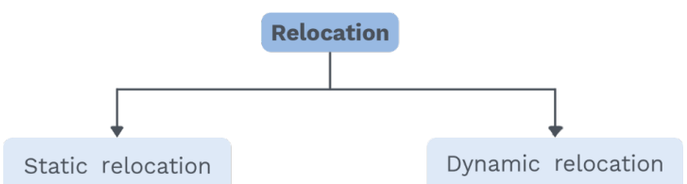
Time = 4000 milliseconds



Objectives of memory management system:

1) Relocation:

- The ability to shift processes around in memory without having any impact on their execution.
- Memory is managed by the operating system rather than the programmer, and processes can be moved into the memory.
- Memory Management (MM) is responsible for translating logical addresses to physical addresses.



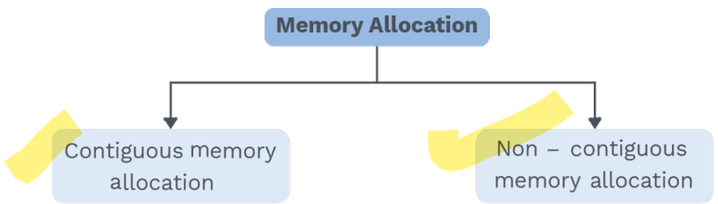
Static relocation:
Before or during the loading of the process into memory, the program must be relocated.
Relocator must be executed again if the program is not loaded into the same address space in memory.

Dynamic relocation:
In memory, processes can freely move about. At runtime, a virtual–to–physical address space mapping is performed.

2) Protection:
- To prevent data and instructions from being over–written.
- For data and instructions security purposes, OS is protected from user processes and on the same note, user processes are protected from other user processes.
- The reason behind this is many of the languages compute address of memory during runtime.

3) Sharing:
- Different processes may need to run the same operation or even access the same data at times.
- Different processes must access the same memory address when they signal or wait for the same semaphore.

Memory allocation:



Contiguous memory allocation:
The RAM is frequently partitioned into two parts: one for the operating the system, and the other for user processes.

Note:
The operating system can be installed in either a low-memory or a high-memory location. It is determined by the Interrupt Vector's location.

Definitions

It is basically a method in which a single contiguous part of memory is allocated to a process.

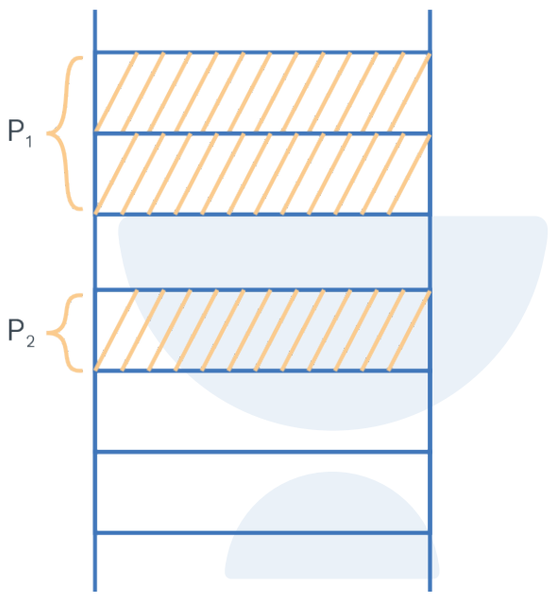


Fig. 5.7 Contiguous Memory Allocation

- Unused memory space is allocated to the same location in contiguous memory.
- Processes are faster in execution.
- Processes are easier for the OS to control.
- Address translation is minimum while executing a process.
- It suffers from both Internal and external fragmentation.

Grey Matter Alert!

Internal fragmentation:
When memory blocks associated with a process have unused space within the block, it is called internal fragmentation.

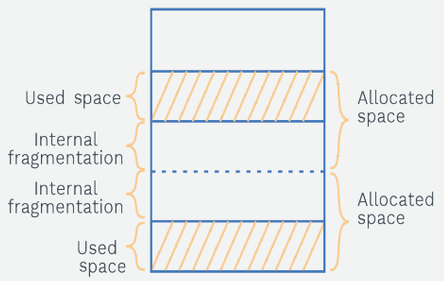


Fig. 5.8 Internal Fragmentation

External fragmentation:
When there is enough space within the memory to satisfy a method's memory request, but the process' memory request cannot be completed because the memory is spread in a non-contiguous manner, several approaches, such as compaction, are applied to solve external fragmentation.

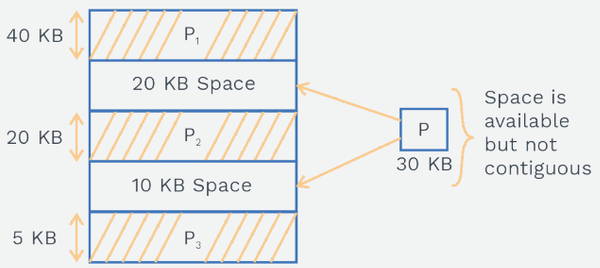
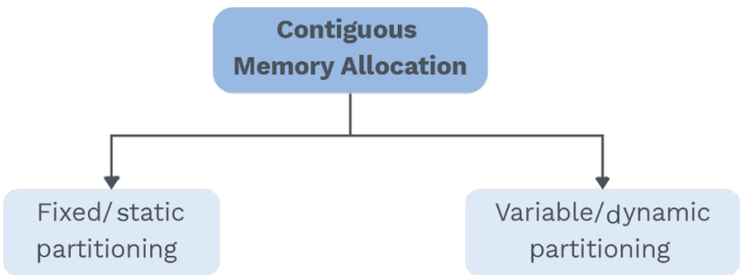


Fig. 5.9 External Fragmentation

In the above figure P1, P2 and P3 have been allocated in RAM, leaving holes of 20KB and 10KB which are not contiguous. When a process P of 30KB request for space in memory; it cannot be allocated because available space is not contiguous.



Fixed partitioning:

- This is the simplest technique used to store more than one process at a time in the main memory.
- In this partitioning, a number of non-overlapping partitions in physical memory is fixed, but the size of each partition may or may not be the same.
- In this, only one process is allowed per partition.

Note:

The operating system is always installed in the first partition, with the remaining partitions being utilised to store user processes.

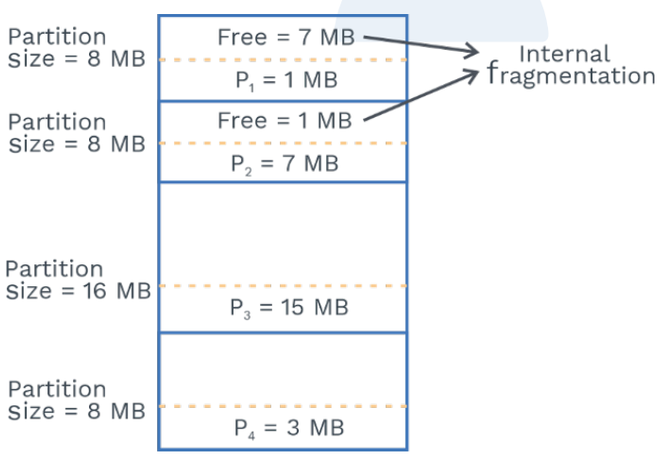


Fig. 5.10 Fixed Size Partition

Advantages:

1) Implementation is easy

2) Fixed partitioning processing necessitates less CPU resources, and minimal OS overhead.

3) Fixed partitioning does not suffer from external fragmentation.

Disadvantages:

1) Internal Fragmentation is suffered in the fixed partition.
2) Because partitions are made before execution, it limits the degree of multiprogramming. For example, if there are 'x' partitions in RAM and 'y' is the number of processes, then the 'y' = 'x' requirement must be met. In fixed partitioning, processes that are larger than partitions (in terms of quantity) are invalid.
3) It restricts the size of the process since it cannot support processes larger than the partition size.
4) Spanning a process into two partitions is not possible.

Variable partitioning:

Initially, the whole user space of memory is free, and partitions are created as needed after the arrival of processes.

- The total number of partitions is not fixed and is determined by the number of incoming processes and the amount of RAM available.
- The operating system takes up the first partition, and the remaining space is dynamically partitioned.

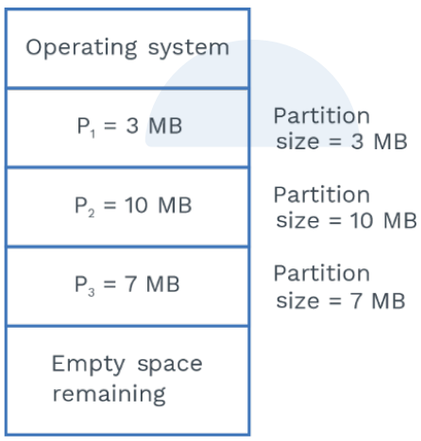


Fig. 5.11 Variable Partitioning

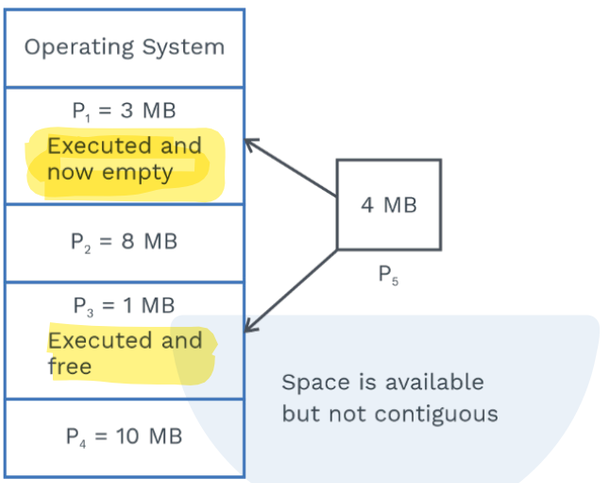
Advantages:

1) There is no internal fragmentation because the main memory space is allocated based on the needs of the process. There will be no unused partition space remaining.
2) There are no restrictions on the amount of multiprogramming that can be done.
3) There is no limit on the size of the process requesting RAM.

Disadvantages:

1) Implementation is complex as the allocation of memory is at runtime.

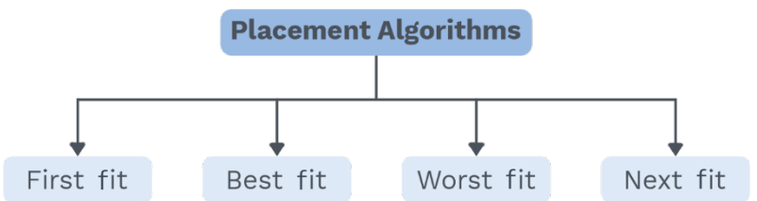
2) It suffers from external fragmentation.



In the above figure P_1, P_2, P_3 and P_4 has been allocated in RAM. Process P_1 and Process P_3 have been executed and freed up space of 3MB and 1MB, which are not contiguous. When process P_5 of 4MB requests for space in memory, but it cannot be allocated because available space is not contiguous.

Allocation policies:

When there are multiple partitions available to accommodate a process, a partition must be chosen using one of the partition allocation policies.



1) First fit:

It assigns the first available free partition (hole) that is large enough. It begins scanning the memory from the beginning and selects the first large enough block available. Among the several allocation policies, it is the quickest.

2) Best fit:

- It allocates the smallest sufficient free block that can hold the process.
- It gives the worst performance overall in terms of allocation time, as every time the smallest hole is found by searching in the whole memory.
- It will give least internal fragmentation.
- Compaction is done here more often.
- Best fit is an efficient allocation policy for fixed partitioning.

3) Worst fit:

- It allocates the largest block among all available blocks that is big enough.
- It is the opposite of best fit allocation.
- Worst fit is efficient allocation policy for variable partitioning as it will create holes of larger size so that other small processes can be placed in those memory holes. .

4) Next fit:

Next fit will start searching from last allocated partition; rest is the same as that of the first fit.

Rack Your Brain

Is Best–fit really the best allocation technique among others in fixed partitioning?

Grey Matter Alert!

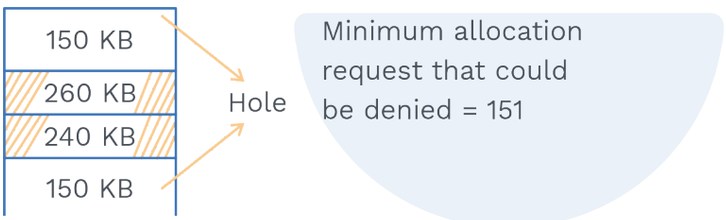
Compaction:
Compaction or shuffling memory contents is a solution for external fragmentation; all free memory is gathered into one single block. Moving should be dynamic in order to make compaction possible. Using a paging or segmentation technique, external fragmentation can be resolved.

SOLVED EXAMPLES

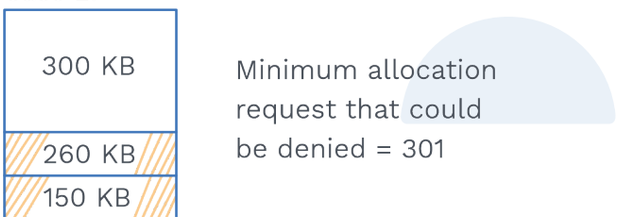
Q2 **A memory of size 800 KB is managed using variable partitioning with no compaction method. Memory currently has the following partitions:**
Partition I – 260 KB
Partition II – 240 KB
What is the smallest allocation request size (in KB) that could be denied?

Sol: **Range: 101–101**

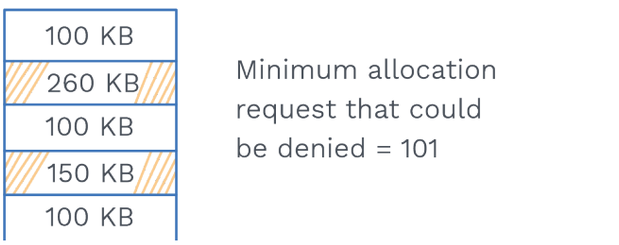
Case 1:



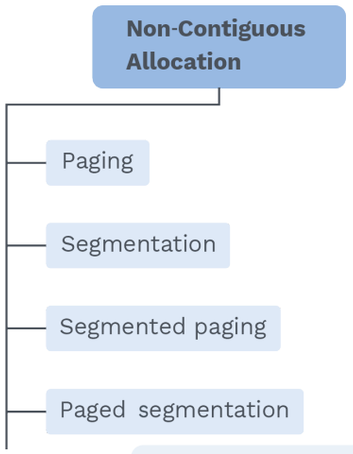
Case 2:



Case 3:



Non–contiguous allocation:
In contrast to contiguous allocation, it is a mechanism that allocates memory space in different locations to the process according to its needs.
All of the available vacant space is dispersed across.
This memory allocation strategy helps to prevent memory waste, which eventually leads to internal and external fragmentation.

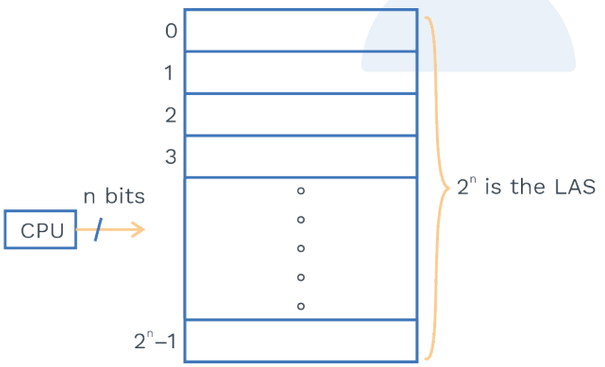


Paging:

- Paging is a storage mechanism used in operating systems to retrieve processes from secondary storage into main memory in the form of pages.

Logical address space (LAS):

It is the collection of logical addresses



LAS = 2^{LA}

Logical Address (LA) = $\log_2$LAS

- It is generated by the CPU and is also referred to as a virtual address.
- It is a reference to a memory location that is independent of the data that is currently assigned to that location.

Translation is required to convert logical to a physical address.

Some terminologies:

Address space:

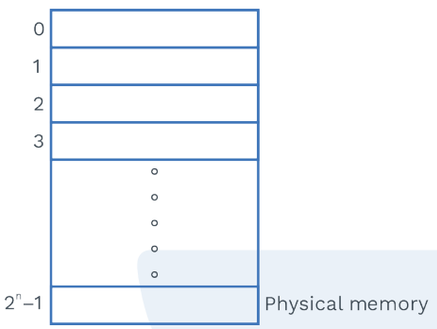
A set of words/memory locations/addresses are called address space.

Two types:

1) Physical Address Space (PAS)

2) Logical Address Space (LAS)

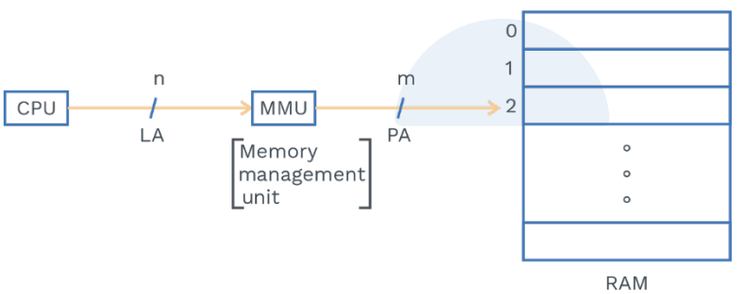
Physical address space:



PAS = 2^{PA}

PA (Physical Address) = ($\log_2$ PAS)

PA is the number of bits to uniquely identify each frame of physical memory



Example: Consider a byte addressable memory, where

LAS = 16 KB = 2^{14}B = LA: 14 bits

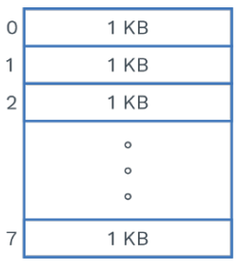
PAS = 4 KB = 2^{12} B = PA: 12 bits

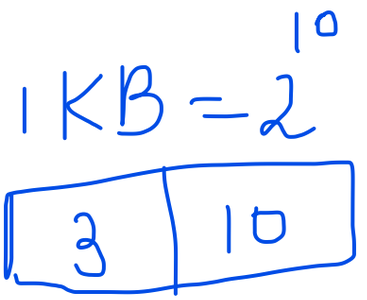
Non-Contiguous memory allocation has mainly four steps.

1) Organisation of LAS:

- It is the view of the process from CPU perspective.
- Pages are equisised.
- Pages are nothing but logical division of long processes into smaller blocks.

Example: Consider the following logical address space(LAS), which is divided into equal size pages:





Size of page = 1KB
LAS: 8 KB = 2^{13} B

So number of pages (P) = $\frac{\text{LAS}}{\text{Page size}} = \frac{2^{13}\text{B}}{2^{10}\text{B}} \Rightarrow 8$

Now to represent eight pages, we need minimum of 3 bits, and 10 bits to identify each byte in pages, assuming memory is byte addressable.
Number of page bits (P) depends on number of pages in LAS(N).
$N = 2^P$
$P = \log_2(N)$

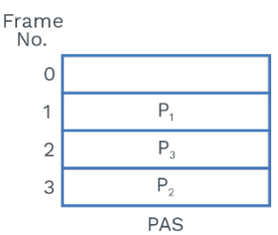
So LA = | P (3) | d (10) | ← 13 →

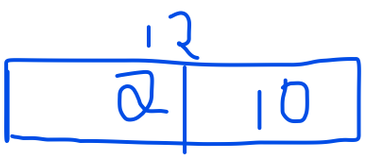
P : Page Number
d : Page offset

2) Organisation of PAS:

- Physical memory is divided equisised into many logical parts known as frames.
- One frame accommodates one page of a process.
- Size of frame is equal to page size.

Example:





Size of PAS = 4 KB = 2^{12}B
PA = 12 bits
Frame Size = 1 KB

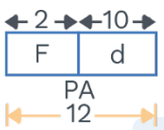
$\Rightarrow$ Number of frames (M) = $\frac{\text{PAS}}{\text{Frame size}}$

$$= \frac{2^{12}B}{2^{10}B} = 4$$

To represent 4 frames, we need atleast 2 bits and 10 bits for page offset.
The number of frames bits (F) depends on number of the frames in PAS(M).

$$M = 2^F$$

$$F = \log_2(M)$$



F : Frame number
d : Frame offset

3) Memory management unit – organisation:

- Each process is associated with its own page table, which are kept in the main memory (overhead).
- Page table is organized into a sequence of entries where entries are equal to the number of entries in logical address space (page table size).
- Page table entry (PTE) contains the essentially frame number of PAS in which the page referred is present.
- PTE denoted by 'e' is measured in bytes. The minimum size of PTE is 1 byte.
- PTS (Page Table Size) is given by

$$PTS = N * e \text{ Bytes}$$

N = No. of pages in LAS

e = Page table entry

- Page table of a process is not accessible to other processes.

4) Address translation:

- The LA (logical address) is generated by 'CPU' divided into a page number and offset, used as an index into the page table and then to frame number.

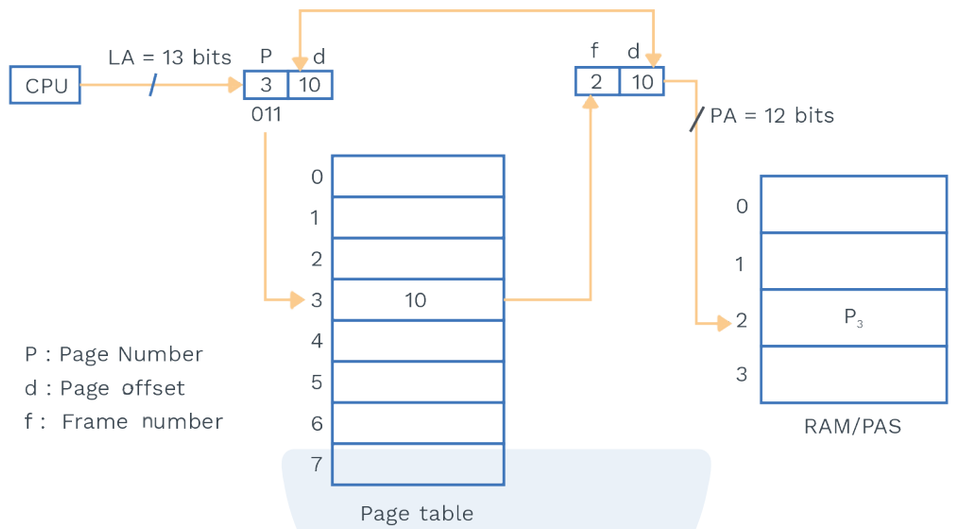


Fig. 5.12 Concept of Paging

SOLVED EXAMPLES

Q3 **Consider the following details of a byte addressable memory:**
LA = 29 bits
PA = 21 bits
Page size = 4 KB
If the page table entry size is 2 bytes.
What is the page table size(in KB)?

$4KB = 2^2 \times 2^{10} = 2^{12}$

LA 29: 17 | 12

Sol: **Range: 256-256**

Let 1 W = 1 Byte

LAS = 2^{29} B

PAS = 2^{21} B

PS (Page Size) = 2^{12} B

PA 21: 9 | 12

$$\text{Number of pages} = \frac{\text{LAS}}{\text{PS}} = \frac{2^{29}}{2^{12}} = 2^{17} \text{ pages}$$

LA = | P | d | P : 17, d : 12

PA = | f | d | f : 9, d : 12

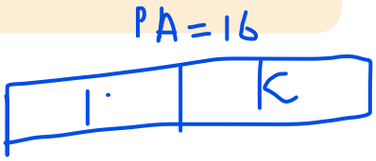
$\Rightarrow$ e = 2B

Page Table Size = $2^{17} \times 2B$

$= 2^{18}$ B

= 256 KB

Q4 **Suppose LAS = PAS = 2^{16} bytes in a paging scheme. Consider page table entry size be 4 Bytes. What should be the page size (in bytes) of a process so that page table fits in exactly one page?**



Sol: **Range: 512-512**

Given process size is 2^{16}B

Let page size be 2^KB

$$\text{So, number of pages} = \frac{\text{Process Size (LAS)}}{\text{Page Size}} = \frac{2^{16}}{2^K}$$

$$= 2^{16-K}$$

$$\text{Page table size} = 2^{16-K} \times 4B$$

$$= 2^{18-K}\ B$$

Now we have to fit this page table in one page.

So, $2^{18-K} = 2^K$

$\Rightarrow 18-K = K$

$\Rightarrow 18-2K = 0$

$\Rightarrow K = 9$

So page size is 2^9B = 512B

Q5 **Consider a system with byte addressable memory, a virtual address space of 36 bits and an 8-KB page size. Size of the main memory is 512 MB. What is the approximate size of the page table (in MB) when PTE contains 2 valid bit, 2 modified bit and 2 reference bit?**

Sol: Range: 24-24

Virtual address = 36 bit

So VAS = 2^{36} bit

$$\text{Number of pages} = \frac{2^{36}}{2^{13}} = 2^{23} \qquad \text{Number of frames} = \frac{2^{29}}{2^{13}} = 2^{16}$$

Page table entry contains frame number(16 bits; along with that, it also has 2 valid bits, 2 modified bits and 2 dirty bits.

Page table entry size = 16 + 2 + 2 + 2 bits

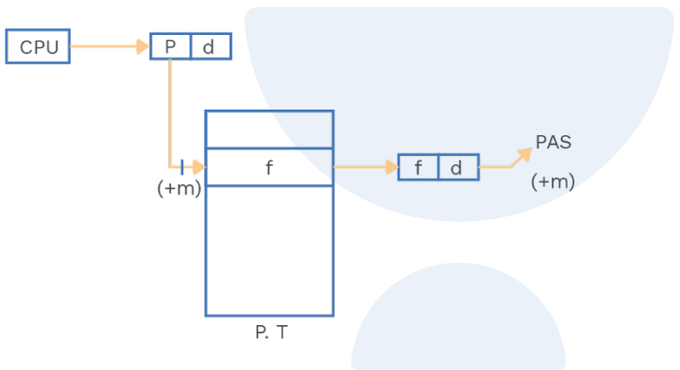
= 22 bits = 24 bits (Since PTE size must be in multiple of bytes)
Page table size = Number of pages × PTE size

$$= 2^{23} \times \frac{24}{8} \text{ B}$$

$$= 24 \times 2^{20} \text{ B}$$
$$= 24 \text{ MB}$$

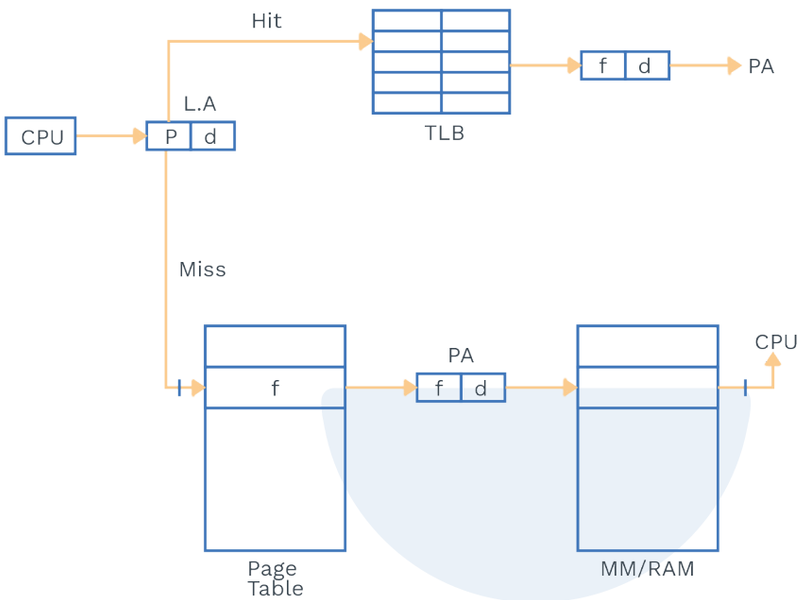
Performance of simple paging:
Two issues with paging technique

1) Time (temporal) issue:



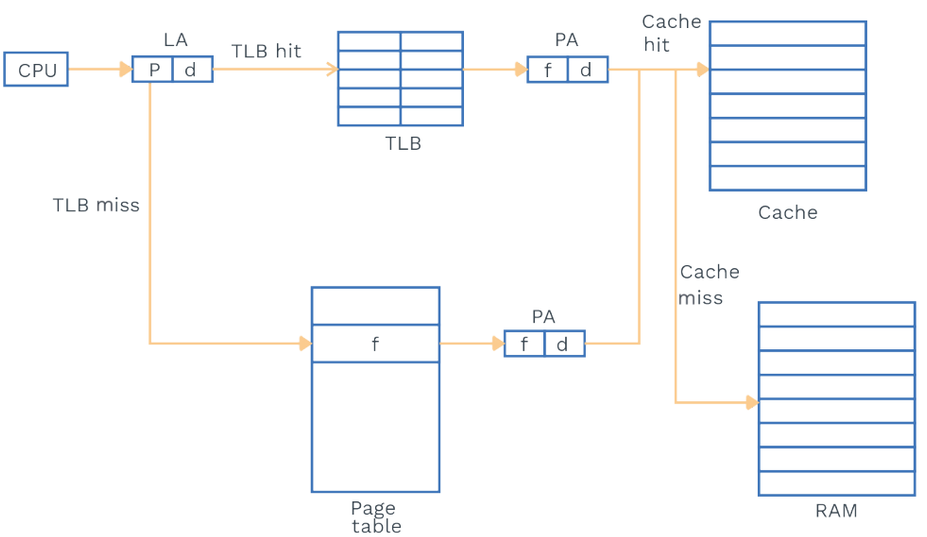
Set the memory access time to 'm' milli seconds.
Effective memory access time (EMAT)
= m(P.T.) + m(word)
= 2m
This EMA includes 1) Page table access time
2) Physical memory access time
So, to reduce EMAT we can use paging with TLB (Translation lookaside buffer)

Paging with TLB:



Let TLB hit ratio be 'x' ; $0 \leq x \leq 1$
TLB miss ratio = (1 – x)
TLB access time 'c': (c << m)
EMAT = x(c + m) + (1 – x) (c + 2m)

Paging with TLB and Cache:



Let TLB hit ratio be 'x': $0 \leq x \leq 1$
TLB miss ratio: (1 – x)
TLB access time be 'c': (c << m)
Cache hit ratio be 'y': $0 \leq y \leq 1$
Cache miss ratio: (1 – y)
Cache access time: K
Memory access time: m
Let all the pages are in main memory.
$EMAT_{(TLB+Cache)}$: x (c + y (K)+ (1–y) (K+m)) + (1–x) (c + m + y(K) + (1–y) (K+m))

SOLVED EXAMPLES

Q6 **Consider a system with a memory access time 280 ns. Assuming a TLB with access time 40 ns is used with the memory. What should be the TLB hit ratio (up to two decimal place) to reduce effective memory access time to 180 ns?**

Sol: Range: 1-1
C = 40ns
$EMAT_{(SP)}$ = 280 ns = 2 × m ⇒ m = 140 ns
$EMAT_{(SP + TLB)}$ = 180 ns = x(40+140) + (1 – x) (40 + 280)

x = 1 ⇒ 100 percent TLB hit when hit ratio is 1.

2) Spatial issue:

- It aims to reduces Page table size overhead.
- Reducing the page table size by increasing page size but, it leads to more internal fragmentation.
- Decreasing the page size will increase the number of pages thus increases Page table size.

Optimal page size:
An optimal page size should minimize both page table size and internal fragmentation.
Let logical address space = S bytes
Page size (PS) = P bytes

Page table entry = e bytes

$\Rightarrow$ So, number of pages in LAS $= \frac{S}{P}$

$\Rightarrow$ page table size (PTS) $= \frac{S}{P} * e$ byte

$\Rightarrow$ We take internal fragmentation due to the wasted memory in the last page of process is

IF (avg) $= \frac{PS}{2} = \frac{P}{2}$

Total overhead = PTS + IF $= \frac{S}{P}(e) + \frac{P}{2}$

We minimize total overhea by taking the first derivative with respect to 'P' and solving it:

$\Rightarrow \frac{d}{dP}\left(\frac{S}{P}(e) + \frac{P}{2}\right) = 0$

$\Rightarrow -\frac{S}{P^2}(e) + \frac{1}{2} = 0$

$\Rightarrow P^2 = 2Se$

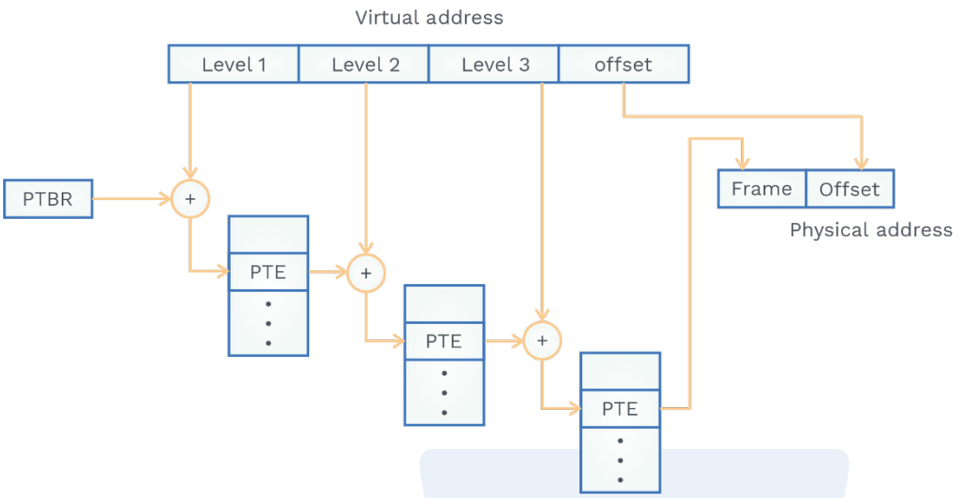
$\Rightarrow P = \sqrt{2Se}$

Conclusion is, $\sqrt{2Se}$ is the optimal page size to have minimum overhead of page table and internal fragmentation.

Multilevel paging:

Multilevel paging is a hierarchical paging technique that consists of two or more levels of page tables.

- This method is to avoid memory overhead of page tables. (i.e. larger page table size is not desirable)
- Multilevel paging refers to paging on a page table, in which the address space of the page table is partitioned into pages of equal-sized.
- The frames of PAS accommodate pages of page tables.
- Pages of the page table are accessed in PAS through another page table known as the outer page table.
- Page table base register stores the address of the level 1 (outermost) page table (PTBR).



1) PTBR value + level 1 offset present in virtual address created by CPU equals reference to PTE in level 1 page table.
2) Base address (present in level 1 PTE) + level 2 offset equals reference to PTE in level 2 page table (present in V.A.)
3) Base address in level 2 PTE + level 3 page offset equals reference to PTE in level 3 page table.
4) PTE is the address of the actual page frame (present in level 3).

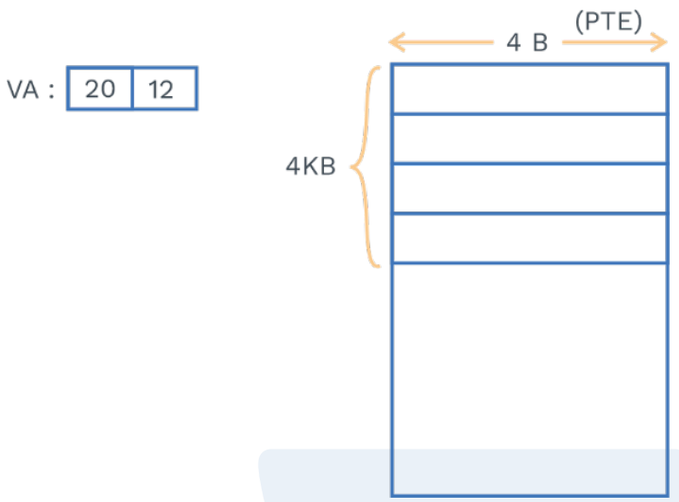
Example: Suppose we have a multilevel paging scheme where LAS = 4 GB. Page size = 4 KB. How many level page table exists. Assume PTE(e) = 4 B.

Sol: Number of pages $= \frac{\text{LAS}}{\text{PS}}$

$= \frac{2^{32}}{2^{12}}$

$\Rightarrow 2^{20}$

So to represent 2^{20} we need 20 bits atleast and 12 bits for page offset.



Number of pages in the innermost page table = 2^{20} pages

Page table size of innermost page table

= Number of pages × PTE size

$2^{20} \times 4B = 2^{22}B$

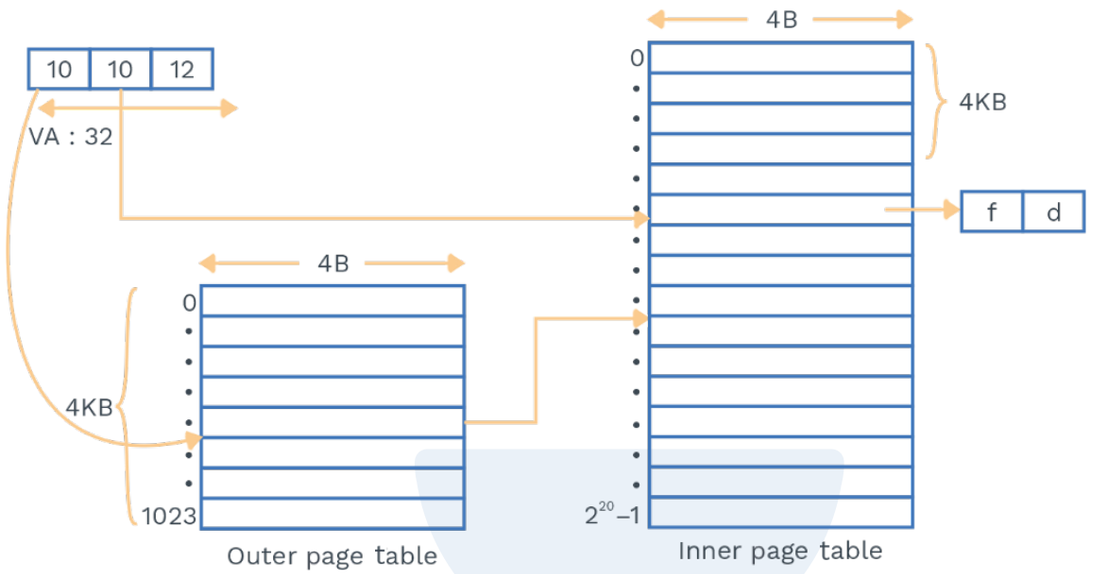
Which is greater than page size.

So more level of paging

Number of pages in next level page table

$$= \frac{\text{Page table size at inner level}}{\text{page size}}$$

$$= \frac{2^{22}B}{2^{12}B} = 2^{10}$$

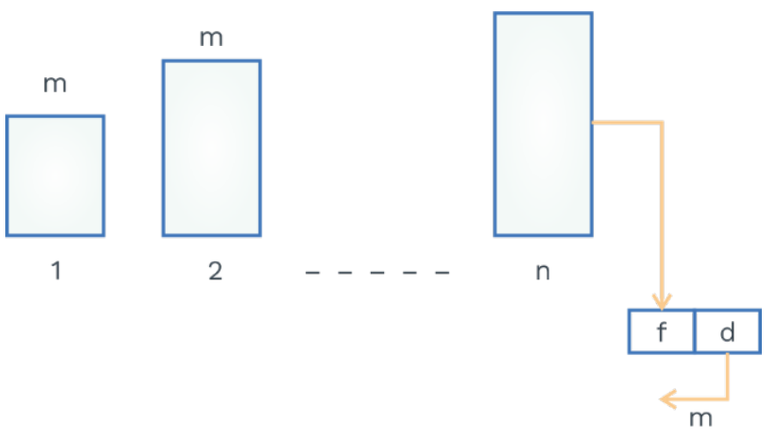


The outer page table has 2^{10} entries that point to the pages of the inner page table.
$= 2^{10} \times 4B = 2^{12}$ which is equal to page size, so no more page table is created.
So two levels of page table exist.

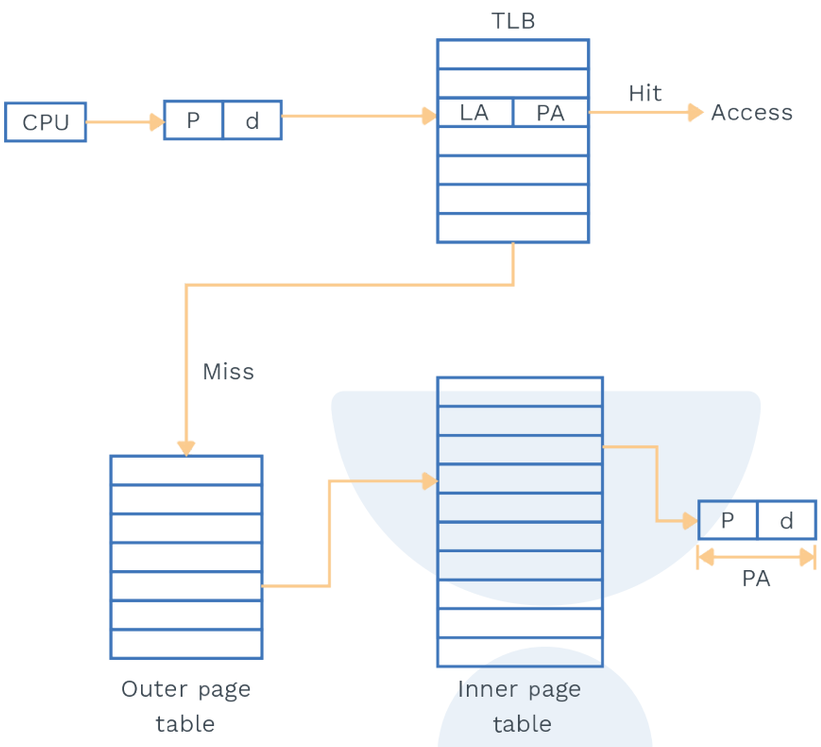
Performance of multi–level paging:

- Effective memory access time of 2–level paging

$(EMAT_{2LP}) = 3 * m$ ns
$\therefore$ m = memory access time
Generated formula for n–level paging
$EMAT_{'n'LP} = (n+1)\ m$ ns

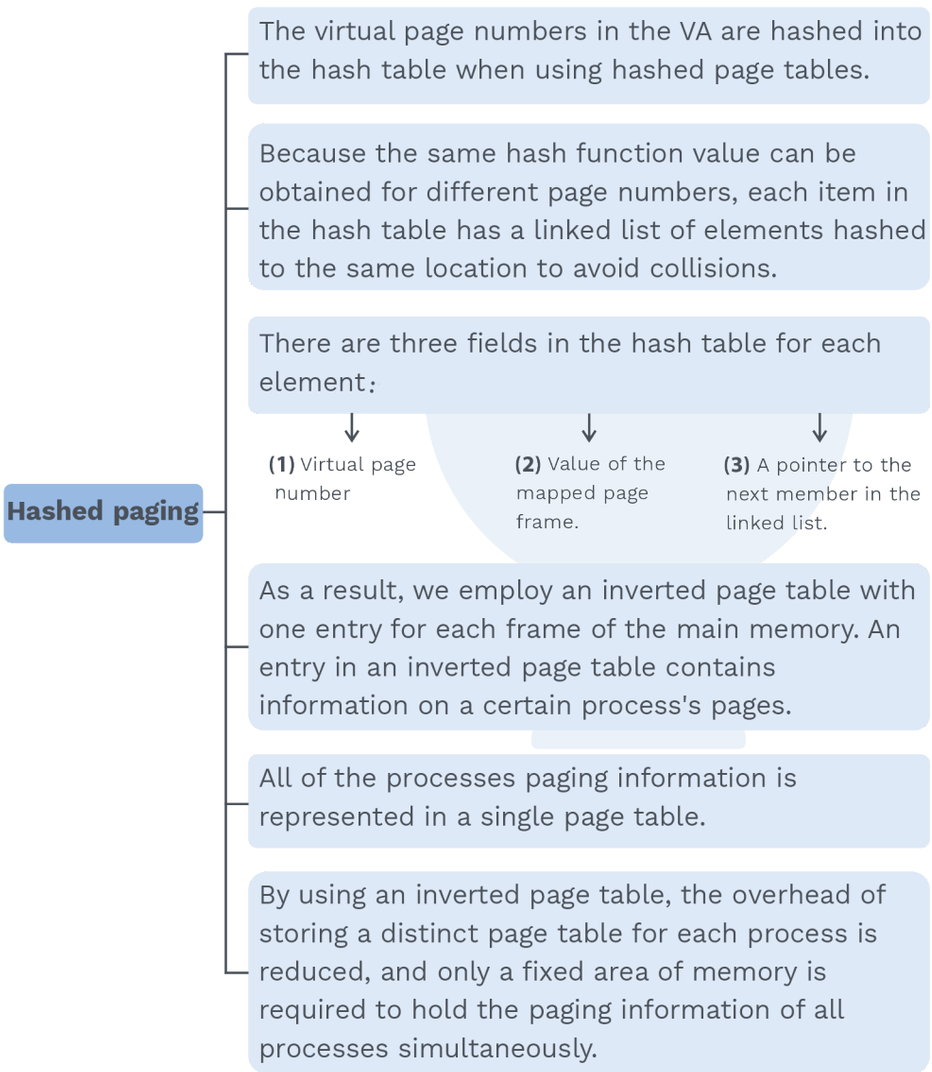


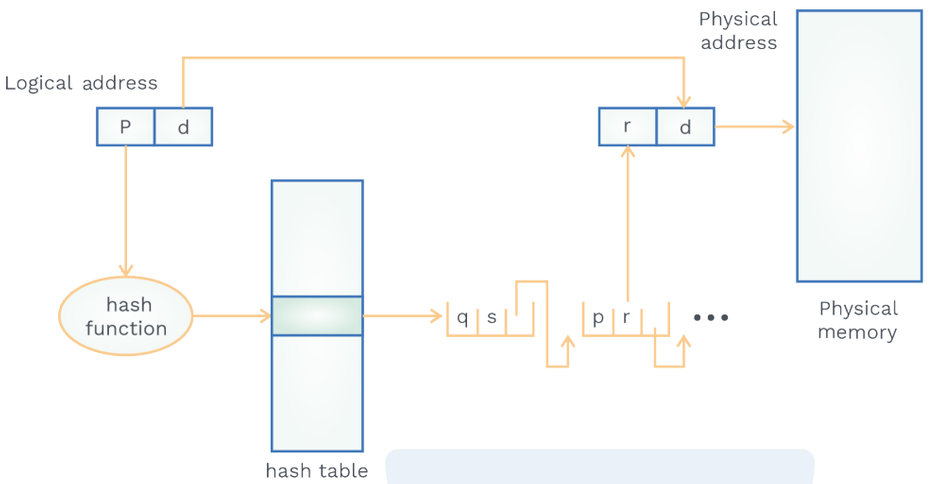
Multilevel paging with TLB:



Let TLB hit rate = x
TLB access time = c
memory access time = m
$EMAT_{2LP+TLB}$ = x (c + m) + (1 – x) (c + m + m + m)
= x (c + m) + (1 – x) (c + 3m)
Generalize formula if n level page table is present
$EMAT_{n\ LP+TLB}$ = x (c + m) + (1 – x) (c + (n + 1) m)

Paging with hashing (using data structures):





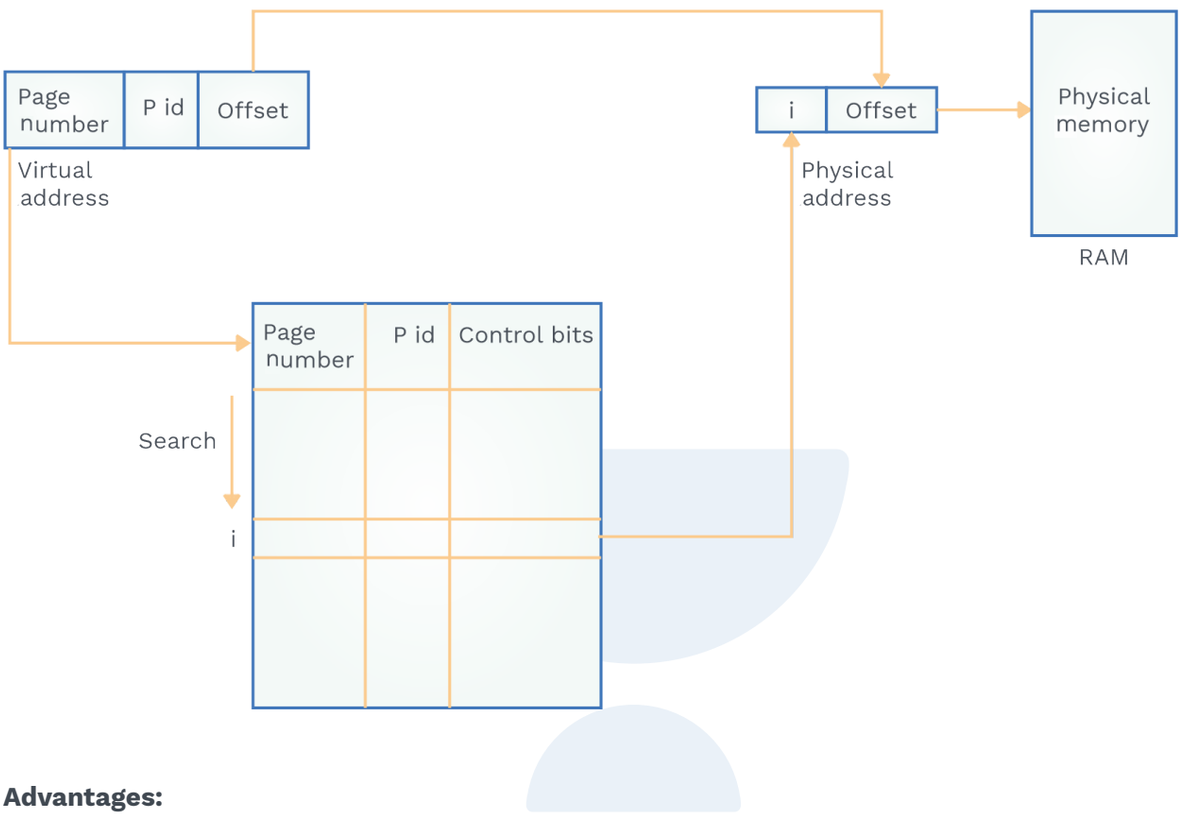
Inverted page table entry

Page number	Process-id	Control bits	Chained pointer
		It includes valid bit, dirty bit, reference bits, protection, etc.	It may be possible sometimes that two or more processes share a part of RAM. In this case, two or more logical pages map to same page table entry then chaining pointer is used to map the details of these logical pages to the root page table.

Working:

1) The fields are contained in the virtual address generated by the CPU, and each page table item contains other pertinent information.

2) When a memory reference occurs, the virtual address is matched by the memory-mapping unit, and an inverted page table is searched for a match with the appropriate frame number acquired from the page table index; if no match is discovered, a segmentation fault is issued.



Advantages:

- Reduced memory space

Disadvantages:

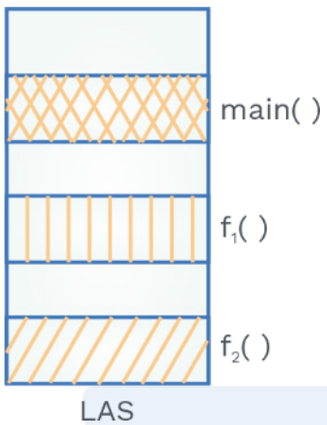
- As a result of the longer search time, the inverted page table is sorted by frame number, but the memory lookup is done with the virtual address in mind.
- Implementing shared memory is tough.

Segmentation:

There are certain limitations to simple paging, which are overcome by segmented paging.

- Paging does not preserve the user's view of memory allocation to the program.
- As per user view, the program is divided into logical parts representing functions, and procedures known as segments.
- Those segments which may be of different sizes are stored in PAS.
- The words of the segments are accessed in PAS through a segment table.
- Virtual address consists of segment number and offset.

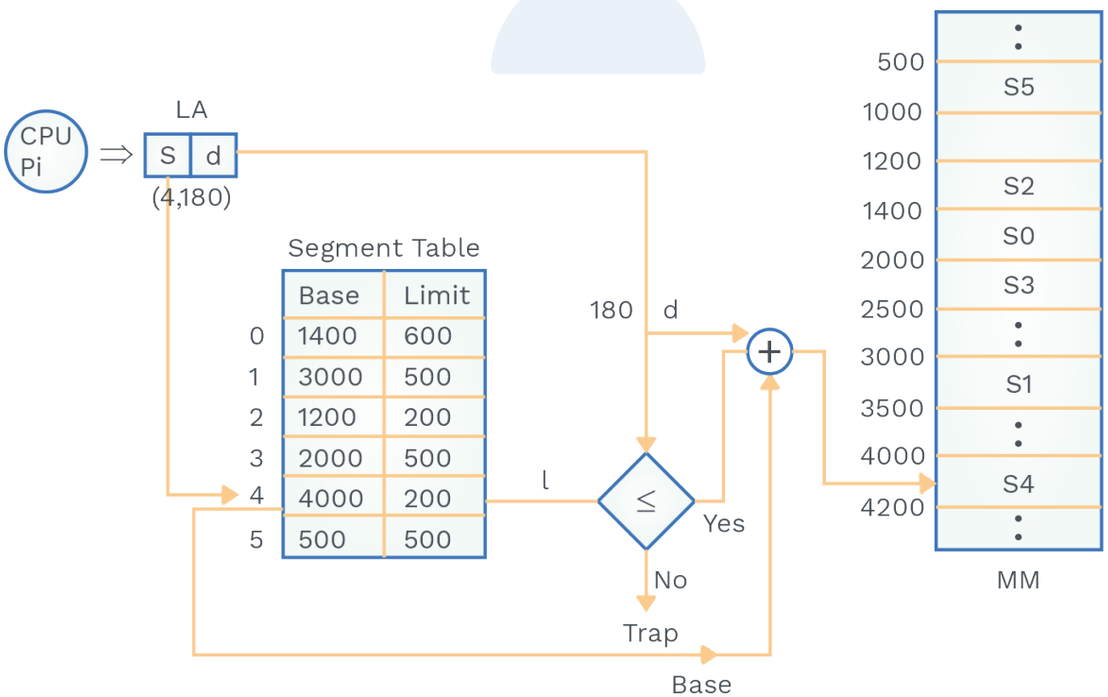
Organisation of LAS in segmentation:



- The whole program is divided into unequal size segments.

Organisation of PAS in segmentation:

- Variable sized partitioning.
- No concept of frames.
- Partition allocation methods such as best fit, Worst fit are used.



Example:

Logical Address	Physical Address
2 | 120	1200 + 120 = 1320 (Valid)
1 | 350	2000 + 350 = 2350 (Valid)
1 | 510	3000 + 510 = 3510 (Invalid)
4 | 180	4000 + 180 = 4180 (Valid)

- Logical address is divided into two parts.

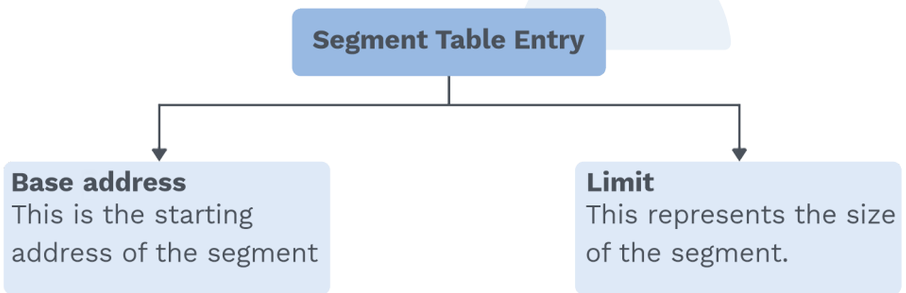
1) Segment number: It represents the segment, CPU requested the word from.

2) Offset: The address of a word displaced from the starting address of the first word of segment.

- Physical address is divided into two parts.

1) Segment starting address

2) Offset



- MMU checks the validity of the memory reference by calculating the absolute address by adding the base address and offset, and if the offset is greater than the limit specified in segment table, then a trap is generated, and MMU invalidates the memory request.

Segmentation v/s paging:

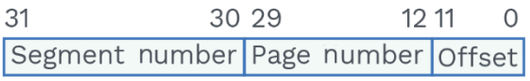
Segmentation	Paging
Non contiguous memory allocation	Non contiguous memory allocation
Variable sized segment	Fixed sized pages
Slower	Faster
It suffers from external fragmentation	It suffers from internal fragmentation
For segmentation, compiler is responsible	For paging, OS is accountable
OS maintains a list of holes in main memory	OS maintains a free frame list.
The segment size is given by user	Page size is determined by hardware
Segment table is used	Page table is used

EMAT in segmentation:
Let memory access time be 'm'.
EMAT (Effective memory access time of segmentation) = 2m
EMAT (Segmentation + TLB) = x (C + m) + (1 – x) (C + 2m)
x = TLB hit ratio
C = TLB access time
m = memory access time

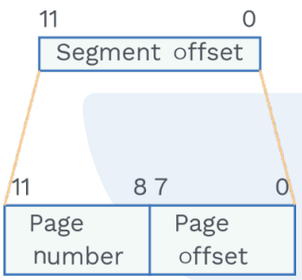
Issues with segmentation:
- External fragmentation occurs.
- Segment table slows down the translation process.
- External fragmentation can be overcome by segmented paging.

Segmented paging:
Segmented paging uses a dual concept of segmentation (partition of a process from user's view) and paging.
- The program is divided into variable-size segments, and each segment is further get divided into fixed-size pages.
- Pages are smaller than segments, and each segment has its own page table, implying that each program has several page tables.

- Segment number, page number, and page offset are used to represent virtual addresses.



- The logical address is made up of segment number and segment offset from the perspective of the programmer, but from the perspective of the operating system, segment offset is made up of page number and page offset for a page within its defined segment.



Example:

Let a segment be of 64 KB, and the virtual address generated by CPU is of 32 bits. Segmented paging is applied with page size (PS) = 1 KB.

$$\text{Number of pages} = \frac{\text{Segment size}}{\text{Page size}} = \frac{64\,\text{KB}}{1\,\text{KB}} = 64$$

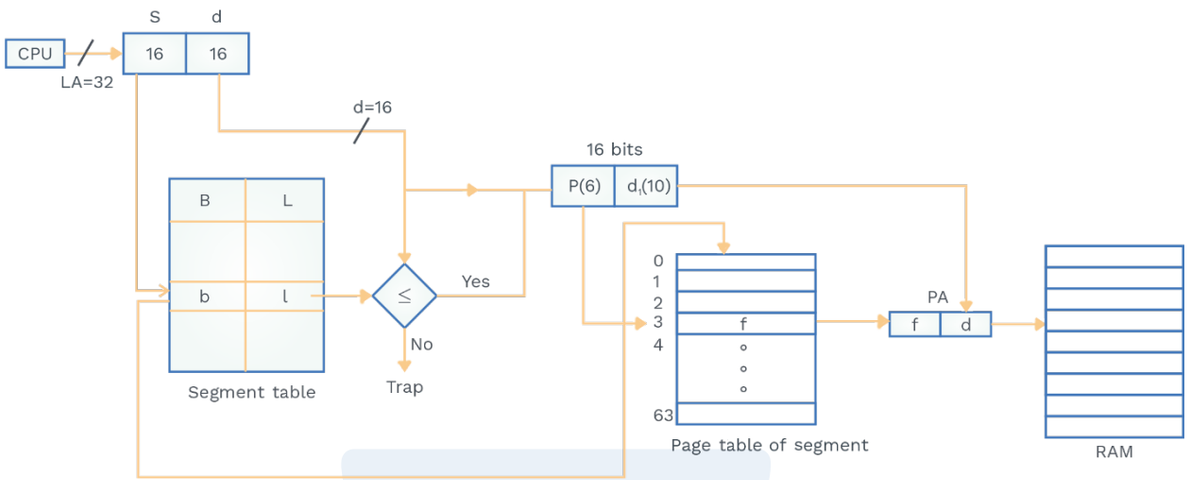


Fig. 5.12 Segmented Paging

Advantages of segmented paging:

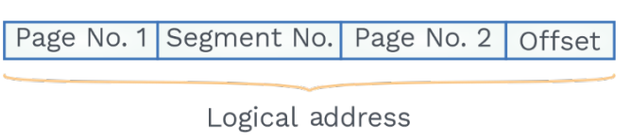
- The page table size is minimised because pages are only present for segment data, lowering memory requirements.
- It provides a user view as well as the benefits of paging.
- When compared to simple segmentation, it reduces external fragmentation.
- Because virtual memory (described later in this chapter) does not require the complete segment to be swapped out, the transition to virtual memory is simple.

Disadvantages of segmented paging:

- Internal fragmentation will still exist in paging.
- External fragmentation happens due to variable widths of page tables and segment tables in today's system, which necessitates the employment of additional hardware.

Paged segmentation:

- To avoid external fragmentation, we employ paging on the segment table, also known as Paged Segmentation.
- The logical address created by the CPU in paged segmentation will now contain of



S: Segment number
P: Page number
d: Offset
x: Starting address of the page of segment table
b: Base address of page table of segment
l: Limit or size of the segment

- Even with segmented paging, the page table can have a number of invalid pages. The issue with large page tables can be resolved by using multi-level paging.

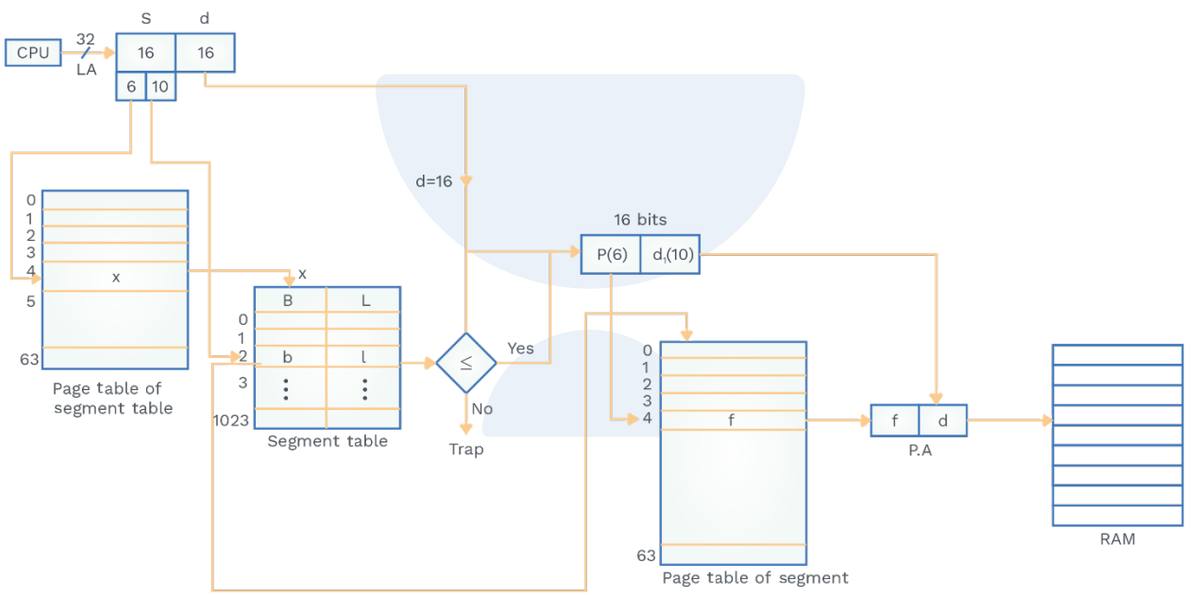


Fig. 5.13 Paged Segmentation

Advantages of paged segmentation:

- Page table size will be reduced.
- There is no external fragmentation, and memory requirements are reduced because the number of pages is limited to the segment size.
- Swapping out the full segment in virtual memory is not required.
- Like segmented paging, the size of the page table is reduced.
- No external fragmentation
- The entire segment need not be swapped out in virtual memory.

Disadvantages of paged segmentation:

- Internal fragmentation still exists.
- Hardware is required, which is more complex than segmented paging.
- Extra layer of paging at the first stage at segment table adds to the delay in access of memory.

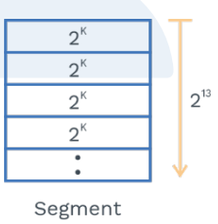
SOLVED EXAMPLES

Q7 **Suppose LAS = PAS = 2^{16} B, LAS is divided into 8 equisized segments and paging on each segment is applied. Let page table entry size be 2 byte. What should be the page size (in bytes) of a segment so that page table of segment fits in exactly one page?**

Sol: Range: 128-128

Size of segment is 2^{13} B. Let page size be 2^K B.

Number of pages in segment $= \frac{2^{13}}{2^K} = 2^{13-K}$



Size of page table = Number of pages × size of PTE

$= 2^{13-K} \times 2B$

$= 2^{14-K}$ B

Now we have to fit this page table in one page.

$$2^{14-K} = 2^K$$

$$14-K = K$$

$$K = 7$$

So page size = 2^7 B = 128B

Q8 **Consider a system with a virtual address of 38 bits that uses 2–level paging. The page directory is indexed with the most significant 10 bits, whereas the page table is indexed with the next 16 bits. In both levels, each entry is 4 bytes long. What is the maximum number of page tables a process can have?**

Sol: Range: 1025-1025
At the first, the level there is a page directory, and each entry in the page directory points to a page table. So, 10 bits are given for the page directory.
Then it points to 2^{10} = 1024 page tables.
So total no. of page tables = 1 + 1024 = 1025

Q9 **Consider a system with TLB support and two–level paging. TLB access takes 20 nanoseconds, while main memory access takes 80 nanoseconds. TLB has references for 130 pages out of 400 pages references. How long (in ms) does it take to access memory effectively?**

Sol: Range: 208 to 208
TLB hit ratio (x) = $\frac{130}{400}$ = 0.325

EMAT = x (C + m) + (1 – x) (C + 3m) ns

C : TLB access time
m : Memory access time

= 0.325 (20 + 80) + 0.675 (20 + 3 * 80)
= 32.5 + 175.5 ns
= 208 ns

Q10 **Consider a paging system with 1 KB pages and 32–bit virtual addresses. Each page table entry necessitates the use of 32 bits. The page table size should be limited to one page. When multilevel paging is used, how many levels are needed?**

Sol: Range: 3-3

Page table size of 1'st level (inner most) = $\frac{2^{32}B}{2^{10}B} \times 4B = 2^{24}$ B

2^{24} B is greater than 1 KB (2^{10} B)

So, page table size of 2nd level = $\frac{2^{24}B}{2^{10}B} \times 4B$

$= 2^{16}$ B

2^{16} B is also greater than 1 KB.

Page table size of 3rd level (Outer most) = $\frac{2^{16}B}{2^{10}B} \times 4B$

$= 2^{8}B$

2^{8} B is lesser than 2^{10} B.

So, 3 level of page table required.

Q11 **Consider a system implementing paging, and in which average process size is 16 MB, and each page table entry size is 16 B. The optimal size of page (in KB, rounded off to 1 decimal digit) to minimize the total overhead due to page table and internal fragmentation is (in KB)?**

Sol: Range: 22.5 to 22.6

Assume page size = x bytes

Process overhead = Page table overhead

+ Overhead due to internal fragmentation

Page table overhead = Number of pages × PTE

$$= \left[\frac{\text{Process Size}}{\text{Page Size}}\right] \times 16\,B$$

$$= \frac{16\,MB}{xB} \times 16\,B$$

Average overhead due to internal fragmentation.

$$= \frac{0 + x}{2} = \frac{x}{2}$$

So, total overhead of paging

$$= \frac{256\,MB}{x} + \frac{x}{2}$$

For minimizing overhead, differentiation with respect to 'x', then

$$-\frac{256\,MB}{x^2}+\frac{1}{2}=0$$

$$x^2 = 2 \times 256\ MB$$

$$x = \sqrt{512\,MB}$$

$$x = 22.6\ KB$$

Q12 **Consider a system with physical address of 40–bit, 16 K frame and page table contains 64 K entries. The number of bits in virtual address will be?**

Sol: Range: 42-42

64K = 16bit = number of pages in page table

It is given 16K frames = 2^{14} frames

= 14 bits

So 26 bits will be the page/frame offset.

Virtual address = No. of pages + Page offset bits

= 42 bits

Q13 **Consider a system with 2-level paging and a TLB with a hit rate of 80%, given TLB access time is 2ns. Find effective memory access time (in ns) if the data cache hit rate is 75% and cache access time is 1 ns, and main memory access time is 100 ns. (Rounded off to nearest integer)**

Sol: Range: 67-67

TLB hit rate= 0.80, TLB miss rate = 0.20, TLB access time = 2 ns

Cache hit rate = 0.75, Cache miss rate = 0.25, Cache access time = 1 ns

Accessing main memory needs 100 ns.

Effective memory access time
= .80 [2 + .75 (1) + 0.25 (1+100)] + 0.20 [2 + 200 + 0.80(1) + 0.20 (1+100)]
= .80 [2 + 0.75 + 0.25 (101)] + 0.20 [2 + 200 + 0.80 + 0.20 (101)]
= 0.80 [28] + 44.6
= 67 ns

Q14 **Consider a system implementing virtual memory of address size 32–bits, having 30–bits of physical address, and a page size of 4 KB. Both physical address space and virtual address space are byte-addressable. The system uses an inverted page table, where each entry has a page number and process identifier (PID) of size 12 bits. Find the size of the inverted page table in Mega Bytes.**

Sol: Range: 1-1
Given,
Physical address Size: 30–bits
Virtual address size: 32–bits
Page size: 4 KB
Process ID size: 12 bits

Physical address space (PAS) = 2^{30} bytes

$$\text{Number of frames} = \frac{\text{PAS}}{\text{Page Size}} = \frac{2^{30}\text{ bytes}}{4\text{KB}}$$

$$= \frac{2^{30}\text{ Bytes}}{2^{12}\text{ Bytes}} = 2^{18}$$

Number of pages = LAS/page size

$$= \frac{2^{32}}{2^{12}} = 2^{20}$$

Number of bits to denote page number
= $\log_2 2^{20}$ = 20 bits
Page table entry size = Page number bits + PID bits
= 20 + 12 = 32 bits = 4 bytes
So, size of inverted page table = Number of frames × e
= $2^{18} \times 2^2$ bytes
= 2^{20} bytes = 1 MB

Q15 **Consider a byte addressable virtual memory system with 34-bits of virtual address. Each page table entry is 4 bytes long. What is the minimum page size (in kilo bytes) required for a three-level paging scheme? (Consider each page table to fit into exactly in one frame).**

Sol: Range: 1-1

Given:

Virtual address = 34 bits, page table entry, page table entry (PTE) = 4 B

The size of the virtual address space (VAS) = 2^{34} B

$$\text{Size of a page table} = \left(\frac{\text{VAS}}{\text{Page size}}\right) * \text{PTE}$$

Let the page size is 2n bytes.

So,

$$\text{Size of page table at first level of paging} = \left(\frac{2^{34}}{2^{n}}\right) * 2^{2}$$

$$\text{Size of page table at second level of paging} = \left(\frac{2^{34}}{2^{2n}}\right) * 4^{2}$$

$$\text{Size of page table at third level of paging} = \left(\frac{2^{34}}{2^{3n}}\right) * 4^{3}$$

Given, every page table should fit into one page,

$$2^{n} \geq \left(\frac{2^{34}}{2^{3n}}\right) * 4^{3}$$

$$\Rightarrow 2^{4n} \geq 2^{40}$$

$$\Rightarrow 4^{*n} \geq 40$$

$\Rightarrow n \geq 10$ i.e., n = 10 (for minimum page size)

Page size = 2^{n} bytes = 2^{10} bytes = 1 KB

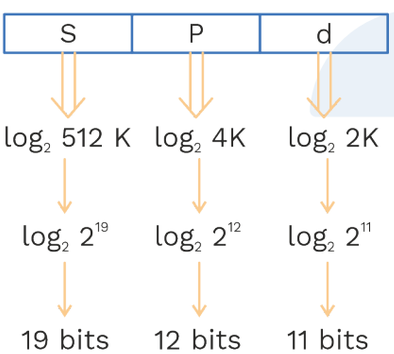
Q16 **Consider a system where the logical address space is divided into 512 K segments. Each segment is divided into 4K equal-sized pages. Each page stores 2 KB information. What are the storage capacities of logical and physical address spaces, respectively? (Assume byte–addressable memory and page table entry size is 2 B).**

a) 4 TB, 128 MB
b) 2 TB, 512 MB
c) 4 TB, 256 MB
d) 8 TB, 128 MB

Sol: **Option: a)**

Logical address structure :	S	P	d

S: The number of bits required to represent all segments of LAS.
P: The number of bits required to represent all pages in one segment.
d: The number of bits required to represent page size of the segment.



- Length of logical address = (19+12+11) bits = 42 bits
- Size of logical address space = 2^{42} B = 4 TB

Physical address structure:	F	d

F: The number of bits required to represent all frames uniquely in physical address space.
d: The frame size of physical address space.
Since, page table entry size is 2B = 16 bits.
- F = 16 bits as page table entry contains frame number.

In segmented paging:
Frame size = Page size of segment
d = 11 bits
The size of physical address = (16+11) bits = 27 bits
Size of physical address space = 2^{27} B = 128 MB

Previous Years' Question

Q: In a system with 32-bit virtual addresses and 1 KB page size, use of one–level page tables for virtual to physical address translation is not practical because of

a) The large amount of internal fragmentation
b) The large amount of external fragmentation
c) The large memory overhead in maintaining page tables
d) The large computation overhead in the translation process

Sol: c) **(GATE CSE-2003)**

Virtual memory:

Definition

Virtual memory is a technique that lets non-completely in-memory processes to be executed.

Several memory management solutions aim to keep many processes in memory at the same time to allow multiprogramming, but they require the full process to be in memory before it can run.

- Virtual memory alleviates programmers' concerns about memory and storage constraints.
- It enables processes to easily share files and implement shared memory.

There are numerous advantages to being able to run a program that is only partially in memory.

1) At any point in time, the size of a program can be larger than the available memory.

2) More programs could be run if each program required less physical memory runs at the same time, resulting in a rise in CPU utilisation and memory usage throughput.
3) Because less I/O is required to load or swap user programs into memory, each user program can be loaded or swapped faster. The user software would run more quickly.

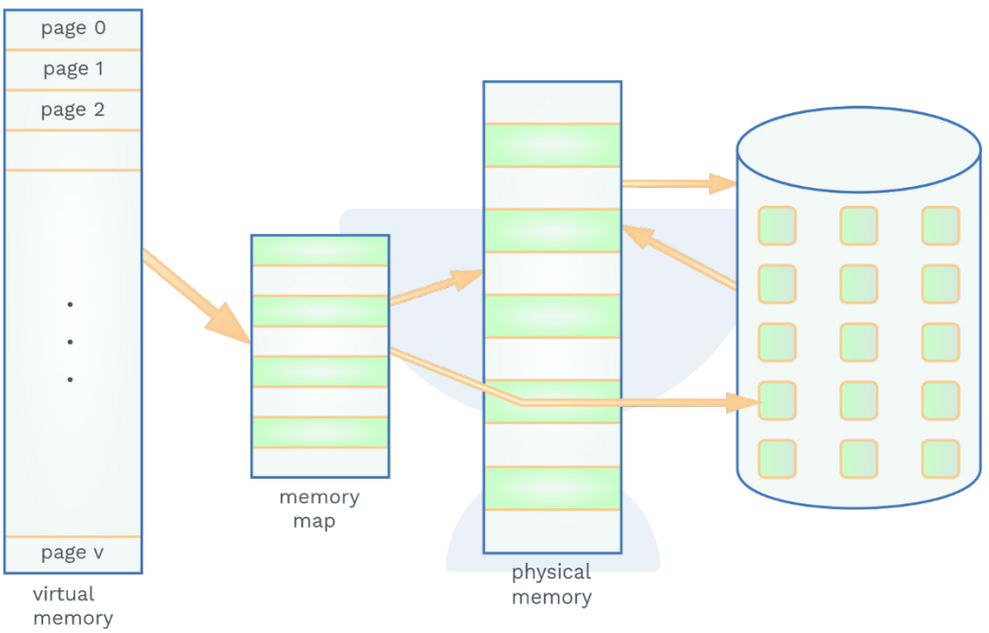


Fig. 5.14 Diagram Showing Virtual Memory that is Larger than Physical Memory

- Virtual memory, in addition to separating logical and physical memory, allows two or more processes to share files and memory via page sharing.

Advantages

- System libraries can be shared by many processes using a virtual address space mapping of the shared item.
- Virtual memory enables one process to generate a memory space that can be shared with another. Although the virtual address space of processes sharing this region is considered part of their virtual address space, the physical pages of memory is shared.
- Using the fork() system call, pages can be shared during process creation, speeding up the process creation process.

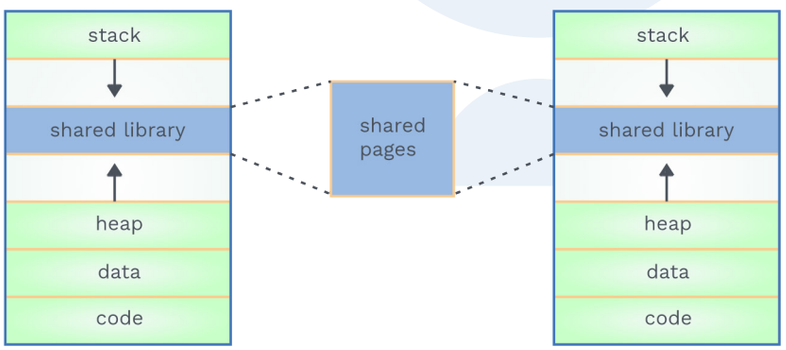


Fig. 5.15 Shared Library Using Virtual Memory

- Within a process, all memory references are logical addresses that are dynamically transformed into physical addresses at run time. This means that during the execution of a process, it can be swapped in and out of main memory, occupying different locations in main memory at different moments.
- A process can be divided down into multiple pieces, and these fragments do not need to be kept in the main memory throughout execution. This is possible because of a mix of dynamic run–time address translation and the use of a page or segment table.

Demand paging:

Demand paging is the process of loading a page into memory on demand (when a page fault occurs).

- However, determining the pages to be kept in the main memory and the pages to be kept in the secondary memory is quite a challenge since to predict when a process will require a specific page at a specific time is unknown.
- It recommends that all frame pages be kept in secondary memory until they are needed, and that no page be loaded into the main memory until it is needed.

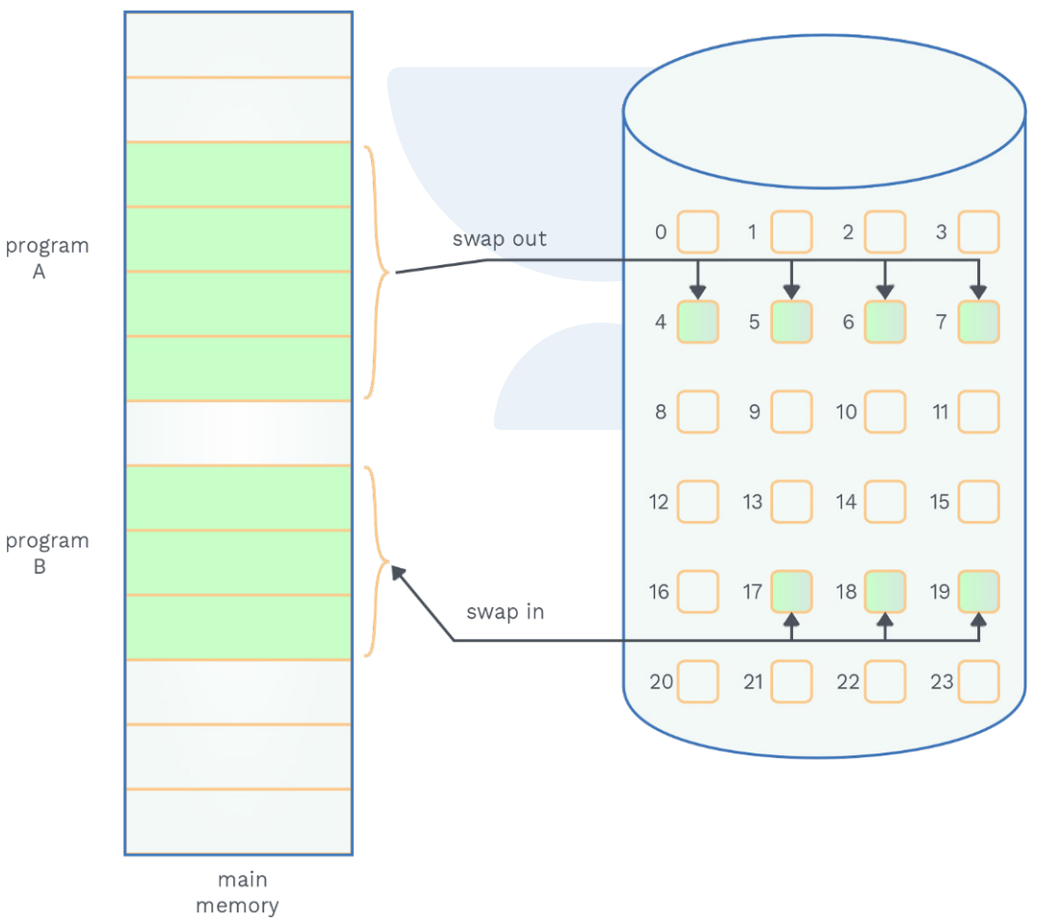


Fig. 5.16 Transfer of a Paged Memory to Contiguous Disk Space

- The operating system guesses which pages will be used when a process is swapped in before it is swapped out again. Instead of storing the entire process, the operating system saves only those pages.

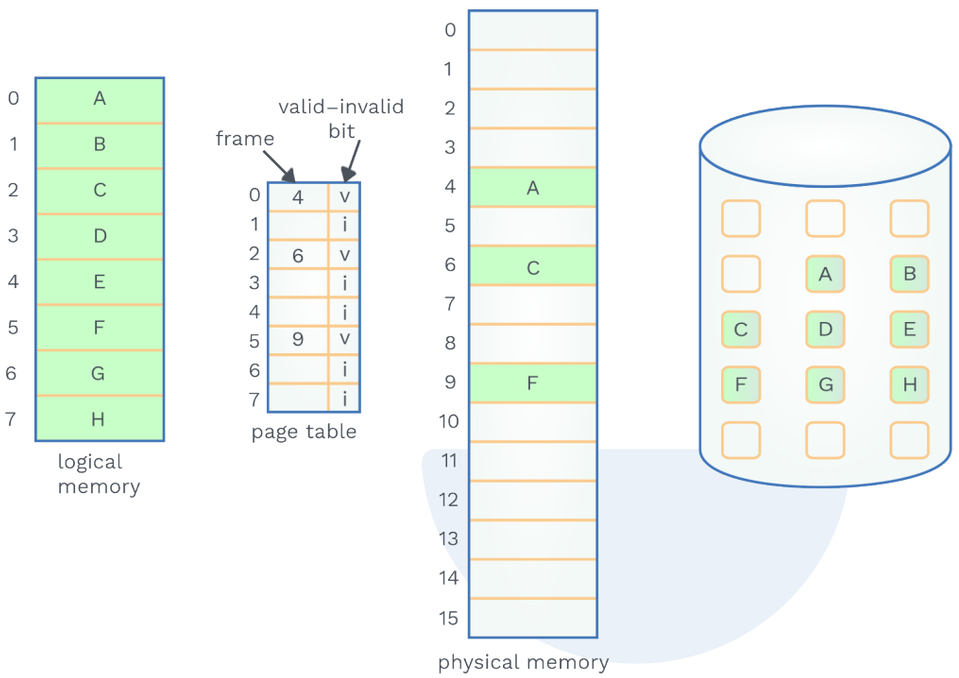


Fig. 5.17 Page Table When Some Pages are Not in Main Memory

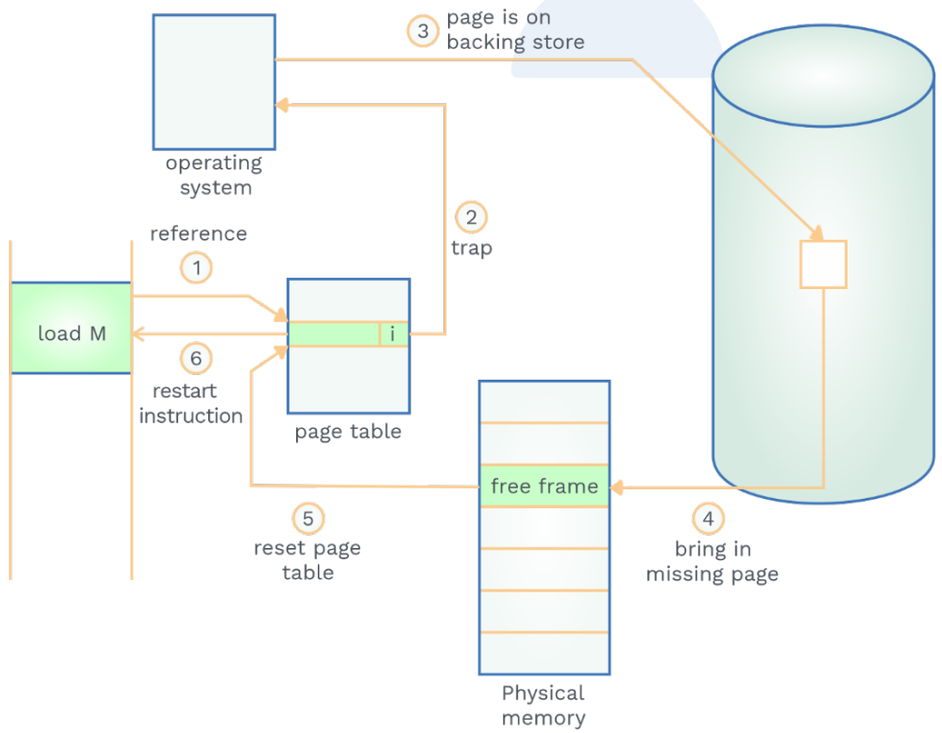


Fig. 5.18 Steps in Handling a Page Fault

- When a program accesses a page that hasn't been loaded into memory, it causes a page fault. While translating the address through the paging hardware.
- It will check the valid–invalid bit in the page table, and if the invalid bit is set, it will produce an error.
- Trap to the operating system, after which a page fault is handled.

The following is the technique for dealing with a page fault:

Technique for dealing with a page fault

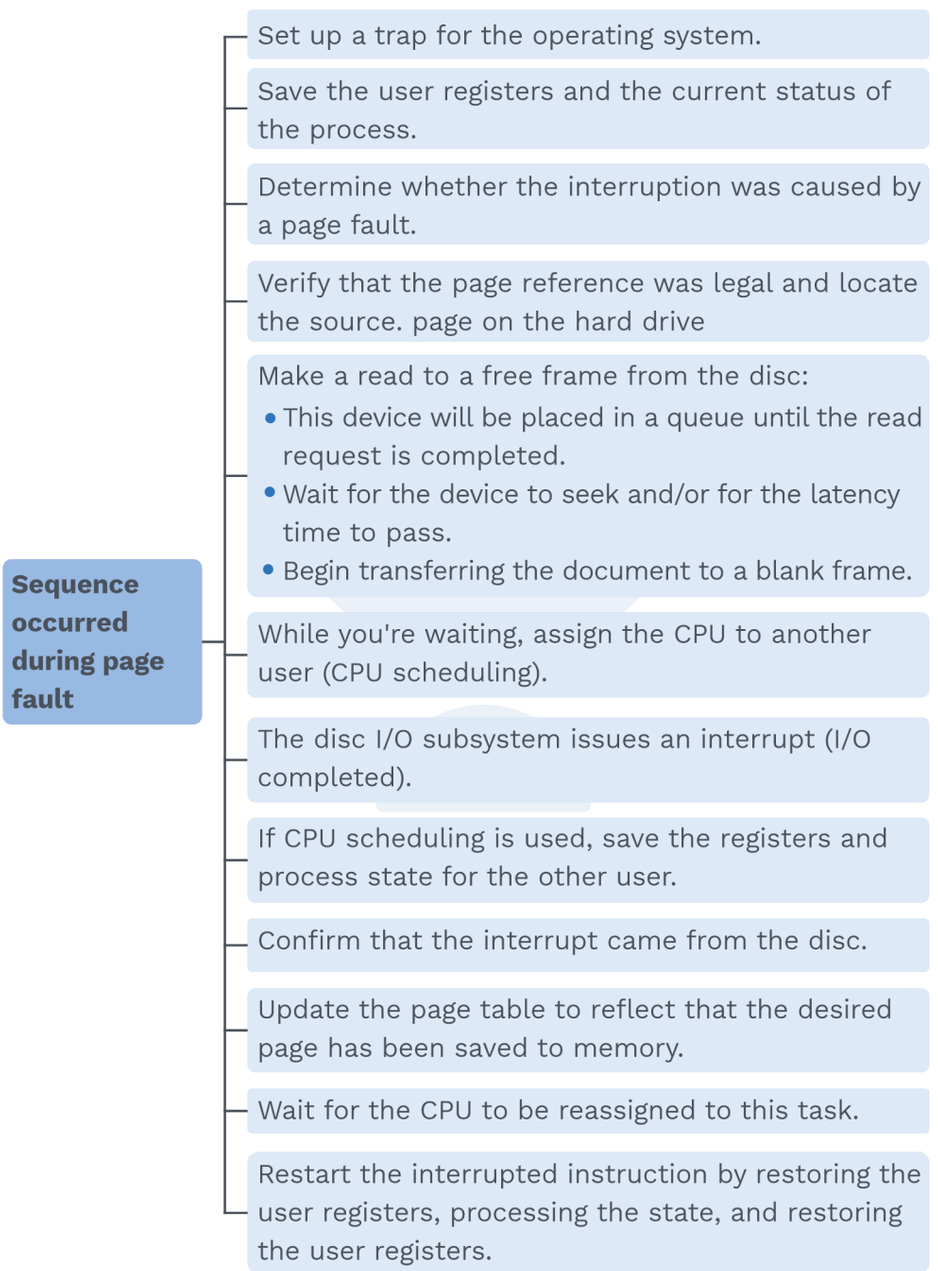
- It looks for this in an internal table (within the process control block). procedure for determining whether a reference was legitimate or not, access to memory
- We stop the process if the reference is invalid. If it was legitimate, but we hadn't yet imported that page, we did so now.
- From the list, we select a free frame.
- A disc operation is scheduled to read the desired page into the newly allocated frame.
- Once the disc read is complete, we update the process' internal table as well as the page table to reflect that the page has been loaded into memory.
- The instruction that was halted by the trap is restarted. The page can now be accessed as if it had always been in memory by the process.

Performance of demand paging:

p = Probability of page fault ($0 \le p \le 1$)

$$\text{EMAT} = (1-p)m + p \times (\text{page fault service time})$$

m = memory access time



- In every scenario, all of these actions are required. However, three components of the page fault service time slows down the system's overall performance.
  1) Service the page–interrupt the fault.
  2) Read the text on the page.
  3) Begin the process all over again.

SOLVED EXAMPLES

Q17 **Consider a system using demand paging architecture that takes 10 ms to serve a page fault the main memory access time is 3 ms. The maximum acceptable page fault rate (in percentage rounded off up to 2 decimal digits) to get the effective memory access not to be more than 3.70 ms is?**

Sol: Range: 10.00-10.00

Let P be the page fault rate.

EMAT $\geq$ P (page fault service time) + (1 – P) (m)

$$3.70 \geq P\,(10) + (1 - P) \times 3$$

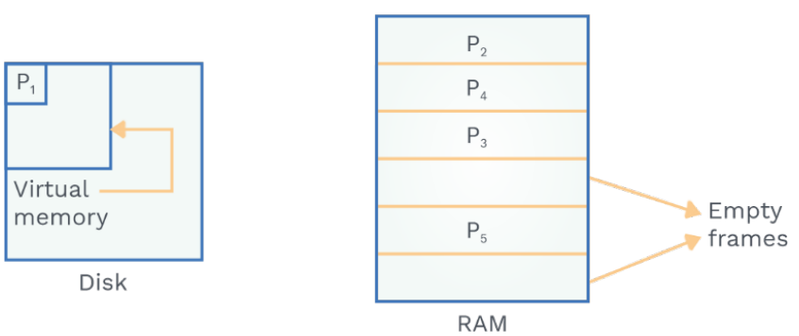
$$.70 \geq 7P$$

$$P = \frac{.7}{7} = 10\%$$

Page fault:

Two cases arises in paging when page fault occurs

Case 1: Empty frame available in physical memory.

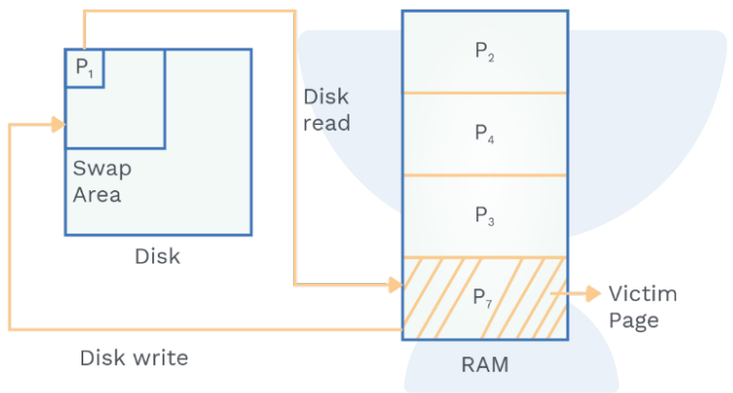


Here P_1 page will move from the swap area on the disk to one of the free frames, and the process will continue executing after page fault.

Case 2: No empty frame in physical memory

Here when a page fault occurs, and process requested page is brought into RAM from the swap area of the disk, there is not a single frame free to

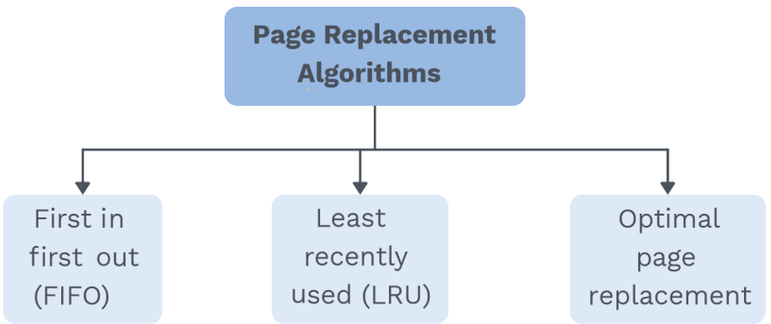
accommodate the page. So some page replacement algorithm is used that selects a victim page of some process that swaps out, and the desired page occupies its space. Victim pages are checked whether they are dirty pages or clean pages. A dirty page is a page when some modification of data has happened. Since it was last brought in memory. A clean page is when no data is altered on the page. A dirty page needs a write back to the disk when it becomes a victim page for some page replacement algorithm. It is a costly operation for the execution of the process as disk read/write are slower significantly in terms of physical memory read/write, whereas clean page don't need a write back to disk. So only disk reads happen in which desired page overwrites the victim page.



Grey Matter Alert!

- Whether the victim page is a dirty page or a clean page can be identified by dirty bit in page table of a process.
- If the dirty bit is set by hardware then it means the page is modified, and if it is replaced, it must be written back to the disk.
- Many systems keep a small list of clean pages that are available immediately for a replacement.

Page replacement algorithms:



1) First in first out (FIFO):

- This is the most basic page replacement method. The operating system maintains a queue for all pages in memory, with the oldest page at the front of the list to be replaced. When a page has to be replaced, the first page in the queue is chosen for replacement.
- On one side, the page replaced may be an initialization part that was utilised a long time ago and is no longer needed, while on the other hand, it may contain a widely used variable that was initialised early and is in constant use.

SOLVED EXAMPLES

Q18 **Consider the page reference strings 1, 7, 0, 7, 5, 6, 3 and the process's allocation of three - page frames. Determine the total number of page errors. Assume you're using the FIFO page replacement technique.**

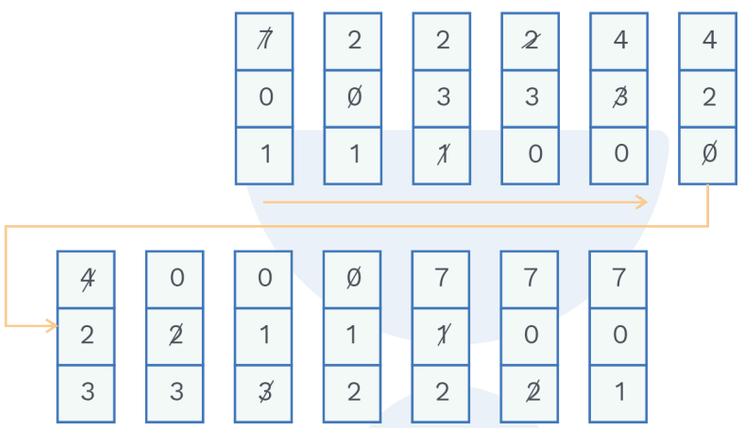
Sol: Range: 6–6

1	7	0	7	5	6	3
		0	0	0	0	3
	7	7	7	7	6	6
1	1	1	1	5	5	5
Miss	Miss	Miss	Hit	Miss	Miss	Miss

Total page fault = 6

Q19 **Given a page reference string 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1. Determine the total number of page errors, when the system allocates three frames to the process and employs the FIFO page replacement technique to replace the pages.**

Sol: **Range: 15-15**



so total fault in string are

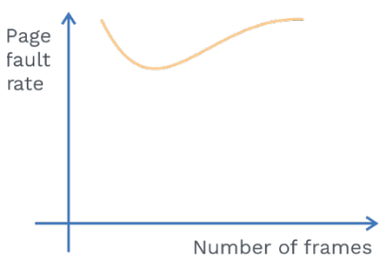
7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

Pages with square box were page fault so total number of page faults are 15.

Note:

If we increase the number of pages frame from 3 to 4. Then the total number of page fault will be 10.

Increasing the number of page frames can sometimes result in an increase in the number of page faults.



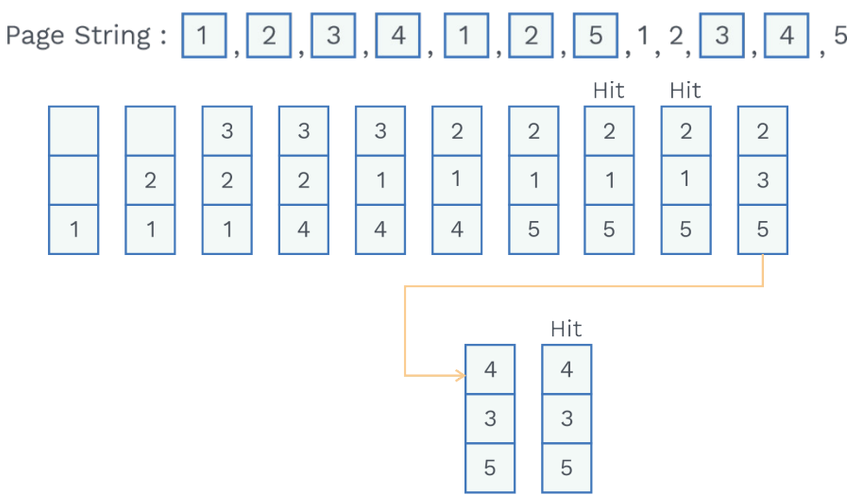
Belady's anomaly:
The phenomenon known as Belady's anomaly, it occurs when the number of page frames for a given memory access pattern increases, resulting in an increase in the frequency of page faults. Belady's anomaly affects FIFO policies and other non-stack based algorithms.

Why does this happen?

- The other two algorithms, Optimal and LRU, are widely utilised, but they never suffer from Belady's anomaly since they are stack-based.
- A stack-based algorithm is one that can display a set of pages in a stack.
- Memory for N frames is always a subset of the total number of pages in the in the total number of N + 1 frames of memory.
- The set of pages in RAM in an LRU policy would be the n most recently referenced pages. If the number of frames grows, these n pages will continue to be the most recently referenced and will remain in memory.
- So, only the FIFO page replacement policy suffers from Belady's anomaly.

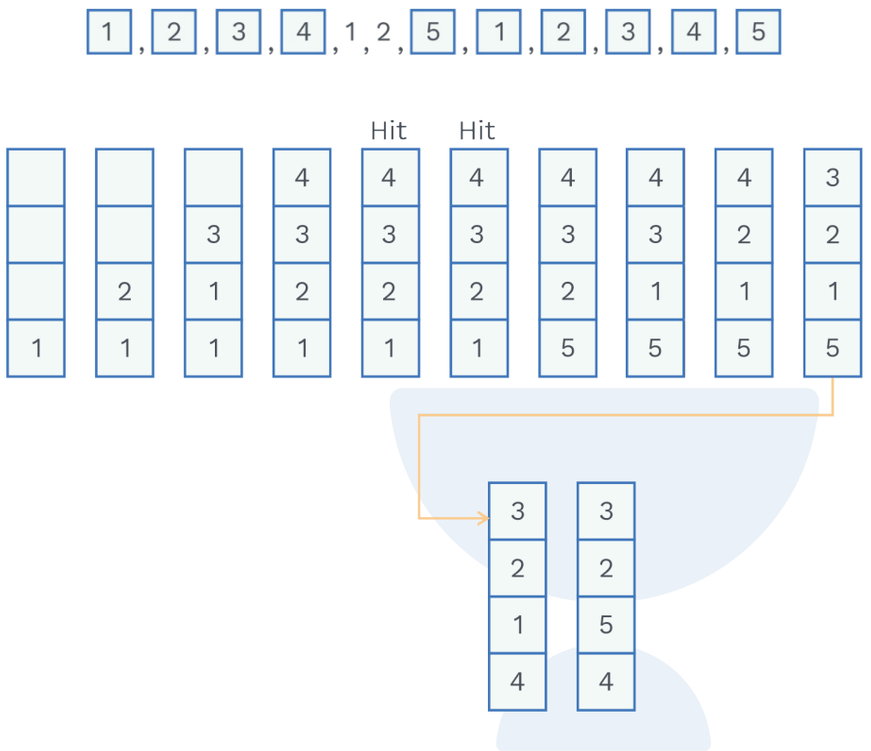
Example:
Consider the following page reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5. When the number of page frames assigned is 3 and 4, calculate the number of page faults.

Sol:
Case 1: M = 3 (Page frames)
First, let the page frame number be 3.



Number of page faults = 9.

Case 2: M = 4 (Page frames)



The total number of the page faults = 10.

Hence, for the given reference string number of page frames are four but the number of page faults is 10.

2) Least recently used (LRU):

In the LRU page replacement policy, if all the frames are in use and a page is asked by the executing process, than the frame to be chosen for replacement is the one which has not been accessed least recently. The LRU page replacement policy is effective. The main issue is figuring out how to replace LRU frames. An LRU page–replacement method needs hardware support.

Implementation:

A stack of page numbers is kept here. When a page is referenced, it is moved to the top of the stack from the bottom. As a result, the most recently used page always appears at the top of the stack, during the least recently used page appears at the bottom.

It's better to use a double linked list with a head and tail pointer to implement the aforesaid strategy. It is an expensive process to remove a page and place it on top of the stack; the tail pointer refers to the bottom of the stack, which is the LRU page.
LRU replacement, unlike FIFO, is not affected by Belady's anomaly.

SOLVED EXAMPLES

Q20 **Given a page reference string 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 3 for a process. The system allocates four frames to the process. Find number of page faults if LRU policy is used.**

Sol: **Range: 6-6**
Page string: 7 0 1 2 0 3 0 4 2 3 0 3 2 3

			2	2	2	2	2	2
		1	1	1	1	1	4	4
	0	0	0	0	0	0	0	0
7	7	7	7	7	3	3	3	3
Miss	Miss	Miss	Miss	Hit	Miss	Hit	Miss	Hit

Total page fault = 6

3) Optimal page replacement (OPT):
The optimal page replacement policy has the lowest page fault rate among all the page replacement algorithms. Belady's anomaly is never a problem for it.
Pages that will not be used for the greatest period of time in the future are chosen for replacement using this technique.
Example:
Consider the page reference string 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, with four-page frames. Determine the total number of page errors. The best policy is used.

Sol:

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

			2	2	2	2	2	2
		1	1	1	1	1	4	4
	0	0	0	0	0	0	0	0
7	7	7	7	7	3	3	3	3
Miss	Miss	Miss	Miss	Hit	Miss	Hit	Miss	Hit

Total page faults = 6

SOLVED EXAMPLES

Q21 **Consider a system which implements an optimal page replacement policy for page replacement.**
Given a page reference string 2, 5, 7, 4, 5, 7, 3, 4, 6, 3, 4, 2, and three frames empty initially. Find the number of page faults.

Sol: **Range: 7-7**

In optimal page replacement policy, when a page fault occurs, the page which is going to be used last in the given page reference string is chosen for replacement.

(2)	(5)	(7)	(4)	5	7	(3)	4	(6)	3	4	(2)
		7	7	7	7	3	3	3	3	3	3
	5	5	5	5	5	5	5	6	6	6	6
2	2	2	4	4	4	4	4	4	4	4	2

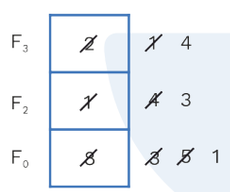
Page number encircled or resulted in page faults, so the total number of page faults is 7.

Q22 **For the reference strings 8, 1, 2, 3, 1, 4, 1, 5, 3, 4, 1, 4, 3, find the sum of page faults that occurred using FIFO, optimal and least recently page replacement algorithms with 3 -page frames. Given initially all frames are empty.**

a) 26
b) 27
c) 28
d) 32

Sol: **Option: a)**

FIFO :

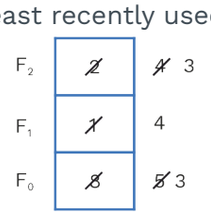


Total number of page fault
= 3 + 1 + 1 + 1 + 3 + 1 = 10

Optimal page replacement algorithm:

Total number of page fault
= 3 + 1 + 1 + 1 + 1 = 7

Least recently used:



Total number of page fault
= 3 + 1 + 1 + 1 + 1 + 1 + 1 = 9

The sum of page faults occurred using FIFO, optimal and least recently page replacement algorithms with 3 page frames = 10 + 7 + 9 = 26

Q23 **Which of the following is/are TRUE with respect to LRU page replacement strategy?**

a) LRU is useful for processes that exhibit spatial locality.

b) LRU incurs the overhead of maintaining an ordered list of pages and reordering that list.

c) LRU is useful for processes that exhibit temporal locality.

d) LRU incurs the overhead of maintaining a list of pages, and several ordered subsets of the list.

Sol: **Options: b), c)**

a) FALSE, LRU is useful for processes exhibiting temporal locality.

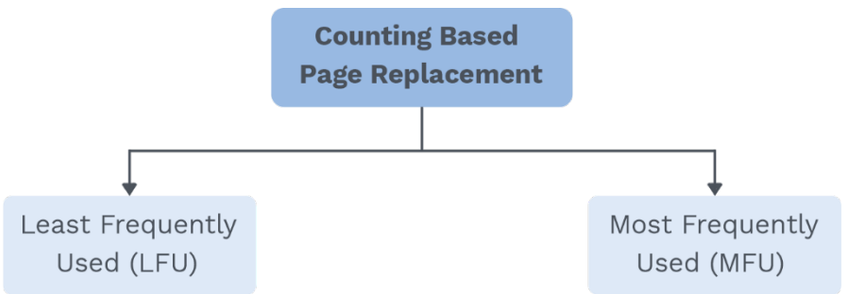
b) TRUE, LRU incurs the overhead of maintaining an ordered list of pages and reordering that list.

c) TRUE, LRU is useful for processes that exhibit temporal locality.

d) FALSE

Counting based page replacement:

Apart from FIFO, LRU, OPT page replacement algorithms, there are few more algorithms exists that can be used.



1) Least frequently used (LFU):

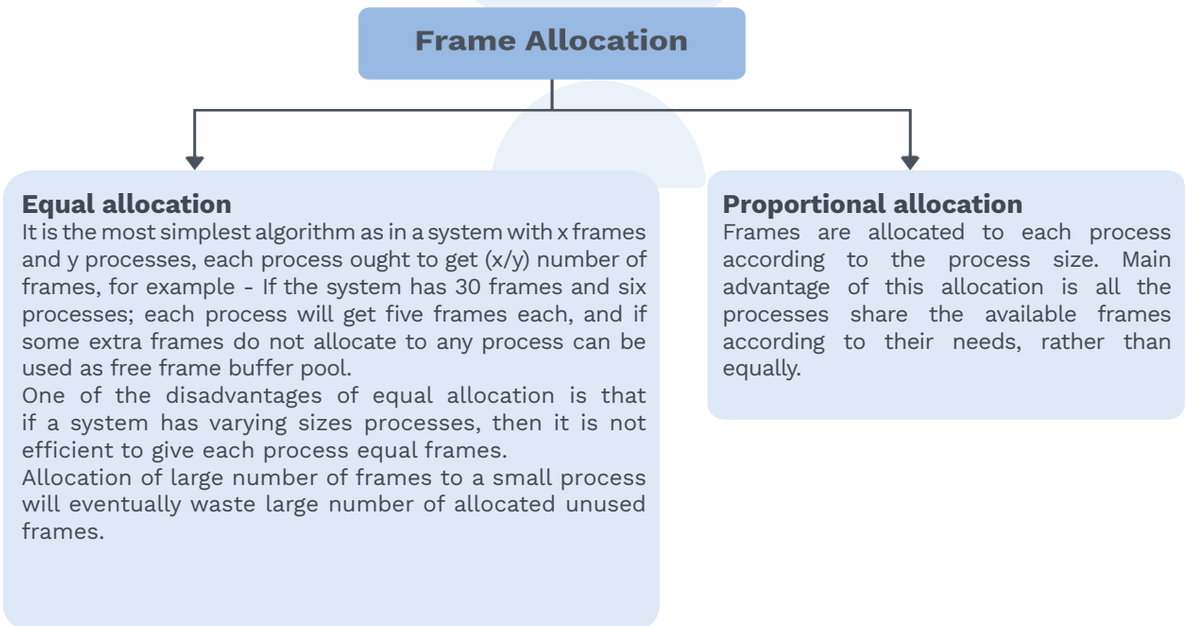
- It is necessary to replace the page with the lowest count. This choice is that a frequently visited page should have a high number of references.
- The issue arises when a page is frequently used during the first part of a procedure but is seldom used again.
- If it was widely used, it will have a large count and will remain in RAM even if it is no longer needed.

2) **Most frequently used (MFU):**
It's the page replacement method, which assumes that the page with the lowest count was probably freshly added and hasn't been used yet. It's the polar opposite of the least-often-used page replacement algorithm.

Allocation of frames:
Page replacement techniques and frame allocation algorithms are required when using demand paging. If you have numerous processes in physical memory, frame allocation methods are employed in the system to get the number of page frames that need to be allotted to each process.
Various challenges that the solutions for frame allocation encounter. You can't use more than the entire amount of frames available.

- Each process should be given at least a certain amount of frames.

Frame allocation algorithms:



Thrashing:
We must suspend a process execution set if the number of frames allocated to it falls below the minimum number required by the computer

architecture, as it will page fault fast. At this stage, it must take the place of one or more pages.
As a result, page faults recur in the same manner. Swap–in and Swap–out of pages will enhance disc activity while decreasing CPU usage. This is referred to as thrashing.

Cause for thrashing:

- The act of thrashing has a harmful effect on the system's performance. When CPU utilisation is low; the OS increases the degree of multiprogramming and employs a global page replacement algorithm that replaces pages independent of their order.
- To change pages in and out, these faulting processes must employ the paging device.
- CPU utilisation will be decreased when a lot of pages are queued up.
- The degree of multiprogramming will be increased. It indicates that more new processes will try to start by stealing resources from existing ones. Resulting in more page faults and a longer paging device queue.
- As a result of the foregoing scenario, system throughput plummets, and the fault rate skyrockets, and effective memory access time skyrockets while no work is being done.

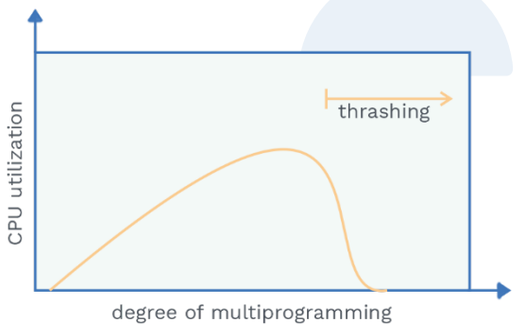
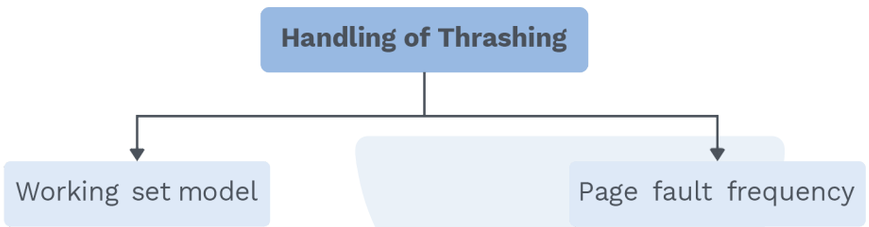


Fig. 5.19 Thrashing

- We can reduce the consequences of thrashing by using the local replacement technique, but it won't fully eliminate it because processes will still queue for the paging device most of the time. We must provide a process with as many frames as it requires to avoid thrashing. There are numerous methods for determining how many frames an operation requires. Working set strategy, for example, begins with determining how many frames a process truly employs. This method establishes a process execution locality model.

- **Locality model:**

We can reduce the consequences of thrashing by using the local replacement technique, but it won't fully eliminate it because processes will still queue for the paging device most of the time. We must provide a process with as many frames as it requires to avoid thrashing. There are numerous methods for determining how many frames an operation requires. Working set strategy, for example, begins with determining how many frames a process truly employs. This method establishes a process execution locality model.



1) Working set model:

- The basic idea of is that the process will fail only when it'll move to new location when enough number of frames are provided to store its current location locality.
- However, if the assigned frames are smaller than the current frame size, the process is doomed to thrash due to its proximity.
- The most frequently used pages would always become part of the working set.
- The working set's accuracy is determined by the size of parameter. If is too large, the working set may overlap, and if it is too small, the locality may not be completely covered.

Page reference table:

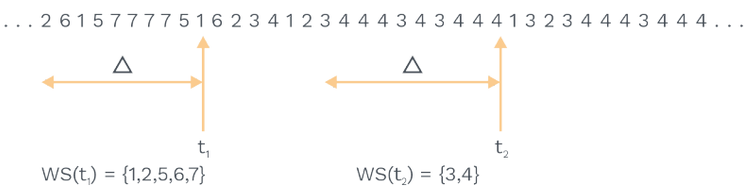


Fig. 5.20: Working Set Model

- Keeping track of the working set is a challenge with the working set model.
- The working set window moves about; a new reference appears at one end of each moving reference, while the oldest one fades away.

2) Page fault frequency:

- It's a more straightforward approach to dealing with thrashing.
- It reduces page fault, even if and page fault rate is high, it consider the process as failed.
- There aren't enough frames. Because a low rate of page faults indicates when there are two or more frames in a process, a precise upper and a lower limit should be set.
- Set the page fault that you want.

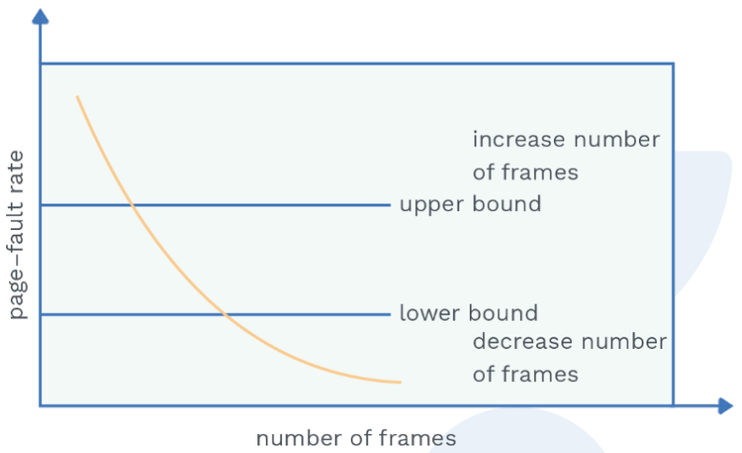


Fig. 5.21 Page–Fault Frequency

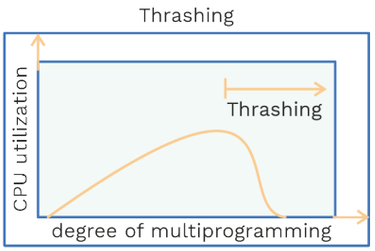
- If, in any case page fault rate < lower limit, then frames will be deleted, otherwise the process will be provided many numbers of frames.
- If there are no free frames and the page fault is high, some processes can be halted, and frames allotted to them can be reallocated.

SOLVED EXAMPLES

Q24 **Which of the following is TRUE about Thrashing?**
a) Thrashing always decreases degree of multiprogramming
b) Thrashing reduces page I/O
c) Thrashing implies excessive page I/O
d) None

Sol: **Option: c)**

In thrashing, maximum CPU time is wasted in I/O and page faults.



If degree of multiprogramming (DOM) gets increased beyond a certain limit, thrashing occurs. Thrashing doesn't decrease the DOM.

Q25 **Which of the following options is/are true with respect to thrashing?**
a) Thrashing decreases the CPU utilisation
b) Thrashing is about spending more time in paging than executing
c) The working set model in memory management is used to implement the concept of thrashing
d) Thrashing leads to high page fault rate

Sol: **Options: a), b), d)**
c) False. The working set model in memory management is used to implement the concept of the "principle of locality".
All other options are true.

IMPORTANT FORMULAE

Organization of the (LAS) logical address space:

If address size of LAS is n bits, and word size is of 1 byte

LAS = 2^n bytes

Logical address (LA) = $\log_2$ (LAS) bits = $\log_2$ (2^n) bits = n bits

Given, the size of each page is K bytes

$$\text{Number of pages in LAS} = \frac{\text{LAS}}{\text{Page size}}$$

Given, the size of each entry of page table as e bytes

$$\text{Then, the size of the Page Table} = \frac{\text{LAS}}{\text{Page size}} * \text{e Bytes}$$

Optimal page size to minimize page table size and internal fragmentation is given as

$$\sqrt{2 * \text{LAS} * \text{e}}$$

Organization of physical address space (PAS):

If address size of PAS is n bits, and word size is of 1 byte

PAS = 2^n bytes

Physical address (PA) = $\log_2$ (PAS) bits = $\log_2(2^n)$ bits = n bits

Given, frame size = K bytes

$$\text{Number of frames in PAS} = \frac{\text{PAS}}{\text{Frame size}}$$

Performance of multi–level paging:

Given main memory access takes m ns.

The average memory access time with n–level paging = (n+1) m ns

Performance of multilevel paging with TLB:

Given, TLB hitting rate = x, Time to access TLB is c

Memory access time = m

Average memory access time with n–level paging and a TLB

= x (c + m) + (1 – x) (c + (n + 1) m)

Performance of segmentation:

Given, main memory access time be 'm'.

Effective memory access time of segmentation (EMAT) = 2m

x = TLB hit ratio

c = TLB access time

m = memory access time

The average memory access time using segmentation and TLB is given as,

x (c + m) + (1 – x) (c + 2m)

concept of the "principle of locality".
All other options are true.

Chapter Summary

- Basics of memory management: – **1)** Registers
  2) Cache
  3) RAM memory
  4) Secondary memory
- Address spaces: – **1)** Logical address space
  2) Physical address space
- Linking: – It is used to generate a single executable code which is obtained by combining the different object files and library functions generated by the compiler.
- Loading: – It is used for bringing the executable code from secondary memory and loading it into the main memory for its execution.
- Swapping: – Movement of process between physical and secondary memory.
- Contiguous memory allocation: – **1)** Fixed partitioning
  2) Variable partitioning
- Allocation policies: – **1)** Best fit
  2) First fit
  3) Worst fit
  4) Next fit
- Non contiguous memory allocation: – **1)** Paging
  2) Segmentation
  3) Segmented paging
  4) Paged segmentation
- Virtual memory: – **1)** Demand paging
  2) Page replacement algo
  - FIFO
  - LRU
  - OPT

- Frame allocation algorithms – **3)** Belady's anomaly
  1) Equal allocation
  2) Proportional allocation
  3) Local allocation
  4) Global allocation
- Thrashing – If number of frames allocated to the process are lesser than what process

6 File System and Disk Storage

6.1 BASICS OF FILE

- The file system provides an abstract data structure of the file present in the hard disk (secondary storage).
- File system provides the online storage and access mechanism for both data and programs of the operating system.
- File system is a combination or collection of two main objects like directories and files.
- For stored information, the uniform logical view is provided by OS to define the file, i.e., logical storage unit, it (OS) abstracts the physical properties of its storage devices.
- File is a collection of related information, and it is last stage/smallest allotment which could be stored on the secondary storage. Any data could be stored only in the form of files onto secondary storage.
- Files have a defined structure, like for instance, a text file .txt which has a sequence of characters organized into lines, and likewise an (.exe) executable file which has code/programs.

File attributes:

A file is named to get it identified uniquely by its end users. A file has various attributes attached to it, which vary from one operating system to another. It typically consists of:

1) **Name:** The symbolic name of the file is the only information kept in readable form.
2) **Identifier:** A file is allocated in the system with a distinctive tag.
3) **Type:** It represents what type of file it is.
4) **Location:** The location of a file is the actual position on the disk where the file is stored.
5) **Size:** This gives info about the present file size, possibly maximum file size extension.
6) **Protection:** Access–control information determines who can read, write and execute.
7) **Time and date:** This attribute says about creation, last modified, last used times of a file.

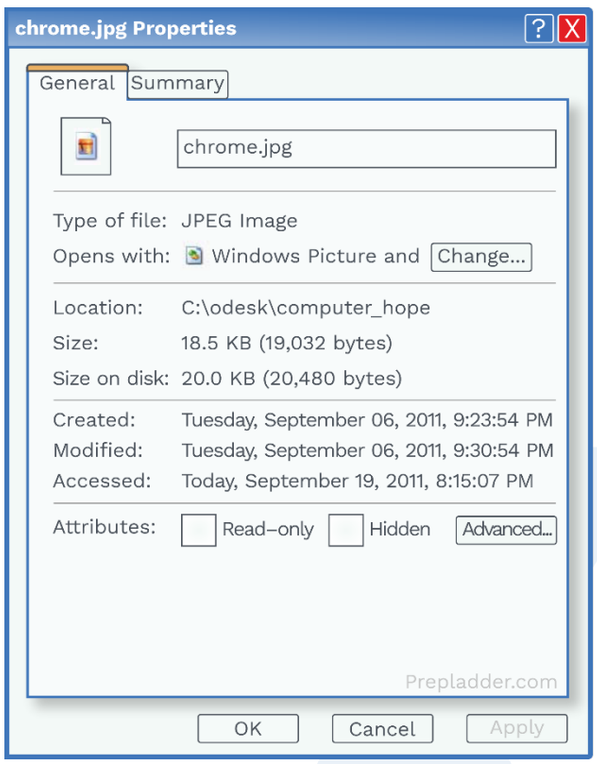


Fig.6.1 File Attributes

Note:

In a system with many files, the size of the directory itself may be in megabytes because files are non_volatile.

File operations:

A file is an abstract data type. Various types of operations can be performed on the file by the resident operating system using system calls like create, read, write, delete and truncate.

- **Creating a file:** For a file creation, the first thing to have is storage space and secondly need to enter the unmatched or unique file name in the directory, these are prerequisites for file creation.
- **Writing a file:** Writing on to the file is done using a system call with specifying both names of the file and the data to be written to the file. A writer pointer is kept to the location of the file from where or the position

which needs to be start writing and pointer need to be updated accordingly as per writes in the file.

- **Reading a file:** To read a file, a read pointer is provided by the OS.
- **Repositioning within a file:** For an appropriate entry, the directory is searched. The pointer is repositioned in the file to the given entry from the current position of the file. There is no I/O involvement for the repositioning of the pointer within a file. This operation is called file seek.
- **Deleting a file:** A file (content of file) is deleted along with its attributes, and space is released. The space is again reusable.
- **Truncating a file:** A file truncation includes the content removal from the file but not the complete deletion of the file. In file truncation, attributes of the file remains but only data/content from the file is removed.

Note:

Other common operations include appending new information to the end of an existing file and renaming an existing file.

Grey Matter Alert!

Open() system call is used by most of the OS in order to avoid constant searching. Information associated with an open() file system call.

1) Access rights

2) File's location on the disk

There are also operating systems which provide the file lockers. So, a file could be safe when it is a sharable resource. If a file is locked by a process, then it would not allow other processes to gain access.

SOLVED EXAMPLES

Q1 **Which of the following statements is/are TRUE?**
a) The file systems manage only secondary storage data
b) Native and mounted file systems must be similar in type
c) A file system creates a hard link to a file in a mounted file system
d) Tape storage is most appropriately managed by a sequential file organization

Sol: **Option: d)**
a) (FALSE) The file system manages files on secondary as well as main storage
b) (FALSE) One of the primary advantages of mounting a file system is to enable multiple heterogeneous file systems
c) (FALSE) Soft links are path name, and hard links are directory entries
d) Tape storage is accessed sequentially

File types:
A file is recognized by the operating system on its type and operates accordingly. The types could be like .doc, .txt, etc.
The extension type of a file indicates the type of operation we can perform on files.

File Type	Usual Extension	Function
executable	exe, com, bin or none	ready–to–run machine–language program
object	obj, o	compiled, machine language, not linked
source code	c, cc, java, pas, asm , a	source code in various languages
batch	bat, sh	commands to the command interpreter
text	doc, txt	textual data, documents

word processor	tex, rtf, doc	various word–processor formats
library	lib, a, so, dll	libraries of routines for programmers
print or view	gif, pdf, jpg	ASCII or binary file in a format for printing or viewing
archive	arc, zip, tar	related files grouped into one file, sometimes compressed, for archiving or storage
multimedia	mpeg, mov, mp3, mp4, avi	binary file containing audio or A/V information

Fig. 6.2 Common File Types

Directory:

- In a computer system, millions of files are stored on random-access storage devices, including the hard disks, optical disks, and memory-based disks.
- To organize these files properly, a directory structure is used by the file system.
- Directory is a collection of correlated files which keeps entries of all files.

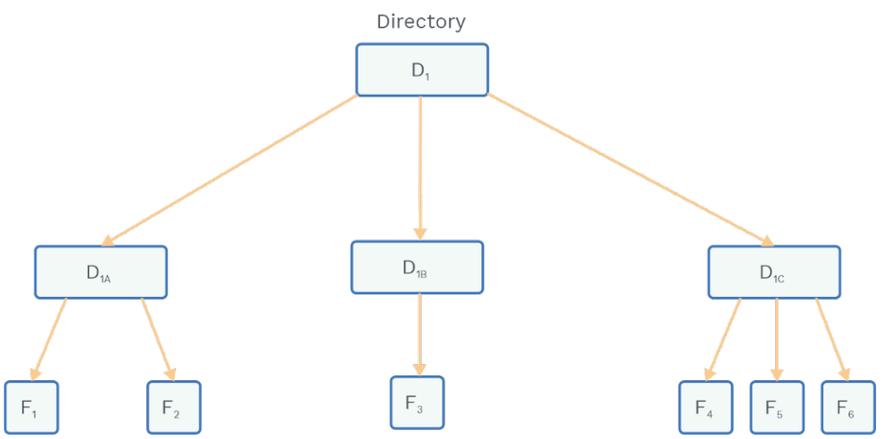
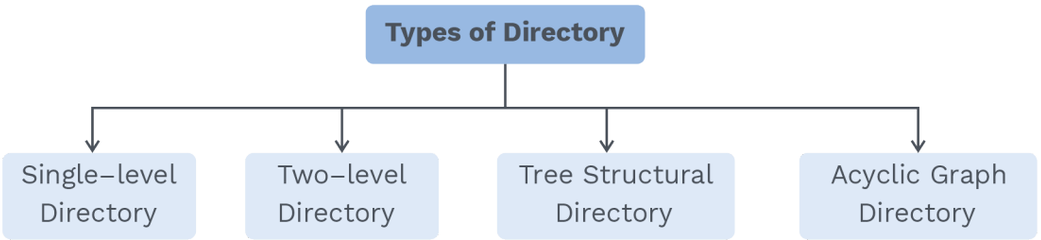


Fig. 6.3 Directory Structure

Types of directory structure:



Single–level directory:

- The simplest directory structure is the single level directory. Which consists of one directory with all the files in the same directory.

Advantages:

1) Since we have only one level structure, implementation becomes easy.
2) It's faster to search a file in this structure if the number of files is low.
3) File operations are easy.

Disadvantages:

1) It consumes more time if files are high in number.
2) Users cannot be separated.

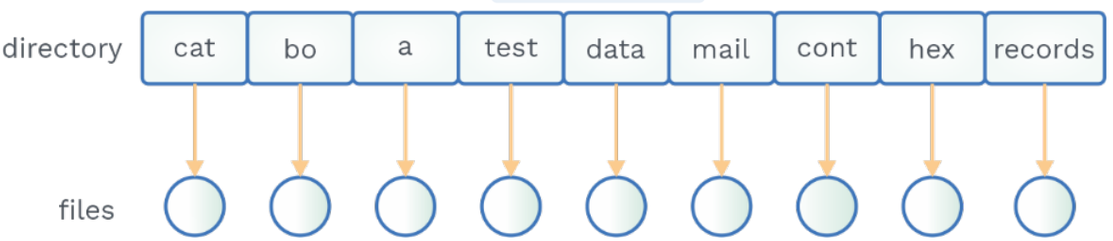


Fig. 6.4 Single-Level Directory

Two–level directory:

- Among multiple users, this (single level directory) structure gives confusion in file names.
- So, to overcome ambiguity, creating a separate directory for an individual user is good.
- Every user has UFD (user file directory) of their own.
- The structure of UFDs is similar, but each directory lists files of single user.
- For unique file naming purpose, user needs to know the path name of file desired.

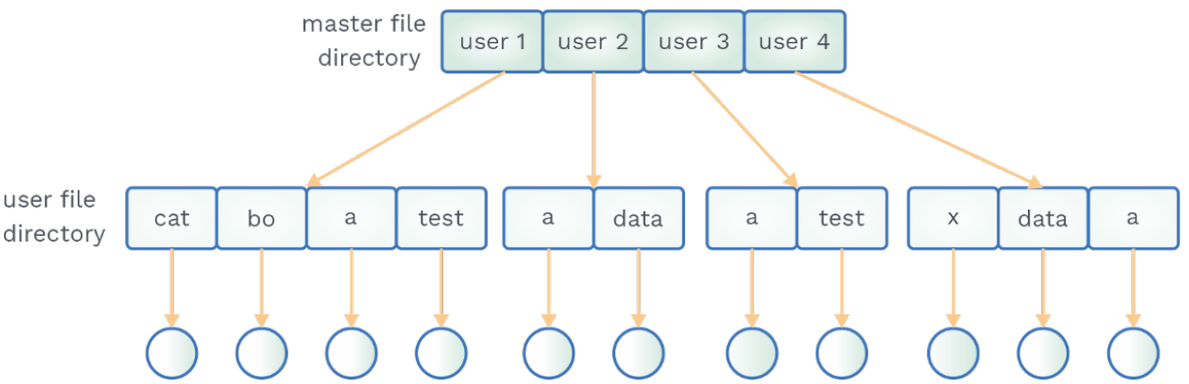


Fig. 6.5 Two-Level Directory

Advantages:

1) Isolation of users from one to another is effectively done in this structure. So, the same directory or same file name could exist for multiple users, because paths are different.

2) Searching of files become easier due to path name and user grouping.

Disadvantages:

1) The implementation of a directory structure is difficult.

2) Files are not permitted to share with users. Thus, if the same file exists in two different directories, then updates made in any one of the files are not reflected in the other file. This leads to the problem of inconsistency.

Tree-structured directory:

- Tree-structured directory is the extension of a two-level directory tree structure to a tree of arbitrary height.
- In this, users can create their own subdirectories and can keep their files in a more organized manner.
- Tree structure is the most frequently used directory structure.
- Unique file route/path for each file and root directory could be found in the tree structure.

Advantages:

1) The probability of name collision is very less as each file has different paths.

2) Full path name of the file is given, which makes searching very easy.

3) By using a relative search or absolute path search, we could find the file.

Disadvantages:

1) Implementation of this directory structure is more complicated.

2) It is not efficient since accessing a file may lead to accessing of multiple directories.

3) Sharing of directories and files is prohibited in this structure.

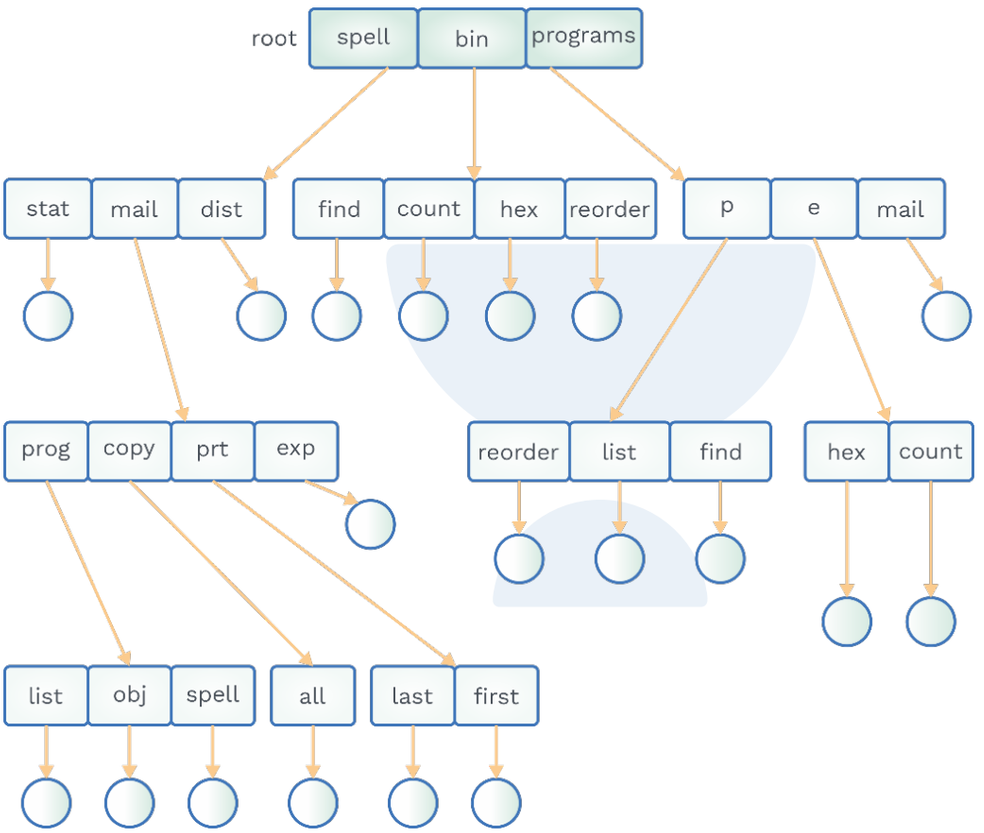


Fig. 6.6 Tree Level Directory

Acyclic-graph directory:

- An acyclic graph directory allows different users to share subdirectories and files.
- In two different directories, there may exist the same files or subdirectories. So, among various users or directories, common subdirectories could be shared.

Note:

Sharing subdirectories (files) is different from copying subdirectories (files) to another place; if a user alters shared subdirectories (files), the changes will get reflected everywhere.

Advantages:

1) There is no problem of inconsistency because files can be shared, and any changes made to one copy of the file are automatically reflected everywhere.
2) Searching a file is easy because it has a unique or individual path.
3) Since there is no cycle, simple graph algorithms can be used to traverse the graph.

Disadvantages:

1) Acyclic-graph directory structure is more complex than a simple tree structure.
2) Deleting files may create a problem because files are shared via links, and it may lead to a dangling pointer.

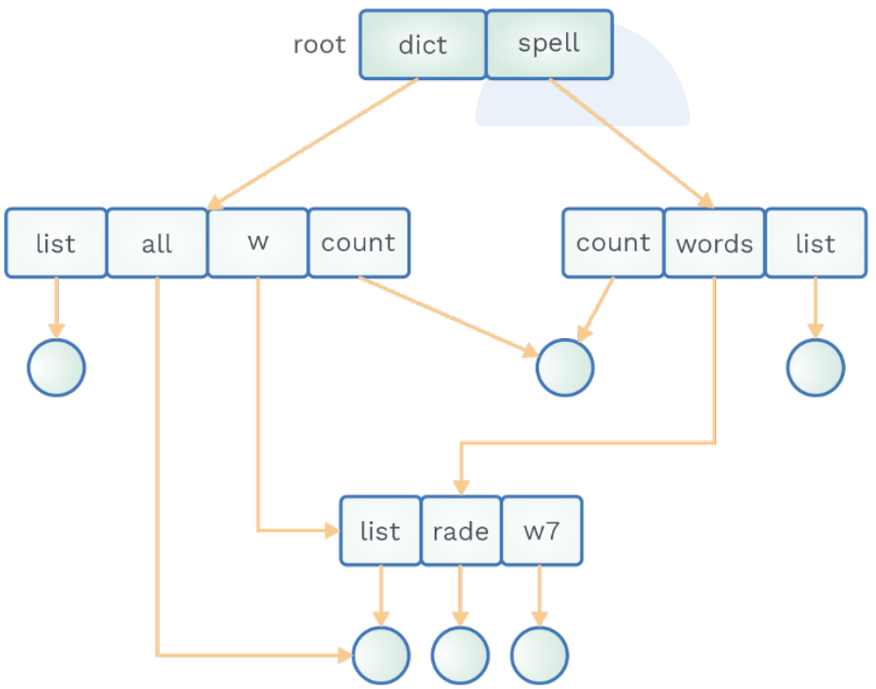


Fig. 6.7 Acyclic–Graph Directory Structure

General graph directory structure:

- Generally, in a graph directory structure, cycles can be present within a directory structure.
- If there is a cycle in the structure, then there are more than one path possible to reach to the file.

Advantages:

1) It allows cycle.

2) It is a flexible structure, since a file can be searched through many paths.

Disadvantages:

It is costly because we might need to design complex algorithms in order to avoid infinite loops while searching through the cycle.

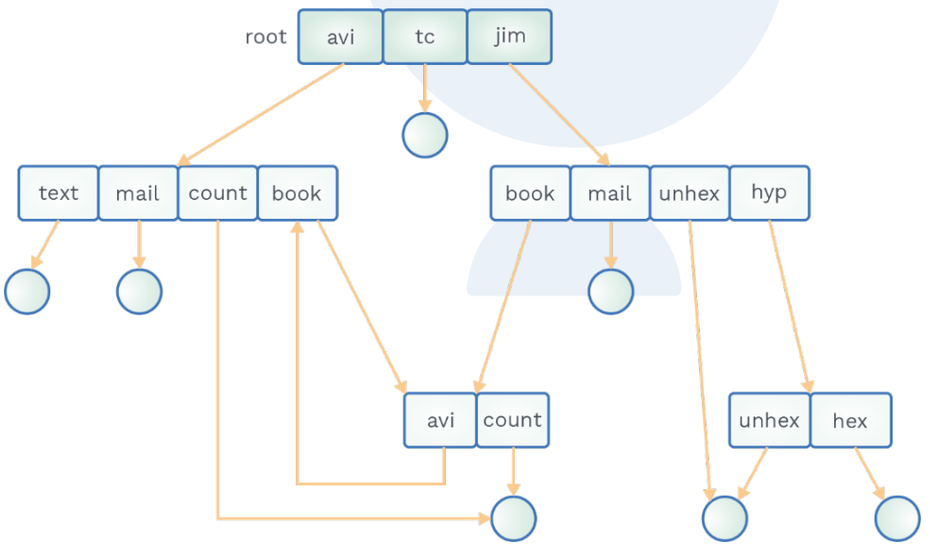


Fig. 6.8 General Graph Directory

SOLVED EXAMPLES

Q2 **Which of the following is/are FALSE?**

a) A hierarchical file system, must have a root directory irrespective of the operating system.

b) In a hierarchical file system a soft link is also known as a relative path.

c) In a hierarchical file system, for a file, multiple soft links but only one hard link can exist.

d) In a hierarchical file system, for a file, only one soft link but multiple hard links can exist.

Sol: **Options: b), d)**

a) (TRUE) A hierarchical file system must have a root directory irrespective of the operating system.

b) (FALSE) In a hierarchical file system, a soft link is a directory entry containing the pathname for another file. The relative path of a file is just a short path of the file in the current working directory.

c) (TRUE) In a hierarchical file system, for a file, multiple soft links but only one hard link can exist.

d) (FALSE) A hard link is a directory entry that specifies the location of the file on the storage device.

Access methods:

There are different methods that can be used to access the information within the file.

a) Sequential access:

Information in the file is accessed and processed orderly (sequentially), i.e., one record after other in this method.

Example: Compiler and editor access a file using sequential access.

Key points are:

1) Orderly, data is accessed one record after another record.

2) A read operation "read next", reads next portion of the file and advances the file pointer.

3) Most of the OS provide this method to access a file.

b) Direct Access:

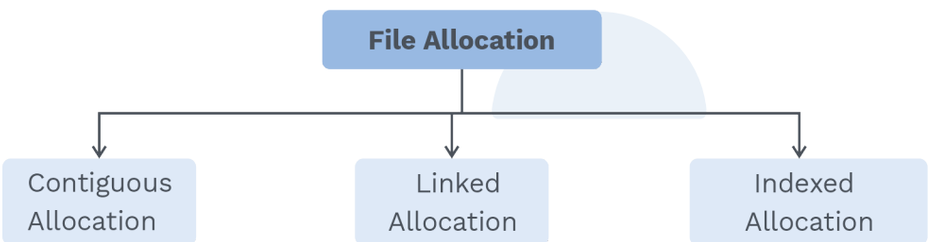
- It is also known as the relative access method.
- In this method, a file can be accessed randomly without any proper sequence(order).
- Random read/write is possible like read on block 10, then 56 and write on block 45 can be performed.

c) Index sequential method:

- Accessing of the file is done by using another file called index file, which is built on sequential method in this method.
- It generally involves index construction for the file, whereas the file contains a pointer to the various blocks of the file.
- For finding a record in the file, first search the index and later with the help of a pointer, directly access the file to find desired the record.

File allocation methods:

There are multiple ways to store a file in the disk blocks; three of them are discussed here:



Motive of file allocation:

Utilization of disk space in an effective way and accessing file blocks in fast manner.

a) Contiguous allocation:

- In this, whenever a file is created, it occupies a contiguous set of blocks on the disk. Let us consider a file that requires 'x' blocks, if a file is assigned to block 'p' then the file will occupy blocks p, p + 1, p + 2, p + 3, ... p + x - 1.
- If starting disk block address and file Length (in terms of blocks) are given, then we can easily find the number of occupied blocks by the file .,
- The directory entry for a file with contiguous allocation contains the first block address and file size.

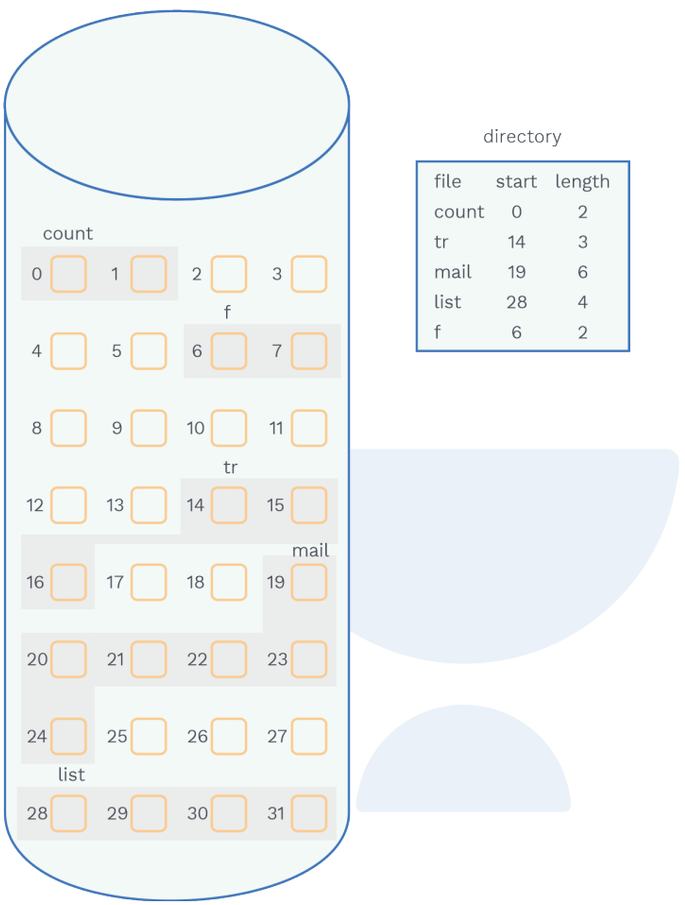


Fig. 6.9 Contiguous Allocation

Advantages:

1) Both sequential and random accesses can be done. For random access, the address of the X^{th} block of the file, which starts at block 'p' can easily be obtained as (p + X).
2) Due to sequential allocation, the number of seeks is minimal, which makes it extremely fast.

Disadvantages:

1) It suffers from external fragmentation because even if the disk block is free, it cannot be allocated, as it may not be a contiguous block.
2) Increasing the file size is difficult because the next contiguous block may or may not be empty.
3) Internal fragmentation may exist in the last disk block allocated to the file.

b) Linked list allocation:

- It is a non-contiguous allocation. In this type of allocation, each file acts as a linked list of disk blocks. Where blocks need not be in a contiguous manner.
- Here, a file could be allocated in disk blocks that are located anywhere on disk. In this allocation, directory contains starting and ending blocks pointers of the file.

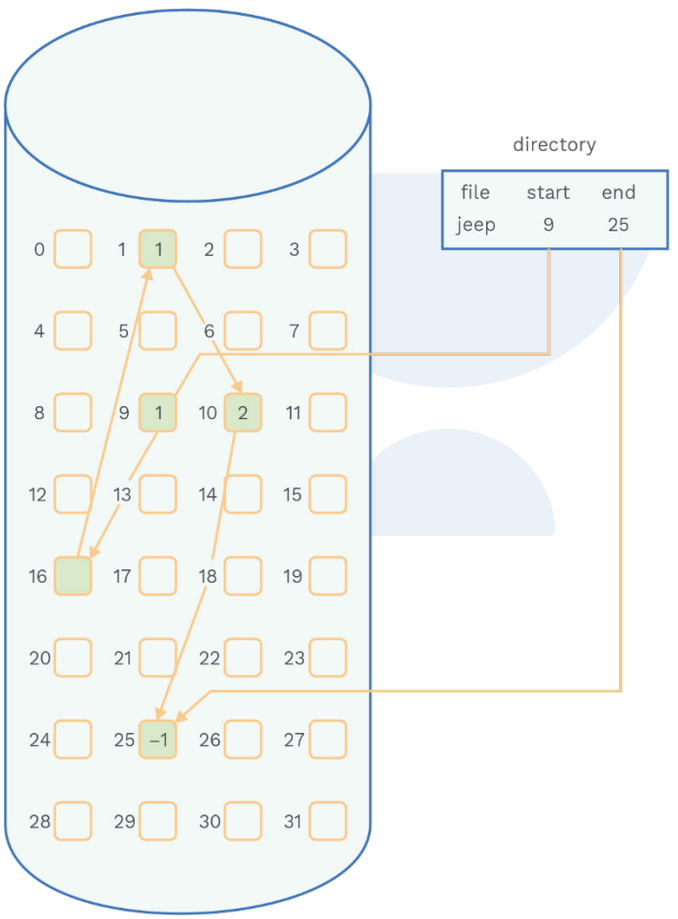


Fig. 6.10 Linked-List Allocation

Advantages:

1) This is flexible allocation because whenever a free disk block is present, a file block can be allocated in that block.

2) External fragmentation is not present. So, no space is lost due to disk fragmentation.

Disadvantages:

1) Need a large number of seeks to access the required disk block, since a file is scattered randomly on the disk.
2) A file can be accessed only in a sequential manner, as random access is not possible.
3) Maintaining the pointer for each disk block is considered as an overhead.

SOLVED EXAMPLES

Q3 **Consider a file which is stored in a disk and occupying disk blocks from 1 to 50. If currently R/W head is on 15th disk block, and a particular record is to be accessed, which is stored on disk block number 42. It takes 2ns to access a disk block.**
Let the time required to access the disk block containing the desired record using the contiguous file allocation method and using the linked file allocation method are x ns and y ns, then find the absolute difference between x and y.

Sol: **Range: 52 – 52**

a) Contiguous file allocation:
In this method, random access is possible.
So, from 15th disk block, we can directly access 42nd disk block.
Hence, only single disk block access is required.
Access time = 2ns

b) Linked file allocation:
In this method, random access is not possible.
So, from 15th block user needs to access all the blocks to reach 42nd block.
Total block access = 27
Access time = 27 * 2 = 54 ns
The absolute difference between x and y is 54 - 2 = 52 ns.

c) Indexed allocation

- In this, every file will have a special block which consists of the pointers to all blocks of that file. This special block is an index block.
- The ith entry of the index block of a file have the adisk address of the ith block of that file. Here each file will have index block.

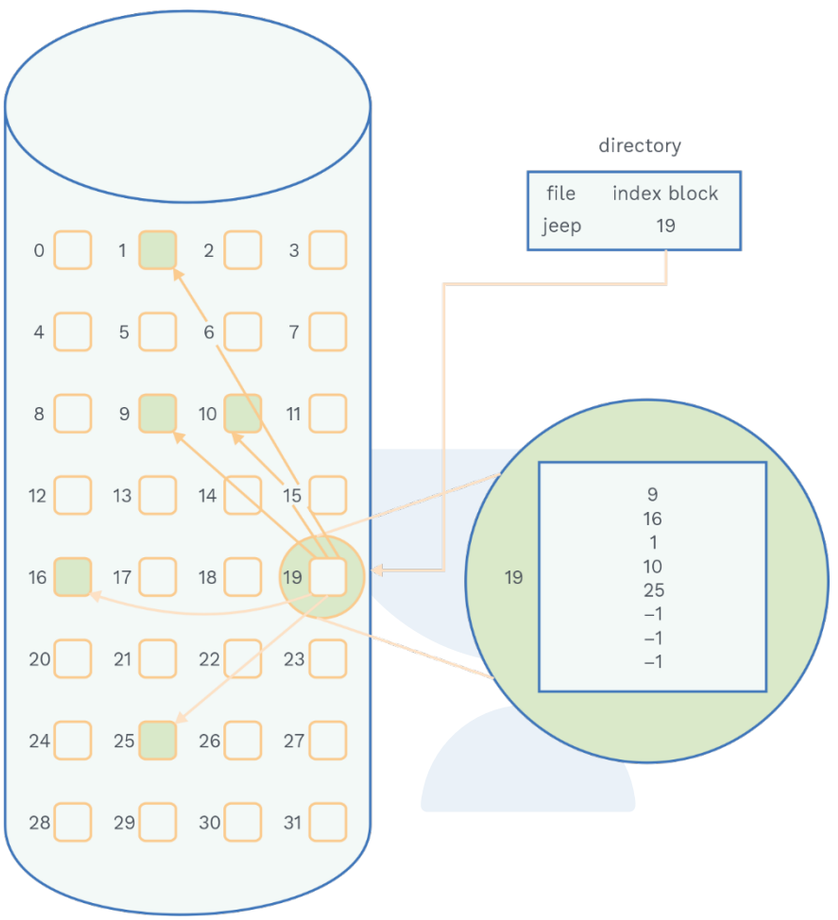


Fig. 6.11 Indexed Allocation of Disk Space

Advantages:

1) It provides random access to the blocks of files, which makes retrieval faster.
2) External Fragmentation is not present.

Disadvantages:

1) Pointer overhead in index allocation is more as compared to linked allocation.
2) A complete block holds the pointers only, which leads to inefficient utilization of space.

Sometimes, a single index block is not sufficient to hold all the pointers for files that are very large.

Thus, there are few mechanisms that is used for this.

- **Linked scheme:**
  As the name specifies, in this scheme,atleast two index blocks are grouped together to hold the pointers. Every index block will have a pointer or address to other index blocks.
- **Multilevel index:**
  There are multiple levels of an index. In case of two levels of an index, the first level index block has pointers to the second level index blocks and the second level block has pointers to the disk blocks of the files stored on the disk.
- **I-node:**
  1) A file attributes like a name, size, permissions, etc are stored in a special block called as I-node (information node).
  2) In an I-node, space which remains empty after storing the meta data of a file is used for storing DBA (disk block address) which has the actual content of a file.
  3) The first few of these pointers in I-node point to the direct blocks. Direct block is the block which has actual data.
  4) A single indirect block exists, whose pointers are pointing to the actual data blocks where the data of a file is stored. This indirect block is never holding the data. But it holds the addresses of the blocks (which holds the actual data).
  5) Similarly, the next few pointers are pointing to double indirect blocks which do not contain the file data but the address of the blocks that contain the address of the blocks containing the file data.

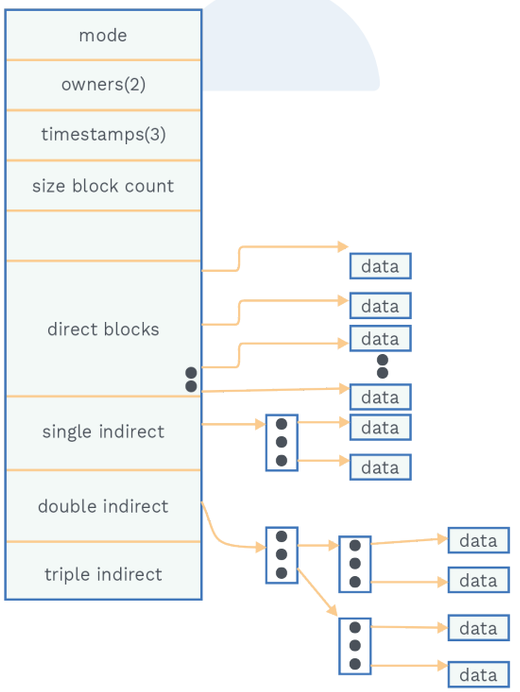


Fig. 6.12 UNIX I-Node

SOLVED EXAMPLES

Q4 **Consider a UNIX-I-NODE structure, which maintains eight direct disk block addresses, two single indirect, one double indirect, one triple indirect disk block addresses. The size of the disk block is 256 bytes, and disk block address requires 32 bit.**
The maximum possible file size and total size of the file system are:
a) 64 MB and 66,594 KB
b) 64 MB and 66,494 KB
c) 32 MB and 66,594 KB
d) 32 MB and 66,494 KB

Sol: **Option: a)**
Disk block size (DB size) = 256 bytes.
Disk block address size (DBA) = 32 bit = $\frac{32}{8}$ = 4 bytes.

a) Maximum file size will be for triple indirect disk block addresses:

$$1 * \left(\frac{\text{DB size}}{\text{DBA}}\right)^3 * \text{DB size}$$

$$= \left(\frac{2^8}{4}\right)^3 * 2^8$$

$$= 2^{18} * 2^8 = 2^{26}\,\text{B}$$

$$= 64\,\text{MB}$$

b) Total size of the file system

$$= \left[8 + 2 * \left(\frac{\text{DB size}}{\text{DBA}}\right) + 1 * \left(\frac{\text{DB size}}{\text{DBA}}\right)^2 + 1 * \left(\frac{\text{DB size}}{\text{DBA}}\right)^3\right] * \text{DB size}$$

$$= \left[8 + 2 * \frac{2^8}{2^2} + \left(\frac{2^8}{2^2}\right)^2 + \left(\frac{2^8}{2^2}\right)^3\right] * 2^8\text{B}$$

$$= \left[8 + 2^7 + 2^{12} + 2^{18}\right] * 2^8\,\text{B}$$

$$= \left[2 + 2^5 + 2^{10} + 2^{16}\right] 2^{10} \text{ B}$$

$$= [2 + 32 + 1024 + 65536] \text{ KB} = 66594 \text{ KB}$$

Q5 **Consider a Unix I–node, which maintains 32 direct disk block addresses, one single indirect, one double indirect and one triple indirect disk block address. The size of a disk block and disk block address is 2 KB and 32 bits. What is the maximum file size possible in the system? _____ (in GB) (rounded off up to two decimal places)**

Sol: **Range: 256.50 – 256.50**

Maximum file size possible –

$$\left[\underset{\substack{\uparrow \\ \text{Direct} \\ \text{DBAS'}}}{32} + \left(\frac{\text{DB Size}}{\text{DBA}}\right) + \left(\frac{\text{DB Size}}{\text{DBA}}\right)^2 + \left(\frac{\text{DB Size}}{\text{DBA}}\right)^3\right] * \text{Disk Block Size}$$

Where DB : Disk Block Address = 32 bits = 4B.

$$= \left[32 + \frac{2\,\text{KB}}{4\,\text{B}} + \left(\frac{2\,\text{KB}}{4\,\text{B}}\right)^2 + \left(\frac{2\,\text{KB}}{4\,\text{B}}\right)^3\right] * 2\,\text{KB}$$

$= [2^5 + 2^9 + (2^9)^2 + (2^9)^3] * 2$ KB

$= 2^5 (1 + 2^4 + 2^{13} + 2^{22}) * 2$ KB

$= 268{,}960{,}832$ KB

$= 256.50$ GB (divided by 2^{20})

Q6 **Consider a disk with FAT system which has FAT entry size of 32 bits. Alice is using a system with 64 GB hard disk. Block size is known to be 32 KB. What is the minimum size (in MB) of file that Alice can store in the hard disk?**

Sol: **Range: 65,528 - 65,528**

Hard disk size = 64 GB = $2^{30} * 26 = 2^{36}$ Bytes

Number of blocks in disk = $2^{36} / 2^{15} = 2^{21}$ blocks

For each block, there will be a FAT entry → (We want to maximize FAT table size)

FAT size = $2^{21} * (32/8) = 2^{21} \times 2^2 = 2^{23}$ Bytes = 8MB

Minimum size of file = $(2^{36} / 2^{20}) - 8 = 2^{16} - 8 = 65{,}528$ MB

Rack Your Brain

Consider the Unix I–node, which maintains 14 direct pointers, one single indirect pointer and one double indirect pointer. The disk block offset is 12,768 bits, and the disk block address is 64 bits long. The maximum file size possible with double indirect is _____ GB.

Free space management:

- As there is limited disk space, we need to reuse the deleted space for the new files.
- There is a free-space list that is managed by the system to keep the track of all the disk space that is free.
- To keep track of all the disk space that is free, a free-space list is managed by the system.

1) Bit vector:

The empty space list is implemented as a bit map. A block is represented using a bit

If bit=1, block is free

If bit=0, block is allocated

Suppose 1,2,4,6,7...blocks are free. So, the free space bit map would be look like.

0 1 1 0 1 0 1 1 .....

Advantages:

i) It is simple to understand.

ii) It's very efficient in finding the first free block.

2) Linked list:

Free space management on the disk can also be done using a linked list; in this method, a free disk block has a pointer to the next free block on the disk, and so on. A pointer to the first free block is stored at a designated location on the disk, and it is cached in memory.

3) Grouping:

In this approach, the addresses of the first 'n' free blocks on the disk are stored in the first free block. In these n blocks, except the last block n-1 blocks are actually free. The last block holds the address of the next 'n' free blocks on the disk.

Main advantage of this method → group of free disk blocks can be found easily.

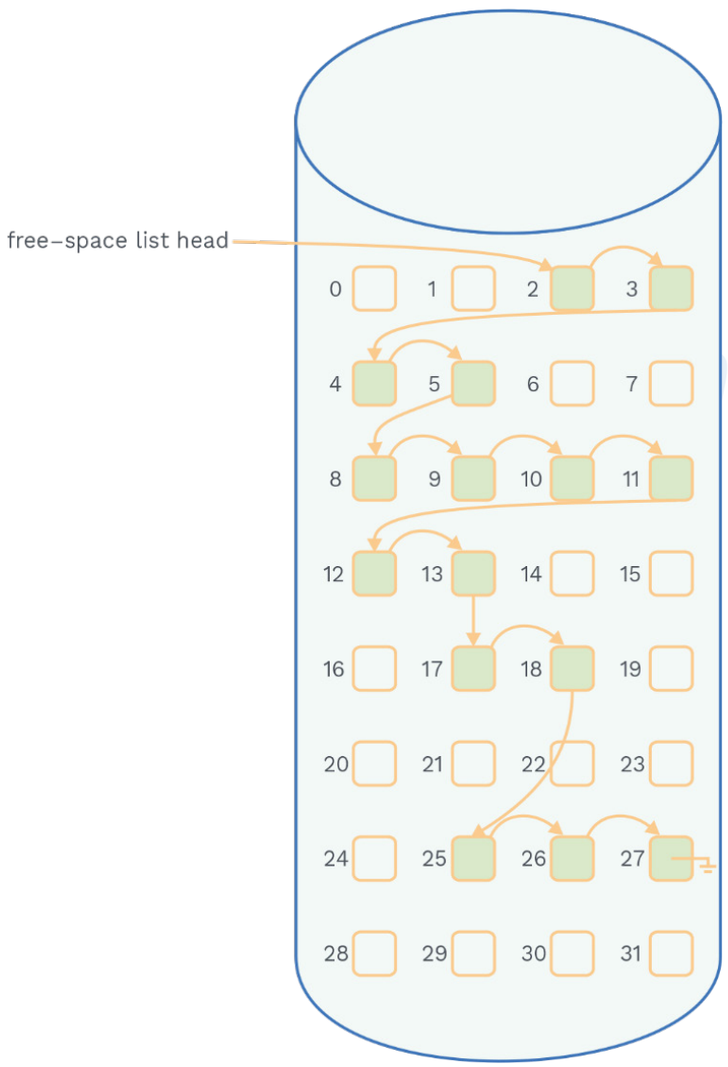


Fig. 6.13 Linked Free–Space List on Disk

Disk storage structures (DSS):

Basics of DSS

Magnetic disks

Bulk secondary storage is provided by magnetic disks for a modern computer systems. Each disk platter has a flat circular shape like CD.

1) The two–surface of the platter is covered with magnetic material. Each surface is divided into tracks.

- Each track is further divided into sectors. Outer tracks are bigger than inner tracks. They have the same number of sectors and storage capacity, so storage density is high in the inner tracks and low in the outer.
- Disk head (R–W head) rotates over the rotating hard disk. This head performs all read–write operations on the disk. Position of the head is the major concern as we need to put R–W head to the position where we want in read/write.

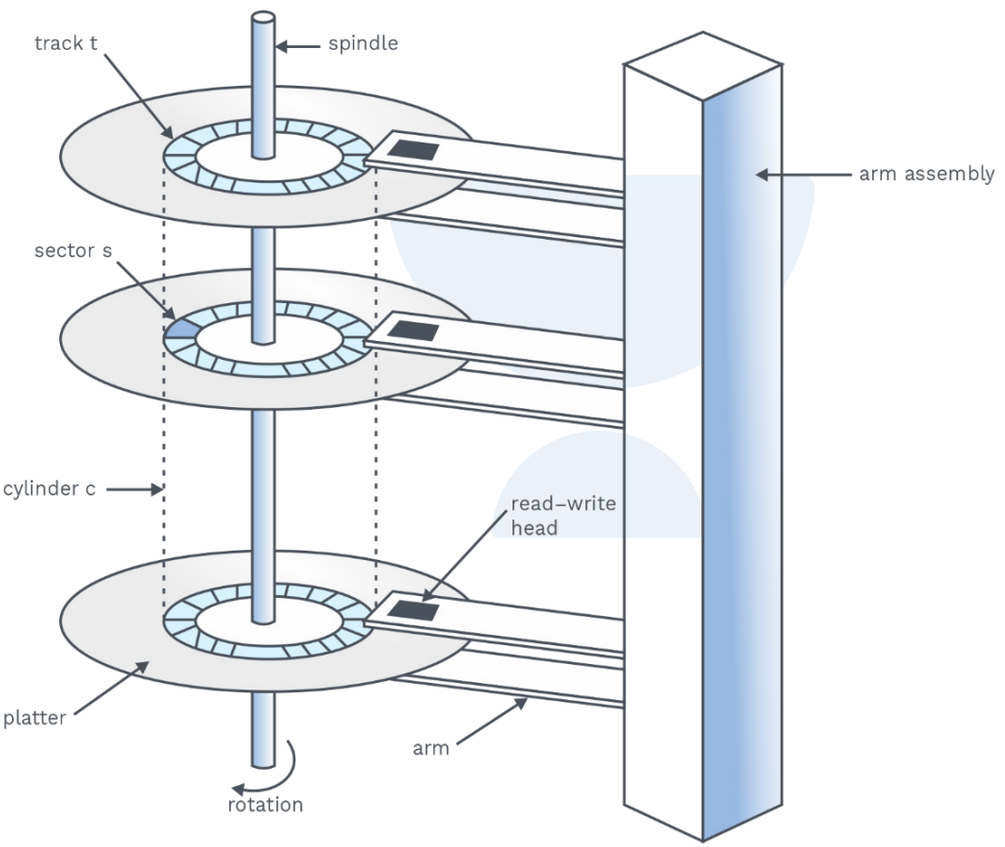


Fig. 6.14 Moving–Head Disk Mechanism

Some terms are:

1) **Seek time:** Time is taken by the read-write head to reach from one track to another one is known as seek time.
2) **Rotational latency:** The amount of time taken by the sector to rotate in such a way that it can be accessed by the read-write head.

3) **Data transfer time:** Time taken to transfer the required amount of data.
4) **Average access time:** = Average rotational Latency + Data transfer + Seek time.

Average Rotational Latency = ½ * Time taken for one full rotation

Disk Scheduling:

Benefits of disk scheduling:

- Even though multiple processes requests for I/O operation, at a time, only a single I/O request is served by the disk controller. The remaining I/O requests need to wait until they get scheduled.
- The greater disk arm movement happens when two or more requests are far from each other.
- HDD is one of the slowest (secondary devices) parts of the computer system. It needs to be accessed in an efficient manner.

1) **FCFS scheduling (first come – first serve):**
It is considered as the easiest disk scheduling algorithm.
In this method, the disk requests are satisfied in the order they arrive in the disk queue.

Example: A disk queue with requests for I/O to blocks on a cylinder.
90, 178, 42, 133, 10, 140, 76, 79 and consider head starts at 50 in a disk system with 200 cylinders.

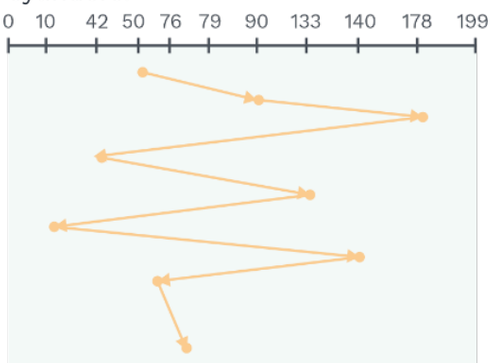


Fig. 6.15 FCFS Algorithm

Total seek time =
(90 - 50) + (178 - 90) + (178 - 42) + (133 - 42) + (133 – 10) + (140 – 10) + (140 – 76) + (79 – 76) = 675

Rack Your Brain

Consider the following parameters
Number of surfaces = 32
Number of tracks/surface= 512
Number of sectors/track = 512
Number of bytes/sector = 1 KB
The number of bits required to specify a particular sector in the disc are?

2) SSTF scheduling (shortest seek time first):

It is reasonable to serve the requests near to present head position before serving far away disk requests. This scheduling selects the request with the least time from the current head position.

With this method, average response time decreases and throughput increases.

Example: A disk queue with requests for I/O to blocks on the cylinder. 82, 170, 43, 140, 24, 16, 190 and consider SSTF scheduling and current head start at 50 in a disk system with 200 cylinders.

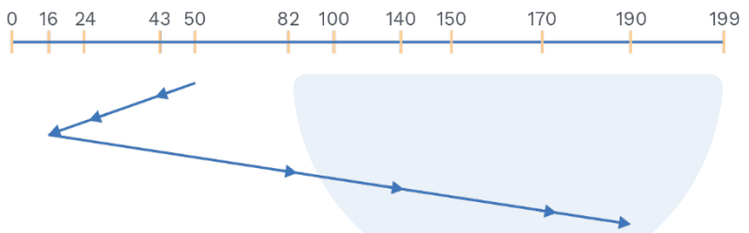


Fig. 6.16 SSTF Algorithm

Total seek time
= (50 – 43) + (43 – 24) + (24 – 16) + (82 – 16) + (140 – 82) + (170 – 140) + (190 – 170) = 208.

Major disadvantages are:

- An overhead of prior seeks time calculation.
- Starvation is possible for a request with a higher seek time compared to the next requests.
- The high variance of response time as SSTF favours only some requests.

3) SCAN scheduling:

The disk arm initiates at one end of the disk and moves towards, the other end of the disk, serving all the requests falling in between. After that, the disk arm changes its direction and serves all the requests in that direction in order. Head scans across the disk, back and forth. This scheduling is also called the elevator algorithm.

Example: A disk queue with requests for I/O to blocks on the cylinder. 90, 178, 42, 133, 10, 140, 76, 79 and consider scan algorithm and head starts at 50 in a disk system with 200 cylinders and it is scanning in left direction.

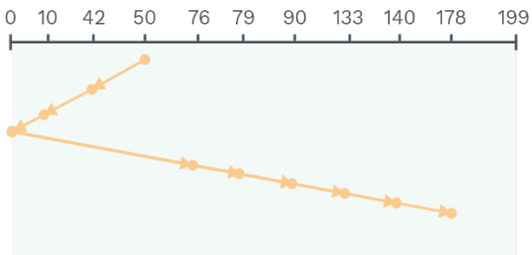


Fig. 6.17 SCAN Scheduling

Total seek time= (50 - 42) + (42 - 10) + (10 - 0) + (76 - 0) + (79 - 76) + (90 - 79) + (133 - 90) + (140 - 133) + (178 - 140) = 228

4) C–SCAN scheduling

C-SCAN is an extension of SCAN. It tries to ensure a uniform waiting time for all requests. In this method, the disk head moves from one end of the disk to the other like SCAN, fulfilling requests on the way. After the head reaches the other end of the disk, it returns to the beginning of the disk, by passing any requests on the way back.

The cylinders are treated as a circular list by the C–SCAN scheduling, which wraps around from the last cylinder to the first.

Example: A disk queue with request for I/O to the cylinder are

90, 178, 42, 133, 10, 140, 76, 79 .

Consider C–SCAN scheduling and head starts at 50 in a disk system with 200 cylinder, and is moving in the right direction.

Sol:

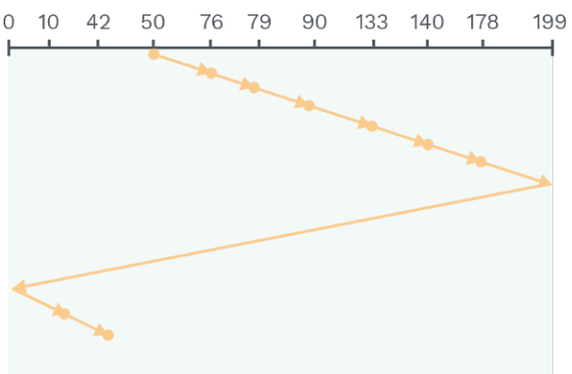


Fig. 6.18 C-SCAN Scheduling

Total seek time = (76 - 50) + (79 - 76) + (90 - 79) + (133 - 90) + (140 - 133) + (178 - 140) + (199 - 178) + (199 - 0) + (10 - 0) + (42 - 10) = 390

5) LOOK scheduling:

It is identical to the SCAN disk scheduling algorithm, except that instead of going to the end of the disk, the disk arm only goes to the last request to be handled in front of the head and then reverses its direction from there. As a result, the extra time caused by unnecessary traversal to the disk's end is avoided.

Example: A disk queue with request for I/O to the cylinder are

90, 178, 42, 133, 10, 140, 76, 79.

Consider LOOK scheduling and head starts at 50 in a disk system with 200 cylinders and is moving in the right direction.

Sol:

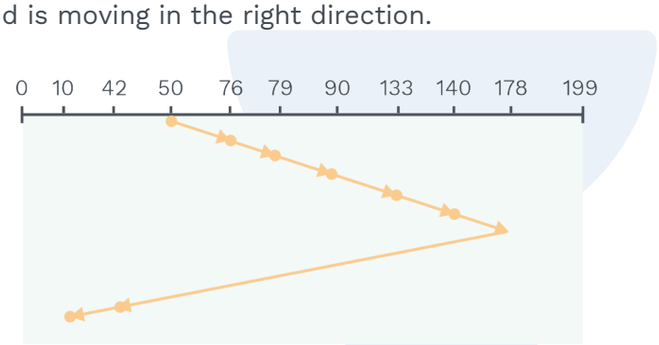


Fig. 6.19 LOOK Scheduling

Total time = (76 - 50) + (79 - 76) + (90 - 79) + (133 - 90) + (140 - 133) +

(178 - 140) + (178 - 42) + (42 - 10) = 296

6) C–LOOK scheduling:

Circular-LOOK (C-LOOK) is similar to C–SCAN scheduling. In this approach, the disk arm moves in a direction serving all requests till the last request in the same direction. After that, the disk arm changes its direction and serves the last request nearest to other ends. Again the disk arm changes the direction and serves the remaining requests in the direction in order. In the given example, the disk arm goes from 178 – 10 and not to 199 and then 0 unlike the C-SCAN. Thus, it also prevents the extra delay which occurred due to unnecessary traversal to the end of the disk.

Example: A disk queue with request for I/O to the cylinder are

90, 178, 42, 133, 10, 140, 76, 79.

Consider LOOK scheduling and head starts at 50 in a disk system with 200 cylinders and is moving in the right direction.

Sol:

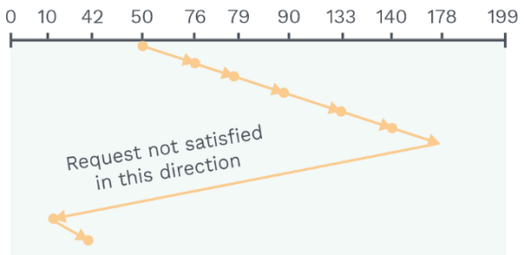


Fig. 6.20 C-LOOK Scheduling

Total seek time = (76 - 50) + (79 - 76) + (90 - 79) + (133 - 90) +
(140 - 133) + (178 - 140) + (178 - 10) + (42 - 10)
= 328

Rack Your Brain

Consider a disk system with 200 cylinders (0–199) and reading data from track 100. Track sequence is given 45, 8, 10, 11, 110, 50, 186, 176, 192. Suppose the FCFS scheduling is used and it takes 1 ms to move from one cylinder to adjacent one. Then total seek time is _____ ms.

SOLVED EXAMPLES

Q7 **Consider a disk drive with 3000 cylinders, numbered in the range [0, 2999]. The disk arm is serving the cylinder 125 now, and it is moving upwards. The pending requests in order are given as: 85, 1390, 900, 1765, 729, 1512, 1025, 1699, 128.**
The total distance (in cylinders) that the disk arm moves from its current position to serve all the pending requests using C–SCAN is _____

Sol: **Range: 5958 – 5958**

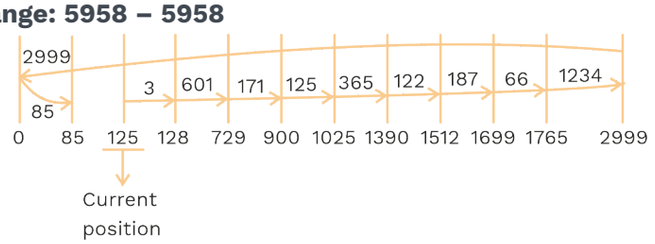


Fig. 6.21 C-SCAN Scheduling

Total distance = 3 + 601 + 171 + 125 + 365 + 122 + 187 + 66 + 1234 + 2999 + 85 = 5958

Q8 **Consider the tracks request sequence of a disk with 200 tracks is 72, 160, 33, 130, 14, 6, 180. Initially, R/W head is at track 50. R/W head moves in an upward direction. How many additional head movements will be traversed by the R/W head when the C-SCAN method is used compared to Shortest Seek Time First (SSTF) method?**

a) 200
b) 163
c) 180
d) 120

Sol: **Option: b)**

SSTF

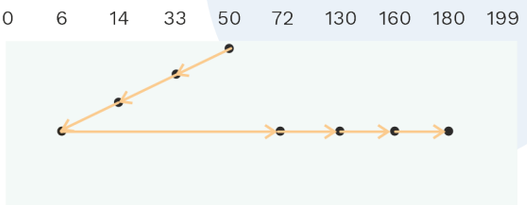


Fig. 6.22 SSTF Scheduling

No. of head movements = (50 - 6)+(180 - 6) = 44 + 174 = 218

C–SCAN

Fig. 6.23 C-SCAN Scheduling

Number of head movements = (199 - 50) + (199 - 0) + (33 - 0) = 149 + 199 + 33 = 381
Additional head movement are: = 381 – 218 = 163

IMPORTANT FORMULAE

Secondary memory:

No. of blocks in secondary memory = (size of the secondary memory / size of a block)

Contiguous allocation:

The X^{th} block of the file, which starts at block 'p' can be obtained at $(p + X)^{th}$ block.

File allocation table (FAT):

No. of entries in FAT = No. of blocks in secondary memory
Size of one entry = Bits required to identify a block
FAT Size = No. of entries in FAT * FAT Entry Size

UNIX I-node:

UNIX I-Node file system with n_0 direct disk block addresses, n_1 single indirect disk block addresses, n_2 doubly indirect DBAs and n_3 triple indirect DBAs. Then,
then maximum file size for triple indirect disk block addresses is

$$= \frac{\text{Disk Block Size}}{\text{Disk Block Address}} * \text{Disk Block Size}$$

Size of the whole file system is give as:

$$\left\{ n_0 + n_1 \times \left\{ \frac{\text{Disk Block Size}}{\text{Disk Block Address}} \right\} + n_2 \times \left\{ \frac{\text{Disk Block Size}}{\text{Disk Block Address}} \right\}^2 + n_3 \left\{ \frac{\text{Disk Block Size}}{\text{Disk Block Address}} \right\}^3 \right\} * \text{Disk Block Size}$$

Chapter Summary

- Basics of file system – The mechanism for online storage and access to both data and programs.
- File attributes – File has various identifiers to uniquely identify it in file system, such as name, size, type, etc.
- File operations – **1)** Creating a file
  2) Writing a file
  3) Reading a file
  4) Repositioning within a file
  5) Deleting a file
  6) Truncating a file
- File types – It represents the type of file and the operating system can recognize it and take appropriate action accordingly.
- Directory – A directory contains various files and folders.
- Types of directory – **1)** Single level
  2) Two level
  3) Tree structured
  4) Acyclic graph
- File access methods – Sequential access
  Direct access
  Index sequential
- File allocation methods – Contiguous allocation
  Linked allocation
  Indexed allocation
  I-node
  File allocation table
- Free space management – Bit vector
  Linked list
  Grouping
- Disk scheduling – FCFS
  SSTF
  SCAN
  C–SCAN
  LOOK
  C–LOOK