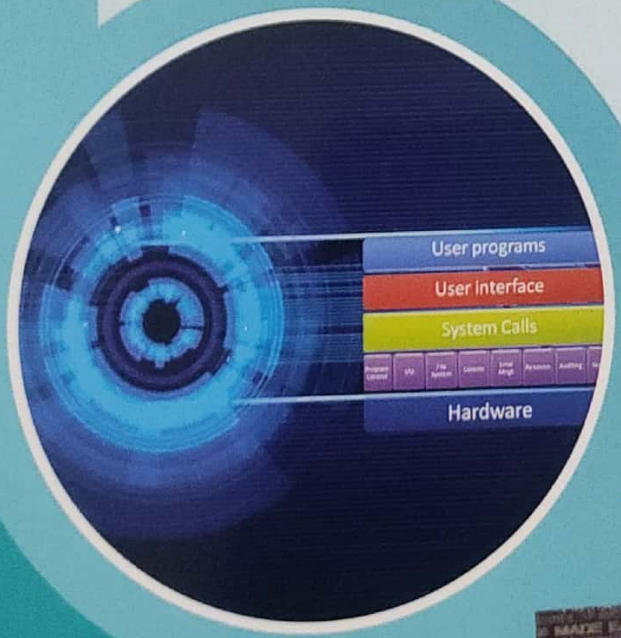
GATE PSUs

MADE EASY Publications

POSTAL STUDY PACKAGE

COMPUTER SCIENCE & IT



2020

THEORY BOOK

Operating System

Well illustrated theory with solved examples

Contents

Operating System

Chapter 1

Basic Concepts of OS 3

1.1 Operating System (OS) 3
1.2 Structure of Computer System 3
1.3 Layered View of Operating System Services 4
1.4 History of Operating System 4
1.5 Types of Operating System 4
1.6 Dual Mode Operations 7
1.7 Functions of Operating System 8
1.8 Operating System Components 9
1.9 System Call 9
1.10 Interrupts 9
1.11 Booting of OS 10
1.12 Types of Kernel Designs 11
Student Assignments 13

Chapter 2

Processes and Threads 16

2.1 Process 16
2.2 Process State Models 17
2.3 Operations on a Process 22
2.4 Scheduling 22
2.5 Thread 24
2.6 Multi Threading 26
2.7 Co-operating Processes (Inter Process Communication) 28
Student Assignments 29

Chapter 3

CPU Scheduling 32

3.1 Introduction 32
3.2 Goals of CPU Scheduling 33
3.3 Kinds of CPU Scheduling Algorithms 33
3.4 Scheduling Algorithms 33
Student Assignments 53

Chapter 4

Process Synchronization 57

4.1 Synchronization 57
4.2 The Critical-Section Problem 57
4.3 Requirements 58
4.4 Synchronization Techniques 59
4.5 Two Process Solution (Software Solution) 59
4.6 Multiple Process Solution (Software Solution) 62
4.7 Bakery's Algorithm (Software Solution) 63
4.8 Interrupt Disabling 64
4.9 Special Machine Instructions 64
4.10 Semaphores 68
4.11 Mutex 72
4.12 Classical Problems of Synchronization with Semaphore Solution 75
4.13 Programming Language Solution: Monitors 84
Student Assignments 86

Chapter 5

Concurrency and Deadlock 91

5.1 Concurrency 91
5.2 Precedence Graph 93
5.3 The Fork and Join Constructs 95
5.4 Parbegin/Parend Concurrent Statement 98
5.5 Deadlock 100
5.6 Conditions for a Deadlock 101
5.7 Methods of Handling Deadlocks 102
5.8 Deadlock Prevention 103
5.9 Deadlock Avoidance 104
5.10 Deadlock Detection and Recovery 106
Student Assignments 113

Chapter 6

Memory Management 117

6.1 Introduction 117
6.2 Logical (Virtual) Vs Physical Address Space 118
6.3 Memory-Management Unit (MMU) 119
6.4 Memory Allocation Techniques 123
6.5 Dynamic Allocation Algorithms 125
6.6 Internal Fragmentation and External Fragmentation 126
6.7 Paging 128
6.8 Segmentation 144
6.9 Segmented Paging 146
6.10 Buddy System 148
Student Assignments 150

Chapter 7

Virtual Memory 153

7.1 Introduction 153
7.2 Page Fault 154
7.3 Page Replacement 156
7.4 Page Replacement Algorithms 156
7.5 Frame Allocation 161
7.6 Dynamic Paging Algorithms 164
7.7 Advantages and Disadvantages of Virtual Memory 164
Student Assignments 165

Chapter 8

File System 168

8.1 Introduction 168
8.2 Directories 168
8.3 File Management System 169
8.4 File System Organization 171
8.5 File Allocation Methods 172
8.6 Free Space Management 173
8.7 Tape 179
Student Assignments 183

Chapter 9

Input/Output System 187

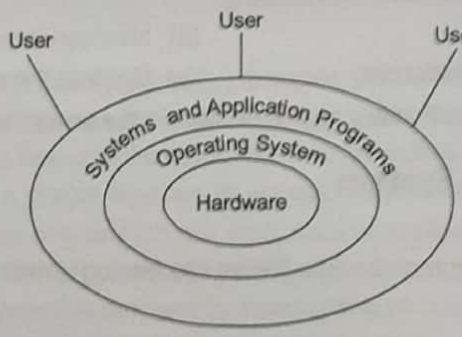
9.1 Introduction 187
9.2 I/O System Structure 188
9.3 Buffering and Caching 188
9.4 Magnetic Storage Devices 189
9.5 Disk Scheduling 191
9.6 Disk Scheduling Algorithms 191
9.7 Comparisons of Disk Scheduling Algorithms 196
9.8 Selection of a Disk Scheduling Algorithm 196
Student Assignments 199

CHAPTER 01

Basic Concepts of Operating System

1.1 Operating System (OS)

It is a program that acts as an interface between a user (applications) and the computer hardware.

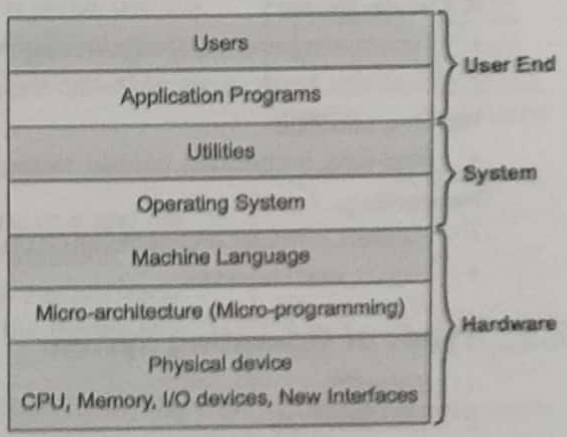


Operating System allows the compute system in more convenient to we and efficient resource utilization.

1.2 Structure of Computer System

A computer system consists of :

- **Users:** People, other computers, machines, etc.
- **Application programs:** Compilers, database systems, video games, business applications, web-browsers, etc.
- **System programs:** Shells, editors, compilers, development tools, etc.
- **Operating System:** It is a system program which controls and coordinates the use of hardware among application programs.
- **Hardware:** CPU, Disk, Memory, I/O devices, etc.



NOTE: Firmware (BIOS) is a software which is permanently stored on chip but upgradable. It loads the operating system during the boot.

1.3 Layered View of Operating System Services

1. **User View:** The OS is an interface, hides the details which must be performed and presents a virtual machine to the user that makes easier to use.
   OS provides the following services to the user.
   (*i*) Execution of a program
   (*ii*) Access to I/O devices
   (*iii*) Controlled access to files
   (*iv*) Error detection (Hardware failures, and software errors)
2. **Hardware View:** The operating system manages the resources efficiently in order to offer the services to the user programs.
   OS acts as resources managers:
   (*i*) Allocation of resources
   (*ii*) Controlling the execution of a program
   (*iii*) Controls the operations of I/O devices
   (*iv*) Protection of resources
   (*v*) Monitors the data
3. **System View:** OS is a program that functions in the same way as other programs. It is a set of instructions that are executed by the processor.
   OS acts as a program to perform the following:
   (*i*) Hardware upgrades
   (*ii*) New services
   (*iii*) Fixes the issues of resources
   (*iv*) Controls the user and hardware operations

Goals of operating system: Primary goal is convienience and secondary goal is efficiency.

1.4 History of Operating System

Tube based:
- Hardware can run one program at a time (Serial Processing) (1945-1955)
- Numerical calculations

Transits based:
- High level languages (Fortran)
- (Batch programing) (1955-1965)

IC Circuits based:
- Time sharing and multiprogramming (1965-1980)
- Business Applications, Scientific and Engineering Applications

Working and PCs:
- Real-time, Embedded, parallel, Network programming (1980-2000)

Networking:
- Parallel Computer Architectures (2000-Present)
- High Speed Networks.

1.5 Types of Operating System

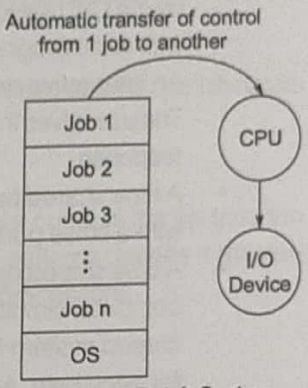
1. Serial OS
2. Batch OS
3. Interactive OS
4. Multiprogrammed OS

5. Time sharing OS
6. Real time OS
7. Network OS
8. Parallel OS
9. Distributed OS
10. Clustered OS
11. Handheld OS

1.5.1 Simple Batch Systems

The operating system in early computers was fairly simple. Its major task was to transfer control automatically from one job to the next. The operating system was always in memory. To speed up processing, jobs with similar needs were batched together and were run through the computer as a group. Thus, the programmers would leave their programs with the operator. The operator would sort programs into batches with similar requirements, and as the computer become available, would run each batch.

- Systems allowed automatic job sequencing by a resident operating system and greatly improved the overall utilization of the computer.
- In batch system there is lack of interaction between the user and the job while the job is executing.
- The CPU utilization was still low. In this execution environment the CPU is often idle. This idleness occurs because the speeds of the mechanical input/output devices are intrinsically slower than those of electronic devices.
- Batch system are appropriate for executing large jobs that need little interaction. *Example:* IBM OS/2.



Batch System

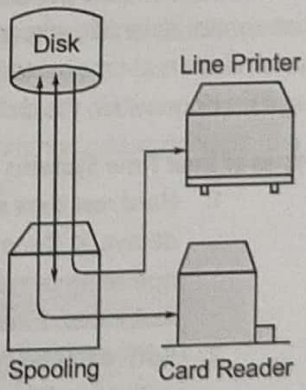
Spooling

Spool is acronym for simultaneous peripheral operation On-Line. Spooling overlaps I/O of one job with the computation of other job. Spooling uses the disk as a huge buffer for reading as far ahead as possible on input devices and for storing output files until the output devices are able to accept them. A spool is a buffer that holds output for a device, such sa a printer, that cannot accept interleaved data streams.

Although a printer can serve only one job at a time, several applications may wish to print their output concurrently, without having their output mixed together. The operating system solves this problem by intercepting as output to the printer. Each applications' output is spooled to a separate disk file.

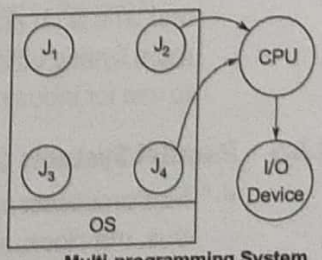
When an application finishes printing, the spooling system queue picks the next spooled-file for input to the printer.

The spooling system copies the queued spool files to the printer one at a time. Even in a simple system, the spooler may be reading the input of one job while printing the output of a different job. Spooling can keep both the CPU and the input/output devices working at much higher rates.



1.5.2 Multi-Programmed Systems

- Several jobs are kept in main memory at the same time and the CPU is multiplexed among them, to increase CPU utilization.
- A job pool on the disk consists of a number of jobs that are ready to be executed. Subsets of these jobs reside in the memory for execution.
- The operating system picks and executes one of the jobs in memory.

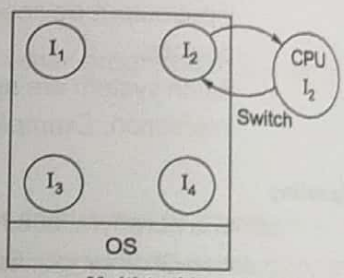


Multi-programming System

- When this job in execution needs an input/output operation to complete, Instead of waiting for the job to complete the input/output, it switches to the subset of jobs waiting for CPU.
- In a non multi programmed system the CPU would sit idle. In multiprogramming system the operating system simple switches to and executes another job.
- If several jobs are ready to be brought into memory and there is not enough room for all of them, then the system must choose among then. Making this decision is job scheduling.
  Example: Windows and Unix.

1.5.3 Time Sharing System (Multitasking)

- Time sharing or multitasking is a logical extension of multiprogramming. Multiple jobs are executed by the CPU switching between them, but the switches occur is frequently that the users may interact with each program while it is running.
- An interactive computer system provides direct communication between the user and the system. The user gives instructions to the operating system or to a program directly and receives an immediate response.
- A time shared operating system uses CPU scheduling and multiprogramming to provide each user with a small portion of a time shared computer.
- A time shared operating system allows the many users to share the computer simultaneously. Since each action or command in a time shared system tends to be short, only a little CPU time is needed for each user. As the system switches rapidly from one user to the next, each user is given the impression that she has her own computer, whereas actually one computer is being shared among many users.



Multi-tasking System

1.5.4 Real Time System

Used when there are rigid time requirements on the operating of a processor or the flow of data. Systems that control scientific experiments, medical imaging systems, industrial control systems, and some display systems are real-time systems. A real time operating system has well defined, fixed time constraints. Processing must be done within the defined constraints or the system will fail. *Example:* RTOS

Types of Real Time Systems

1. ***Hard real time system:*** Guarantees that critical tasks completed on time. This goal requires that all delays in the system be bounded from the retrieval of stored data to the time that it takes the operating system to finish any request made to it. Kernel delays need to be bounded and more restrictive. ***Example:*** Satelite, Missile System.
2. ***Soft real time system:*** A less restrictive type of real time system is a soft real time system, where a critical real time task gets priority over other tasks, and retains that priority until it completes. Soft real time is an achievable goal that can be mixed with other types of systems. However they have more limited utility than do hard real time system. Given their late of deadline support they are risky to use for industrial control and robotics. ***Example:*** Banking system.

1.5.5 Parallel Systems (Multiprocessors)

- Multiprocessor system have more than one processor in close communication, sharing the computer bus, the clock, and sometimes memory and peripheral devices. These systems are referred to as tightly coupled systems.

- One advantage of building this kind of system is increased throughput. By increasing the number of processors we hope to get more work done in a shorter period of time.
- Multiprocessors can also save money because the processors can share peripherals, cabinets and power supplies. If several programs are to operate on the same set of data, it is cheaper to store those data on one disk and to have all the processors share them, rather than to have many computers with local disks and many copies of the data.
- Another advantage is increased reliability.
- If functions can be distributed properly among several processors, then the failure of one processor will not halt the system, but rather will only slow it down.
- The ability to continue providing service proportional to the level of surviving hardware is called graceful degradation. Systems that are designed for graceful degradation also called fault-tolerant.

Symmetric Multiprocessing Model

In this each processor runs an identical copy of the operating system and these copies communicate with one another as needed.

Asymmetric Multiprocessing Model

In this model each processor is assigned a specific task. A master processor controls the system; the other processors either look to master for instruction or have predefined tasks. This scheme defines a master-slave relationship. The master processor schedules and allocates work to the slave processors.

1.5.6 Distributed Systems

The processors do not share memory or a clock, instead each processor has its own local memory. The systems are also referred as loosely coupled systems.

Advantages of Distributed Systems

- **Resource sharing:** If a number of different sites are connected to one another then, a user at one site may be able to use the resources available at another.
- **Computation speedup:** If a particular computation can be partitioned into a number of subcomputations that can run concurrently then a distributed system may allow us to distribute the computation among the various sites to run that computation concurrently.
- **Reliability:** If one site fails in a distributed system, the remaining silt can potentially continue operating.
- **Communication:** When many sites are connected to one another by a communication network, the processes at different sites have the opportunity to exchange information.

1.5.7 Difference between Distributed OS and Network OS

Network Operating System	Distributed Operating System
1. Each computer has its own OS 2. It allows interaction between the machines by having a common communication architecture 3. Independent machines accessed by the user	1. Common OS shared by a network of computers 2. Single OS controlling the network 3. Dependent machines accessed by the user to share the resources

1.6 Dual Mode Operations

A processor can support two modes of execution:

1. Kernel / Protected / Supervisor / System/Monitor / Privileged mode.
2. User mode / Non-privileged mode

Operating system runs in Kernel mode and user programs run in user mode. Mode bit is used to decide the mode of operating system. If mode bit is 0, it operates in kernel mode. Otherwise it operates in user mode when mode bit is 1.

User mode (Non-privileged mode)	Mode bit = 1
Kernel mode (Privileged mode)	Mode bit = 0

1.6.1 Kernel Mode

A process running in Kernel mode has following:

- Full access to machine instruction set.
- Direct access to hardware (memory, I/O devices, etc).

OS and device drivers must run in Kernel mode. MS DOS only support Kernel mode.

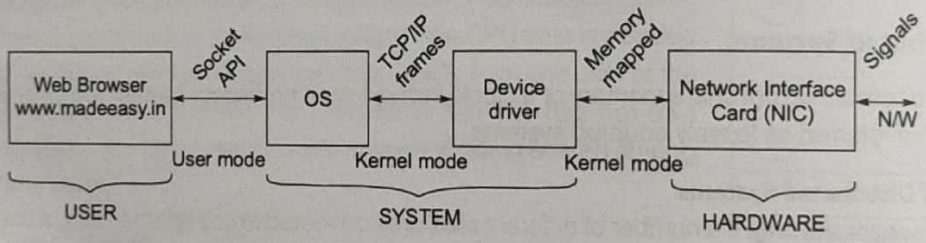
Privileged instructions: Set timer, Clear memory, Disable interrupts, Access I/O devices, etc.

1.6.2 User Mode

A process running in user mode has following:

- Access to the limited set of machine instructions.
- No direct access to hardware.
- Hardware access is coordinated by OS.

Non-privileged instructions: Read clock, Generate trap, User to Kernel mode switch, etc.



1.6.3 Privileged Instructions and Non-privileged Instructions

Privileged Instructions	Non-Privileged Instructions
I/O operations Context switching Disabling the interrupts Set system lock Remove process from memory Changing memory LOC of process	Reading system time Reading the states of CPU Sending the final printout of printer

Note: In boot time the system always start in kernel. The OS always run in kernel mode.

1.7 Functions of Operating System

- It controls all of computer resources.
- It provides valuable services to user programs.
- It coordinates the execution of user programs.
- It provides resources to user programs.
- It provides an interface (virtual machine) to the user.
- It hides the complexity of software.
- It supports the multiple execution modes.
- It monitors the execution of user programs to prevent errors.

1.8 Operating System Components

- **Process Management:** Operating system manages the process creation and deletion.
- **Main Memory Management:** Operating system keeps track of allocation and deallocation of main memory.
- **Secondary Storage Management:** Operating system uses secondary storage (Disk) to extend main memory.
- **I/O System Management:** Operating system uses I/O system to manage the devices.
- **File Management:** Operating system manages the file creation, deletion and permissions.
- **Protection System:** Operating system protects the resources by authorization.
- **Interface:** Operating system provides an interface to the user using the facilities by "command interpreter" or "Graphical user interfaces".
- **Networking:** The processors in the system are connected through a communication network. Communication takes place using a protocol. A distributed system provides user access to various system resources.

1.9 System Call

System call provides the services of operating system to the user programs via Application Program Interface (API). It provides an interface between a process and operating system to allow user level processes to request services of operating system. System calls are the only entry points into the Kernel System. All programs needing resources must use system calls.

Services Provided by System Calls

1. Process creation and management
2. Main memory management
3. File Access, Directory and File system management
4. Device handling (I/O)
5. Protection
6. Networking, etc.

Types of System Calls

1. **Process control:** end, abort, create, terminate, allocate and free memory.
2. **File management:** create, open, close, delete, read file etc.
3. Device management
4. Information maintenance
5. Communication

1.10 Interrupts

There are two types of interrupts:

1. **Interrupt:** Interrupt is a hardware generated which changes the flow of execution within the system. Interrupt handler deals with hardware generated interrupts to control such interrupts. Interrupt can be used to signal the completion of I/O.
   - External event
   - Asynchronous event
   - Independent of the currently executed process instructions.

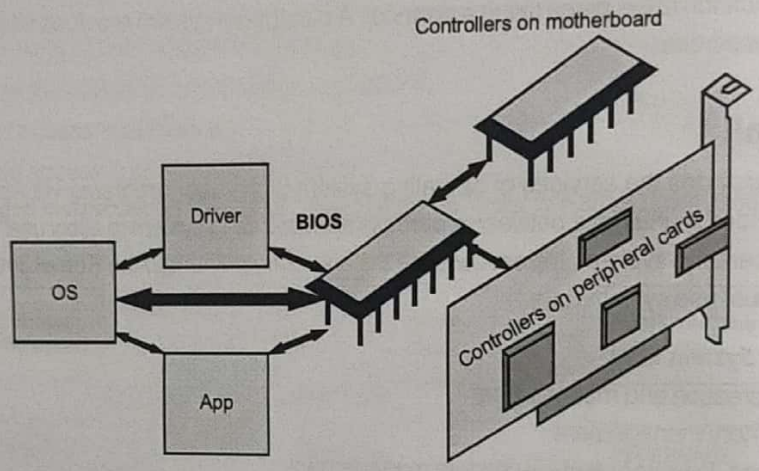
   Examples: Clock interrupt, I/O interrupt and Memory fault.

2. **Trap:** Trap is a software generated signal either to call operating system routines or to catch the arithmetic errors.
   - Exceptions and system calls
   - Synchronous event
   - Internal (exceptions) events or external events.

1.11 Booting of OS

Hardware doesn't know where the operating system resides and how to load it. Hence it needs a special program to do this job i.e., Bootstrap loader. ***Example:*** BIOS (Boot Input Output System).

Bootstrap loader locates the kernel, loads it into main memory and starts its execution. In some systems, a simple bootstrap loader fetches a more complex boot program from disk, which in turn loads the kernel.



BIOS Interaction

Booting Process

- Reset event on CPU (power up, reboot) causes instruction register to be loaded with a predefined memory location. It contains a jump instruction that transfers execution to the location of Bootstrap program.
- This program is form of ROM, since RAM is in unknown state at system startup. ROM is convenient as it needs no initialization and can't be affected by virus.
- Run diagnostics to determine the state of machine. If diagnostics pass, booting continues.
- Runs a **Power-On Self Test (POST)** to check the devices that the computer will rely on, are functioning.
- BIOS goes through a preconfigured list of devices until it finds one that is bootable. If it finds no such device, an error is given and the boot process stops.
- Initializes CPU registers, device controllers and contents of the main memory. After this, it loads the OS.
- On finding a bootable device, the BIOS loads and executes its **boot sector**. In the case of a hard drive, this is referred to as the **Master Boot Record (MBR)** and is often not OS specific.
- The MBR code checks the **partition table** for an active partition. If one is found, the MBR code loads that partitions **boot sector** and executes it.
- The boot sector is often **operating system** specific, however in most operating systems its main function is to load and execute a **kernel**, which continues startup.

- If there is no active partition or the active partition's boot sector is invalid, the MBR may load a secondary boot load and pass control to it and this secondary boot loader will select a partition (often via user input) and load its boot sector.
- Examples of secondary boot loaders:
  (a) **GRUB** : GRand Unified Bootloader
  (b) **LILO** : LInux LOader
  (c) **NTLDR** : NT Loader
- System such as cellular phones, PDAs and game consoles stores entire OS on ROM. Done only for small OS, simple supporting hardware, and rugged operation.
- Changing bootstrap code would require changing ROM chips.
  EPROM : Erasable Programmable ROM

1.12 Types of Kernel Designs

Monolithic Kernel

- All OS services operate in kernel space.
- Good performance.
- **Disadvantages:** Dependencies between system component. Complex and huge (millions(!) of lines of code).

Microkernel

- Minimalist approach
- IPC, virtual memory, thread scheduling
- Put the rest into user space
- Device drivers, networking, file system, user interface
- More stable with less services in kernel space
- **Disadvantages:** Lots of system calls and context switches.
  Example: Mach, L4, AmigaOS, Minix, K42.

Hybrid Kernel

It combines the best of both worlds

- Speed and simple design of a monolithic kernel
- Modularity and stability of a microkernel
- Still similar to a monolithic kernel
  Example: Windows NT, NetWare, BeOS

Exokernel

- Follows end-to-end principle
- Extremely minimal
- Fewest hardware abstractions as possible
- Just allocates physical resources to apps
- **Disadvantages:** More work for application developers. *Example:* Nemesis, ExOS

Remember

Q.1 In a multiprogramming and time-sharing environment, several users share the system simultaneously. This situation can result in various security problems.

(a) What are two such problems?

(b) Can we ensure the same degree of security in a time-shared machine as we have in a dedicated machine? Explain your answer.

Ans: (a) Stealing or copying one's programs or data; using system resources (CPU, memory, disk space, peripherals) without proper accounting.

(b) Probably not, since any protection scheme devised by humans can inevitably be broken by a human, and the more complex the scheme, the more difficult it is to feel confident of its correct implementation.

Q.2 What is the main advantage of multiprogramming?

Ans: Multiprogramming makes efficient use of the CPU by overlapping the demands for the CPU and its I/O devices from various users. It attempts to increase CPU utilization by always having something for the CPU to execute.

Q.3 List the steps that are necessary to run a program on a completely dedicated machine.

Ans: (a) Reserve machine (b) Manually load program into memory (c) Load starting address and begin execution (d) Monitor and control execution of program form console.

Q.4 Writing an operating system that can operate without interference from malicious or undebugged user programs requires some hardware assistance. Name three hardware aids for writing an operating system.

Ans: (a) Monitor/user mode (b) Privileged instructions (c) Timer (d) Memory protection.

Q.5 What is the purpose of the command interpreter? Why is it usually separate from the kernel?

Ans: It reads commands from the user or from a file of commands and executes them, usually by turning them into one or more system calls. It is usually not part of the Kernel since the command interpreter is subject to changes.

Q.6 What is the main advantage of the layered approach to system design?

Ans: As in all cases of modular design, designing an operating system in a modular way has several advantages. The system is easier to debug and modify because changes affect only limited sections of the system rather than touching all sections of the operating system. Information is kept only where it is needed and is accessible only within a defined and restricted area, so any bugs affecting that data must be limited to a specific module or layer.

Q.7 What are the main advantages of the microkernel approach to system design?

Ans: Benefits typically include the following (a) adding a new service does not require modifying the Kernel, (b) it is more secure as more operations are done in user mode than in Kernel mode, and (c) a simpler Kernel design and functionality typically results in a more reliable operating system.

Summary

- **Multitasking:** Interaction between multiple processes on the same processor.
- **Multiprogramming:** Interaction between multiple processes in the system. (either on same processor or on different processor)
- **Multiprocessing:** Interaction between multiple processors to execute the processes parallely.
- **Concurrency includes:**
  - Communication among processes/threads.
  - Sharing system resources.
  - Cooperative processing of shared data.
  - Synchronization of process/thread activities.
  - Organized CPU scheduling.
  - Solving deadlock and starvation problems.
- **Concurrency arises:**
  - Interaction between multiple processes running on one CPU.
  - Interaction between multiple threads running in one process.
  - Interaction between multiple processors running multiple processes/threads.
- **Multicomputing:** Interaction between multiple computers running distributed processes.

Student's Assignment

Q.1 The software that contains the core components of the operating system is called
(a) Controller (b) Root
(c) Kernel (d) None of the above

Q.2 When a computer is switched on, where is the operating system loaded?
(a) BIOS (b) ROM
(c) POST (d) RAM

Q.3 On which chip is the BIOS program permanently stored?
(a) RAM (b) ROM
(c) SIMBA (d) none of these

Q.4 Spooling helps because
(a) it is a more secure method of accessing data
(b) print jobs go more smoothly with less stop and go
(c) the computer is released to do other things while still printing
(d) None of the above

Q.5 When a computer is "swapping", it is
(a) moving data from hard drive to floppy drive
(b) moving data from memory to the swap file on the hard drive
(c) moving data between registers in memory
(d) None of the above

Q.6 The Operating System is responsible for
(a) controlling peripheral devices such as monitor, printers, disk drives
(b) provide an interface that allows users to choose programs to run/manipulate files
(c) manage users' files on disk
(d) All of the above

Q.7 Which of the following does an operating system do in a stand-alone computer system?
(a) manages the user's files
(b) provides the interface to allow the user to communicate with the computer
(c) controls the various peripherals
(d) All of the above

Q.8 Which of the following is true about a terminal on a time-sharing computer system?

(a) has its own CPU and some memory
(b) has no memory or CPU of its own
(c) has its own CPU but no memory
(d) has its own memory but no CPU

Q.9 The boot process happens in the order
(a) POST test, activate BIOS, check settings, load OS into RAM
(b) Activate BIOS, POST test, load OS into RAM, check settings
(c) Check settings, load OS into RAM, activate BIOS, POST test
(d) Load OS into RAM, check settings, activate BIOS, POST test

Q.10 An interrupt handler is a
(a) location in memory that keeps track of recently generated interrupts
(b) peripheral device
(c) utility program
(d) special numeric code that indicates the priority of a request

Q.11 Loading the OS into the memory of a PC is called
(a) thrashing (b) booting
(c) formatting (d) spooling

Q.12 An operating system is
(a) application (b) supervisory program
(c) set of users (d) set of programs

Q.13 This part of the operating system manages the essential peripherals, such as the keyboard, screen, disk drives, and parallel and serial ports.
(a) basic input/output system
(b) secondary input/output system
(c) peripheral input/output system
(d) marginal input/output system

Q.14 Which of the following functions is not controlled by the operating system?
(a) managing memory
(b) managing programs and data
(c) managing input and output
(d) all of the above are controlled by the operating system

Q.15 Executing more than one program concurrently by one user on one computer is known as
(a) multiprogramming
(b) time-sharing
(c) multitasking
(d) multiprocessing

Q.16 The simultaneous processing of two or more programs by multiple processors is
(a) multitasking (b) multiprogramming
(c) time-sharing (d) multiprocessing

Q.17 This comprises the detailed machine language necessary to control a specific device and is controlled by the operating system.
(a) driver
(b) utility program
(c) virtual memory
(d) peripheral device

Q.18 Which of the following controls the manner of interaction between the user and the operating system?
(a) language translator
(b) platform
(c) user interface
(d) none of the above is correct.

Q.19 Overlay is
(a) a part of an OS
(b) a specific memory location
(c) a single contiguous memory that was used in the olden days for running large programs by swapping
(d) overloading the system with many user fills

Q.20 Which of the following is real time system?
(a) an on line railway reservation system
(b) aircraft control system
(c) payroll processing system
(d) All of the above.

Q.21 Where does the swap space residable?
(a) RAM (b) disk
(c) ROM d) on-chip cache

Q.22 What is not the major objective of an operating system?

(a) to act as a resource manager for multiple tasks running on the CPU, the memory and disk resources

(b) to provide a programming interface to the user

(c) to act as an uniform abstract machine on top of a variety of different hardware platforms

(d) to enable loading and execution of binary code with minimum intervention by the user

Q.23 System call is used to access

(a) I/O functionality

(b) Operating system functionality

(c) Application functionality

(d) None of the above

Q.24 Which of the following statement is true with respect to Interrupts?

(a) interrupts are external to the CPU

(b) when an interrupt occurs, the CPU stops executing the current process and immediately transfers the execution to an appropriate Interrupt service routine

(c) when higher priority Interrupt occurs while processing the lower priority interrupt, then CPU transfers the execution to handle the high priority interrupts

(d) All of the above

Q.25 In which of the following situation, spooling is not essential.

1. Swapping page to disk from main memory
2. Terminal used to enter data for a program being executed
3. CPU sends data to remote printer using communication patch

(a) Only 2 (b) 2 and 3

(c) Only 3 (d) None of the above

Answer Key:

1. (c)	**2.** (d)	**3.** (b)	**4.** (c)	**5.** (b)
6. (d)	**7.** (d)	**8.** (b)	**9.** (a)	**10.** (c)
11. (b)	**12.** (b)	**13.** (a)	**14.** (d)	**15.** (c)
16. (d)	**17.** (a)	**18.** (c)	**19.** (c)	**20.** (b)
21. (b)	**22.** (d)	**23.** (b)	**24.** (d)	**25.** (d)

CHAPTER 02

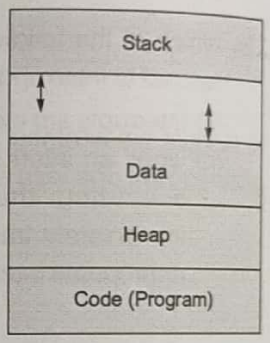
Processes and Threads

2.1 Process

A process is an activity of executing a program. It is a program under execution. Every process needs certain resources to complete its task.

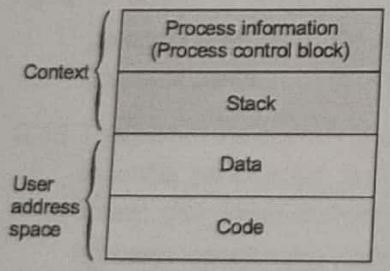
There are two types of processes:

1. **User process:** User processes are executed in user mode and user processes can be preempted while executing.
2. **System process:** System processes are executed in privileged mode. System process executed automatically without preemption.



Abstraction view of a process

2.1.1 Process Description



Every process has an image consists of three components.

1. Executable program [code section]
2. Associated data needed by the program [data section]
3. Execution context of the process [context section]

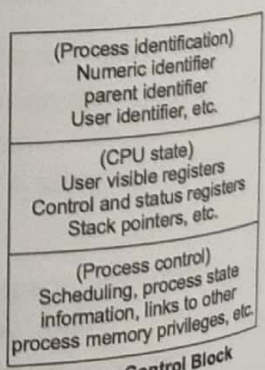
Code and data section of a program are called as "user address space". Context section of a process is managed by operating system which contains stack and process control information.

2.1.2 Process Control Block (PCB)

A process context saved in particular block to control the process is called as "Process Control Block". PCB is also called as task control block. Every process has a PCB that contains the following information.

1. Process Identification Data
2. CPU state information
3. Process control information

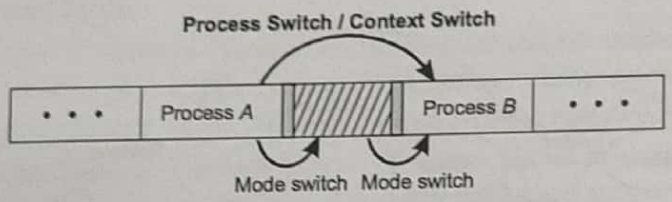
OS maintains a process to manage the processes by maintaining PCB of every process as an entry of process table.



Process Control Block

2.1.3 Process Switch (Context Switch)

1. **Process switch:** Process switch also called as context switch which involves saving the current CPU information, updating the control information and restore the CPU information.
   Process switch includes the following steps:
   (*i*) Save CPU context [Mode switch from user mode to Kernel mode using mode bit].
   (*ii*) Update PCB of current process.
   (*iii*) Move PCB of current process to appropriate queue.
   (*iv*) Select another process for execution [By CPU scheduler].
   (*v*) Update PCB of selected process.
   (*vi*) Update memory management structures.
   (*vii*) Restore CPU context of new PCB [Mode switch from Kernel mode to user mode].



2. Context switch time is dependent on hardware and it is overhead because during context switch the system does not useful work.

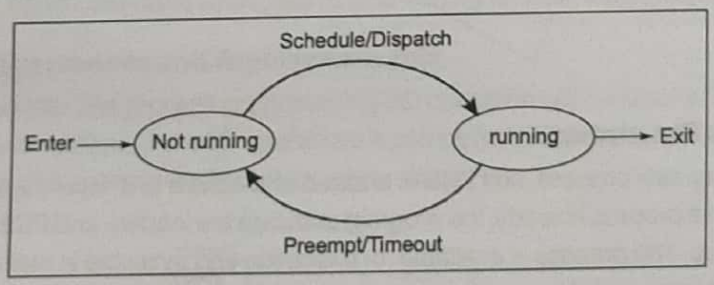
2.2 Process State Models

Process state defines the current activity of the process. Process states are: new, running, ready, blocked, suspended (suspended ready, suspended blocked), etc.

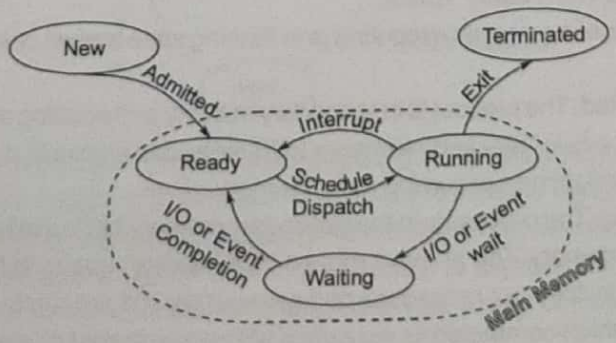
2.2.1 Types of Process State Models

Number of process states are decided by the type of process state model.

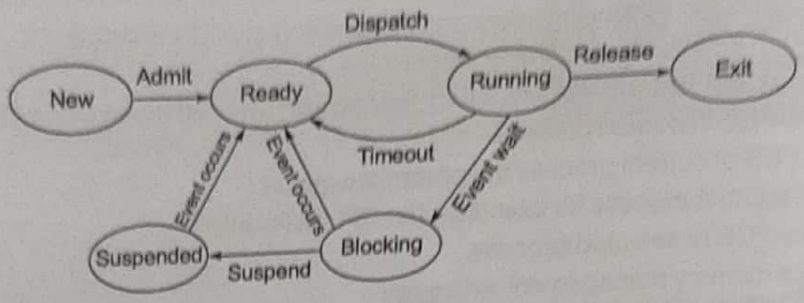
1. **Two state process model:**



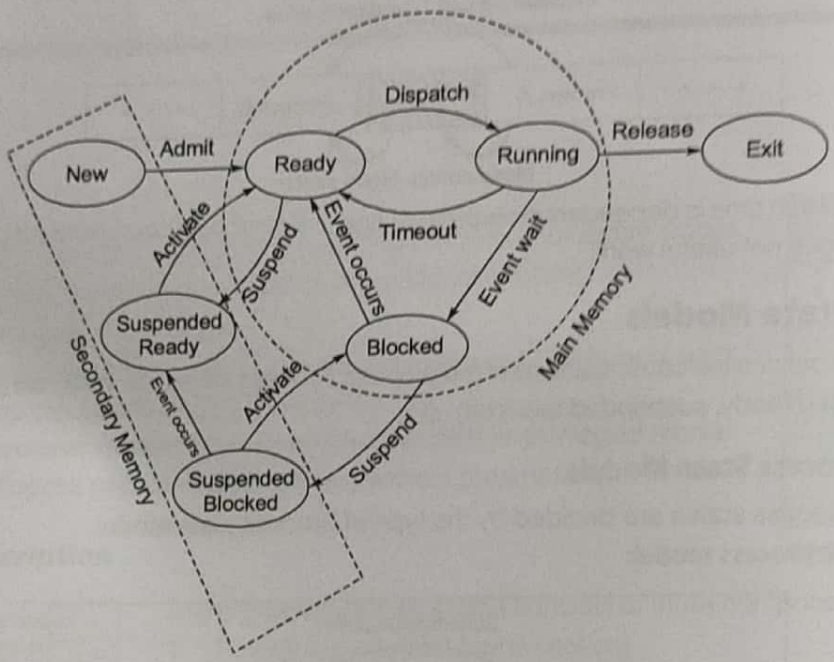
2. **Five state process model:**



3. Six state process model:



4. Seven state process model:



2.2.2 Queues / States Description

- **New:** For every new process, first PCB is created and added to a "new" queue.
- **Ready:** When a process is ready, the program and data are loaded, and PCB of that process kept in "Ready" queue. The process is available for execution and available in main memory.
- **Blocked:** The process is in main memory awaiting an event will be in blocked. When event occurs, the process enters into "Ready" queue.
- **Running:** The process currently executing is in running state atmost one process is in running state at any time.
- **Suspended blocked:** The process is in secondary memory and awaiting an event will be in suspended blocked. The process in "Blocked" will move to "Suspended blocked" due to many reasons such as blocked process might be consuming more memory.
- **Suspended ready:** The process is in the secondary memory but is available for execution whenever it is loaded into memory. The process moved from "Ready" queue to "Suspended Ready" queue due to more priority for other processes and several reasons are exist.
- **Exit:** A process which completed its execution will be terminated by swapping out.

2.2.3 Active and Inactive Jobs

The following processes (active) are in main memory.

- Ready
- Running
- Blocked

The following processes (inactive) are in secondary memory.

- New
- Suspended blocked
- Suspended ready
- Exit

2.2.4 Actions Performed by OS

Timeout/Preemption: The process receives a timer interrupt and relinquishes control back to the OS dispatcher. The OS puts the process in "Ready" queue and dispatches another process to the CPU.

Dispatch: A process in "Ready" queue has been chosen to be the next running process.

Event Wait (Block/I/O Wait): A process invokes an I/O system call for an event that blocks the currently executing process. OS puts the process in "Blocked" queue and dispatches another process to the CPU from "Ready" queue.

Event Occurs (Unblock/ I/O Completion): An I/O system sends an interrupt to CPU to inform the completion of its task. OS may then decide to unblock the process which is waiting in blocked "queue" and puts in "Ready" queue.

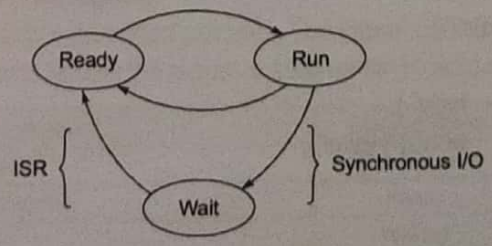
2.2.5 Types of Process

Processes may be categorized as:

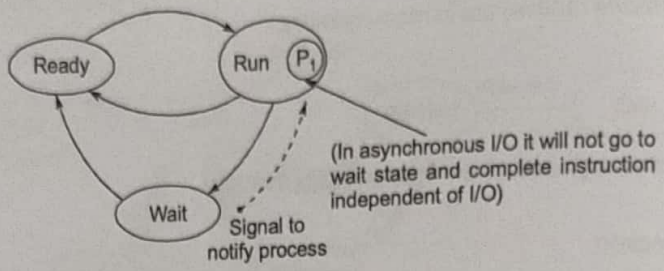
- CPU-bound: Process does not need much I/O service, almost always want the CPU.
- I/O-bound: Short CPU burst times, needs lots of I/O service.
- Interactive: Short CPU burst times, lots of time waiting for user input (keyboard, mouse).

2.2.6 Types of I/O (Syncronous and Asyncronous I/O)

1. **Synchronous I/O:** The process programming I/O operation will be blocked in the block state till I/O operation is completed. Once I/O operations is completed ISR (Interrupt Service Routine) will initiated which places the process from block to ready state.



2. **Asynchronous I/O:** An asynchronous I/O while initiating I/O request a handler function will be registered. The process is not placed in block state it continues to execute remaining code after initiating I/O request. At the point of I/O request is completed the signal mechanism is used to notify the process that data is available and registered handler function is asynchronously invoked. All information about process will be store in handler function like type, what it does etc.?



2.2.7 Process Memory Image

Memory image of a process has following 5 sections which are used while running a process.

- **Text Segment/Test Region:** It stores the machine instructions of stored program and size is fixed.
- **Data Segment:** It stores initialized static and global data of stored program and size is fixed.
- **BSS (Block Stated by Symbol) Segment:** It stores uninitialized static data that is defined in the program (e.g., global uninitialized strings, numbers, structures). Size of BSS is fixed.
- **Heap Segment:** It is dynamically allocated memory. Memory can be allocated and deallocated dynamically at run time.
- **Stack Segment:** It used to maintain the call stack, which holds return addresses, local variables, temporary data, and saved registers. Stack is dynamic.

2.2.8 Important Commands (System Calls)

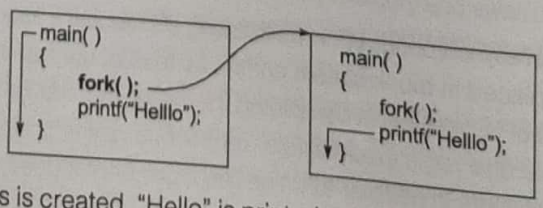
- **Fork:** *Fork* creates a new process. The new process is a copy of the parent. It's running the same program and has the same open files. It is, however, a copy and not a reference to the parent. Any memory modifications or changes to open file status will be invisible to the parent and vice versa. fork system call : (i) Returns –ve value if process creation is unsuccessful, (ii) Returns +ve value if process creation is sucecesful, (iii) Returns 0 to newly created process.
  The parent and child process have same virtual address but physical address will be different.

NOTE: For N fork statement 2^N-1 child process will be created.

Example - 2.1 Find the number of child processes created for the following code and also find how many times "Hello" is printed.

```
main( )
{
    fork( );
    printf("Hello");
}
```

Solution:

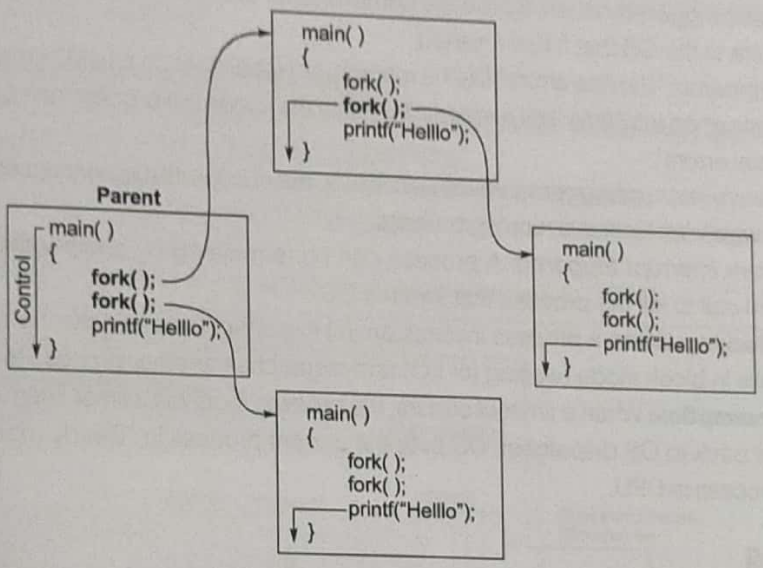


Only one child process is created. "Hello" is printed twice for the given code.

Example - 2.2 Find the number of child processes created for the following code and also find how many times "Hello" is printed.

```
main( )
{
    fork( );
    fork( );
    printf("Hello");
}
```

Solution:



Three child processes are created. "Hello" is printed four times.

- **execve:** *execve* does not create a new process. It loads a new program from the file system to overwrite the current process, reinitializing the process' memory map. It is often used immediately after *fork* to allow have the child process created by *fork* to load and run a new program.
- **exit:** *exit* tells the operating system to terminate the current process.
- **wait:** *wait* allows a parent to wait for and detect the termination of child processes.
- **signal:** *signal* allows a process to detect signals, which include software exceptions, user-defined signals, and death of child signals.
- **kill:** *kill* sends a signal to a specified process. The signal can usually be detected with *signal*. It does not kill the process unless the signal is a kill (or terminate) signal.

Event	Starting State	Ending State
Program started by user	Does not exist	Running
Scheduler dispatches process	Ready	Ready
I/O complete	Waiting	Ready
Process admitted to ready queue for first time	New	Ready
Scheduler preemtps (interrupts) process	Running	Ready
Process finishes execution (task is complete)	Running	Terminated
Process initiates I/O	Running	Waiting

Events of a Process

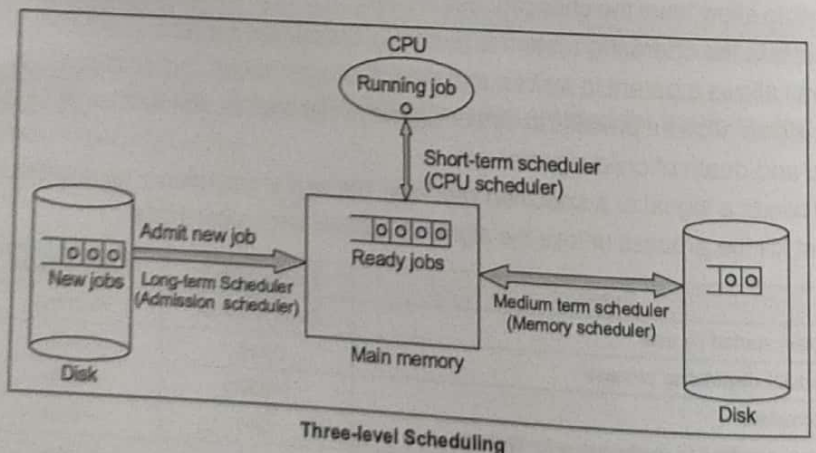
2.3 Operations on a Process

1. **Process creation:** The following events lead to the process creation.
   (a) System initialization (system booting) creates several background processes (email, logon, etc).
   (b) A user requests to create a new process.
   (c) Already running to process (existing) can create a new process.
   (d) Batch system takes initiation of a batch job
2. **Process termination:** The following events lead to the process termination
   (a) *Process-triggered:* When a process completes its execution it executes "exit" system call to indicate to the OS that it has finished.
   (b) *OS-triggered:* "Service errors" like no memory left for allocation and I/O errors "Preemption" of a process when total time limit exceeded. In both the cases the process can be terminated, called as "total errors".
   (c) *Hardware interrupt triggered:* Arithmetic errors, out of bounds memory access, etc. are program bugs which terminates a running process.
   (d) *Software interrupt triggered:* A process can be terminated by another process by executing system call to kill the process that informs OS.
3. **Process blocking:** When a process invokes an I/O system call that blocks the process and OS put this process in block mode (waiting for I/O) and dispatches another process to CPU.
4. **Process preemption:** When a timeout occurs, the process receives a timer interrupt and relinquishes the control back to OS dispatcher. OS puts the current process in "Ready mode" and dispatches another process to CPU.

2.4 Scheduling

Scheduling depends on three scheduler in the system

1. Long-term scheduler
2. Medium-term scheduler
3. Short-term scheduler



Three-level Scheduling

2.4.1 Long-term Scheduler (LTS)

- It involves in a decision to add a job to the pool of processes to be executed.
- It controls the degree of programming by taking the decision that the number of processes can be ready for execution by keeping in ready queue (in main memory).

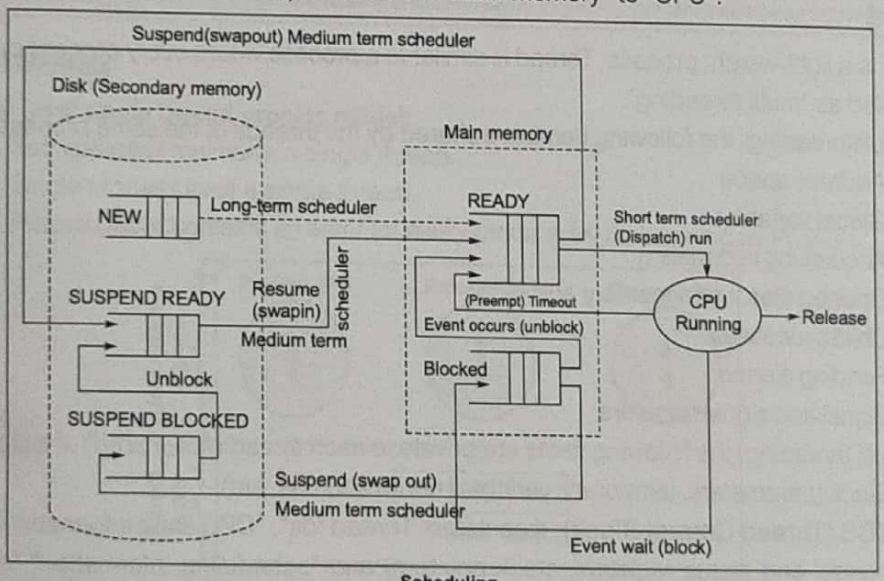
- It creates a new process and also decides which process move from "New" to "Suspended Ready" (disk) / "Ready" (main memory).
- It involves in a transition of a job from **"secondary memory to main memory"**.

2.4.2 Medium-term Scheduler

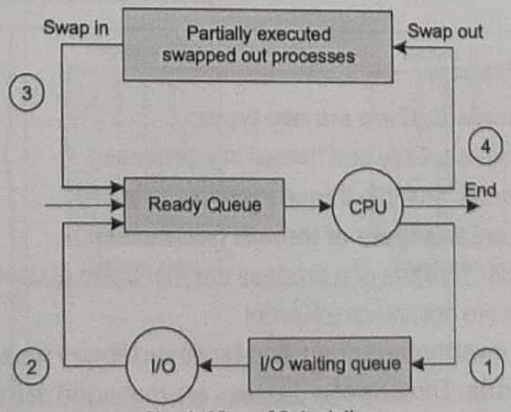
- It involves in the decision to swap processes that partially or fully in main memory.
- It acts as swapper by doing swap in and swap out operations.
- Swap in operation moves a process from suspended ready (disk) to ready (main memory) state.
- Swap out operation moves a process from suspended blocked (disk) to blocked (main memory) state.
- It involves in a transition of a job from **"main memory to secondary memory"** or **"secondary memory to main memory"**.

2.4.3 Short-term Scheduler (CPU Scheduling)

- It involves in a decision to select a process from ready state (in memory) which will be executed by the processor. This process is called "dispatching".
- It selects a process from ready state and changes to running state.
- Dispatcher may help to dispatch the select process by short-term scheduling as quickly as possible to the CPU.
- It involves in a transition of a process from "main memory" to "CPU".



Scheduling



Simple View of Scheduling

2.4.4 CPU Scheduler Vs Short-term Scheduler

Long-term scheduler

- Selects processes from job queue
- Loads the selected processes into memory for execution
- Updates the ready queue
- Controls the *degree of multiprogramming* (the number of processes in the main memory)
- Not executed as frequently as the short-term scheduler
- Should generate a good mix of CPU-bound and I/O-bound processes
- May not be present in some systems (like time sharing systems)

Short-term scheduler

- Selects processes from ready queue
- Allocates CPU to the selected process
- Dispatches the process
- Executed frequently (every few milliseconds, like 10 msec)
- Must make a decision quickly *(* must be extremely fast)

2.5 Thread

A thread is a light-weight process. Thread is similar to a process where every process can have one or more threads called as "multi threading".

- In multithreading, the following items are shared by the threads of the same process.
  (a) Address space
  (b) Global variables
  (c) Accounting information
  (d) Opened files, used memory and I/O
  (e) Child processes
  (f) Pending alarms
  (g) Signal and signal handlers
- In multi threading, the following items are private to each thread (not shared) of a process.
  (a) Stack (parameters, temporary, variables return address, etc).
  (b) **TCB (Thread Control Block):** It contains "Thread Ids", "CPU state information" (user visible, control and status registers, stack pointers) and "scheduling information" (state of thread priority, etc).

2.5.1 Types of Threads per Process

- **Based on number of threads, there are two types:**
  (a) **Single thread process:** Only one thread in a process.
  (b) **Multi thread process:** Multiple threads within a process.
- **Based on level, there are two types of threads per process:**
  (a) **User-level threads:** Threads of a process are managed at user level.
  User level threads are not known toKernel
  Execution of user level thread scheduling by thread library (user mode).
  (b) **Kernel-level threads:** Threads of a process are managed at Kernel level.
  Kernel level threads are known to Kernel and scheduled by OS (Kernel mode).

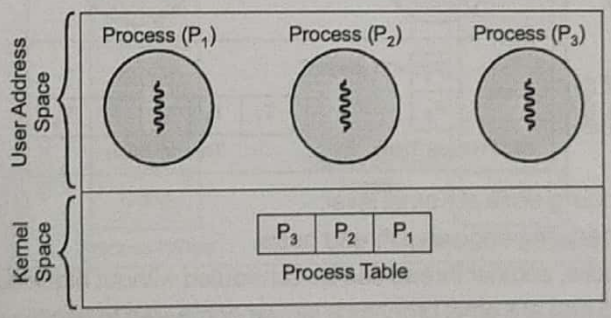
- **User-level Threads per Process Vs Kernel level threads per process Vs Process:**

S.No.	Process	Kernel Thread	User Thread
1.	Process is high overhead	Kernel thread is medium overhead	User thread is low overhead
2.	No sharing between processes	Share address space	Share address space
3.	Communication between the processes need OS support	Communication between the threads need OS support	Communication between the threads is done at user level (does not require OS support).
4.	Blocking one process does not affect the other processes	Blocking one thread does not affect the other threads of a process	Blocking one thread will block whole process of the thread
5.	Each process can run on different processor in multiprocessor system to support parallelism	Each thread can run on different processor in multi-processor system to support parallelism and all threads of a process are managed by Kernel.	All threads should run on only one processor and only one thread runs at a time. Because all threads of a process are dependent and managed by process.
6.	Types of a process: 1. User level single thread process 2. User level multi-thread process 3. Kernel level single thread process 4. Kernel level multi-thread process	Types of a process: 1. Kernel level single thread process 2. Kernel level multi-thread process	Types of a process: 1. User level single thread process 2. User level multi-thread process
7.	Process scheduling done by OS using process table	Thread scheduling done by OS using thread table	Thread scheduling done by thread library using thread table
8.	Process is a heavy weight process	Kernel thread is a light weight process and need not be associated with process	User thread is a light weight process and it belongs to a process
9.	A process can be suspended without affecting other processes	Thread suspend not possible suspending a process makes all of its thread stop running.	Thread suspend not possible suspending a process makes the current running thread stops.

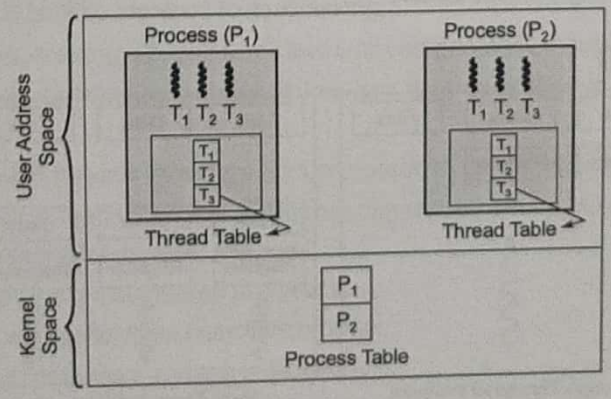
2.5.2 Thread Models

1. User level single thread process model:

- Each process maintains a single thread.
- Single process itself a single thread.
- Process table contains an entry for every process by maintaining its PCB.

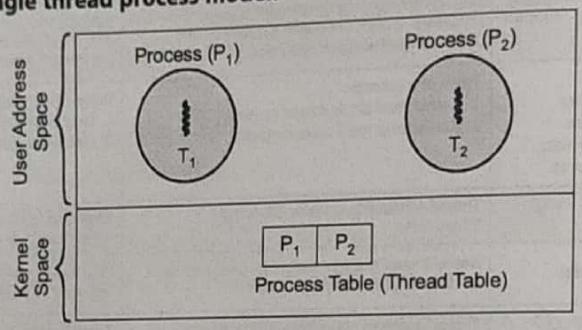


2. User level multi thread process model:



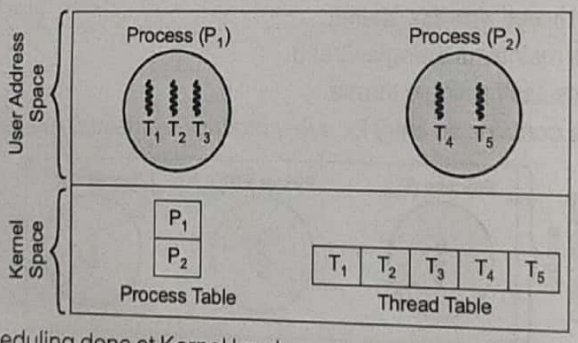
- Each process contains multiple threads.
- All threads of the process scheduled by a thread library at user level.
- Thread switching can be done faster than process switching.
- Thread switching is independent of OS which can be done within a process.
- Blocking one thread makes blocking of entire process.
- Thread table maintains TCB of each thread of a process.
- Thread scheduling happens within a process and not known to Kernel.

3. Kernel level single thread process model:



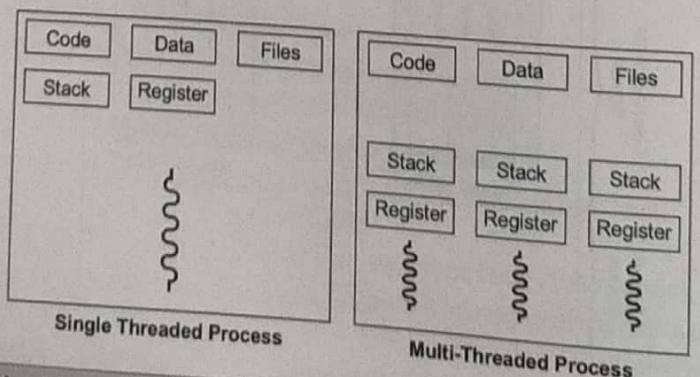
- Every process has single kernel level thread and when a process has single thread, process table can act as thread table.

4. Kernel level multi thread process model:



- Thread scheduling done at Kernel level.
- Fine grain scheduling done on a thread basis.
- If a thread blocks, another thread can be scheduled without blocking the whole process.
- Thread scheduling at Kernel process is slower compared to user level thread scheduling.
- Thread switching involves switch.

2.6 Multi Threading



It refers to the ability of an OS to support multiple threads of execution within a single process. Multi threading gives an advantage of separation of resource ownership (Address space, and used files I/O) and execution (TCB and scheduling).

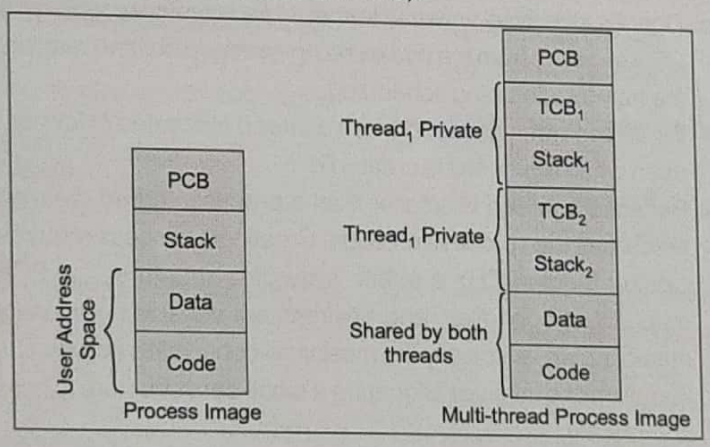
Resources like code data, files and memory will be shared among all threads within process but stacks and registered can't be shared and every new thread will have their on stacks and registers (shown in figure).

Advantage of Multi Threading

- Performs foreground and background work in parallel.
- Increases responsiveness
- Allows synchronous processing (separates the execution of independent tasks).
- Increases execution speed (overlaps CPU execution time and I/O wait time).
- Concurrency can be achieved by multi threading.
- Reduces context switch time.
- Enhances the throughput.

2.6.1 Process Image Vs Multi Thread Process Image

- Process image contains PCB, Stack, Data, and Code.
- Multi thread process images contains PCB for parent process, TCB for each thread, stack for each thread, common address space (data and code).



2.6.2 Multithreading Vs Multitasking

- Multithreading is faster compared to multitasking.
- In multithreading, creating a thread takes less time compared to creation of a process in multitasking.
- In multithreading, terminating a thread takes less time compared to termination of a process in multitasking.
- Switching between threads takes less time compared to switching between processes.
- In multithreading, threads of the same process can share memory, I/O and files. So communication between threads can be done at user level.
- Threads are light weight compared to a process.
- Multithreading applications are webclient browser.
- Webserver, word processor and spreadsheets.

2.7 Co-operating Processes (Inter Process Communication)

The concurrent processes executing in the operating system may be either independent processes or co-operating processes. A process is independent if it cannot affect or be affected by the other processes executing in the system. A process is co-operating if it can affect or be affected by the other processes executing in the system. Any process that shares data with other processes is a co-operating process.

There are several reasons for providing an environment that allows process co-operation: Information Sharing, Computation speedup, Modularity and Convenience.

Co-operating processes must communicate via interprocess communication. Some interprocess communications are: Message-Passing System, Naming, Synchronization and Buffering.

Remember

Q.1 Describe the actions a kernel takes to context switch between processes.

Ans: In general, the operating system must save the state of the currently running process and restore the state of the currently running process and restore the state of the process scheduled to be run next. Saving the state of a process typically includes the values of all the CPU registers in addition to memory allocation. Context switches must also perform many architecture-specific operations, including flushing data and instruction caches.

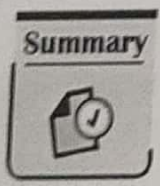
Q.2 Describe the actions taken by a kernel to context switch between kernel-level threads.

Ans: Context switching between kernel threads typically requires saving the value of the CPU registers from the thread being switched out and restoring the CPU registers of the new thread being scheduled.

Q.3 What resources are used when a thread is created? How do they differ from those used when a process is created?

Ans: Because a thread is smaller than a process, thread creation typically uses fewer resources than process creation. Creating a process requires allocating a Process Control Block (PCB), a rather large data structure. The PCB includes a memory map, list of open files, and environment variables. Allocating and managing the memory map is typically the most time-consuming activity. Creating either a user or kernel thread involves allocating a small data structure to hold a register set, stack, and priority.

Summary



- **Process:** An instance of execution of a program that is identifiable and controllable by the operating system.
- **Process Control Block (PCB):** A data structure that contains information about the current status and characteristics of a process.
- **Process status:** Information stored in the job's PCB that indicates the current position of the job and the resources responsible for that status.
- **Thread:** A portion of a program that can run independently of other portions. Multithreaded applications programs can have several threads running at one time with the same or different priorities. A process consists of the actual resources and actions of a computation. A thread is a light-weight process
- **Queue:** A linked list of PCBs that indicates the order in which jobs or processes will be serviced.

- **Task/Process:** The basic unit of concurrent programming languages that defines a sequence of instructions that may be executed in parallel with other similar units.
- **Job Scheduler/High level Scheduler/Long-term Scheduler:** Selects jobs from a job queue of incoming jobs based on each job's characteristics.
- **Middle-level scheduler/Medium term Scheduler:** A scheduler used by the Processor Manager when the system to remove active processes from memory becomes overloaded. The middle-level scheduler swaps these processes back into memory when the system overload has cleared.
- **CPU Scheduler/Low-level scheduler/Process Scheduler:** The low-level scheduler of the Processor Manager that establishes the order in which processes in the READY queue will be served by the CPU.
- **Running process:** The process is currently executing code on the CPU
- **Ready process:** The process is not currently executing code but is ready to do so if the operating system would give it the chance
- **Blocked (or waiting) process:** The process is waiting for some event to occur and will not ready to execute instructions until this event is ready.

Student's Assignment

Q.1 Consider the following components in a computer system

C1: Computer Hardware
C2: Application Programs
C3: Utilities

Find the place of operating system using the above components.

(a) Inbetween C1 and C2
(b) Inbetween C2 and C3
(c) Inbetween C1 and C3
(d) None of the above

Q.2 Which of the following instruction is allowed only in Kernel mode?

(a) Switch user mode to Kernel mode
(b) Read the time
(c) Disable all interrupts
(d) None of these

Q.3 Which of the following is advantage of microkernel approach to the system design?

(a) Adding a new service does not require modifying the Kernel.
(b) It is more secure because more operations are done in user mode than in Kernel mode.
(c) Both (a) and (b)
(d) Neither (a) nor (b)

Q.4 Which of the following operations need not to be privileged?

(a) Read the clock (b) Access I/O device
(c) Clear the memory (d) None of these

Q.5 Identify one of the following which need not be part of operating system?

(a) CPU scheduling
(b) Page replacement
(c) Demand paging and virtual memory
(d) Compiler

Q.6 Which of the following is hardware generated signal?

(a) Interrupt (b) Trap
(c) Both (a) and (b) (d) Neither (a) nor (b)

Q.7 Identify the scheduler which involves only in the decision for selection of partially serviced jobs?

(a) Short-term scheduler
(b) Long-term scheduler
(c) Medium-term scheduler
(d) None of these

Q.8 In multi thread process, which of the following is shared by the threads of a process?
(a) CPU state (b) Stack
(c) Address space (d) All of these

Q.9 Which of the following statements are correct?
(a) Blocking one thread does not affect other threads of the same process in Kernel mode.
(b) Blocking one thread will block the whole process in user mode.
(c) Both (a) and (b)
(d) None of these

Q.10 Match the following groups

Group-I (Scheduler)
A. Long-term scheduler
B. Medium-term scheduler
C. Short-term scheduler

Group-2 (Transition of process)
1. New to ready state
2. Ready to running state
3. Suspended to blocked

Codes:

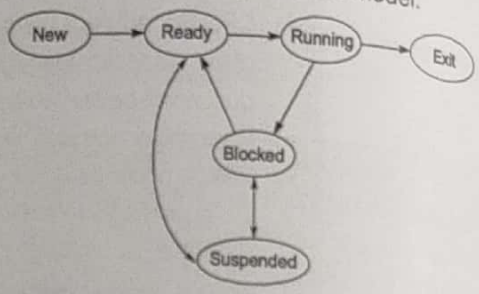
	A	B	C
(a)	1	2	3
(b)	1	3	2
(c)	3	1	2
(d)	2	3	1

Q.11 Identify the correct statement from the following
(a) Job may transit from "Ready" to "Suspend ready"
(b) Job may transit from "Suspend blocked" to "Suspend ready"
(c) Job may transit from "Suspend ready" to "Ready"
(d) All of these

Q.12 Which of the following scheduling can be done by thread library?
(a) process scheduling
(b) Kernel thread scheduling
(c) User thread scheduling
(d) None of these

Q.13 If a system uses 5-state model for processes execution (New, Ready, Running, Blocked, Exit), then which of the following state processes are in the main memory?
(a) New, Ready
(b) Ready, Running
(c) Ready, Running, blocked
(d) New, Ready, Running, blocked

Q.14 Consider the following six-state model.



Identify the states in which processes are in the secondary memory (disk).
(a) New
(b) New, Suspended
(c) New, Suspended, blocked
(d) New, Suspended, blocked, Ready

Q.15 Let t_1 be the time taken for mode switch and t_2 be the time taken for process switch. Then find the relation between t_1 and t_2.
(a) $t_1 = t_2$ (b) $t_1 > t_2$
(c) $t_1 < t_2$ (d) None of these

Q.16 A system can support a process with user level threads or Kernel level threads. Identify the correct statement from the following.
(a) User thread is known to Kernel, blocking one thread causes entire thread is blocked.
(b) Kernel level threads: Contain separate TCB for each thread within a process, blocking one thread is independent to other thread
(c) Both (a) and (b)
(d) Neither (a) nor (b)

Q.17 Which of the following is not TCB information.
(a) Thread identifier
(b) CPU state information: control and status registers, stack pointers
(c) Scheduling: state, priority awaited event
(d) Used memory and I/O, program code (Address space)

Q.18 Choose the incorrect statement from the following.
(a) Thread can't suspended (Swapped out) from blocked state to suspended state.
(b) Processes can be suspended from blocked to suspended state
(c) Threads are lightweight processes
(d) None of these

Q.19 Monitor mode is called as
(a) Privileged mode (b) Kernel mode
(c) System mode (d) All of these

Q.20 Which of the following statement is false.
(a) Interrupt is hardware generated interrupt
(b) Trap is software generated interrupt
(c) Trap can be used to call OS routines or to catch arithmetic errors
(d) None of these

Q.21 Which of the following statements are correct when user level threads are compared to Kernel level threads.
(a) User level threads require memory management where Kernel threads do not.
(b) User level thread scheduling is faster than Kernel thread scheduling.
(c) Both (a) and (b)
(d) Neither (a) nor (b)

Q.22 Which of the following can run in parallel on different processors in a multiprocessor?
(a) Processes
(b) Kernel threads of a process
(c) Both (a) and (b)
(d) User threads of a process

Q.23 Which of the following are equal things when there is only one CPU in a system.
(a) Multiprogramming and Multitasking
(b) Multiprocessing and Multiprogramming
(c) Multitasking and Multiprocessing
(d) None of these

Q.24 Which of the following item is not a part of Process Control Block (PCB)?
(a) CPU state information
(b) Scheduling information of a process
(c) Stack section
(d) None of these

Q.25 In multithreading, which of the following statements is correct?
(a) All threads of a process share same address space and resources
(b) Creation of a new thread only involves allocating a new stack and a new thread Control Block
(c) Both (a) and (b)
(d) Neither (a) nor (b)

Answer Key:

1. (a)	2. (c)	3. (c)	4. (a)	5. (d)
6. (a)	7. (c)	8. (c)	9. (c)	10. (b)
11. (d)	12. (c)	13. (c)	14. (b)	15. (c)
16. (b)	17. (d)	18. (d)	19. (d)	20. (d)
21. (b)	22. (c)	23. (a)	24. (c)	25. (c)

CHAPTER 03

CPU Scheduling

3.1 Introduction

Short-term scheduling is called as CPU scheduling or process scheduling or memory scheduling.

CPU scheduler schedules a process which is loaded in memory (Ready state) to run in CPU. The action of loading the process state into the CPU is known as "dispatching". CPU scheduler decides which "Ready" process to run next in CPU and which "Running" process to time-out.

Process may use both CPU and I/O. If process need more CPU execution and less I/O service then such process is called as "CPU bound process" or "Compute bound process". If process need less CPU execution and more I/O service then such process is called as "IO-bound process". CPU bound processes keep the CPU busy and IO bound processes keep the disk busy.

CPU scheduler is used in multiprogramming system (batch and interactive) by switching the processes called as "context switch" or "process switch". During context switch there is no useful work done by CPU to user.

The type of processes in the system will affect the performance of scheduling algorithms. A short-term CPU scheduling decision is needed when a process:

1. switches from a running to a waiting state (non-preemptive)
2. switches from a running to a ready state (preemptive)
3. switches from a waiting to a ready state (preemptive)
4. terminates (non-preemptive)

The dispatcher is the module/function that gives control of the CPU to the process selected by the short-term scheduler. This function involves: Switching context, switching to user mode and jumping to the proper location in the user program to restart the program. The time it takes for the dispatcher to stop one process and start another running is known as the dispatch latency.

In Scheduling, selection of a process may occurs:

- When a new process arrives.
- When a waiting process changes state to ready.
- When a running process changes state to waiting (I/O request).

- When a running process changes state to ready (interrupt).
- Every q seconds (q is quantum or time slice or time quantum).
- When priority of a ready process exceeds the priority of a running process.

Example-3.1 Consider a system with 'n' CPU processors than what is minimum and maximum number of processor that may be present on running, ready 0 block stage.

Solution:

	Min	Max
Ready	0	Any
Run	0	n
Block	0	Any

'n' processes, hence n running states

Depends on maximum degree of multiprogramming in a system.

3.2 Goals of CPU Scheduling

1. **General goals for all systems:**
   (a) Fairness (comparable process get comparable service)
   (b) Compliance to systems policy
   (c) Keep system busy (CPU and I/O devices should be utilized fully)
2. **Interactive system specific goals:**
   (a) Response time (respond to requests quickly)
   (b) Proportionality time (meet users expectation)
3. **Batch system specific goals:**
   (a) Throughput (Maximize the number of complete jobs per time unit)
   (b) Turn around time (latency) minimizes the time between submission and termination of job.
   (c) CPU utilization (keep CPU busy all the time)
4. **Real time system specific goals:**
   (a) Meeting deadlines
   (b) Predictability

3.3 Kinds of CPU Scheduling Algorithms

There are two kinds of scheduling algorithms

(a) **Co-operative scheduling (Non-preemptive scheduling):** A process in CPU runs until it blocks an I/O or terminated by its completion.
Non-preemptive scheduler gives full permission to a process to execute in CPU. Other processes may be responded slowly.

(b) **Pre-emptive scheduling:** A process runs by following clock interrupts, when a time-out occurs it forcibly switched from running state to ready state.
Pre-emptive scheduler preempts a process which involves process switching. Many process switching will be not good for overall utilization of CPU.

3.4 Scheduling Algorithms

Scheduling Metrics

- Arrival Time (AT): The time at which the process became "Ready".
- Wait Time (WT): The time spent waiting for CPU to complete the execution of a process.
- Service Time (ST): The time spent executing in CPU. "Service time" also called as "Burst time (BT)" or "Processing time" or "execution time".

- **Turn Around Time (TAT):** The total time spent waiting for CPU and executing in CPU. Turn around time also called as "Response time".

$$TAT = WT + ST$$
$$WT = TAT - ST = TAT - BT$$

- **Completion Time (CT):** The time at which the process finishes its execution in CPU. Completion time is also called as "Finish time".

$$TAT = CT - AT = WT + ST$$

- **Averages (mean) Waiting Time (WT_{Avg})** $= \dfrac{\text{Sum of waiting time of processes}}{\text{Number of processes}}$

$$WT_{Avg} = \frac{\sum_{i=1}^{n} WT_i}{n}$$; where n is number of processes.

- **Average (mean) Turn Around Time (TAT_{Avg})** $= \dfrac{\text{Sum of TATs of processes}}{\text{Number of processes}}$

$$TAT_{Avg} = \frac{\sum_{i=1}^{n} TAT_i}{n}$$; where n is number of processes.

- **Processor Utilization:** The ratio of busy time of the processor to the total time passes for processes to finish. We would like to keep the processor as busy as possible.

  Processor Utilization = (Processor buy time) / (Processor busy time + Processor idle time)

- **Throughput:** The measure of work done in a unit time interval.

  Throughput = (Number of processes completed) / (Time Unit)

- **Response Time (RT):** The amount of time it takes to start responding to a request. This criterion is important for interactive systems.

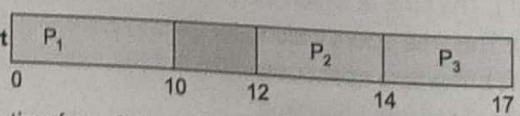
$$RT = t_{(\text{first response})} - t_{(\text{submission of request})}$$

Scheduling Algorithms

1. **Scheduling in batch systems:**
   - First-Come-First-Served (FCFS)
   - Shortest Job First (SJF)
   - Shortest Remaining Time First (SRTF)
2. **Scheduling in interactive systems:**
   - Round-Robin (RR)
   - Highest Response Ratio Next (HRRN)
   - Priority scheduling with multilevel queue
   - Priority scheduling with multilevel queue feedback

Gantt Chart

Gantt Chart shows the execution time period of processes. Sample Gantt chart is shown below, which begins from 0 time unit.



- Process P_1 started executing from 0 time unit and finished just before 10 unit.

- From 10 unit to 12 unit duration, the CPU is idle because no process in the ready queue.
- Process P_2 started executing from 12 unit time and finished at 14 unit.
- Process P_3 started executing from 14 unit time and finished at 17 unit.

Remember

Dispatcher: The dispatcher is the module that gives control of the CPU to the process selected by the short term scheduler. This function involves: (a) Switching context, (b) Switching to user mode and (c) Jumping to the proper location in the user program to restart that program.

Dispatch Latency: The time it takes for the dispatcher to stop one process and start another running is known as the dispatch latency.

3.4.1 FCFS Scheduling Algorithm

- FCFS is implemented using queue which serves first in first out manner.
- Processes are assigned to the CPU based on their order of requests (order of arrivals)
- "Ready queue" maintains all processes in their arrival orders. First arrived process will be executed first. Second process will be waiting in ready queue as first when first process executing in CPU.
- FCFS scheduling is non-preemptive algorithm.
- When a process competes its execution in CPU, the next process selected from the "ready queue".
- If a process blocked to service I/O, then blocked process kept in blocked queue (device queue) and next process is selected from ready queue to execute in CPU.

Advantage: Simple to implement.

Disadvantage: Convoy effect (shorter jobs get stuck behind longer jobs, it effects lower utilization of CPU and I/O). If shorter processes arrived after longer jobs, then shorter processes responds very late.

Example-3.2 Consider the following table with service times of processes. Assume that all processes are arrived at time 0. Ready queue has P_1 first, then P_2, P_3 and P_4 in order

Process	P_1	P_2	P_3	P_4
Service time	3	5	2	4

Computer the following metrics using FCFS scheduling.

(*i*) Waiting time of each process

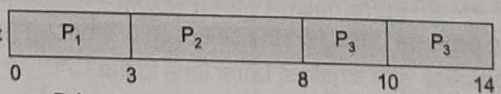
(*ii*) Turn around time of all processes

(*iii*) Average waiting time of all processes

(*iv*) Average turn around time of all processes.

Solution:

Gantt Chart



(*i*) WT_1 (waiting time of process P_1) = 0

$WT_2 = 3$ (P_2 started executing from 3 and arrived at 0)

$WT_3 = 8$

$WT_4 = 14$

(*ii*) $TAT_1 = WT_1 + ST_1 = 0 + 3 = 3$ or $TAT_1 = CT_1 - AT_1 = 3 - 0 = 3$

$TAT_2 = CT_2 - AT_2 = 8 - 0 = 8$

$TAT_3 = CT_3 - AT_3 = 10 - 0 = 10$

$TAT_4 = CT_4 - AT_4 = 14 - 0 = 14$

(*iii*) Average Waiting Time $(WT_{Avg}) = \frac{\text{Sum of waiting time of processes}}{\text{Number of processes}}$

$$WT_{Avg} = \frac{0+3+8+10}{4} = \frac{21}{4} = 5.25$$

(*iv*) Average Turn Around Time $(TAT_{Avg}) = \frac{\text{Sum of TATs of processes}}{\text{Number of processes}}$

$$TAT_{Avg} = \frac{3+8+10+14}{4} = \frac{35}{4} = 8.75$$

Example-3.3 Consider the following table with Processes, Arrival times and Burst times.

Process	P_1	P_2	P_3	P_4
Arrival time	0	1	3	9
Bust time (Service time)	3	5	2	4

Compute the following metrics using the FCFS scheduling.
(*i*) Waiting time of each process, (*ii*) Turn around time of all processes, (*iii*) Average waiting time of all processes and (*iv*) Average turn around time of all processes.

Solution:

Gantt Chart

P_1	P_2	P_3	P_3
0	3	8	10	14

(*i*) WT_1 (waiting time of each process P_1) = 0
$WT_2 = 3 - 1 = 2$ (P_2 arrived at 1 but executed at 3), $WT_3 = 8 - 3 = 5$, $WT_4 = 10 - 9 = 1$

(*ii*) Turn around time of each process
$TAT_1 = 3 - 0 = 3$, $TAT_2 = 8 - 1 = 7$, $TAT_3 = 10 - 3 = 7$, $TAT_4 = 14 - 9 = 5$

(*iii*) Average Waiting Time $(WT_{Avg}) = \frac{0+2+5+1}{4} = 2$

(*iv*) Average Turn Around Time $(TAT_{Avg}) = \frac{3+7+7+5}{4} = 5.5$

3.4.2 Shortest Job First Scheduling Algorithm (SJF) Or Shortest Process Next

- SJF schedules a process based on the selection of smallest (shortest) burst time of processes.
- SJF assigns a process with smallest burst time to the CPU.
- SJF is non-preemptive algorithm.
- In SJF, service time of every process known before execution.
- Shortest job selected first, because it finishes earliest. SJF is also called as "shortest time to competion first".
- If all jobs are having same service time, SJF works same as FCFS.

Advantage: Minimizes average response time (optimal).

Disadvantage: Hard to predict the future (service time of process). Starvation may occur if so many shortest processes are requesting then longer processes may starve.

Example-3.4 Consider the following table with four processes and their service times.

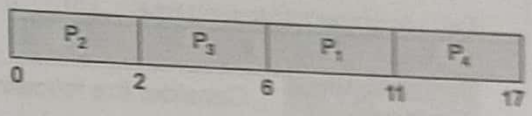
Process	Burst Time (Service Time)
P_1	5
P_2	2
P_3	4
P_4	6

Assume that all processes are arrived at the same time. First process starts execution at time 0. Compute the following with SJF scheduling.

(*i*) Waiting time of each process
(*ii*) Turn around time of each process
(*iii*) Average waiting time of processes
(*iv*) Average turn around time of processes

Solution:

(*i*) Waiting time of each process



$WT_1 = 6 - 0 = 6$
$WT_2 = 0$
$WT_3 = 2$
$WT_4 = 11$

(*ii*) Turn around time of each process

$TAT_1 = 11 - 0 = 11$
$TAT_2 = 2$
$TAT_3 = 6$
$TAT_4 = 17$

(*iii*) Average waiting time

$$WT_{Avg} = \frac{6+0+2+11}{4} = \frac{19}{4} = 4.75$$

(*iv*) Average turnaround time

$$TAT_{Avg} = \frac{11+2+6+17}{4} = \frac{36}{4} = 9$$

Example-3.5 Consider the following table

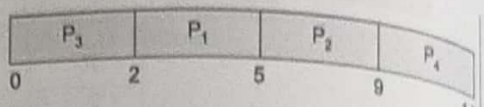
Process	Service Time	Arrival Time
P_1	3	2
P_2	4	0
P_3	2	0
P_4	5	3

Compute the following with SJF scheduling.

(*i*) Waiting time of each process
(*ii*) Turn around time of each process
(*iii*) Average waiting time of processes
(*iv*) Average turn around time of processes

Solution:

(*i*) Waiting time of each process



$WT_1 = 2 - 2 = 0$
$WT_2 = 5 - 0 = 5$
$WT_3 = 0$
$WT_4 = 9 - 3 = 6$

(*ii*) Turn around time of each process

$TAT_1 = 5 - 2 = 3$
$TAT_2 = 9 - 0 = 9$
$TAT_3 = 2 - 0 = 2$
$TAT_4 = 14 - 3 = 11$

(*iii*) Average waiting time $WT_{Avg} = \frac{0+5+0+6}{4} = \frac{11}{4} = 2.75$

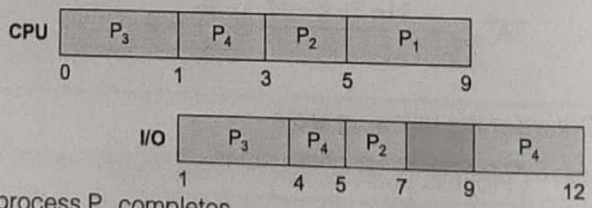
(*iv*) Average turnaround time $TAT_{Avg} = \frac{3+9+2+11}{4} = \frac{25}{4} = 6.25$

Example-3.6 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 the same time. First process starts executed from '0' time unit. Every process completes its CPU request then gets I/O service.
If any two processes need same amount of CPU service time then prefer the process which has less I/0 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.

Solution:



At time unit '5', the process P_4 completes.

3.4.3 Shortest Remaining Time First (SRTF)

- Preemptive SJF is called as 'Shortest Remaining Time First'.
- SRTF scheduling is similar to SJF, only the difference is SRTF can preempt the job while it executing in CPU if any smaller job needs to execute compared to remaining time of current job.
- SRTF can lead to starvation if smaller jobs are many.
- SRTF minimizes the average turnaround time.
- If all processes are arrived at the same time, the SRTF works same as SJF algorithm.
- It is hard to predict the future to find service time of every process before execution in SRTF algorithm.

Example-3.7 Consider the following table

Process	Arrival Time	Service Time
P_1	0	6
P_2	1	3
P_3	2	1
P_4	3	4

Compute the following using the SRTF scheduling.

(*i*) Waiting time of each process

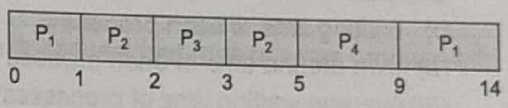
(*ii*) Turn around time of each process

(*iii*) Average waiting time of processes

(*iv*) Average turn around time of processes

Solution:

(*i*) Waiting time of each process



$WT_1 = 8$

$WT_2 = 1$

$WT_3 = 0$

$WT_4 = 2$

(*ii*) Turn around time of each process

$TAT_1 = 14$

$TAT_2 = 5 - 1 = 4$

$TAT_3 = 3 - 2 = 1$

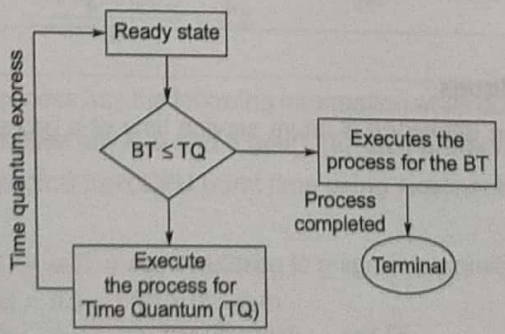
$TAT_4 = 9 - 3 = 6$

(*iii*) Average waiting time $WT_{Avg} = \frac{8+1+0+2}{4} = \frac{11}{4} = 2.75$

(*iv*) Average turnaround time $TAT_{Avg} = \frac{14+4+1+6}{4} = \frac{25}{4} = 6.25$

3.4.4 Round Robin Scheduling (RR)

- RR scheduling is a preemptive FCFS based on a **timeout interval** (quantum or time slice).
- A running process is preempted (interrupted) by the clock and process will be kept in ready state then submits a process from ready queue into CPU.
- Small time quantum leads to more **context switches** (overhead).



- Very large time quantum (time quantum greater than every process service time) makes RR scheduling will work same as FCFS.

- RR scheduling is good for shorter jobs.
- RR scheduling is poor when all jobs are having same length.
- If n processes are in ready queue and time quantum is q the every process gets CPU service less than $(n-1)q$ time units.

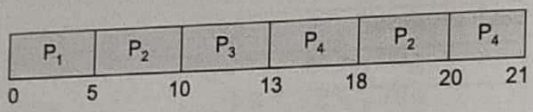
Example-3.8 Consider the following table

Process	Service Time	Arrival Time
P_1	5	0
P_2	7	1
P_3	3	3
P_4	6	4

Compute the following using the RR scheduling with time quantum = 5.

(*i*) Waiting time of each process
(*ii*) Turn around time of each process
(*iii*) Average waiting time of processes
(*iv*) Average turn around time of processes

Solution:



(*i*) Waiting time of each process

$$WT_1 = 0$$
$$WT_2 = (5-1) + (18-10) = 4 + 8 = 12$$
$$WT_3 = 10 - 3 = 7$$
$$WT_4 = (13-4) + (20-18) = 9 + 2 = 11$$

(*ii*) Turn around time of each process

$TAT_1 = 5 - 0 = 5$, $TAT_2 = 20 - 1 = 19$, $TAT_3 = 13 - 3 = 10$, $TAT_4 = 21 - 4 = 17$

(*iii*) Average waiting time $WT_{Avg} = \frac{0+12+7+11}{4} = \frac{30}{4} = 7.5$

(*iv*) Average turnaround time $TAT_{Avg} = \frac{5+19+10+17}{4} = \frac{51}{4} = 12.75$

3.4.5 SJF Estimations Techniques

It is same as SJF only the difference is future service time of a process is estimated from the past behaviors (durations).

Simple Averaging

Predicted service time = Simple averaging of past run times

$$S(n+1) = \frac{\sum_{i=1}^{n} T(i)}{n} = \frac{T(n) + \sum_{i=1}^{n} T(i)}{n} = \frac{T(n)}{n} + \frac{1}{n} \cdot \sum_{i=1}^{n-1} T(i) = \frac{T(n)}{n} + \frac{(n-1)}{n} \frac{\sum_{i=1}^{n-1} T(i)}{(n-1)}$$

$$S(n+1) = \frac{T(n)}{n} + \left(1 - \frac{1}{n}\right) \cdot S(n)$$

$$\therefore \quad S(n+1) = \frac{\sum T(i)}{n} = \frac{T(n)}{n} + \left(1 - \frac{1}{n}\right) \cdot S(n)$$

NOTE

- $S(n+1)$ is the next <CPU burst time (future prediction).
- $S(n)$ is the past predicted CPU burst time (previous).
- $T(n)$ is the past actual CPU burst time (previous).

Exponential Averaging

- $S(n+1) = \alpha \cdot T(n) + (1-\alpha)\, S(n)$, $0 < \alpha \leq 1$.
- If α is very large, it forgets past runs and depends only on last service time of the process.
- If α is very small, it remembers past runs for a long-time.

Example-3.9 Consider the following table which has last four runs of same process with corresponding CPU burst time

Run	Service Time
1	4
2	5
3	2
4	3

What will be the next predict of CPU burst time using shortest process next with simple averaging technique?

Solution:

$T(i)$ is actual service time of process at i^{th} run.

$S(k)$ is predict of CPU burst time at k^{th} run.

$T(1) = 4,\ T(2) = 5,\ T(3) = 2,\ T(4) = 3$

$$S(n+1) = \frac{\sum T(n)}{n}$$

$$S(5) = \frac{T(1)+T(2)+T(3)+T(4)}{4}$$

$$= \frac{4+5+2+3}{4} = 3.5$$

Example-3.10 A process has the following information while scheduling by shortest process next (i) Last three runs of a process are 4, 5 and 4 (last), (ii) Last predicted burst time of a process is 5 and (iii) $\alpha = 0.8$. Find the (predict) next CPU burst time using exponential averaging.

Solution:

$$S(n+1) = \alpha \cdot T(n) + (1-\alpha)\, S(n)$$

$$S(4) = 0.8 \times T(3) + (1 - 0.8)$$

$$S(3) = 5$$

$$S(4) = 0.8 \times 4 + 0.2 \times 5 = 4.2 \qquad [T(1) = 4,\ T(2) = 5,\ T(3) = 4]$$

3.4.6 Higher Response Ratio Next (HRRN) Scheduling

- Response ratio = $\frac{\text{Response time}}{\text{Service time}} = \frac{\text{Turnaround time}}{\text{Service time}} = \frac{WT+ST}{ST} = \frac{CT-AT}{ST}$
- HRRN scheduling selects a process which has highest response ratio to schedule as next process into CPU.
- HRRN scheduler is non-preemptive algorithm.
- It minimizes the average turnaround time.
- HRRN scheduler compromises between FCFS and SJF.
- It favors both longer processes and short process.
- The processes are waiting from longer will also get scheduled by HRRN scheduler.

Example-3.11 Consider the following set of process with CPU burst times. The system uses highest response ratio next scheduling to schedule the process.

Process	CPU burst Time
P_1	5
P_2	7
P_3	2
P_4	4

Compute the following if all the processes are arrived at the same time and processes are entered into the ready in order as P_1, P_2, P_3 and P_4.

(*i*) Turnaround time of each process. (*ii*) Average waiting time.

(*iii*) Average turn around time.

Solution:

Response ratio of first process

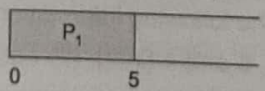
$$RR_1 = \frac{WT_1+BT_1}{BT_1} = \frac{0+5}{5} = 1$$

$$RR_2 = \frac{0+7}{7} = 1$$

$$RR_3 = \frac{0+2}{2} = 1$$

$$RR_4 = \frac{0+4}{4} = 1$$

P_1 is scheduled first, because all processes are having same response ratio.

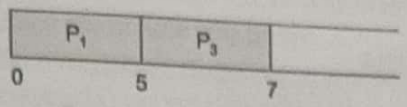


$$RR_2 = \frac{5+7}{7} = \frac{12}{7} = 1.7$$

$$RR_3 = \frac{5+2}{2} = \frac{7}{2} = 3.5$$

$$RR_4 = \frac{5+4}{4} = \frac{9}{4} = 2.25$$

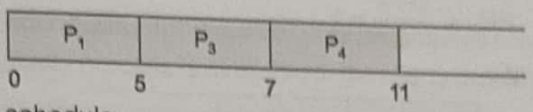
Process P_3 has highest response ration.



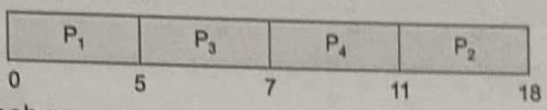
$$RR_2 = \frac{7+7}{7} = 2$$

$$RR_4 = \frac{7+4}{4} = \frac{11}{2} = 5.7$$

Process P_4 has highest response ratio



Only one process left to schedule



(*i*) Waiting time of each process

$WT_1 = 0$
$WT_2 = 11 - 0 = 11$
$WT_3 = 5$
$WT_4 = 7$

(*ii*) Turn around time

$TAT_1 = 5 - 0 = 5$
$TAT_2 = 18 - 0 = 18$
$TAT_3 = 7$
$TAT_4 = 11$

(*iii*) Waiting time of each process

$$WT_{Avg} = \frac{0+11+5+7}{4} = \frac{23}{4} = 5.75$$

(*iv*) Waiting time of each process

$$WT_{Avg} = \frac{5+18+7+11}{4} = \frac{41}{4} = 10.2$$

Example-3.12 Consider the following set of process with the arrival times and burst times

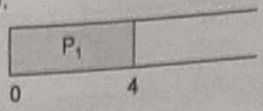
Process	Arrival time	Burst Time
P_1	0	4
P_2	2	1
P_3	5	2
P_4	4	2

Highest response ratio next scheduling is used to schedule the processes in the system with the above four process. Compute the following.

(*i*) Waiting time of each process
(*ii*) Turn around time of each process
(*iii*) Average waiting time
(*iv*) Average turnaround time

Solution:

At $t = 0$, only P_1 is in ready queue.

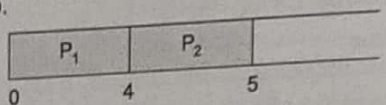


At $t = 4$, only P_2 and P_4 are in ready queue.

$$RR_2 = \frac{WT_2 + BT_2}{BT_2} = \frac{2+1}{1} = 3$$

$$RR_4 = \frac{0+2}{2} = 1$$

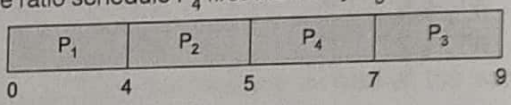
P_2 has highest response ratio.



$$RR_4 = \frac{1+2}{2} = \frac{3}{2} = 1.5$$

$$RR_3 = \frac{0+5}{5} = 1$$

P_4 has highest response ratio schedule P_4 first then only P_3 left to schedule.



(*i*) Waiting time of each process

$$WT_1 = 0$$
$$WT_2 = 4 - 2 = 2$$
$$WT_3 = 7 - 5 = 2$$
$$WT_4 = 5 - 4 = 1$$

(*ii*) Turn around time of each process

$$TAT_1 = 4 - 0 = 0$$
$$TAT_2 = 5 - 2 = 3$$
$$TAT_3 = 9 - 5 = 4$$
$$TAT_4 = 7 - 4 = 3$$

(*iii*) Average waiting time

$$WT_{Avg} = \frac{0+2+2+1}{4} = \frac{5}{4} = 1.25$$

(*iv*) Average turn around time

$$TAT_{Avg} = \frac{0+3+4+3}{4} = \frac{10}{4} = 2.5$$

3.4.7 Priority Scheduling

It selects the highest priority process to run. Each process is assigned a priority. Of all processes ready to run, the one with the highest priority gets to run next .

- Priorities may be **internal** or **external**.
- **Internal priorities** are determined by the system.
- **External priorities** are assigned by administrators.
- Priorities may also be **static** or **dynamic**.
- A process with a **static priority** keeps that priority for the entire life of the process.

- A process with a **dynamic priority** will have that priority changed by the scheduler during its course of execution to achieve its scheduling goals.
- For example, the scheduler may decide to decrease a process priority to give a chance for lower-priority job to run. If a process is I/O bound (spending most of its time waiting on I/O), the scheduler may give it a higher priority so that it can get off the run queue quickly and schedule another I/O operation.

Priority Scheduling can be of two types:

1. **Non-Preemptive priority scheduling:** The scheduler simply picks the process with the highest priority to run next.
2. **Preemptive priority scheduling:** If the system uses preemptive scheduling, a process is preempted whenever a higher priority process is available in the ready queue.

NOTE: If all processes are having equal priority then priority scheduling works similar to FCFS.

Advantage: Priority scheduling provides a good mechanism where the relative importance of each process may be precisely defined.

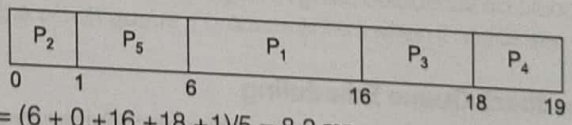
Disadvantage:

- If high priority processes use up a lot of CPU time, lower priority processes may starve and be postponed indefinitely (called **starvation**). This problem can be solved with AGING. Aging is a technique which gradually increases the priority of processes that wait in the system for a long time.
- Another problem with priority scheduling is deciding which process gets which priority level assigned to it.

Example-3.13 Consider the following 5 processes with burst times and their priorities. Find the average waiting time using non-preemptive priority scheduling. Assume that all processes are arrived at 0 time units. Assume lower priority number as highest.

Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	3
P_4	1	4
P_5	5	2

Solution:

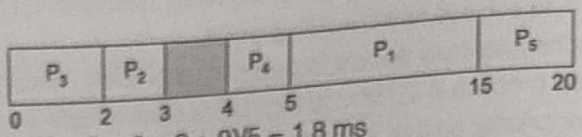


Average waiting time = (6 + 0 +16 +18 +1)/5 = 8.2 ms

Example-3.14 Consider the following 5 processes with burst times, arrival times and their priorities. Find the average waiting time using non-preemptive priority scheduling.

Process	Burst Time	Priority	Arrival Time
P_1	10	3	5
P_2	1	1	2
P_3	2	3	0
P_4	1	4	4
P_5	5	2	6

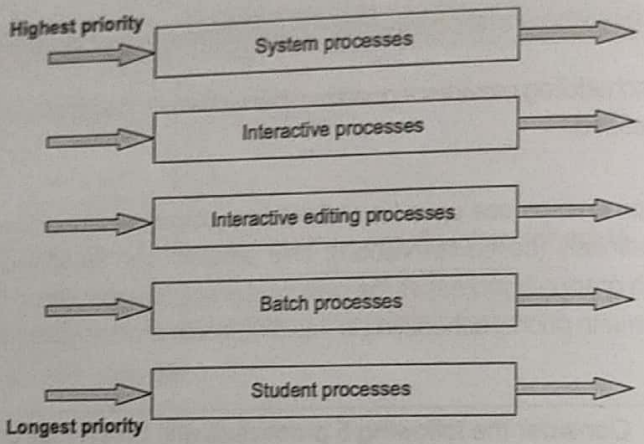
Solution:



Average waiting time = (0 + 0 + 0 + 9)/5 = 1.8 ms

3.4.8 Multi Level Queue Scheduling

Instead of a pure priority scheduler that expects processes to have unique priorities, we can set up priority classes. Each class has its own unique priority and within each class we have a queue of processes that are associated with that priority class. Each priority class can have its own scheduling algorithm to determine how to pick processes from that class.



Make a distinction between foreground (interactive) and background (batch) processes.

Different response time requirements for the two types of processes and hence, different scheduling needs. Separate queue for different types of processes, with the process priority being defined by the queue separate processes with different CPU burst requirements:

- Too much CPU time (lower priority)
- I/O-bound and interactive process (higher priority)
- *Aging* can be used to prevent starvation.
- Each queue can have its own scheduling algorithm like FCFS or RR. For example, Queue for interactive processes could be scheduled using RR algorithm where queue for batch processes may use FCFS algorithm.

3.4.9 Multi Level Feedback Queue Scheduling

It allows a process to move between queues. It uses many ready queues and associates a different priority with each queue. High priority queues often have a short time slice associated with them.

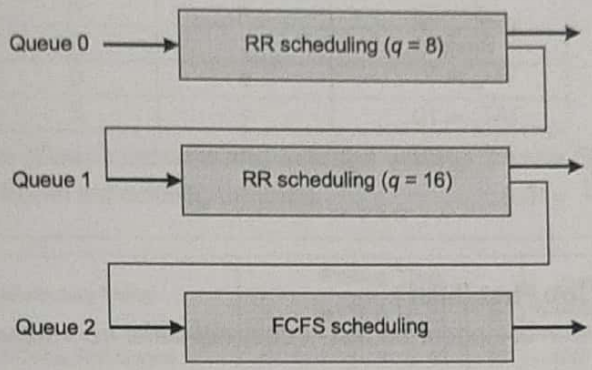
If a process reaches the end of its time slice rather than blocking, the scheduler demotes it to the next lower-priority queue. Lower priority queues are then assigned increasingly longer time slices. The process will get knocked to a lower priority level each time it runs for the full quantum.

The goal is to keep interactive processes with short CPU bursts at high priorities and lower the priorities of CPU-bound processes while giving them longer stretches of time to run (but less often). A drawback is that if an interactive process has a stretch of computation, it will get knocked to a low level and never get pushed up. Some systems deal with this by raising a process priority whenever it blocks on I/O. Another approach is to raise the priority of a process that has not had a chance to run for a while. This is called **process aging**.

Consider a scheduler with n queues Q_0, Q_1, Q_2 ... Q_n. Queue Q_i executes scheduling algorithm A_i with time quantum q_i to the currently scheduled process.

$q_0 < q_1 < q_2 < \ldots q_n$. Q_1 have highest priority processes than Q_2. Similarly Q_2 have highest than Q_3 and so on.

Example:



- It chooses to process with highest priority from the occupied queue and run that process either preemptively or non-preemptively. If the process uses too much CPU time it will moved to a lower-priority queue.
- If a process that wait too long in the lower-priority queue may be moved to a higher-priority queue may be moved to a highest-priority queue (prevents starvation).
- It gives priority to interactive threads (those with short CPU bursts)
- It never chooses a thread in queue i if there are threads in any queue $j < i$. Threads in queue i use quantum q_i, and $q_i < q_j$ if $i < j$.
- Newly ready threads go in to queue 0.
- Level i thread that is preempted goes into the level $i + 1$ ready queue.

A multi-level feedback queue scheduler is defined by the following parameters:

(a) Number of queues

(b) Scheduling algorithm for each queue

(c) Method used to determine when to upgrade a process to higher priority queue

(d) Method used to determine when to demote a process

(e) Method used to determine which queue a process enters when it needs service

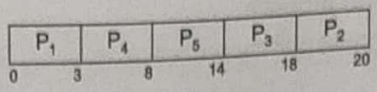
3.4.10 Longest Job First

- This scheduling is not preferable because throughput decreases.
- It is non-preemptive.
- Job with longest burst time is selected.
- If burst time priority are matching then schedule the process that has lowest overall time.

Example-3.15 Find average turnaround time for the following table with five processes.

Process	1	2	3	4	5
BT	3	2	4	5	6
AT	0	1	2	3	4

Solution:



$CT_1 = 3$ $TAT_1 = 3$

$CT_2 = 20$ $TAT_2 = 19$

$CT_3 = 18$ $TAT_3 = 16$

$CT_4 = 8$ $TAT_4 = 5$

$CT_5 = 14$ $TAT_5 = 10$

$$\text{Average TAT} = \frac{\sum TAT_i}{5} = 10.6$$

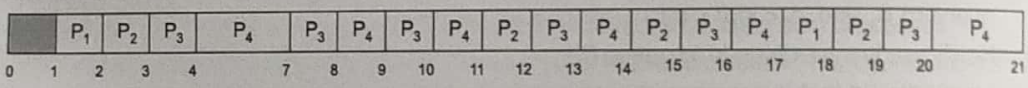
3.4.11 Longest Remaining Job First (LRJF)

It is the preemptive version of longest job first. It does not suffer from starvation.

Example-3.16 Find average turnaround time for the following table with four processes.

Process Number	AT	BT
1	1	2
2	2	4
3	3	6
4	4	8

Solution:



$TAT_1 = 18 - 1 = 17$, $TAT_2 = 19 - 2 = 17$, $TAT_3 = 20 - 3 = 17$, $TAT_4 = 21 - 4 = 17$

Average TAT = (17+17+17+17)/4 = 17

Comparison Table of Scheduling Algorithms

	FCFS	Round robin	SPN	SRTF	HRRN	Feedback	LRJF
Selection function	Max [w]	Constant	min [s]	min [s – e]	$\max\left(\frac{w+s}{s}\right)$	–	–
Decision mode	Non-preemptive	Preemptive (at time quantum)	Non-preemptive	Preemptive (at arrival)	Non-preemptive	Preemptive (at time quantum)	Preemptive
Throughput	Not emphasized	May be low if quantum is too small	High	High	High	Not emphasized	Low
Response time	May be high, espacially if there is a large variance in process execution times	Provides good response time for short processes	Provides good response time for short processes	Provides good response time	Provides good response time	Not emphasized	Good
Overhead	Minimum	Minimum	Can be high	Can be high	Can be high	Can be high	High
Effect on processes	Penalizes short processes I/O bound processes	Fair treatment	Penalizes long processes	Penalizes long processes	Good balance	May favor I/O bound processes	Favour high CPU bound process
Starvation	No	No	Possible	Possible	No	Possible	No

Example-3.17 Consider the following table with five processes *A*, *B*, *C*, *D* and *E* arrive in this order at the same time with the given CPU burst and priority values. Ignore context switching overhead.

	CPU Burst	Priority
A	7	3
B	2	5(Lowest)
C	3	1(Highest)
D	6	4
E	4	2

Find the *waiting time of each process* and *average waiting time* for FCFS, Nonpreemptive SJF, Priority, and Round Robin scheduling policies.

Solution:

Scheduling Policy	Waiting Time					Average Waiting Time
	A	B	C	D	E	
First-Come-First-Served	0	7	9	12	18	46/5 = 9.2
Non-Pre-emptive Shortest-Job First	15	0	2	9	5	31/5 = 6.2
Priority	7	20	0	14	3	44/5 = 8.8
Round-Robin (time quantum=2)	15	2	10	15	13	55/5 = 11

Example-3.18 Can any of the three scheduling schemes (FCFS, SRTF, or RR) result in starvation?

Solution:

The SRTF algorithm can result starvation of long tasks if short tasks keep arriving.

To fix SRTF, we might add some priority mechanism to make sure that threads that have waited for a long time without running get at least some of the CPU now and then (priority increments when threads don't get to run). The multi-level scheduler is a good approximation to SRTF that can be designed to avoid starvation. (the result wouldn't be quite SRTF any more).

FCFS and RR algorithms do not suffers from starvation. (FCFS algorithm will result in starvation only if some thread runs forever. RR does not result in starvation, since all threads get some of the CPU. To fix FCFS, we would need to find some way of preempting threads that have run forever, possibly by giving some CPU to other threads.

Example-3.19 Round robin scheduling requires a time quantum (q). What is the effect if q is too large?

Solution:

If too large scheduling is same as first come first served scheduling algorithm.

Example-3.20 What is the effect (i.e. what bad thing happens) if q is too small?

Solution:

If too small too much time (overhead) is spent switching processes.

Example-3.21 Consider the following processes which are submitted to an operating system with priority and burst times. Assume a time quantum of 2, where applicable. A lower number indicates a greater priority.

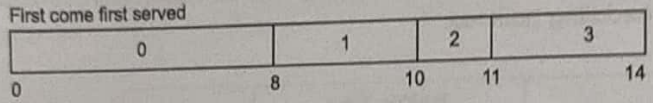
Process	Arrival time	Burst time	Priority
0	0	8	5
1	2	2	4
2	4	1	6
3	6	3	3

Draw the Gantt chart for the following scheduling algorithms. (1) FCFS, (2) SJF, (3) SRTF, (4) Priority (Non-preemptive), (5) Priority (pre-emptive), and (6) Round-Robin.

Solution:

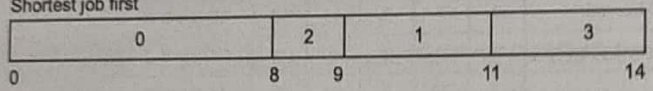
(1) FCFS: First come first served. Processes dispatched in order of arrival

First come first served



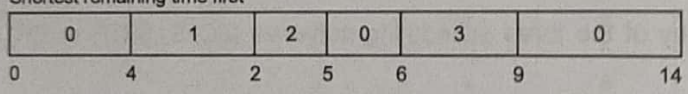
(2) SJF: Shortest job first. Process with shortest burst time runs next. Non-preemptive

Shortest job first



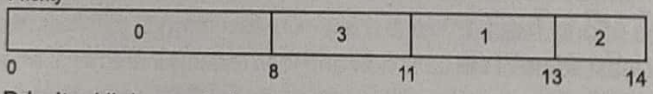
(3) SRTF: Shortest remaining time first. Process with Shortest burst time runs, pre-emptive.

Shortest remaining time first



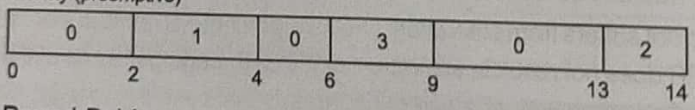
(4) Priority: Highest priority runs first Non-preemptive

Priority



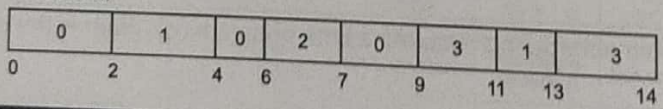
(5) Priority: Highest priority runs first Preemptive

Priority (preemptive)



(6) Round-Robin

Round robin

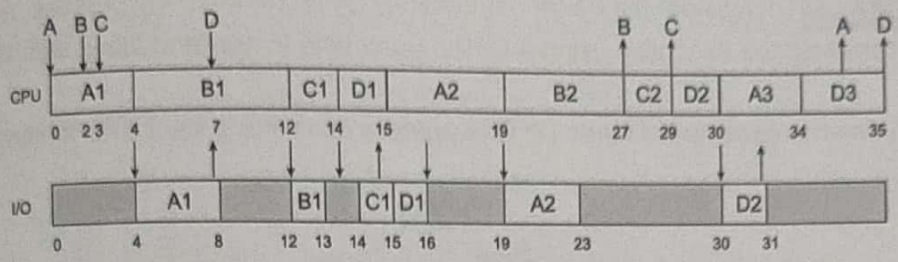


Example-3.22 Consider the following information of each process with arrival times and their execution times of CPU and I/O. FCFS is used to schedule CPU and I/O jobs.

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4	4	4	4	4
B	2	8	1	8	–	–
C	3	2	1	2	–	–
D	7	1	1	1	1	1

Compute the Processor utilization, Throughput, Average waiting time and Average turn around time.

Solution:



Processor utilization = (35/35) * 100 = 100%

Throughput = 4/35 = 0.11

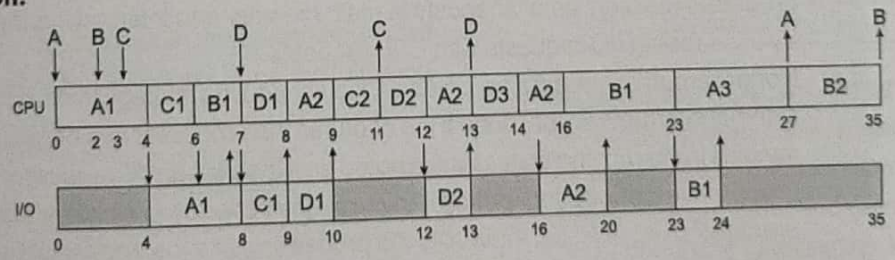
Process	TAT	WT
A	34 – 0 = 34	34 – 12 = 22
B	27 – 2 = 25	25 – 16 = 9
C	29 – 3 = 26	26 – 4 = 22
D	35 – 7 = 28	28 – 3 = 25
	Avg TAT = 28.25	Avg WT = 19.5

***Example*-3.23** Consider the following information of each process with arrival times and their execution times of CPU and I/O. SJF is used to schedule CPU jobs and FCFS is used to schedule I/O jobs.

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4	4	4	4	4
B	2	8	1	8	–	–
C	3	2	1	2	–	–
D	7	1	1	1	1	1

Compute the Processor utilization, Throughput, Average waiting time and Average turn around time.

Solution:



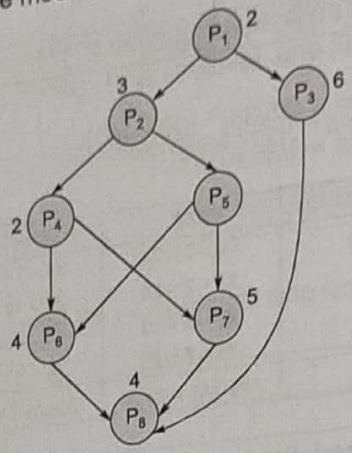
Processor utilization = (35/35)*100 = 100%

Throughput = 4/35 = 0.11

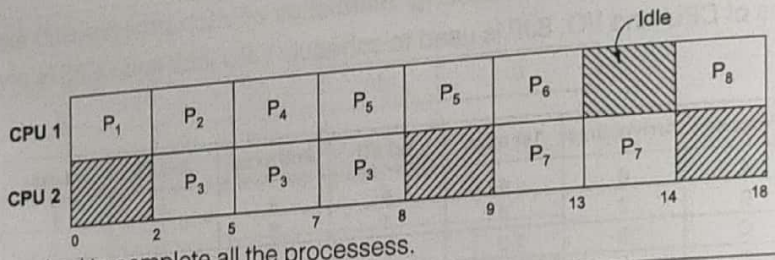
Process	TAT	WT
A	35 – 0 = 35	35 – 12 = 23
B	31 – 2 = 29	29 – 16 = 13
C	17 – 3 = 14	14 – 4 = 10
D	23 – 7 = 16	16 – 3 = 13
	Avg TAT = 23.5	Avg WT = 14.75

Example - 3.24 Consider the below dependency graph between the process. At what time all the process complete the execution using 2 CPUs. Burst time is provided along with the process node.

Note: (1) Use non preemptive mode (2) One process can't share the 2 CPU at same time.



Solution:



18 units is required to complete all the processess.

Summary

- **Multiprogramming:** A technique that allows a single processor to process several programs residing simultaneously in main memory and interleaving their execution by overlapping I/O requests with CPU requests.
- **Nonpreemptive scheduling policy:** A job scheduling strategy that functions without external interrupts so that once a job captures the processor and begins execution, it remains in the running state uninterrupted until it issues an I/O request or it's finished.
- **Preemptive scheduling policy:** Any process scheduling strategy that, based on predetermined policies, interrupts the processing of a job and transfers the CPU to another job. It is widely used in time-sharing environments.
- **First-come, first-served (FCFS):** A nonpreemptive process scheduling policy (or algorithm) that handles jobs according to their arrival time.
- **Round robin:** A preemptive process scheduling policy (or algorithm) that allocates to each job one unit of processing time per turn to ensure that the CPU is equally shared among all active processes and isn't monopolized by any one job.
- **Time quantum:** A period of time assigned to a process for execution before it is preempted.

- **Shortest job next (SJN):** A nonpreemptive process scheduling policy (or algorithm) that selects the waiting job with the shortest CPU cycle time.
- **Shortest remaining time first (SRTF):** A preemptive process scheduling policy (or algorithm) similar to the SJN algorithm that allocates the processor to the job closest to completion.
- **Priority scheduling:** A nonpreemptive process scheduling policy (or algorithm) that allows for the execution of high-priority jobs before low-priority jobs.
- **Multiple-level queues:** A process scheduling scheme (used with other scheduling algorithms) that groups jobs according to a common characteristic.
- **Aging:** A policy used to ensure that jobs that have been in the system for a long time in the lower-level queues will eventually complete their execution.
- **Turnaround time:** A measure of a system's efficiency that tracks the time required to execute a job and return output to the user.
- **Response time:** A measure of the efficiency of an interactive system that tracks the speed with which the system will respond to a user's command.

Student's Assignment

Q.1 Which of the following information is sufficient to compute the turn around time of a process.
(a) Process arrival time and process successful termination time
(b) Process waiting time and process service time
(c) Both (a) and (b)
(d) Neither (a) nor (b)

Q.2 Consider a system has 4 processes with arrival times and burst times as shown in the following table.

Process	P_1	P_2	P_3	P_4
Arrival time	0	1	3	9
Burst time	3	5	2	4

Compute the average waiting time of processes using the FCFS scheduling?
(a) 0 (b) 2
(c) 5.5 (d) 4

Q.3 Consider the following table

Process	P_1	P_2	P_3	P_4
Service time	5	2	4	6

Assume that all the processes are arrived at time 0. Compute the average waiting time of processes in the system using shortest job first scheduling.
(a) 4.75 (b) 5.25
(c) 7.65 (d) 9

Q.4 Consider the following table which has 4 processes.

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

All processes are arrived at time '0'. CPU scheduling uses SJF algorithm and I/O scheduling uses FCFS algorithm. Every process gets CPU service first and then it requests I/O. CPU scheduling favours the less I/O service process whenever two or more processes have same CPU service time.

What is the total time in which I/O device will not service any request from time '0 to the completion of last job? (Idle time of I/O)
(a) 0 (b) 1
(c) 2 (d) 3

Q.5 Consider the following table:

Process	P_1	P_2	P_3	P_4
Arrival time	0	1	2	3
Service time	6	3	1	4

Compute the average turn around time when processes are scheduled with SRTF.

(a) 4.15 (b) 5.20
(c) 6.25 (d) 7.30

Q.6 Consider the following table which contain CPU burst times for last four runs of the same process.

Run	P_1	P_2	P_3	P_4
Service time	4	5	2	3

What will be the next predict of CPU burst time using shortest process next will simple averaging technique?

(a) 3 (b) 3.5
(c) 4 (d) 4.5

Q.7 Consider the following table:

Process	P_1	P_2	P_3	P_4
Service time	5	7	3	6
Arrival time	0	1	3	4

Compute the average waiting time of processes using Round-Robin algorithm with time slice of 5.

(a) 10 (b) 11.25
(c) 12.75 (d) None of these

Q.8 Assume that a process has CPU burst times for last three runs as 4, 5 and 4 (last). Given that last predict burst time was 5 and $\alpha = 0.8$. What is the next predict of CPU burst time for the process when CPU scheduler is shortest process next with exponential averaging?

(a) 4.2 (b) 3.4
(c) 4 (d) 3

Q.9 Let WT_i be the waiting time of process 'i', TAT_i be the turn around time of process i, BT_i be the burst time of process i. Then find which of the following is correct.

(a) $WT_i = TAT_i - BT_i$ (b) $TAT_i = WT_i - BT_i$
(c) $WT_i = BT_i - TAT_i$ (d) None of these

Q.10 RR schedular maintains larger quantum then choose the wrong statement from the following.

(a) Larger quantum gives more efficient than smaller quantum used by RR scheduler.
(b) Number of context switches can be less.
(c) It may suffer interactive response time for processes.
(d) None of these

Q.11 Which of the following may suffer from starvation

(a) FCFS scheduling
(b) SJF scheduling
(c) RR scheduling
(d) All of the these

Q.12 Assume time quantum of RR scheduler is greater than longest CPU burst time of all processes. Which of the following is correct?

(a) RR scheduler is better than FCFS
(b) FCFS scheduler is better than RR
(c) RR scheduler is same as FCFS
(d) No comparison between RR and FCFS

Q.13 Consider the following set of processes with the arrival time and CPU burst time.

Process	P_1	P_2	P_3	P_4
Arrival time	0	2	4	5
CPU burst time	5	6	3	1

Assume that above processes are scheduled by RR scheduler with time quantum of 2 units. What is the average turn around time of all processes?

(a) 10 (b) 11.5
(c) 12.3 (d) 13.1

Q.14 Round Robin scheduler implemented a version with feedback queues. There are two Ready queues in the system. All new processes are placed in the higher priority Ready Queue and interrupted processes are placed in the lower priority ready queue. If there are any processes in the higher priority ready queue then first scheduled them by RR before scheduling lower priority ready queue processes.

Consider the following processes with the arrival times and burst times.

Process	P_1	P_2	P_3	P_4	P_5
Arrival time	0	2	3	5	6
CPU burst time	3	5	2	4	3

What is the average turn around time of all processes in the given system with time quantum of 2 units for the both ready queues?

(a) 10 (b) 8.2
(c) 7.3 (d) 9.8

Q.15 Consider the following set of processes with CPU burst times. The system uses highest response ratio next scheduling to schedule the processes.

Process	P_1	P_2	P_3	P_4
CPU burst time	5	7	2	4

What is the average turnaround time of processes when all processes are arrived at the same time? Assume the processes are entered into the order queue (in order) P_1, P_2, P_3 and P_4.

(a) 9.7 (b) 10.2
(c) 11.3 (d) 12.4

Q.16 Consider the following set of processes with the arrival times and burst times. Processes are scheduled using highest response ratio next.

Process	P_1	P_2	P_3	P_4
Arrival time	0	2	5	4
CPU burst time	4	1	2	2

What is the average waiting time of processes?

(a) 1 (b) 1.25
(c) 1.50 (d) 1.75

Q.17 Consider the following processes with burst times, priority and arrival times.

Process	P_1	P_2	P_3	P_4
Burst time	5	2	6	4
Priority	4 (low)	3	1(high)	2
Arrival time	0	2	2	6

Using FCFS scheduling compute the average waiting time of processes. (FCFS selects highest priority process when more than one process arrived at the same time).

(a) 6.4 (b) 3.25
(c) 5.15 (d) 4.75

Q.18 Consider the following processes with burst times, priority and arrival times.

Process	P_1	P_2	P_3	P_4	P_5
Arrival time	0	1	2	3	5
Burst time	4	3	3	3	2
Priority	5(lowest)	4	2	1(highest)	3

SJF scheduling selects always shortest job but when there is ambiguous it chooses highest priority process first. What is the average turn around time of the processes using SJF?

(a) 6.7 (b) 6.3
(c) 7.2 (d) 7.9

Q.19 Consider the following set of processes.

Process	P_1	P_2	P_3	P_4
Burst time	4	5	3	2
Priority	3	1 (high)	4 (low)	2
Arrival time	2	0	3	1

If shortest remaining time first scheduler uses priority to schedule the processes, what is the average waiting time of processes?

(a) 3.05 (b) 3.25
(c) 4.15 (d) 4.75

Q.20 Consider the following work load.

Process	P_1	P_2	P_3	P_4	P_5
Arrival time	0	3	1	2	2
Burst time	2	4	3	2	4
Priority	1(high)	4	5 (lowest)	2	3

Processes are scheduled with priority scheduler. Assume that priority scheduler is non-preemptive. What is the turn around time with the priority scheduler?

(a) 7.3 (b) 6.1
(c) 6.6 (d) 7.2

Q.21 Which of the following condition is used for every process by "Highest Response Ratio Next" scheduling to select the process with highest response ratio?

(a) Response ratio = $\frac{\text{Wait time} + \text{Service time}}{\text{Service time}}$

(b) Response ratio = $\frac{\text{Wait time} + \text{Service time}}{\text{Wait time}}$

(c) Response ratio = $\frac{\text{Service time}}{\text{Wait time} + \text{Service time}}$

(d) Response ratio = $\frac{\text{Wait time}}{\text{Wait time} + \text{Service time}}$

Q.22 Consider the following processes with priority

Process	P_1	P_2	P_3	P_4
Burst time	5	4	3	2
Priority	2	1(highest)	3	4
Arrival time	0	2	3	4

Processes are scheduled using priority scheduler with preemption of a process whenever a new process arrived with high priority. A process can continue the execution if arrived process in ready queue has lower priority than the executing process. What is the average waiting time of process?

(a) 5 (b) 6.2
(c) 4.3 (d) None of these

Q.23 There are six processes waiting in ready queue with the burst time are 9, 7, 4, 2, 1 and x. In what order should they run to minimize the average waiting time when the value of x is either 5 or 6.

(a) 1, 2, 4, x, 7, 9
(b) 9, 7, x, 4, 2, 1
(c) x, 7, 9, 4, 2, 1
(d) Order can not be decided

Q.24 A process burst time can be using exponential averaging technique of shortest process next scheduling.

Assume $E(t)$ is the estimation burst time of a process at time t, burst (t) is the actual burst time of a process at time t and $0 \le \alpha \le 1$. What is the formula to predict the burst time of t?

(a) $E(t) = B(t) \cdot \alpha + (1-\alpha) \cdot B(t-1)$
(b) $E(t) = \alpha \cdot E(t-1) + (1-\alpha) \cdot E(t-2)$
(c) $E(t) = \alpha \cdot B(t-1) + (1-\alpha) \cdot B(t-2)$
(d) $E(t) = \alpha \cdot B(t-1) + (1-\alpha) \cdot E(t-1)$

Q.25 Match the following groups:

Group-I

A. FCFS
B. Round Robin
C. SRTF
D. Priority scheduler

Group-II

1. Important processes get execute first.
2. Minimizes the average waiting time.
3. The processes run in the order they arrived.
4. Every process get a chance to execute.

Codes:

	A	B	C	D
(a)	1	2	3	4
(b)	4	3	2	1
(c)	3	4	2	1
(d)	2	1	3	4

Answer Key:

1. (c)	**2.** (b)	**3.** (a)	**4.** (d)	**5.** (c)
6. (b)	**7.** (d)	**8.** (a)	**9.** (a)	**10.** (d)
11. (b)	**12.** (c)	**13.** (a)	**14.** (d)	**15.** (b)
16. (b)	**17.** (d)	**18.** (c)	**19.** (b)	**20.** (c)
21. (a)	**22.** (a)	**23.** (a)	**24.** (d)	**25.** (c)

CHAPTER 04

Process Synchronization

4.1 Synchronization

Threads (or processes) are **concurrent** if they exist at the same time. They are **asynchronous** if they need to synchronize with each other occasionally. They are **independent** if they have no dependence on each other: it does not affect a thread whether another thread exists or not. Threads are **synchronous** if they synchronize frequently so that their relative order of execution is guaranteed.

A **race condition** is a bug where the outcome of concurrent threads is dependent on the precise sequence of execution (particularly, the interleaving of executing two threads). A race condition typically occurs when two or more threads try to read, write, and possibly make decisions based on the contents of memory that they are accessing concurrently. The regions of a program that try to access shared resources and cause race conditions are called **critical sections**.

To avoid race conditions, we want to make sure that only one thread at a time can execute within a critical section. Enforcing this is called **mutual exclusion**.

4.2 The Critical-Section Problem

1. The critical-section problem is to design a protocol that the processes can cooperate. The protocol must ensure that when one process is executing in its critical section, no other process is allowed to execute in its critical section.
2. The critical section problem is to design a protocol that the processes can use so that their action will not depend on the order in which their execution is interleaved (possibly on many processors).

Template for each process that involves critical section:

```
do
{
      /* Entry section; */
      critical_section();
      remainder_section();/*Exit section */
} while (1);
```

4.3 Requirements

Assumptions:

- No assumption about the hardware instructions
- No assumption about the number of processors supported
- Basic machine language instructions executed atomically

Design of a protocol to be used by the processes to cooperate with following constraints:

- Mutual exclusion
- Progress, and
- Bounded wait

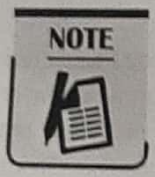
4.3.1 Mutual Exclusion

If a process is executing in its critical section, then no other processes can be executing in their critical sections. (Safety)

4.3.2 Progress

Progress conditions holds the following:

- If no process is executing in its critical section, the selection of a process that will be allowed to enter its critical section cannot be postponed indefinitely.
- A process executing outside of its critical section cannot prevent other processes from entering theirs critical section; processes attempting to enter their critical sections simultaneously must decide which process enters eventually.



- Progress condition also holds following implicitly.
- If only one process wants to enter, it should be able to.
- If two or more want to enter, one of them should succeed.
- A Process that is seeking to enter the critical section will eventually succeed.

4.3.3 Bounded Waiting

There must exist a bound on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted

- Assume that each process executes at a nonzero speed
- No assumption concerning relative speed of the *n* processes.

Critical section requirements

- Mutual exclusion must be enforced: Only one process at a time is allowed into its critical section, among all processes that have critical sections for the same resource or shared object.
- A process that halts in its non critical section must do so without interfering with other processes.
- It must not be possible for a process requiring access to a critical section to be delayed indefinitely; no deadlock or starvation.
- When no process is in a critical section, any process that requests entry to its critical section must be permitted to enter without delay.
- No assumptions are made about relative process speeds or number of processors.
- A process remains inside its critical section for a finite time only.

4.4 Synchronization Techniques

4.4.1 Software Solutions

Algorithms whose correctness does not rely on any other assumptions.

1. Algorithm1-Strict alternation: turn variable (Dekkers Algorithm)
2. Algorithm2-Use of flag variable
3. Algorithm3-Peterson's approach: (Combine Algorithm1 and Algorithm2)
4. Multiple Process Solution
5. Bakery's Algorithm

Drawback of Software Solutions

1. Complicated to program.
2. Busy waiting is possible.
3. If critical sections are long, it would be more efficient to block processes that are waiting.
4. Makes difficult assumptions about the memory system.

4.4.2 Hardware Solutions

Most processors have specific instructions that aid in programming critical sections. The key to these instructions is that the operations that they perform are ***atomic***, meaning that they are *indivisible*. The following mechanisms depends on machine instructions

1. Disabling Interrupts
2. Test and Set (Test and Set Lock) instruction
3. Swap instruction

Drawback of Hardware Solutions

Busy waiting, Deadlock and starvations may possible

4.4.3 Semaphore (OS) Solutions

Provide some functions and data structures to the programmer through system/library calls.

1. Binary Semaphore
2. Counting Semaphore

Drawback of Semaphores

1. Simple algorithms require more than one semaphore. This increases the complexity of semaphore solutions to such algorithms.
2. Semaphores are too low level. So it is easy to make programming mistakes.
3. The programmer must keep track of all calls to wait and to signal the semaphore. If this is not done in the correct order, programmer error can cause deadlock.
4. **Programming Language Solutions:** Linguistic constructs provided as part of a language (Condition variable/Monitor).

4.5 Two Process Solution (Software Solution)

4.5.1 Turn variable (Algorithm 1)

This approach assumes that only two processes are trying to synchronize, and they both know their own process ID (me) and the other's process ID (other).

```
void lock()
{
        while(turn != me);
}
```

```
void unlock()
{
        turn = other;
}
```

Implementation:

```
extern int turn;   /* Shared variable between both
processes */
void process ( const int me ) /* me can be 0 or 1 */
{
      int other = 1 - me;
      do
      {
            while ( turn != me ) ; /* do nothing */
            critical_section();
            turn = other;
            remainder_section();
      } while ( 1 );
}
```

```
extern int turn; /* Shared variable */
void process ( const int i ) /* i = 0 or1*/
{
      int j =1- i;
      do
      {
      ...
      while ( turn != i );
      critical_section();
      turn = j;
      ...
      } while (1);
}
```

1. Guarantees mutual exclusion.
2. Does not guarantee progress — enforces strict alternation of processes entering CS's.
3. Bounded waiting violated — suppose one process terminates while its turn?

Here we have a global variable turn that tracks which process can enter its critical section next. When a process exits its critical section, it marks turn to say that the next process to enter its critical section will be other_pid.

This has the mutual exclusion property, so it's basically valid. But it's problematic for another reason: Let's say *A* hits its critical section quite often (it's a very fast process), but *B* does not. This will force *A* to wait for *B* every time it hits its critical section, effectively slowing *A* to *B*'s pace. This is a large performance hit that isn't really acceptable.

4.5.2 Use of flag Variable (Algorithm 2)

1. Satisfies the mutual exclusion requirement
2. Does not satisfy the progress requirement
   (*i*) Time T0 p0 sets flag[0] to true
   (*ii*) Time T1 p1 sets flag[1] to true
   Processes p0 and p1 loop forever in their respective while statements
3. Critically dependent on the exact timing of two processes
4. Switch the order of instructions in entry section: No mutual exclusion

```
extern int flag[2];      /* Shared variable; one for each process */
void process ( const int me)  /* me can be 0 or 1 */
{
     int other = 1 - me;
     do
     {
          flag[me] = 1;      /* true */
          while ( flag[other] );
          critical_section();
          flag[me] = 0;      /* false */
          remainder_section();
     } while ( 1 );
}
```

4.5.3 Peterson's Approach (Algorithm 3)

Peterson's solution is based on two processes, P0 and P1, which alternate between their critical sections and remainder sections.

"me" process is P_i

"other" process is P_j (i.e. $j = 1 - i$)

```
void lock()
{
     interested[me] = true;
     turn = other;
     while(turn != me && interested[other]);
}
```

```
void unlock()
{
     interested[me] = false;
}
```

Does this provide the mutual exclusion property?

Let's say A and B have both executed the first instruction of lock ( ). Whichever assigns turn second will be forced to wait: turn will hold the process ID of the first process to assign turn, and that process will be able to continue past the while loop into the critical section. The second process will not be able to continue until that process finally marks itself as being no longer interested.

Peterson's approach is quite solid, but it suffers from two major shortcomings: First, it busy-waits, wasting CPU time in an idle while loop when there's work to be done in other processes. Second, it only applies to two processes. You might be able to work around the second of these problems, but the first one is pretty much endemic to the solution.

Implementation:

Peterson's solution requires two shared data items: turn and flag.

int turn : If turn = = i, then process i is allowed into their critical section.

boolean flag [2]: When process i wants to enter the critical section, it sets flag[i] to true.

```
extern int flag[2];      /* Shared variables */
extern int turn;         /* Shared variable */
void process (const int me)   /* me can be 0 or 1 */
{
      int other = 1 - me;
      do
      {
            /* Entry section */
            flag[me] = true;  /* Raise my flag */
            turn = other;     /* set turn to other process*/
            while (flag[other] && turn == other) ;
            critical_section();
            /* Exit section */
            flag[me] = false;
            remainder_section();
      } while (1);
}
```

This implementation *ensures mutual exclusion and avoids starvation, but works only for two processes.*

4.6 Multiple Process Solution (Software Solution)

- The array flag can take one of the three values (idle, want_in, in_cs)

```
enum state { idle, want_in, in_cs };
extern int turn; extern state flag[n]; // Flag corresponding to each process
process (const int i)
{
     int j;    // Local to each process
     do {
          do {
               flag[i] = want_in; // Raise my flag
               j = turn;   // Set local variable
               while (j !=i) j = (flag[j] != idle) ? turn: (j+1) % n;
               // Declare intention to enter critical section
               flag[i] = in_cs;
               // Check that no one else is in critical section
               for (j = 0; j < n; j++)
               if ((j != i) && (flag[j] == in_cs))
                  break;
          } while (j < n) || (turn != i && flag[turn] != idle);
          // Assign turn to self and enter critical section
          turn = i;
          critical_section();
          // Exit section
          j = (turn + 1) % n;
          while (flag[j] == idle)
          j = (j + 1) % n;
          //Assign turn to next waiting process; change own flag to idle
          turn = j;
          flag[i] = idle;
          remainder_section();
     } while (1);
}
```

- p_i enters the critical section only if flag[j] '" in_cs for all j '" i.
- turn can be modified only upon entry to and exit from the critical section. The first contending process enters its critical section.
- Upon exit, the successor process is designated to be the one following the current process.
- **Mutual Exclusion**
  (*i*) p_i enters the critical section only if flag[j] '" in_cs for all j '" i.
  (*ii*) Only p_i can set flag[i] = in_cs.
  (*iii*) p_i inspects flag[j] only while flag[i] = in_cs.
- **Progress**
  (*i*) turn can be modified only upon entry to and exit from the critical section.
  (*ii*) No process is executing or leaving its critical section => turn remains constant.
  (*ii*) First contending process in the cyclic ordering (turn, turn + 1, ..., $n-1$, 0, ..., turn-1) enters its critical section.
- **Bounded Wait**
  (*i*) Upon exit from the critical section, a process must designate its unique successor the first contending process in the cyclic ordering turn + 1, ..., $n-1$, 0, ..., turn-1, turn.
  (*ii*) Any process waiting to enter its critical section will do so in at most $n-1$ turns.

4.7 Bakery's Algorithm (Software Solution)

- Each process has a unique id
- Process id is assigned in a completely ordered manner

```
extern bool choosing[n]; /* Shared Boolean array */
extern int number[n]; /* Shared integer array to hold turn number */
void process_i (const int i) /* i^th Process */
{
    do
    {
        choosing[i] = true;
        number[i] = 1 + max(number[0],... , number[n-1]);
        choosing[i] = false;
        for (int j = 0; j < n; j++)
        {
            while (choosing[j]);/*Wait while someone else is choosing */
            while ((number[j]) && (number[j],j) < (number[i],i));
        }
        critical_section();
        number[i] = 0;
        remainder_section();
    }while (1);
}
```

- If p_i is in its critical section and p_k ($k = i$) has already chosen its number[k] = 0, then (number[i], i) < (number[k], k).

Advantages of Bakery's Algorithm

1. **Correctness:** Only one process is in the critical section at a time. (Mutual Exclusion)
2. **Progress:** There is no deadlock, and some process will eventually get into the critical section.

3. **Fairness:** There is no starvation, and no process will wait indefinitely while other processes continuously sneak into the critical section. (Bounded waiting)
4. **Generality:** It works for N processes

4.8 Interrupt Disabling

- A process runs until it invokes an operating-system service or until it is interrupted
- Disabling interrupts guarantees mutual exclusion
- Processor is limited in its ability to interleave programs
- Multiprocessing: disabling interrupts on one processor will not guarantee mutual exclusion
- Brute-force approach
- Not proper to give users the power to disable interrupts
  (*i*) User may not enable interrupts after being done
  (*ii*) Multiple CPU configuration
- In current systems, interrupts must be disabled inside some critical kernel regions: Critical regions must be limited because kernel and interrupt handlers should be able to run most of the time to take care of any event.

Disadvantages of Interrupt Disabling

- It works only in a single processor environment.
- Interrupts can be lost if not serviced promptly.
- A process waiting to enter its critical section could suffer from starvation.

4.9 Special Machine Instructions

- Normally, access to a memory location excludes other access to that same location.
- **Extension:** designers have proposed machines instructions that perform 2 actions atomically (indivisible) on the same memory location (e.g., reading and writing).
- The execution of such an instruction is also mutually exclusive (even on Multiprocessors).
- They can be used to provide mutual exclusion but need to be complemented by other mechanisms to satisfy the other 2 requirements of the CS problem.
- The following techniques are used for hardware solution to the critical solution problem.
  (a) Test And Set instruction (TAS)
  (b) Swap Instruction
  (c) Compare and Swap Instruction
  (d) Fetch and Increment Instruction

4.9.1 Test and Set Instruction

Set a memory location to 1 and return the previous value of the location. If the returned value is a 1 that means lock acquired by someone else. If the returned value is 0 then lock is free and it will set to 1. Test and modify the content of a word atomically.

```
int test_and_set (int& target )
{
      int tmp;
      tmp = target;
      target = 1; /* True */
      return ( tmp );
}
```

Implementing Mutual Exclusion with test and set:

```
extern bool lock (false);
do
{
      while (test_and_set (lock));/* Wait for lock */
      critical_section();
      lock = false;      /* Release the lock */
      remainder_section();
}while (1);
```

Advantages of test and set:

1. It is simple and easy to verify.
2. It is applicable to any number of processes.
3. It can be used to support multiple critical section.

Disadvantages of test and set:

1. Busy waiting is possible.
2. Starvation is also possible.

Solution	Mutual Exclusion	Progress	Bounded waiting
1. Lock variable	✖	✓	✖
2. Strict alteration	✓	✖	✓
3. Peterson solution	✓	✓	✓
4. TSL instruciton	✓	✓	✖

4.9.2 Swap Instruction

- Atomically swaps two variables.
- Another variation on the test and set is an atomic swap of two boolean variables.

```
void Swap(boolean & a, boolean & b)
{
      boolean temp = a;
      a = b;
      b = temp;
}
```

Implementation:

```
Shared data:
            boolean lock = false;

Process Pi :
            do
            {
                  key = true;
                  while (key == true)
                        Swap(lock, key);
                  < critical section >
                  lock = false;
                  < remainder section>
            }
```

4.9.3 Compare and Swap Instruction

Compare the value of a memory location with an old value that is passed to the instruction. If the two values match then write the new value into that memory location.

With this instruction, we set a memory location to a specific value *provided* that somebody did not change the contents since we last read them. If they did, our new value will not be set.

The compare-and-swap instruction compares the value of a memory location with an "old" value that is given to it. If the two values match then the "new" value is written into that memory location.

```
int compare_and_swap(int *x, int old, int new)
{
    int save = *x;     /* current contents of the memory */
    if (save == old)   *x = new;
    return save;       /* tell the caller what was read */
}
```

Implementation of compare and swap instruction:

```
while (compare_and_swap(&locked, 0, 1) != 0); /* spin until locked == 0 */
/* if we got here, locked got set to 1 and we have it */
<Critical Section>
locked = 0; /* release the lock */
<remainder Section>
```

Here, we spin on compare_and_swap and keep trying to set *locked* to 1 (last argument) if *lock*'s current value is 0. Let's consider a few cases:

- If nobody has the region locked, then *locked* is 0. We execute compare_and_swap and tell it to set *locked* to 1 if the current value is 0. The instruction does this and returns the old value (0) to us so we break out of the while loop.
- If somebody has the region locked, then *locked* is 1. We execute compare_and_swap and tell it to set *locked* to 1 if the old value was 0. But the old (current) value is 1, so compare_and_swap does not set it (it wouldn't matter) and returns the value that was already in *locked*, 1. Hence, we stay in the loop.
- Let's consider the potential race condition when two threads try to grab a lock at the same time. Thread 1 gets to run *compare_and_swap* first, so the instruction sets *locked* to 1 and returns the previous value of 0. When Thread 2 runs the instruction, *locked* is already set to 1 so the instruction returns 1 to the thread, causing it to keep looping.

In summary, we're telling the *compare_and_swap* instruction to set the contents of *locked* to 1 *only if locked* is still set to 0. If another thread came in and set the value of *locked* to non-zero, then the instruction will see that the contents do not match 0 (*old*) and therefore will not set the contents to *new*.

4.9.4 Fetch and Increment Instruction

This instruction increments a memory location and returns the previous value of that location.

You grab a ticket (fetch-and-increment) and wait until it's your turn. When you're done, you increment turn so that the thread that grabbed the next ticket gets to go.

This allows you to "take a ticket". Think of fetch-and-increment as one of those take-a-number machines at the counter in a supermarket:

```
int fetch_and_increment(int *x)
{
      last_value = *x;
      *x = *x + 1;
      return last_value;
}
```

The return value, *last_value* corresponds to the number on your ticket while the machine is now ready to generate the next higher number. Implementation of critical section:

```
ticket = 0;
turn = 0;
...
myturn = fetch_and_increment(&ticket);
while (turn != myturn); /* busy wait */
<Critical Section>
fetch_and_increment(&turn);
```

The variable *ticket* represents the ticket number in the machine. The variable *myturn* represents your ticket number and *turn* represents the number that is currently being served. Each time somebody takes a ticket via *fetch_and_increment*, the value of *ticket* is incremented. Each time somebody is done with their critical section and is ready for the next thread to get served, they increment *turn*.

Disadvantage of Atomic Instructions

These atomic-instruction-based mechanisms all require looping in software to wait until the lock is released. This is called **busy waiting** or a **spinlock** (no useful work to do during wait).

With a priority scheduler, it may always get scheduled to run, starving a low priority thread that may be the one that has the lock that needs to be released. This situation is known as **priority inversion**.

A more desirable approach than spinlocks is to have the operating system provide user processes with system calls that we can use for mutual exclusion and put processes that are waiting for a critical section into a waiting state. This way, a waiting process will not be running until it is allowed entry into the critical section.

Example-4.1 Suppose each thread does the following:

```
while (TestAndSet(flag) == false);
Execute Critical Section;
flag = false;
```

Will this code protect the critical section?

Solution:

No. Since TestAndSet sets a memory location to 1 (true), the locked condition is indicated by the value of flag == true. Hence, the above code doesn't wait when the lock is already taken. Hence, it doesn't protect the critical section.

To fix the code, replace TestAndSet(flag) == false with TestAndSet(flag) == true.

Example-4.2 Consider a deposit of $100 and a withdrawal of $50 from account A. Initially, account A holds $1000. Assume the two operations are implemented by two threads, and assume that each instruction is atomic.

Thread1:	Thread2:
t1 = A;	t2 = A;
t1 = t1+$100;	t2 = t2 – 50;
A = t1;	A = t2;

(a) What are the possible outcomes of the above transfer? For each outcome give a possible execution of the threads leading to that outcome.

(b) Assume Thread1 repeats the deposit operation 3 times and Thread2 executes the withdraw operation 2 times (i.e., there are three depostis of $100 each, and 2 withdrawals of $50 each).

What are the potential outcome?

Solution:

(a) Case 1: A = 1050

```
t1 = A;
t1 = t1 + 100;
A = t1;
t2 = A;
t2 = t2 – 50;
A = t2
```

Case 2: A = 950 (Thread1's update is lost)

```
t1 = A;                    t2 = A;
t1 = t1+100;               t2 = t2 – 50;
A = t1;                    A = t2
```

Case 3: A = 1000 (Thread2's update is lost)

```
t1 = A;                    t2 = A;
t1 = t1 + 100;             t2 = t2 – 50;
A = t1;                    A = t2
```

(b) 900, 1000, 1050, 1100, 1150, 1200, 1250, 1300

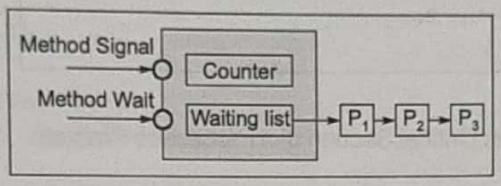
4.10 Semaphores

In general, the semaphore is initialized to the number of concurrent threads that you want to enter a section before they block. Logically, a semaphore S is an integer variable that, apart from initialization, can only be accessed through two atomic and mutually exclusive operations:

1. **Wait(S) or P:** If the semaphore value is greater than 0, decrement the value. Otherwise, wait until the value is greater than 0 and then decrement it.
2. **Signal(S) or V:** Increment the value of the semaphore.

Operation	Semaphore	Dutch	Meaning
wait() or down()	P	Proberen	Test for decrement
signal() Or up()	V	Verhogen	Increment

A semaphore is an object that consists of a counter, a waiting list of processes, Signal and Wait functions.



Semaphore

The most basic use of a semaphore is to initialize it to 1. When a thread want to enter a critical section, it calls *down* and enters the section. When another thread tries to do the same thing, the operating system will put it to sleep because the value of the semaphore is already 0 due to the previous call to *down*. When the first thread is finished with the critical section, it calls *up*, which wakes up the other thread that's waiting to enter.

4.10.1 Types of Semaphores

1. **Counting semaphore:** Integer value can range over an unrestricted domain.

```
down(sem s)
{
     if (s > 0)  {s = s - 1;}
     else {/* put the thread to sleep on event s */}
}
up (sem s)
{
     if (one or more threads are waiting on s)
        {/* wake up one of the threads sleeping on s */}
     Else { s = s + 1;}
}
```

Counting Semaphore/General Semaphore

2. **Binary semaphore:** Integer value can range only between 0 and 1; can be simpler to implement. We can implement a counting semaphore *S* as a binary semaphore. Semaphores are used by two or more threads to ensure that only one of them can enter a critical section. Such semaphores are known as **binary semaphores**. Here is an example of accessing a critical section using a binary semaphore:

```
sem mutex = 1; /* initialize semaphore */
. . .
down(&mutex);
/* do critical section */
up(&mutex);
```

4.10.2 Semaphore Solution with Busy Waiting

If a process is in critical-section, the other process that tries to enter its critical-section must loop continuously in the entry code. The classical definitions for wait and signal are:

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

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

Implementing Semaphores: Critical Section of *n* Processes Problem

```
Shared data: semaphore mutex; //initially mutex=1
```

```
Process Pi:
do
{
    wait(mutex);
    <critical section>
    signal(mutex);
    <remainder section>
} while (1)
```

4.10.3 Semaphore solution with Block and Wakeup

In Busy-wait Problem, processes waste CPU cycles while waiting to enter their critical sections.

Modify wait operation into the block operation. The process can block itself rather than busy-waiting. Place the process into a wait queue associated with the critical section. Modify signal operation into the wakeup operation. Change the state of the process from wait to ready.

When a process executes the wait operation and finds that the semaphore value is not positive, the process can block itself. The block operation places the process into a waiting queue associated with the semaphore. A process that is blocked waiting on a semaphore should be restarted when some other process executes a signal operation. The blocked process should be restarted by a wakeup operation which put that process into ready queue.

- To implemented the semaphore, we define a semaphore as a record as:

```
typedef struct
{
    int value;
    struct process *L;
} semaphore;
```

- Assume two simple operations:
  (*i*) block suspends the process that invokes it.
  (*ii*) wakeup(*P*) resumes the execution of a blocked process P.

- Semaphore operations defined as:

```
wait(S)
{
     S.value--;              if (S.value < 0)
     {
         add this process to S.L;
         block;
     }
}
```

```
signal(S)
{
     S.value++;
     if (S.value <= 0)
     {
         remove a process P from S.L;
         wakeup(P);
     }
}
```

Advantages of Semaphores

1. Semaphores are machine independent.
2. Semaphores are simple to implement.
3. Correctness is easy to determine.
4. Can have many different critical sections with different semaphores.
5. Semaphore acquire many resources simultaneously.
6. No waste of resources due to busy waiting.

Drawback of Semaphore

1. They are essentially shared global variables.
2. Access to semaphores can come from anywhere in a program.
3. There is no control or guarantee of proper usage.
4. There is no linguistic connection between the semaphore and the data to which the semaphore controls access.
5. They serve two purposes, mutual exclusion and scheduling constraints.
6. Processes using semaphores must be aware of each other and coordinate accordingly.
7. Improper use of Semaphore may lead to deadlock and starvation

To achieve the desired effect, we can view the semaphore as a variable that has an integer value upon which three operations are defined:

- A semaphore may be initialized to a non-negative value
- The wait operation decrements the semaphore value. If the value becomes negative, then the process executing the wait is blocked
- The signal operation increments the semaphore value. If the value is not positive, then a process blocked by wait operation is unblocked.

- Semaphores are operated on by a signal operation, which increments the semaphore value and the wait operation, which decreases it. The initial value of semaphore indicates the number of wait operations that can be performed on the semaphore.

(*i*) $V = I - W + S$
where I is the initial value of the semaphore
W is the number of completed wait operations performed on the semaphore
S is the number of signal operations performed on it
V is the current value of the semaphore (which must be greater than or equal to zero).

(*ii*) V is ≥ 0 implies $I - W + S \geq 0$, which gives $I + S \geq W$ or $W \geq I + S$. Thus, the number of wait operations must be less than or equal to the initial value of the semaphore, plus the number of signal operations. A binary semaphore will have an initial value of 1 ($I = 1$), $W \leq S + 1$.

- In mutual exclusion, waits always occur before signals, as waits happen at the start of a critical piece of code, with a signal at the end of it. The above equation states that no more than one wait may run to completion before a signal has been performed. Thus no more than one process may enter the critical section at a time as required.

4.11 Mutex

A mutex is a lock. Only one state (locked/unlocked) is associated with it. However, a *recursive mutex* can be locked more than once (POSIX complaint systems), in which a count is associated with it, yet retains only one state (locked/unlocked). The programmer must unlock the mutex as many number times as it was locked.

If a thread which had already locked a mutex, tries to lock the mutex again, it will enter into the waiting list of that mutex, which results in deadlock. It is because no other thread can unlock the mutex.

Mutex Vs Semaphore

- A mutex is **locking mechanism** used to synchronize access to a resource. Only one task can acquire the mutex. It means there will be ownership associated with mutex, and only the owner can release the lock (mutex).

NOTE: Semaphore is **signaling mechanism** and different from mutex. Mutex may be referred as binary semaphore.

- Mutex can be released only by thread that had acquired it, while you can signal semaphore from any other thread (or process), so semaphores are more suitable for some synchronization problems like producer-consumer.
- A *Mutex* controls access to a single shared resource. It provides operations to *acquire()* access to that resource and *release()* it when done.
- A *Semaphore* controls access to a shared pool of resources. It provides operations to *Wait()* until one of the resources in the pool becomes available, and *Signal()* when it is given back to the pool.
- When number of resources a Semaphore protects is greater than 1, it is called a *Counting Semaphore*. When it controls one resource, it is called a *Boolean Semaphore*. A boolean semaphore is equivalent to a mutex.
- Thus a Semaphore is a higher level abstraction than Mutex. A Mutex can be implemented using a Semaphore but not the other way around.
- A semaphore can be a Mutex but a Mutex can never be semaphore. This simply means that a binary semaphore can be used as Mutex, but a Mutex can never exhibit the functionality of semaphore.
- Both semaphores and Mutex are non -recursive in nature.
- No one owns semaphores, whereas Mutex are owned and the owner is held responsible for them.
- A Mutex, by definition, is used to serialize access to a section of re-entrant code that cannot be executed concurrently by more than one thread. A semaphore, by definition, restricts the number of

simultaneous users of a shared resource up to a maximum number semaphores are system-wide and remain in the form of files on the filesystem, unless otherwise cleaned up. Mutex are process-wide and get cleaned up automatically when a process exits.

- Mutex are lighter when compared to semaphores. What this means is that a program with semaphore usage has a higher memory footprint when compared to a program having Mutex. With 'mutex' this can't happen. No other thread can unlock the lock in your thread, with 'binary-semaphore' this can happen. Any other thread can unlock the lock in your thread.

Example-4.3 Suppose that there are two processes, A and B, which share semaphores R and S. Initially R = 0 and S = 1. Consider the following codes for A and B,

Process A	Process B
P(R);	Operation B.1;
P(S);	V(R);
Operation A.1;	Operation B. 2;
V(S);	P(S);
Operation A.2;	Operation B.3;
V(R);	V(S);

Which of the following statements are true?

(a) Operation A.1 cannot start until operation B.1 is complete

(b) Operations A.1 and B.2 cannot execute simultaneously

(c) Operations A.1 and B.3 cannot execute simultaneously

(d) Operations A.2 and B.3 cannot execute simultaneously

(e) The system may deadlock

Solution:

(a) **True.** Process A cannot complete its call to P(R), and therefore cannot start operation A.1, until process B executes V(R). B does not execute V(R) until after it completes operation B.1.

(b) **False.** After B has executed V(R), A can proceed with executing P(R), P(S) and enter operation A.1, and B can simultaneously enter operation B.2.

(c) **True.** Mutual exclusion between A.1 and B.3 is enforced by the semaphore S. Both A.1 and B.3 are preceded by P(S) and followed by V(S).

(d) **False.** Operation A.2 is protected by R while operation B.3 is protected by semaphore S.

(e) **False.** Process B does not do any "hold and wait", so deadlock is impossible.

Example-4.4 Suppose that there are two processes A and B running the following code:

Process A	Process B
DOWN(S)	DOWN(T)
DOWN(T)	DOWN(S)
<Critical Section>	<Critical Section>
UP(T)	UP(S)
UP(S)	UP(T)

(a) Is it possible for both processes to be in their critical sections simultaneously?

(b) Is it possible for the two processes to deadlock?

Solution:

(a) The two processes cannot be in their critical section simultaneously. Suppose that A executes DOWN(S) before B does. B can only get to its critical section after it executes DOWN(S). B can only proceed past the execution of DOWN(S) after A has executed UP(S), which is not until after A has completed its critical section.

(b) Yes. Suppose that A executes DOWN(S), is preempted, and B executes DOWN(T). Then when A is continued, it will execute DOWN(T) and block; when B is continued, it will execute DOWN(S) and block. Neither can proceed until the other executes an UP, so they are deadlocked.

Example-4.5 Consider the following synchronization problem: There are three processes, P, Q, and R. All three are executing a loop from 1 to N. You wish to enforce the condition that no process may enter the $(I+1)^{st}$ iteration until the other two processes have completed the I^{th} iteration. Show how this condition can be enforced using three semaphores.

Solution:

Semaphores sP, sQ, sR are initialized to 0

P	Q	R

for I : = 1 to N
{
up(sQ)
up(sR)
down(sP)
down(sP)
}
	for I : = 1 to N
	{
	up(sP)
	up(sR)
	down(sQ)
	down(sQ)
	}
		for I : = 1 to N
		{
		up(sP)
		up(sQ)
		down(sR)
		down(sR)
		}

Thus, process P cannot complete its two calls to down(sP) until both Q and R have completed their calls to up(sP), and likewise for the other processes.

Example-4.6 Suppose that there are three processes P, Q, and R, and two semaphores *s* and *t*. Initially $s = 1$ and $t = 2$. The following operations occur:

P executes DOWN(s)
Q executes DOWN(t)
Q executes DOWN(s)
R executes DOWN(t)
R executes DOWN(s)
P executes UP(s)
P executes DOWN(t)

Which of the following is a *possible* state of the system after these operations?

(a) All three processes are blocked.
(b) P is blocked; Q and R are not blocked.
(c) P and Q are blocked; R is not blocked.
(d) P and Q are not blocked; R is blocked.

Solution:

(c) When P executes UP(s), either process Q or process R may be unblocked. If the semaphore uses a FIFO queue, then Q would unblock; but a semaphore is not *required* to use FIFO queue.

Example - 4.7 Consider the Dining Philosophers problem, in which a set of Philosophers sit around a table with one chopstick between each of them. Let the Philosophers be numbered from 0 to $n-1$ and be represented by separate threads. Each Philosopher executes Dine(i), where "i" is the Philosopher's number. Assume that there is an array of semaphores, Chop[i] that represents the chopstick to the left of Philosopher i. These semaphores are initialized to 1.

```
void Dine(int i)
{
    Chop[i].P( );                /* Grab left chopstick*/
    Chop[(i +1)%n].P( );         /* Grab right chopstick*/
    EatAsMuchAsYouCan( );
    Chop[i].V();                 /* Release left chopstick*/
    Chop[(i +1)%n].V( );         /* Release right chopstick*/
}
```

Find whether this solution lead to deadlock. Explain the four conditions of deadlock.

Solution:

This solution can deadlock.

1. **Mutual exclusion:** Semaphores are initialized to 1; consequently, each chopstick can only be held by one thread at a time.
2. **No preemption:** The chopsticks cannot be taken away from a task without violating the semantics of semaphores and hence the assumptions of the code.
3. **Hold and wait:** During a deadlock, the second P( ) call above causes the thread to wait while it is holding another chopstick (first P( ) call).
4. **Circular wait:** Philosopher i grabs Chop[i] and waits for Philosopher $(i+1)\%n$ to release Chop[$(i+1)\%n$]. This is a cycle since Philosopher $n-1$ completes the cycle by grabbing Chop[$n-1$] and waiting for Philosopher 0 to release Chop[0].

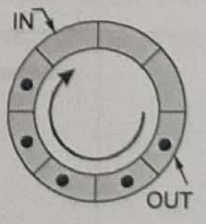
4.12 Classical Problems of Synchronization with Semaphore Solution

The following problems of synchronization are considered as classical problems:

- Bounded-Buffer Problem
- Dining-Philosophers Problem
- Readers and Writers Problem
- Sleeping Barber

4.12.1 Bounded-Buffer Problem

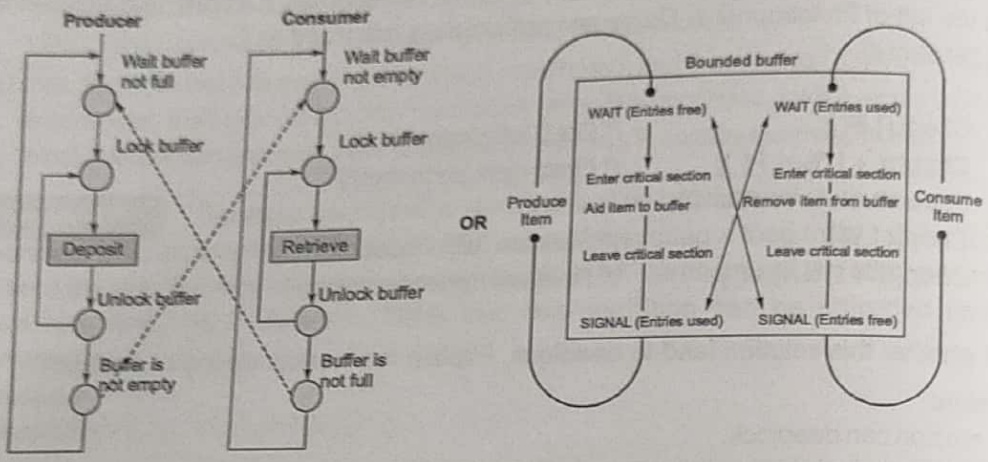
Assume there are two processes named as producer and consumer.
A *producer* that generates items that go into a buffer.
A *consumer* that takes things out of the buffer.
The buffer has a maximum capacity of N items.



Suppose we have a circular buffer with two pointers in and out to indicate the next available position for depositing data and the position that contains the next data to be retrieved (See the Figure).

There are two groups of threads, producers and consumers. Each producer deposits a data items into the in position and advances the pointer in, and each consumer retrieves the data item in position outand advances the pointer out.

Before a producer or a consumer can have access to the buffer, it must lock the buffer. After a producer and consumer finishes using the buffer, it must unlock the buffer. Combined these activities together, we have the following diagram:



Conditions

- If a producer produces faster than the consumer consumes then it will have to go to sleep until there's enough room in the buffer.
- If the consumer consumes faster than the producer produces, the consumer will go to sleep until there is something for it to consume.

Semaphore Solution

Implementation:

Shared Data:

```
semaphore mutex=1;      /*for mutual exclusion */
semaphore empty=N;      /*keep producer from over-producing */
semaphore full=0;       /*keep consumer sleeping if nothing to consume*/
```

Producer Process:

```
producer()
{
      while(1)
      {
          produce_item(item); /* produce something */
          P(empty);           /* decrement empty count */
          P(mutex);           /* start critical section */
          enter_item(item);   /* put item in buffer */
          V(mutex);           /* end critical section */
          V(full);            /* increment full slot */
      }
}
```

Consumer Process:

```
consumer()
{
      while(1)
      {
          P(full);            /* one less item */
          P(mutex);           /* start critical section */
          remove_item(item); /* get the item from the buffer */
          V(mutex);           /* end critical section */
          V(empty);           /* one more empty slot */
          consume_item(item);/* consume it */
      }
}
```

Consider the following three semaphore variables which are used to implement a solution to bounded buffer problem.

1. **Mutex (Mutual exclusion):** It is initialized to 1 so that we can use it to guarantee mutual exclusion whenever we add or remove something from the buffer.
2. **Empty (Number of free slots in the buffer):** It is initialized to N.

   If empty = 0, the producer goes to sleep (when there is no more room in the buffer).

   If empty > 0, wake up the producer (when there is free room in the buffer).

   When we execute *P(empty)*, the operation will decrement *empty* and return to the thread until *empty* is 0 (no more slots). At that point, any successive calls to *P(empty)* will cause the producer to go to sleep. When the consumer consumes something, it will execute *V(empty)*, which will wake up the producer if it is sleeping on the semaphore and allow it to add one more item to the buffer.

   If the producer was not sleeping because there is still room in the buffer, *V(empty)* will simply increment empty to indicate that there is one more space in the buffer.

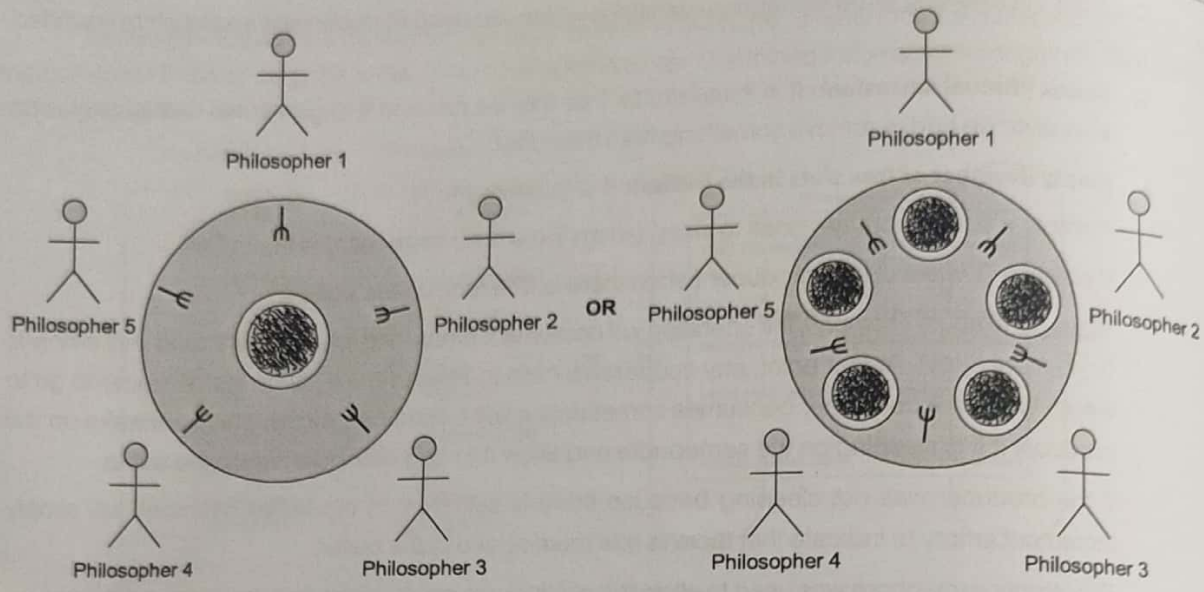
   The *empty* semaphore was used to allow the producer to go to sleep when there was no more room in the buffer and wake up when there were free slots.
3. **Full (Number of filled slots in the buffer):** We want the consumer to go to sleep when the producer has not generated anything for the consumer to consume. To put the consumer to sleep, we will create yet another semaphore called *full* and have it count the number of items in the buffer.
   Since there are no items in the buffer initially, we will initialize *full* to 0.
   Initially, when the consumer starts running, it will run *P(full)*, which will put it to sleep.
   When the producer produces something, it calls *V(full)*, which will cause the consumer to wake up.
   If the producer works faster than the consumer, successive calls to *V(full)* will cause *full* to get incremented each time.

   When the consumer gets to consuming, successive calls to *P(full)* will just cause *full* to get decremented until *full* reaches 0 (no more items to consume), at which point the consumer will go to sleep again.

4.12.2 Dining Philosophers Problem

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

The dining philosophers problem is invented by E.W. Dijkstra.



Dining Philosophers Problem

Dining Philosophers Problem

Solution 1: (Suffers with Deadlock)

```
void philosopher()
{
        while(1)
        {
                sleep();
                get_left_fork();
                get_right_fork();
                eat();
                put_left_fork();
                put_right_fork();
        }
}
```

If every philosopher picks up the left fork at the same time, none gets to eat ever.

Solution 2: (Correct, philosophers manipulate the state of their neighbours directly)

Pick up the left fork, if the right fork isn't available for a given time, put the left fork down, wait and try again. (Big problem if all philosophers wait the same time - we get the same failure mode as before, but repeated.) Even if each philosopher waits a different random time, an unlucky philosopher may starve (in the literal or technical sense). Require all philosophers to acquire a binary semaphore before picking up any forks. This guarantees that no philosopher starves (assuming that the semaphore is fair) but limits parallelism.

Implementation:

```
#define N 5                    /* Number of philosphers */
#define RIGHT(i) (((i)+1) %N)
#define LEFT(i) (((i)==N) ? 0 : (i)+1)
typedef enum {THINKING, HUNGRY, EATING} phil_state;
phil_state state[N];
semaphore mutex =1;
semaphore s[N];            /* one per philosopher, all 0 */
```

```
void test(int i)
{
      if ( state[i] == HUNGRY &&
      state[LEFT(i)] != EATING &&
      state[RIGHT(i)] != EATING ) {state[i] = EATING; V(s[i]);}
}
```

```
void get_forks(int i)
{
      P(mutex);
      state[i] = HUNGRY;
      test(i);
      V(mutex);
      P(s[i]);
}
```

```
void put_forks(int i)
{
      P(mutex);
      state[i]= THINKING;
      test(LEFT(i));
      test(RIGHT(i));
      V(mutex);
}
```

```
void philosopher(int process)
{
      while(1)
      {
            think();
            get_forks(process);
            eat();
            put_forks(process);
      }
}
```

When a philosopher is hungry it uses test to try to eat. If test fails, it waits on a semaphore until some other process sets its state to EATING. Whenever a philosopher puts down forks, it invokes test in its neighbours. (Note that test does nothing if the process is not hungry, and that mutual exclusion prevents races.) So this code is correct, but somewhat obscure. And more importantly, it doesn't encapsulate the philosopher - philosophers manipulate the state of their neighbours directly.

Solution 3: (Correct, not required to write another process's state)

The following Solution does not require a process to write another process's state, and gets equivalent parallelism.

Implementation:

```
#define N 5                    /* Number of philosphers */
#define RIGHT(i) (((i)+1) %N)
#define LEFT(i) (((i)==N) ? 0 : (i)+1)
typedef enum {THINKING, HUNGRY, EATING} phil_state;
phil_state state[N];
semaphore mutex =1;
semaphore s[N];                /* one per philosopher, all 0 */
```

```
void get_forks(int i)
{
        state[i] = HUNGRY;
        while ( state[i] == HUNGRY )
        {
                P(mutex);
                if ( state[i] == HUNGRY &&
                state[LEFT] != EATING &&
                state[RIGHT(i)] != EATING )
                {
                        state[i] = EATING;
                        V(s[i]);
                }
                V(mutex);
                P(s[i]);
        }
}
```

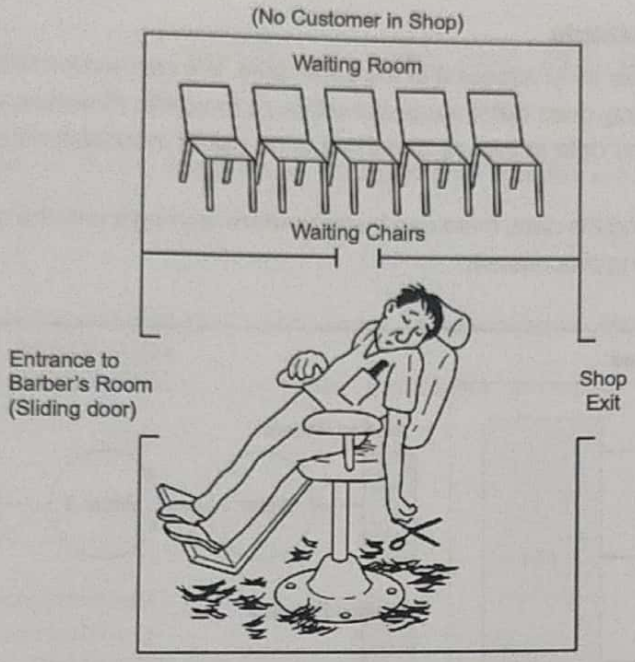
```
void put_forks(int i)
{
        P(mutex);
        state[i]= THINKING;
        if ( state[LEFT(i)] == HUNGRY ) V(s[LEFT(i)]);
        if ( state[RIGHT(i)] == HUNGRY) V(s[RIGHT(i)]);
        V(mutex);
}
```

```
void philosopher(int process)
{
        while(1)
        {
                think();
                get_forks(process);
                eat();
                put_forks();
        }
}
```

If you really don't want to touch other processes' state at all, you can always do the V to the left and right when a philosopher puts down the forks.

4.12.3 Sleeping Barber Problem

Barber shop with one barber, one barber chair and N chairs to wait in. When no customers the barber goes to sleep in barber chair and must be woken when a customer comes in. When barber is cutting hair new customers take empty seats to wait, or leave if no vacancy.



The following code uses three semaphores.

- *customers:* number of waiting customers
- *barbers:* number of barbers (0 or 1) that are idle
- *mutex:* semaphore variable

Implementation:

```
#define CHAIRS 5            /* # of chairs */
semaphore customers = 0;
semaphore barbers = 0;
semaphore mutex = 1;
int waiting = 0;
```

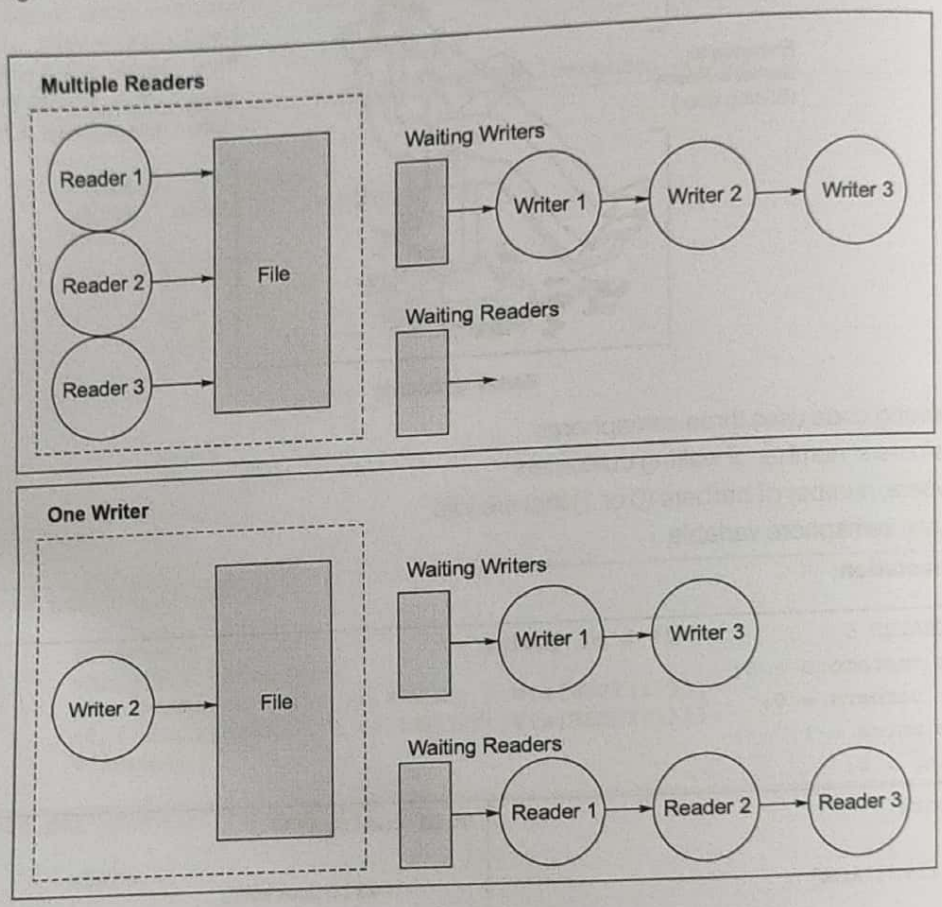
```
void barber()
{
      while (TRUE)
      {
            wait(customers);
            wait(mutex);
            waiting = waiting - 1;
            signal(barbers);
            signal(mutex);
            cut_hair();
      }
}
```

```
void customer()
{
      wait(mutex);
      if ( waiting < CHAIRS )
      {
            waiting = waiting + 1;
            signal(customers);
            signal(mutex);
            wait(barbers);
            get_haircut();
      }
      else
      {
            signal (mutex);
      }
}
```

4.12.4 Readers-Writers Example

Multiple processes may try to access it at the same time. We can allow multiple processes to read the data concurrently since reading does not change the value of the data. However, we can allow at most one process at a time to modify the data to ensure that there is no risk of inconsistencies arising from concurrent modifications.

If a process is modifying the data, there can be no readers reading it until the modifications are complete (shown in the Figure with waiting processes).



- To implement this, we use a semaphore (*canwrite*) to control mutual exclusion between a writer or a reader. This is a simple binary semaphore that will allow just one process access to the critical section. Used this way, it would only allow one of anything at a time: one writer or one reader. Now we need a hack to allow multiple readers.
- What we do is to allow other readers access to the critical section by bypassing requesting a lock if we have at least one reader that has the lock and is in the critical section. To keep track of readers, we keep a count (*readcount*).
- If there are no readers, the first reader gets a lock. After that point, *readcount* is 1. When the last reader exits the critical section (*readcount* goes to 0), that reader releases the lock.
- We also use a second semaphore, *mutex*, to protect the region where we change *readcount* and then test it.

Implementation:

```
sem mutex=1;        /* critical sections used only by the reader
*/
sem canwrite=1;     /*  critical  section  for  N  readers  vs.  1
writer*/
int readcount = 0; /* number of concurrent readers */
writer()
{
     for (;;)
     {
          down(&canwrite);  /* block if we cannot write */
          // write data
          up(&canwrite);    /* end critical section */
     }
}
reader()
{
     for (;;)
     {
          down(&mutex);
          readcount++;
          if (readcount == 1)
          down(canwrite);
          /* sleep or disallow the writer from writing */
          up(&mutex);
          // do the read
          down(&mutex);
          readcount--;
          if (readcount == 0)
          up(writer);
          /* no more readers! Allow the writer access */
          up(&mutex);
          // other stuff
     }
}
```

We have two chunks protected by the *mutex* semaphore.

Chunk 1:

```
readcount++;
if (readcount == 1)
down(&canwrite); /* sleep or disallow the writer from writing */
```

Chunk 2:

```
readcount–;
if (readcount == 0)
up(&canwrite); /* no more readers! Allow the writer access */
```

The *mutex* keeps us from having two threads update the value of *readcount* or check its value at the same time. The *if* statements in the above chunks simply check if we already have a reader inside the thread. If we do, then we don't touch the *canwrite* semaphore.

If we're the first reader, then we do a down(&canwrite) so that another thread will get blocked if it calls *writer()* since that function does a *down(&canwrite)* as well. Alternatively, we may end up blocking if another thread is currently active *inside writer()*.

If a thread calls *reader()*, then we bypass the *down()* because *readcount* > 1. At the end, we check if we were the last reader. If so, then we *up()* the *canwrite* semaphore. This will now allow any thread that is blocked in *writer()* to proceed.

4.13 Programming Language Solution: Monitors

- A monitor is a high-level abstraction that provides process safety.
- A monitor is a programming language construct that supports synchronized access to data.
- A monitor is a software module consisting of one or more procedures, an initialization sequence and local data. Components of monitors are shared data declaration, shared data initialization, operations on shared data and synchronization statement.
- A monitor is essentially an object for which
  (*i*) Object state is accessible only through the object's methods
  (*ii*) Only one method may be active at a time
- Only one process may be active within the monitor at a time. If two processes attempt to execute methods at the same time, one will be blocked until the other finishes.
- A monitor is a collection of procedures and data with the following characteristics:
  (*i*) Processes may call the functions in a monitor but may not access the monitor's internal data structures.
  (*ii*) Only one process may be active in a monitor at any instant. Any other process that tries to access an active monitor will be suspended.
- Two operations are supported by monitors (condition variables):
  (*i*) **Wait:** Causes the calling thread to block, and to release the monitor. The monitor is now available for use by another process.
  (*ii*) **Signal:** Resume execution of some process blocked after a wait on the same condition. If there are several blocked processes, choose one of them; If there is no blocked process, do nothing.

Advantage of Monitors: Monitors have the advantage of making parallel programming easier and less error prone than using techniques such as semaphores.

Disadvantage of Monitors: Monitors have to be implemented as part of the programming language. The compiler must generate code for them. This gives the compiler the additional burden of having to know what operating system facilities are available to control access to critical sections in concurrent processes. Some languages that do support monitors are Java, C#, Visual Basic, Ada, and Concurrent Euclid.

4.13.1 Bounded Buffer Using a Monitor

```
item buffer[N];                    /*buffer with capacity N */
int count;                         /*initially 0 */
cv *notfull, *notempty;            /*Condition variables */
```

```
Produce(item)
{
   While (count == N) wait(notfull);
   <add item to buffer>
   count = count + 1;
   signal(notempty);
}
```

```
Consume(item)
{
   while (count == 0) wait(notempty);
   <remove item from buffer>
   count = count - 1;
   signal(notfull);
}
```

A producer can add characters to the buffer only by means of the procedure append inside the monitor; the producer does not have direct access to buffer. The procedure first-checks the condition not full to determine if there is space available in the buffer. If not, the process executing the monitor is blocked on that condition.

Summary

- An independent process cannot affect or be affected by the execution of another process.
- A cooperating process can affect or be affected by the execution of another process.
- **Advantages of process cooperation:** Information sharing, Computation speed-up, Modularity and Convenience.
- **Race condition:** It is a situation where the semantics of an operation on shared memory are affected by the arbitrary timing sequence of collaborating processes.
- **Critical region:** It is a portion of a process that accesses shared memory.
- A solution to the critical-section problem must satisfy the three requirements : Mutual exclusion, Progress and Bounded waiting.
- **Mutual exclusion:** It is a mechanism to enforce that only one process at a time is allowed into a critical region.
- **Progress:** If no process is executing in its critical section and there exist some processes that wish to enter their critical sections, then only those processes that are not executing in their remainder section can participate in the decision of which will enter its critical section next, and this selection cannot be postponed indefinitely.
- **Bounded Waiting:** There exist a bound on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.
- **Semaphore:** A semaphore is a non-negative integer value. Two atomic operations are supported on a semaphore. (a) down: decrements the value, since the value cannot be negative, the process blocks if the value is zero. (b) up: increments the value; if there are any processes waiting to perform a down, then they are unblocked.
- **Deadlock** is when two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes.
- **Starvation** is indefinite blocking when a process is never removed from the semaphore queue in which it was suspended.
- **Monitor:** A monitor is a high-lever programming language construct that packages data and the procedures to modify the data. The data can only be modified using the supplied procedures. Only one process is allowed inside the monitor at a time.
- Monitors also have condition variables that have two operations, wait and signal. Calling wait blocks the calling process. Calling signal unblocks waiting processes. Since condition variables are inside of a monitor there is no race condition when accessing them.
- **Classic synchronization problems:** Bounded-buffer, Readers-writers, Dining philosophers and Sleeping barber.
- **Solution to critical section problem:** Software Solutions, Hardware Solutions, Semaphores, etc.

Student's Assignment

Q.1 Which of the following is not a solution to the critical section problem?
(a) Semaphore (b) Monitor
(c) Peterson's solution (d) Shared memory

Q.2 "The final value of the shared data depends upon which process finishes last when there are several processes access and manipulated the shared data concurrently". The above situation is called as _____.
(a) Mutual exclusion (b) Deadblock
(c) Race condition (d) Starvation

Q.3 Let A and B be two semaphore variables and they are initialized with 3 and 0 respectively. Consider the following three concurrent processes.

Process-1	Process-2	Process-3

while (1)
{
P(A);
Printf("0");
V(B);
}
	while (1)
	{
	P(B);
	Printf("1");
	Printf("2");
	V(B);
	}
		while (1)
		{
		P(B);
		Printf("3");
		}

At most how many times "3" is printed by process-3 when the above processes are executing concurrently?
(a) 1 (b) 3
(c) 5 (d) None of these

Q.4 Let A and B be two semaphore variables and they are initialized with 3 and 0 respectively. Consider following three concurrent processes.

Process-1	Process-2	Process-3

while (1)
{
P(A);
Printf("0");
V(B);
}
	while (1)
	{
	P(B);
	Printf("1");
	V(B);
	}
		while (1)
		{
		P(B);
		Printf("3");
		}

What is the minimum number of 1's printed by the execution of the above processes?
(a) 0 (b) 1
(c) 3 (d) None of these

Q.5 Which of the following implementation may waste CPU cycles?
(a) Busy waiting (b) Sleep and wake
(c) Both (a) and (b) (d) Neither (a) nor (b)

Q.6 Match List-I with List-II and select the correct answer using the codes given below the lists:

List-I
A. Mechanism that abstracts away protection and scheduling access to shared data through the use of a lock and zero or more condition variables.
B. A variable is accessed only through the atomic operations.
C. Mechanism that allows a process to wait in a loop continuously when other process executing in the critical section.
D. A process will wait indefinitely to enter the critical section but some other process entering the critical section oftenly.

List-II
1. Semaphore
2. Monitor
3. Starvation
4. Spin lock

Codes:

	A	B	C	D
(a)	1	2	3	4
(b)	2	1	3	4
(c)	2	1	4	3
(d)	1	2	4	3

Q.7 Consider two processes P_0 and P_1 which shares a global variable 'flag'. The value of flag is either 0 or 1 where 0 indicates "P_0 is permitted to enter the critical section" and 1 indicates for P_1.
Assume P_0 and P_1 are concurrent processes which are executing the following code with initial value of flag as 0.

```
while (1)
{
while (flag != i);
< critical section >
   flag = 1 – i;
<Remainder section>
}
```

If P_0 starts executing first, which of the following holds by the above execution?

(a) Mutual exclusion (b) Progress
(c) Bounded wait (d) Both (a) and (c)

Q.8 Consider the following codes:

P_1	P_2	P_3
P(S1);	P(S2);	P(S3);
P(S2);	P(S3);	P(S1);
P(S3);	P(S1);	P(S2);
< critical section >	< critical section >	< critical section >
V(S3);	V(S1);	V(S2);
V(S2);	V(S3);	V(S1);
V(S1);	V(S2);	V(S3);

Assume that S1, S2 and S3 are shared binary semaphore variables. Initially S1 = 1, S2 = 1, and S3 = 1. If processes P_1, P_2 and P_3 executes concurrently then which of the following satisfied by them?

(a) Mutual exclusion and No deadlock
(b) No mutual exclusion but deadlock
(c) Mutual exclusion and deadlock
(d) Neither mutual exclusion nor deadlock

Q.9 Assume that two processes P_0 and P_1 shares one global boolean array "flag [ ]" and integer variable "turn". Initially flag [0] and [1] are false.

Consider the following code is executed by process P_i where i = 0 or 1.

```
while (true)
{
flag [i] = True;
  while (flag [j])
                {
                if (turn == j)
                 {
                   flag [i] = False;
                   while (turn == j);
                   flag [i] = True;
                 }
                }
<critical section>
        turn = j;
flag [i] = False;
<Remainder section>
}
```

If current process is P_i then other process is assumed as P_j where $j = 1 - i$. Which of the following condition is satisfied by the given algorithm?

(a) Mutual exclusion
(b) Mutual exclusion, progress
(c) Mutual exclusion, progress and bounded waiting
(d) None of these

Q.10 Consider the following 3-processes with the semaphore variables S_1, S_2 and S_3.

P_1	P_2	P_3
while (1)	while (1)	while (1)
{	{	{
$P(S_3)$;	$P(S_1)$;	$P(S_2)$;
print "0";	print "1";	print "2";
$V(S_2)$;	$V(S_3)$;	$V(S_1)$;
}	}	}

To print the output 21021021021, which of the following could be the initial values of semaphores when three processes executed concurentely?

(a) $S_1 = 0, S_2 = 0, S_3 = 0$
(b) $S_1 = 1, S_2 = 0, S_3 = 0$
(c) $S_1 = 0, S_2 = 0, S_3 = 1$
(d) $S_1 = 0, S_2 = 1, S_3 = 0$

Q.11 Consider the following codes for Reader-Writer problem. Initial values of semaphore variables are W = 1 and mutex = 1. Initial value of rcount = 0 to indicate the read count.

```
writer                    Reader
P(W);                     P(mutex);
<write operation >        rcount ++;
V(W);                     if (rcount ==1) ____A____;
                          V(mutex);
                          < Read operation >
```

```
P(mutex);
rcount--;
if (rcount ==0) ____B____;
V(mutex);
```

Find the missing statements at A and B respectively to give the semaphore based solution for reader-writer problem. (Multiple readers may allowed but only one writer).

(a) P(W) and V(W) (b) P(W) and P(W)
(c) V(W) and V(W) (d) V(W) and P(W)

Q.12 Consider the following bounded-buffer problem. Assume that the buffer is initially empty and buffer has upperbound as *n*-items.

```
Semaphore full = 0; // Number of items in buffer
Semaphore empty = N; // Number of empty cells
Semaphore mutex = 1;
```

Producer (item)	Consumer (item)
{	{
P(empty);	P(__A__);
P(mutex);	P(__B__)
<producer (item)>	<consumer (item)>
V(mutex);	V(__C__)
V(full);	V(__D__)
}	}

Find the missing semaphore variables at A, B, C and D respectively to implement the bounded-buffer synchronization problem correctly for mutual exclusion.

(a) empty, mutex, mutex, empty
(b) full, mutex, mutex, full
(c) empty, mutex, mutex, full
(d) full, mutex, mutex, empty

Q.13 Which of the following is not correct about "Monitors"?

(a) Use condition variables for synchronization
(b) Prevent multiple processes from executing monitor code at the same time
(c) Hide mutual exclusion details from calling routines
(d) None of these

Q.14 "Test and Set" instruction can be used to protect the shared data between _____.

(a) Processes running on multiple networked computers
(b) Processes running on a single CPU
(c) Both (a) and (b)
(d) Neither (a) nor (b)

Q.15 Which of the following might be the reason that mutual exclusion mechanisms (semaphores, monitors, etc.) are used to solve the producer-consumer problem?

(a) To slow down the consumer that run faster than producer
(b) To slow down the producer that run faster than consumer
(c) To prevent the shared buffer from being corrupted
(d) All of these

Q.16 Consider the following code:

```
while (1)
{
while (turn ! = i);
<critical section>
        turn = j;
<remainder section>
}
```

If P_i executes the above code (where *i* refers the current process and j refers the other one), which of the following can not hold by the given solution with two concurrent processes P_i and P_j?

(a) Mutual exclusion
(b) Progress
(c) Bounded wait
(d) None of these

Q.17 Consider a process 'P' tries to acquire a lock using the test-and-set instruction as following.

```
while (Test-and-Set (lock));
```

If the lock is busy (some other process acquired the lock)and all interrupts are disabled then which of the following is correct?

(a) Process 'P' blocks, waiting for the lock to be available
(b) Process 'P' wastes processor and memory cycles
(c) Both (a) and (b)
(d) Neither (a) nor (b)

Q.18 Consider the following producer and consumer code.

```
# define N = 100
int mutex = 1;// Binary semaphore variable
int empty = N;// Counting semaphore variable
int full = 0;// counting semaphore variable
```

```
Void producer ()
{
  int item;
  while (1)
  {
    item = producer-item ();
    down (empty);
    down (mutex);
    insert_item (item);
    up (mutex);
    up (full);
  }
}
```

```
Void consumer ()
{
  int item;
  while (1)
  {
    down (mutex);
    down (full);
    item = consume_item ();
    up (mutex);
    up (empty);
  }
}
```

In the above code **mutex**, **empty** and **full** are semaphore shared variables and **item** is local to the both producer and consumer.

Inser_item (item) function will place "item" into buffer and consume_item function removes an item from the buffer. Which of the following holds by the code?

(a) Satisfies mutual exclusion and no deadlock occurs
(b) Satisfies mutual exclusion but deadlock may occur
(c) Not satisfies mutual exclusion and no deadlock occurs
(d) Not satisfies mutual exclusion but deadlock may occur

Q.19 Consider the following code to solve the critical section problem for two processes P_0 and P_1. Initially flag [i] contain false for $i = 0$ and 1.

```
while (1)
{
 flag [i] = true;
while (flag [j]);
<critical section>
flag [i] = flase;
<remainder section>
}
```

Assume that i refers to the current process P_i and j refers the other process P_j. If two processes executing above code concurrently then which of the following does not satisfy the above solution?

(a) Mutual exclusion and progress
(b) Mutual exclusion and bounded wait
(c) Progress and bounded wait
(d) None of these

Q.20 Consider the following reader-writer problem.

```
Semaphore mutex, write;
int readcount = 0;
mutex = 1;
write = 1;
```

```
Writer                  Reader
P(write);               P(mutex);
<write document>        readcount ++;
V(write);               if (readcount == 1) P(write);
                        <read document>
                        readcount —;
                        if (readcount == 0) V(write);
                        V(mutex);
```

Which of the following statement is true for the above problem?

(a) Mutual exclusion satisfied when multiple readers are reading the shared document
(b) Writer and multiple readers are simultaneously reading and writing into the document
(c) Both (a) and (b)
(d) Neither (a) nor (b)

Q.21 Let P and V be samaphore operations. P represents wait and V represents signal operation. Counting semaphore variable S is initialized to 1 and no blocked processes in the system. If the following operations are performed in the given order then what is the value of S?

P, V, P, V, V, P, P, V, V, V, P, V, V, V, P, P

(a) 0 (b) 1
(c) 3 (d) 5

Q.22 Let S be the binary semaphore variable S = 1 initially. Assume that no blocked processes exist in hte system. If the following operations performed, how many blocked processes are present in the system at the end?

5P, 7V, 10P, 14V, 15P, 21V

(a) 0 (b) 14
(c) 20 (d) 12

Q.23 Let S be the binary semaphore variable and S = 1 initially. Assume that there are no blocked processes in the system. If the following operations performed in the given order then how many blocked processes are present in the system at the end?

7V, 5P, 14V, 10P, 21V, 15P

(a) 0 (b) 14
(c) 20 (d) 12

Q.24 Consider the following code for two processes P_1 and P_2.

P_1	P_2
$P(A)$;	$P(A)$;
$P(B)$;	$V(B)$;
$x = y + 2$;	$y = x + 1$;
$V(B)$;	$V(B)$;
$V(A)$;	$V(A)$;

Assume A and B are binary semaphore variables initialized with values 1 and 0 respectively. x and y are shared variables initialized with 5 and 0 respectively. If there is no deadlock occurs while executing P_1 and P_2 then what are the values of x and y respectively after the execution of given code?

(a) 2 and 6 (b) 8 and 6
(c) 2 and 3 (d) None of these

Q.25 The dining Philosophers problem with N Philosophers has the solution for synchronization without deadlock. Which of the following is correct solution?

(a) (N – 1) Philosophers take the left fork first and later right fork. Remaining one Philosopher take right fork first then left fork.
(b) Odd Philosopher follows "take left fork first then right fork" and even Philosopher follows "take right fork first then left fork"
(c) Both (a) and (b) are correct solutions.
(d) Neither (a) nor (b)

Answer Key:

1. (d)	**2.** (c)	**3.** (b)	**4.** (a)	**5.** (a)
6. (c)	**7.** (d)	**8.** (c)	**9.** (c)	**10.** (d)
11. (a)	**12.** (d)	**13.** (d)	**14.** (b)	**15.** (d)
16. (b)	**17.** (b)	**18.** (b)	**19.** (c)	**20.** (a)
21. (c)	**22.** (a)	**23.** (b)	**24.** (b)	**25.** (c)

CHAPTER 05

Concurrency and Deadlock

5.1 Concurrency

A state in which process exist simultaneously with another process then those it is said to be concurrent. *Concurrency* allows multiple applications to share resources in such a way that applications appear to run at the same time. Since a typical application does not consume all resources at a given time, a careful coordination can make each application run as if it owns the entire machine.

Concurrent processing is basis of multi-programmed operating systems.

Process will need certain resources such as CPU time, memory, files, I/O devices, etc., to accomplish its task. Resources are given to the process when it is created.

There are several motivations for allowing concurrent execution:

- **Physical resource sharing:** Multiuser environment since hardware resources are limited.
- **Logical resource sharing:** Shared file (same piece of information.
- **Computation speedup:** Parallel execution.
- **Modularity:** Divide system functions into separation processes.

5.1.1 Relation between Processes

The processes executing in the operating system is one of the following two types:

- Independent processes
- Cooperating processes.

Independent Process

Its state is not shared to any other process.

- The result of execution depends only on the input state (Deterministic).
- The result of the execution will always be the same for the same input.
- The termination of the independent process will not terminate any other.

Cooperating Process

Its state is shared along other processes.

- The result of the execution depends on relative execution sequence and cannot be predicted in advanced (Non-deterministic).

- The result of the execution will not always be the same for the same input.
- The termination of the cooperating process may affect other process.

5.1.2 Operation on A Process

Most of systems support at least two types of operations that can be invoked on a process; process creation and process deletion.

Process Creation

A parent process and then children of that process can be created. When more than one process is created several possible implementations exist.

- Parent and child can execute concurrently. (Concurrent Processes)
- The parent waits until all of its children have terminated. (Sequential)
- The parent and children share all resources in common.
- The children share only a subset of their parent's resources.
- The parent and children share no resources in common.

Process Termination

A child process can be terminated in the following ways:

- A parent may terminate the execution of one of its children for a following reasons:
  (*i*) The child has exceeded its allocated resource usage.
  (*ii*) The task assigned to the child is no longer required.
- If a parent has terminated the its children must be terminated. (Cascading termination)

5.1.3 Principles of Concurrency

Both **interleaved and overlapped processes** can be viewed as examples of concurrent processes, they both present the same problems.

The relative speed of execution cannot be predicted. It depends on the activities of other processes, the way the operating system handles interrupts and the scheduling policies of the operating system.

Difficulties

- **Sharing of global resources:** If two processes both make use of a global variable and both perform reads and writes on that variable, then the order in which the various reads and writes are executed is critical.
- It is difficult for the operating system to manage the allocation of resources optimally.
- It may be inefficient for the operating system to simply lock the channel and prevents its use by other processes.
- It becomes very difficult to locate a programming error, because reports are usually not reproducible.

Advantages of Concurrency

- Able to run multiple applications at the same time.
- **Better resource utilization:** Resources that are unused by one application can be used for other applications
- **Better average response time:** Without concurrency, each application has to be run to completion before the next one can be run.
- **Better performance:** If one application uses only the processor, while another application uses only the disk drive, the time to run both applications concurrently to completion will be shorter than the time to run each application consecutively.

Drawbacks of Concurrency

- Multiple applications need to be protected from one another.
- Multiple applications may need to coordinate through additional mechanisms.
- Switching among applications requires additional performance overheads and complexities in operating systems (e.g., deciding which application to run next).
- In extreme cases of running too many applications concurrently will lead to severely degraded performance.

5.1.4 Issues of Concurrency

- **Non-atomic:** Operations that are not atomic, but interruptible by multiple processes can cause problems.
- **Race conditions:** A race condition occurs if the outcome depends on which of several processes gets to a point first.
- **Blocking:** Processes can *block* waiting for resources. A process could be blocked for a long period of time waiting for input from a terminal. If the process is required to periodically update some data, this would be very undesirable.
- **Starvation:** It occurs when a process does not obtain service to progress.
- **Deadlock:** It occurs when two processes are blocked and hence neither can proceed to execute.

5.2 Precedence Graph

A precedence graph is a directed acyclic graph with the following properties:

- Nodes of graph correspond to individual statements of program code.
- Edge between two nodes represents the execution order
- A directed edge from node A to node B shows that statement A executes first and then Statement B executes (B depends on A)
- Consider the following code.

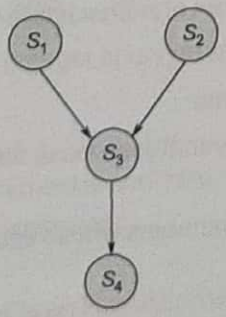
  (S1) $a := x + y$;
  (S2) $b := z + 1$;
  (S3) $c := a - b$;
  (S4) $w := c + l$;

  If above code is executed concurrently, the following precedence relations are exist:

  (*i*) $c := a - b$ cannot be executed before both a and b have been assigned values. (Directed edge is from S_1 to S_2)

  (*ii*) $w := c + 1$ cannot be executed before the new values of c has been computed.

  (*iii*) The statements $a := x + y$ and $b := z + 1$ could be executed concurrently.



Bernstein's Conditions

Following three conditions must be satisfied for two successive statements S1 and S2 to be *executed concurrently* and still produce the same result.

(*i*) $R(S1) \cap W(S2) = \{\}$

(*ii*) $W(S1) \cap R(S2) = \{\}$

(*iii*) $W(S1) \cap W(S2) = \{\}$

The read and write set for the statement $c := a - b$;

- $R(c := a - b) = \{a, b\}$
- $W(c := a - b) = \{c\}$

The read and write set for the statement $w := c + 1$;

- $R(w := c + 1) = \{c\}$
- $W(w := c + 1) = \{w\}$

The read and write set for the statement $x := x + 1$;

- $R(x := x + 2) = W(x := x + 2) = \{x\}$

Example-5.1 Consider the following precedence relations of some program.

S2 and S3 can executed after S1 completes.

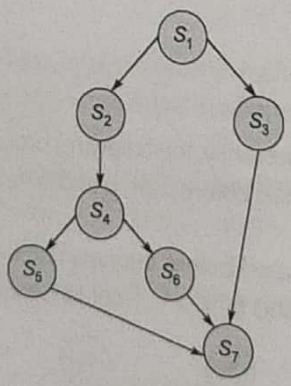
S4 can be executed after S2 completes.

S5 and S6 can be executed after S4 completes.

S7 can be executed only after S5, S6, and S3 complete.

Draw a precedence graph.

Solution:



Concurrency conditions for read and write:

Read set of Si : R(Si). It is the set of all variables whose values are referenced in statement Si during the execution.

Write set of Si : W(Si). It is the set of all variables whose values are changed (written) by the execution of statement Si.

Example-5.2 Let S1 : $a := x + y$ and S2 : $b := z + 1$. Check whether two statements S1 and S2 satisfies Bernstein's conditions.

Solution:

$R(S1) = \{x, y\}$

$R(S2) = \{z\}$

$W(S1) = \{a\}$

$W(S2) = \{b\}$

(*i*) $R(S1) \cap W(S2) = \{x, y\} \cap \{b\} = \{\}$

(*ii*) $W(S1) \cap R(S2) = \{a\} \cap \{z\} = \{\}$

(*iii*) $W(S1) \cap W(S2) = \{a\} \cap \{b\} = \{\}$

Therefore S1 and S2 can be executed concurrently (satisfy the Bernstein's conditions).

Example-5.3 Let S1 : $a := x + y$ and S2 : $b := z + 1$ and S3: $c := a - b$. Check whether two statements S1 and S2 satisfies Bernstein's conditions.

Solution:

$R(S2) = \{z\}$

$R(S3) = \{a, b\}$

$W(S2) = \{b\}$

$W(S3) = \{w\}$

(*i*) $R(S2) \cap W(S3) = \{z\} \cap \{w\} = \{\}$

(*ii*) $W(S2) \cap R(S3) = \{b\} \cap \{a, b\} = \{b\}$

(*iii*) $W(S2) \cap W(S3) = \{b\} \cap \{w\} = \{\}$

S2 cannot be executed concurrently with S3, since $W(S2) \cap R(S3) = \{b\}$.

5.3 The Fork and Join Constructs

Fork

- The *fork L* instruction produces two concurrent executions in a program.
- One execution starts at the statement labeled L, while the other is the continuation of the execution at the statement following the fork instruction.
- The *fork* system call assignment has one parameter (label L).

Join

- The *join instruction* (atomic) provides the means to recombine two concurrent computations into one.
- The join instruction has parameter (integer count) to specify the number of computations which are to be joined.
- Join decrements the integer by one.
- If the value of the integer after decrement is non-zero, the process terminates. Otherwise the process continues execution with the next statement.

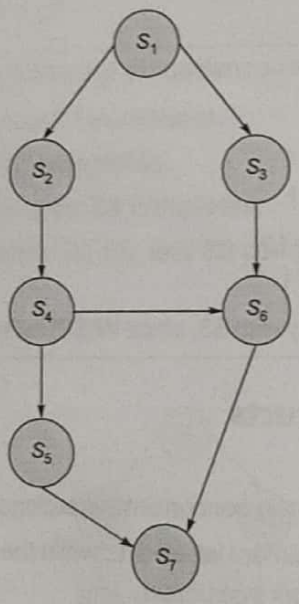
  count : = count - 1;

  IF count ! = 0 THEN quit; where quit is an instruction which results in the termination of the execution.

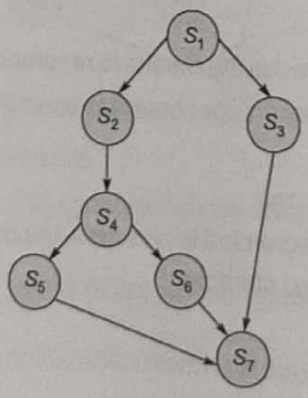
Example-5.4 Construct a precedence graph for the following fork/join program.

```
        S1;
        count1: = 2;
        forkL1;
        S2;
        S4;
        count2: = 2;
        fork L2:
        S5:
        Go to L3;
L1:     S3;
L2:     join count1;
        S6;
L3:     join count2;
        S7;
```

Solution:



Example-5.5 Write a fork/join program for the following precedence graph.



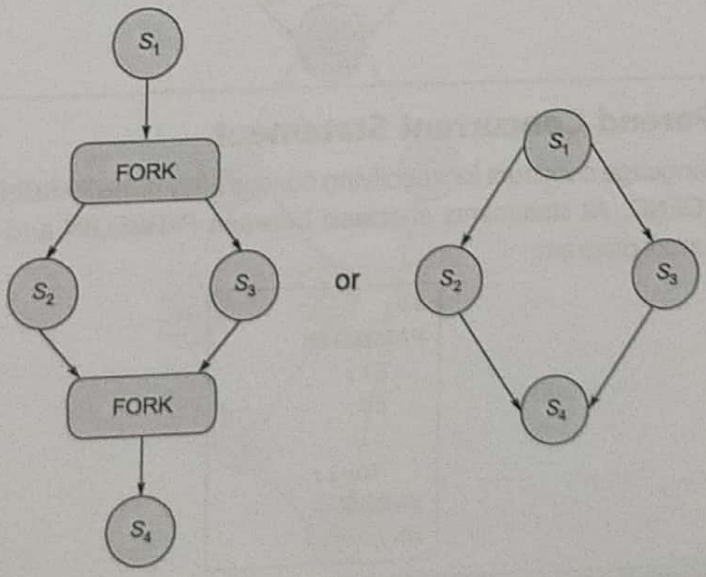
Solution:
After S1 there is a need of fork statement to create a child.
After S4 there is a need of fork statement to create a child.
All processes should join before S7.

```
        S1;
        count: = 3;
        FORK L1;
        S2;
        S4;
        FORK L2;
        S5;
        GOTO L3;
L2:     S6;
        GOTO L3;
L1:     S3;
L3:     JOIN count;
        S7;
```

Example-5.6 Construct a precedence graph for the following concurrency code.

```
        Int count = 2;
        S1;
        FORK (L1);
        S2;
        GOTO L2;
L1:     S3
L2:     JOIN (count);
        S4;
```

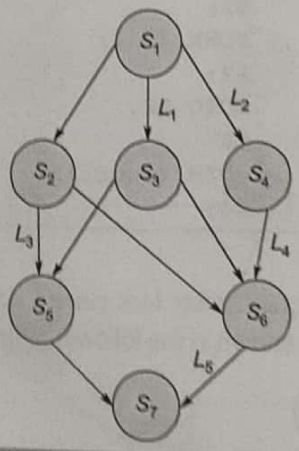
Solution:
After S1 fork statement creates a child. After fork parent continue its flow of execution and child creates new flow execution which is shown in the following Figure.



Example-5.7 Construct a precedence graph for the following concurrency code.

```
        Int count 5 = 2;
        Int count 6 = 3;
        Int count 7 = 2;
        S1;
        FORK (L1);
        FORK (L2);
        S2;
        FORK (L3);
        GOTO L3;
L1 :    S3;
        FORK (L4);
        GOTO L3;
L2 :    S4;
        L4:   JOIN (count 6);
        S6;
L3:     JOIN (count 5);
        S5;
L4:     JOINT (count 7);
        S7;
```

Solution:

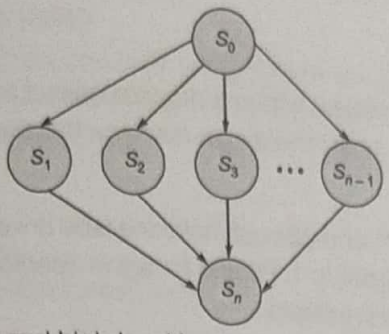


5.4 Parbegin/Parend Concurrent Statement

A higher-level language construct for specifying concurrency is the PARBEGIN/PAREND statement also called as COBEGIN/COEND. All statements enclosed between PARBEGIN and PAREND can be executed concurrently. The concurrent program:

```
S0;
PARBEGIN
  S1;
  S2;
  ...
  Sn-1;
PAREND;
Sn;
```

The above program is equivalent to the following precedence graph.



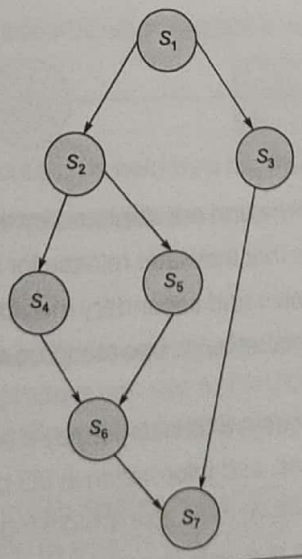
Advantage: It is block-structured high-level language and have the advantages of structured control statements (add other mechanisms such as semaphores).

Disadvantages: The concurrent statement (PARBEGIN/PAREND) is not powerful enough to model all possible precedence graph. In terms of modelling precedence graphs, the FORK/JOIN construct is more powerful than the concurrent statement.

Example - 5.8 Construct the precedence graph for the following cobegin/coend program.

```
begin
S1;
      cobegin
      S3;
      begin
      S2;
           cobegin
           S4;
           S5;
           Coend;
           S6;
      end;
      coend;
S7;
end;
```

Solution:



5.5 Deadlock

A deadlock is a condition in a system where a process cannot proceed because it needs to obtain a resource held by another process but it itself is holding a resource that the other process needs.

Examples of Deadlock

1. System has 2 tape drives. P1 and P2 each hold one tape drive and each needs another one.
2. Semaphores *A* and *B*, initialized to 1. P0 and P1 are in deadlock as follows:
   (a) P0 executes wait(A) and preempts.
   (b) P1 executes wait(B).
   (c) Now P0 and P1 enters in deadlock.

P0	P1
wait (A); wait (B);	wait(B) wait(A)

3. Assume the space is available for allocation of 200 K bytes, and the following sequence of events occur.

P0	P1
... Request 80KB; ... Request 60KB;	... Request 70KB; ... Request 80KB;

Deadlock occurs if both processes progress to their second request.

5.5.1 Resources

System Model

- Resource types R_1, R_2,..., R_m CPU cycles, memory space, I/O devices
- Each resource type R_i has W_i instances.
- Each process utilizes a resource as follows:
  (*i*) request
  (*ii*) use
  (*iii*) release

Types of Resources

1. **Reusable Resources:**
   - Used by one process at a time and not depleted by that use
   - Processes obtain resources that they later release for reuse by other processes
   - Processors, I/O channels, main and secondary memory, files, databases, and semaphores
   - Deadlock occurs if each process holds one resource and requests the other.
2. **Consumable Resources:**
   - Created (produced) and destroyed (consumed) by a process
   - Interrupts, signals, messages, and information in I/O buffers
   - Deadlock may occur if a Receive message is blocking
   - May take a rare combination of events to cause deadlock

5.5.2 Resource Allocation Graph (RAG)

Deadlocks can be described more precisely using Resource allocation graphs. RAG is directed graph which consists of a set of vertices V and a set of edges E. V is partitioned into set of processes (P) and set of resources (R):

$P = \{P_1, P_2, \ldots P_n\}$

$R = \{R_1, R_2, \ldots R_m\}$

E is partitioned into two types:

(a) Request edge: directed edge from P_i to R_j

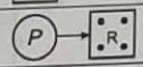
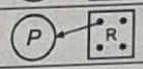
$P_i \rightarrow R_j$

(b) Assignment edge:

(c) Directed edge from R_j to P_i

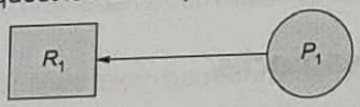
$R_j \rightarrow P_i$

5.5.3 Symbols and Representation

Process P	
Multi instance resource R (R with 4 instances)	
P_i requests a resource	
P_i is holding an instance	

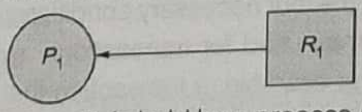
5.5.4 Sequence of Events Required to Use a Resource

1. **Request a resource:**
   (a) Request is made through a system call
   (b) Process must wait if request is denied (Process may be blocked)



Request edge (A resource R_1 is requested by a process P_1)

2. **Use the resource:**
   (a) The process can use the resource when request is successful.



   (b) Assignment edge(A resource R_1 is held by a process P_1)

3. **Release the resource:** The process releases the resource through a system call.

5.6 Conditions for a Deadlock

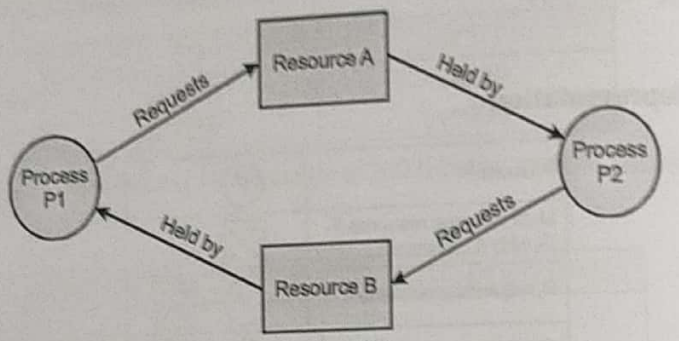
Four conditions must be simultaneously present for a deadlock to occur:

- **Mutual Exclusion:** Only one process can use a resource at a time.
- **No preemption:** No process is allowed to preempt another process forcibly in order to gain a release of a resource from it.
- **Hold and wait:** A process acquires resources piecemeal. While it awaits allocation and assignment of other resources it holds on to its acquired resources.

- Circular wait: Processes waiting for resources from the others form a circular chain. In that chain, each process holds at least one unit resource which in turn is demanded by next process in chain.

Example-5.9 There exists a set {P0, P1,..., P0} of waiting processes such that
P0 is waiting for a resource that is held by P1,
P1 is waiting for a resource that is held by P2,
...
P_{n-1} is waiting for a resource that is held by P_n, and
P0 is waiting for a resource that is held by P0. Draw the resource request graph.

Solution:



Circular Wait

Remember

Is it possible to have a deadlock involving only one single process?
Ans: No. This follows directly from the hold-and-wait condition, that is, with a single process, it is impossible that the hold and wait condition exists.

5.7 Methods of Handling Deadlocks

- **Ignore the deadlock**
- **Prevention:**
  (*i*) Ensure that the system will *never* enter a deadlock state
  (*ii*) Requires negating one of the four necessary conditions
  (*iii*) This approach is sometimes used for transactional systems. Every transaction (process) is assigned a timestamp. For example, a transaction that wants to use a resource that an older process is holding will kill itself and restart later. This ensures that the resource allocation graph always flows from older to younger processes and cycles are impossible. It works for transactional systems since transactions, by their nature, are designed to be restartable (with rollback of system state when they abort). It is not a practical approach for general purpose processes.
- **Avoidance:**
  (*i*) Require careful resource allocation
  (*ii*) Provide advance information to the operating system on which resources a process will request throughout its lifetime.
  (*iii*) The operating system can then decide if the process should be allowed to wait for resources or not and may be able to coordinate the scheduling of processes to ensure that deadlock does not occur.

(iv) This is an impractical approach. A process is likely not to know ahead of time what resources it will be using and coordinating the scheduling of processes based on this information may not always be possible.

- **Detection and recovery:**
  (i) Allow the system to enter a deadlock state and then recover
  (ii) We need some methods to determine whether or not the system has entered into deadlock.
  (iii) We also need algorithms to recover from the deadlock.

5.8 Deadlock Prevention

Design the system in such a way that the possibility of deadlock is excluded from the beginning. Invalidate any of the four necessary conditions and the deadlock will not occur.

- Prevent/deny Mutual Exclusion condition: Use shareable resource: Impossible for practical system.
- Prevent/Deny Hold and Wait condition:
  (a) Pre-allocation: Require processes to request resources before starting. A process never has to wait for what it needs.
  (b) Process must give up all resources and then request all immediately needed

  Problems:
  (a) May not know required resources at start of run
  (b) Low resource utilization. Many resources may be allocated but not used for long time
  (c) Starvation possible. A process may have to wait indefinitely for popular resources.
- Prevent/deny No Preemption condition:
  (a) If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released.
  Preempted resources are added to the list of resources for which the process is waiting.
  Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting.
  (b) The required resource(s) is/are taken back from the process(s) holding it/them and given to the requesting process

  Problems:
  (a) Some resources (e.g. printer, tap drives) cannot be preempted
  (b) May require the job to restart
- Prevent/Deny Circular Wait:
  (a) Order resources (each resource type is assigned a unique integer) and allow process to request for them only in increasing order
  (b) If a process needs several instances of the same resource, it should issue a single request for all of them.
  (c) Alternatively, we can require that whenever a process requests an instance of a resource type it has released all the resources which are assigned a smaller inter value.

  Problem:
  (a) Adding a new resource that upsets ordering requires all code ever written to be modified.
  (b) Resource numbering affects efficiency.
  (c) A process may have to request a resource well before it needs it, just because of the requirement that it must request resources in ascending order.

5.9 Deadlock Avoidance

OS never allocates resources in a way that could lead to a deadlock. Processes must tell OS in advance how many resources they will request. Simplest and most useful model requires that each process declare the maximum number of resources of each type that it may need. The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition.

Resource-allocation state is defined by the number of available and allocated resources, and the maximum demands of the processes.

5.9.1 State of a System

An enumeration of which processes hold, are waiting for or might request which resource

Safe state: There is at least one sequence that does not result in deadlock.

1. No process is deadlocked, and there exists no possible sequence of future request in which deadlock could occur
2. No process is deadlocked and the current state will not lead to a dead lock state

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

- If a system is in safe state then system has no deadlock.
- If a system is in unsafe state then there is possibility of deadlock.
- Deadlock Avoidance ensures that a system will never enter an unsafe state.

Approaches:

- Do not start a process if its maximum requirement might lead to deadlock.
- Do not grant an incremental resource request if this allocation might lead to deadlock.

5.9.2 Deadlock Avoidance with Resource Allocation Graph

- This algorithm can be used if we have only one instance of each resource type.
- In addition to the request and assignment edges, a *claim edge* is also introduced. *Claim edge* $P_i \rightarrow R_j$ indicated that process P_j may request resource R_j in future; represented by a dashed line.
- Claim edge converts to request edge when a process requests a resource. When a resource is released by a process, assignment edge reconverts to a claim edge.
- Resources must be claimed *a priori* in the system. That is, before a process starts executing, all of its claim edges must already appear in the resource allocation graph.
- Suppose that process P_i requests resource R_j. The request can be granted only if converting the request edge if converting the request edge $P_i \rightarrow R_j$ to an assignment edge does not result in a cycle in the resource-allocation graph. That is we use a cycle detection algorithm is used. If no cycle exits, the process P_i will have to wait.

5.9.3 Banker's Algorithm

- It is applicable to the system with multiple instances of resource types.
- Each process must *a priori* claim maximum use.
- When a process requests a resource it may have to wait.
- When a process gets all its resources it must return them in a finite amount of time.
- Banker's algorithm runs each time:
  (a) A process requests resource: Is it safe?
  (b) 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.

Implementation

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

Safety Algorithm

1. Let Work and Finish be vectors of length m and n, respectively.
   Initialize: Work = Available
   Finish $[i]$ = false for $i = 1, 3, \ldots, n$.
2. Find i such that both: Finish $[i]$ = false and $\text{Need}_i \leq$ Work.
   If no such i exists, go to step 4.
3. Work = Work + Allocation_i; Finish $[i]$ = true; go to step 2.
4. If Finish $[i]$ == true for all i, then the system is in a safe state. Otherwise it is unsafe.

5.9.4 Resource Request Algorithm for Process P_i

"Request" is a request vector for process P_i. If Requesti $[j] = k$ then process P_i wants k instances of resource type R_j. Consider the following steps:

1. If $\text{Request}_i \leq \text{Need}_i$ go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim.
2. If $\text{Request}_i \leq$ Available, go to step 3. Otherwise P_i must wait, since resources are not available.
3. Pretend to allocate requested resources to P_i by modifying the state as follows:
   Available = Available = Request_i;
   $\text{Allocation}_i = \text{Allocation}_i + \text{Request}_i$;
   $\text{Need}_i = \text{Need}_i - \text{Request}_i$
4. If it is safe then the resources are allocated to P_i. If it is unsafe then P_i must wait and the old resource-allocation state is restored.

Example-5.10 Consider the following snapshot of a system with five processes

	Allocation	Max	Available
	A B C	A B C	A B C
P0	0 1 0	7 5 3	3 3 2
P1	2 0 0	3 2 2	
P2	3 0 2	9 0 2	
P3	2 1 1	2 2 2	
P4	0 0 2	4 3 3	

Is the system in safe state?

Solution:

Need = Max – Allocation.

Process	Need
	A B C
P0	7 4 3
P1	1 2 2
P2	6 0 0
P3	0 1 1
P4	4 3 1

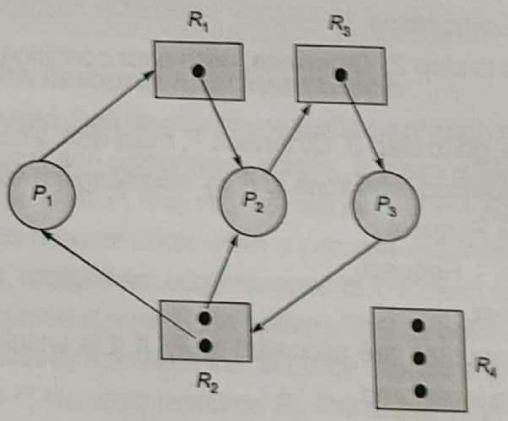
The system is in a safe state since the sequence <P1, P3, P4, P2, P0> satisfies safety criteria.

5.10 Deadlock Detection and Recovery

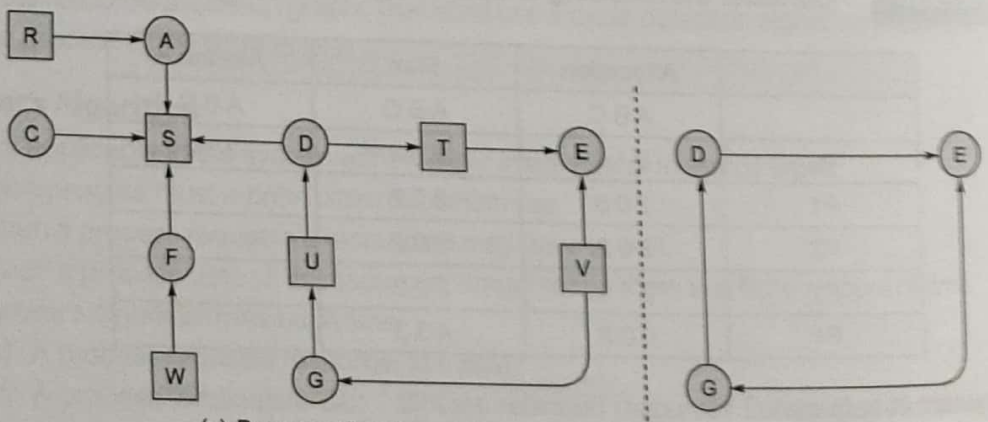
- Allow system to enter deadlock state and then recover it.
- It need a detection algorithm and a recovery algorithm.

5.10.1 How to Detect a Deadlock Using a Resource-Graph?

If each resource type has exactly one instance and the graph has a cycle then a deadlock has occurred. It implies if the cycle involves only a set of resource types, each of which has only a single instance, then the deadlock has occurred. Therefore, a cycle in the graph is both a necessary and sufficient condition for the existence of a deadlock. ***Example***: the following resource graph has deadlock.



Resource Allocation Graph Vs Wait for Graph



(a) Resource allocation graph

(b) Wait for graph

5.10.2 Recovery from Deadlock

1. Process Termination
   - Abort all deadlocked processes. Abort one process at a time until the deadlock cycle is eliminated.
   - In which order should we choose to abort?
     (*i*) Priority of the process.
     (*ii*) How long process has computed, and how much longer to completion.
     (*iii*) Resources the process has used.
     (*iv*) Resources process needs to complete.
     (*v*) How many processes will need to be terminated?
     (*vi*) Is process interactive or batch?
   - Selecting a victim: minimize cost.
   - Rollback: Return to some safe state, restart process for that state.
   - Starvation: Same process may always be picked as victim, include number of rollback in cost factor.
2. Resource Preemption
3. Ostritch algorithm: Ignore deadlock

Remember

How to recover from a deadlock?
Ans: Abort all deadlocked processes and reclaim resources. **OR**
Back up all the deadlocked processes to a predefined checkpoint and restart. **OR**
Gradually abort processes in deadlock one by one until deadlock is resolved. Successfully preempt resources until deadlock is removed.

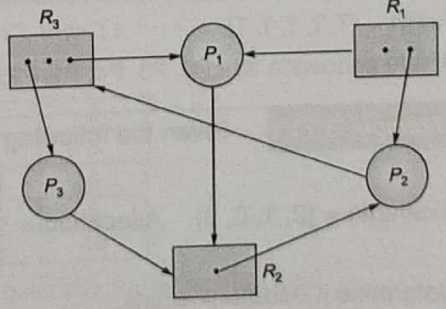
5.10.3 Combined Approach to Deadlock Handling

- Combine the three basic approaches
  (a) Prevention
  (b) Avoidance
  (c) Detection
- Allowing the use of the optimal approach for each of resources in the system.
- Partition resources into hierarchically ordered classes.
- Use most appropriate technique for handling deadlocks within each class.

Example-5.11 A system has three processes P_1, P_2, P_3 and three resources R_1, R_2, R_3. There are two instances of R_1. There is one instance of R_2. There are three instances of R_3. P_1 holds an R_1 and an R_3 and is requesting an R_2. P_2 holds an R_1 and an R_2 and is requesting an R_3. P_3 hold an R_3 and is requesting an R_2. Draw the resource allocation graph for this situation. Does a deadlock exist?

Solution:

Above processes can be satisfied its need. P_2 can hold one instance of R_3 hence P_2 can release all its resources. Therefore deadlock does not exist.



Example-5.12 Given the following system state that defines how 4 types of resources are allocated to 5 running processes.

Available = {2, 1, 0, 0 }

$$\text{Allocation} = \begin{array}{c} \\ P0 \\ P1 \\ P2 \\ P3 \\ P4 \end{array}\begin{array}{c} \begin{matrix} r0 & r1 & r2 & r3 \end{matrix} \\ \begin{bmatrix} 0 & 0 & 1 & 2 \\ 2 & 0 & 0 & 0 \\ 0 & 0 & 3 & 4 \\ 2 & 3 & 5 & 4 \\ 0 & 3 & 3 & 2 \end{bmatrix} \end{array} \qquad \text{Max} = \begin{bmatrix} 0 & 0 & 1 & 2 \\ 2 & 7 & 5 & 0 \\ 6 & 6 & 5 & 6 \\ 4 & 3 & 5 & 6 \\ 0 & 6 & 5 & 2 \end{bmatrix}$$

(a) Compute the need matrix.

(b) Is the system in a safe state?

Solution:

(a) $\text{Need} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 7 & 5 & 0 \\ 6 & 6 & 2 & 2 \\ 2 & 0 & 0 & 2 \\ 0 & 3 & 2 & 0 \end{bmatrix}$

(b) Initialize: Work = {2, 1, 0, 0}, Finish = {F, F, F, F,F}.

Select P0 that does not need any additional resources

Work = {2,1,0,0} + {0,0,1,2} = {2, 1, 1, 2 }

Finish = {T,F,F,F,F}

Select P3 that can request up to {2,0,0,2}

Work = {2, 1, 1, 2} + {2, 3, 5, 4} = {4, 4, 6, 6}

Finish = {T, F, F, T, F}

Select P4 that can request up to {0,3,2,0}

Work = {4, 4, 6, 6} + {0, 3, 3, 2} = {4,7,9,8}

Finish = {T, F, F, T, T}

Select P1 that can request up to {0,7,5,0}

Work = {4, 7, 9, 8} + {2, 0, 0, 0} = {6, 7, 9, 8}

Finish = {T, T, F, T, T}

Select P2 that can request up to {6, 6, 2, 2}

Work = {6, 7, 9, 8} + {0, 0, 3, 4} = {6, 7, 12, 12}

Finish = {T, T, T, T, T}

A safe sequence is: <P0, P3, P4, P1, P2>

Example-5.13 Given the following system resource allocation state:

Available = {2, 1, 0, 0}, $\text{Allocation} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{bmatrix}$ and $\text{Request} = \begin{array}{c} P0 \\ P1 \\ P2 \end{array}\begin{bmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{bmatrix}$

Determine if deadlock exists.

Solution:

Using deadlock detection algorithm:

Initialize: Work = {2, 1, 0, 0}

Finish = {F, F, F}

Select P2 for termination, Work contains enough resources to satisfy P2.

Work = {2, 1, 0, 0} + {0,1,2,0} = {2, 2, 2, 0}

Finish = {F, F, T}

Select P1 for termination, Work contains enough resources to satisfy P1.

Work = {2, 2, 2, 0} + {2,0,0,1} = {4, 2, 2, 1}

Finish = {T, F, T}

Select P0 for termination, Work contains enough resources to satisfy P0.

Work = {4, 2, 2, 1} + {0,0,1,0} = {4, 2, 3, 1}

Finish = {T, T, T}

A safe sequence has been found <P2, P1, P0>, no deadlock exists.

Example-5.14 Suppose that we have the following resources: A, B, C and threads T1, T2, T3, T4. The total number of each resource is:

Total		
A	B	C
12	9	12

Assume that the processes have the following maximum requirements and current allocations:

Thread ID	Current Allocation			Maximum		
	A	B	C	A	B	C
T1	2	1	3	4	3	4
T2	1	2	3	5	3	3
T3	5	4	3	6	4	3
T4	2	1	2	4	1	2

Is the system potentially deadlocked (i.e., unsafe)?

Solution:

Available		
A	B	C
2	1	1

Need			
Thread	A	B	C
T1	2	2	1
T2	4	1	0
T3	1	0	0
T4	2	0	0

No. running the Banker's Algorithm, we see that we can given T4 all the resources it might need and run it to completion. After T4, we do the same for T3, then T2, then T1.

Available after running:

Thread	A	B	C
T1	4	2	3
T2	9	6	6
T3	10	8	9
T4	12	9	12

Many other orderings are possible (T3 or T4 first, then basically any order after that)

Example-5.15 Consider a system that starts with a total of 150 units of memory. Which is then allocated to three processes as shown in the following table of processes, their maximum resource requirements, and their current allocations:

Process	Max Demand	Currently Holds
P1	70	45
P2	60	40
P3	60	15
P4	60	

(a) A fourth process arrives, with a maximum memory need of 60 and an initial request for 25 units.

(b) Using the original table above, a fourth process arrives, with a maximum memory need of 60 and an initial requests for 35 units.

Determine whether it would be safe to grant each of the above requests. If YES, give an execution order that could be guaranteed possible. If NO, show the resulting allocation table.

Solution:

(a) YES, safe 25 units will be left after the allocation for P4, so there is sufficient memory to guarantee the termination of either P1 or P2. After that, the remaining three jobs can be completed in any order.

(b) NO, unsafe. 15 units will be left after the allocation for P4, so there is insufficient memory to guarantee the termination of any process.

Example-5.16 Consider the following snapshot of a system.

	Allocation				Max				Need				Available			
	U	V	W	X	U	V	W	X	U	V	W	X	U	V	W	X
A	0	0	1	2	0	0	1	2	0	0	0	0	1	5	2	0
B	1	0	0	0	1	7	5	0	0	7	5	0				
C	1	3	5	4	2	3	5	6	1	0	0	2				
D	0	6	3	2	0	6	5	2	0	0	2	0				
E	0	0	1	4	0	6	5	6	0	6	4	2				

Is the system in a safe state?

Solution:

The following shows the steps to find a safe sequence.

Available = [1, 5, 2, 0] and is greater than A's Need = [0, 0, 0, 0]

After A completes, Available = [1, 5, 2, 0] + [0, 0, 1, 2] = [1, 5, 3, 2].

Available = [1, 5, 3, 2] is greater than C's Need = [1, 0, 0, 2]

After C completes, Available = [1, 5, 3, 2] + [1, 3, 5, 4] = [2, 8, 8, 6].

Available = [2, 8, 8, 6] is greater than B's Need = [0, 7, 5, 0],

After B completes, Available = [2, 8, 8, 6] + [1, 0, 0, 0] = [3, 8, 8, 6].

Available = [3, 8, 8, 6] is greater than D's Need = [0, 0, 2, 0],

After D completes, Available = [3, 8, 8, 6] + [0, 6, 3, 2] = [3, 14, 11, 8].

Available = [3, 14, 11, 8] is greater than E's Need = [0, 6, 4, 2], we can run E.

Therefore, if the five processes are run in the order of A, C, B, D and E, all of them can finish and the system is safe. < A,C,B,D,E > is a safe sequence.

Note that there are other safe sequences (e.g.,< A,D,C,B,E >).

Example-5.17 Consider the following snapshot of a system:

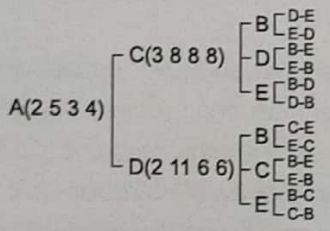
Process	Currently Allocated				Maximum				Needed			
	U	V	W	X	U	V	W	X	U	V	W	X
A	0	0	1	2	0	0	1	2	0	0	0	0
B	1	0	0	0	1	7	5	0	0	7	5	0
C	1	3	5	4	2	3	5	6	1	0	0	2
D	0	6	3	2	0	6	5	2	0	0	2	0
E	0	0	1	4	0	6	5	6	0	6	4	2

Given that currently available resources in the system as: (U, V, W, X) = (1, 5, 2, 2). How many total safe sequences are possible in the above system with 5 processes?

Solution:

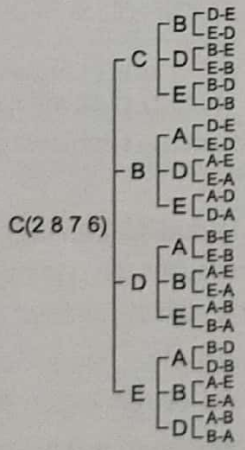
Process	Currently Allocated				Maximum				Available				Needed			
	U	V	W	X	U	V	W	X	1	5	2	2	U	V	W	X
A	0	0	1	2	0	0	1	2					0	0	0	0
B	1	0	0	0	1	7	5	0					0	7	5	0
C	1	3	5	4	2	3	5	6					1	0	0	2
D	0	6	3	2	0	6	5	2					0	0	2	0
E	0	0	1	4	0	6	5	6					0	6	4	2

Number of possibilities when resource is served first for process 'A'.



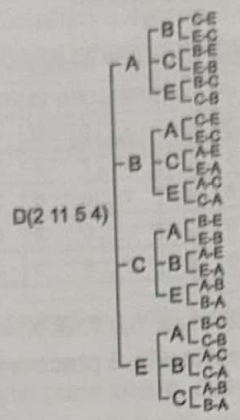
By doing 'A' first we have 12 possibilities to satisfy the need of all processes in order to get the safe sequence.

Number of possibilities when resource is served first for process 'C'.



By doing 'C' first we have 24 possibilities to satisfy the need of all processes in order to get the safe sequence.

Number of possibilities when resource is served first for process 'D'.



Total number of safe sequences are 12 + 24 + 24 = 60.

Summary

- **Starvation:** It implies that a thread cannot make progress because other threads are using resources it needs. Starvation can be recovered from if, for example, the other processes finish.
- **Deadlock:** A problem occurring when the resources needed by some jobs to finish execution are held by other jobs, which, in turn, are waiting for other resources to become available.
- Four conditions have to be met for a deadlock to occur in a system: (1) *Mutual exclusion*: A resource can be held by at most one process. (2) *Hold and wait:* processes that already hold resources can wait for another resource. (3) *Non-preemption*: a resource, once granted, cannot be taken away. (4) *Circular wait*: two or more processes are waiting for resources held by one of the other processes.
- **Deadlock Prevention:** Eliminate one of the factors will create deadlock. Ensuring that at least one of the four conditions for deadlock never holds (mutual exclusion, hold and wait, circular wait, no preemption)
- **Deadlock Avoidance:** The dynamic strategy of deadlock avoidance that attempts to ensure that resources are never allocated in such a way as to place a system in an unsafe state.
- **Deadlock Detection and recovery:** Periodically analyze the state of a system and preempt (destroy/kill) one or more processes if a deadlock has occurred. Allowing deadlock to occur, and then recovering from the deadlock.
- **Safe state:** The situation in which the system has enough available resources to guarantee the completion of at least one job running on the system.
- **Unsafe state:** A situation in which the system has too few available resources to guarantee the completion of at least one job running on the system. It can lead to deadlock.

Student's Assignment

Q.1 Which of the following is not used for deadlock handling?
(a) Deadlock prevention
(b) Deadlock avoidance
(c) Deadlock detection and recovery
(d) None of these

Q.2 Consider a system having 7 resources of same type. Each process may need 3 resources. How many (maximum) processes could be in the system to be deadlock free?
(a) 1 (b) 2
(c) 3 (d) 4

Q.3 Consider a system with 4 processes and 3 resources. Total resources in the system given as: $R_1 = 12$, $R_2 = 9$, $R_3 = 12$. Assume that the system has following information.

Process	Currently Allocated			Maximum		
	R_1	R_2	R_3	R_1	R_2	R_3
P_1	2	1	3	4	3	4
P_2	1	2	3	5	3	3
P_3	5	4	3	6	4	3
P_4	2	1	2	4	1	2

Which of the following is correct statement when above system runs the deadlock avoidance algorithm?
(a) Exactly one safe sequence is possible
(b) More than one safe sequence is possibel
(c) Atmost one safe sequence is possible
(d) No safe sequence exist in the system

Q.4 Consider the following table with 4 processes,

Process	Maximum Demand	Currently holds
P_1	70	45
P_2	60	40
P_3	60	X
P_4	40	Y

If a system has total of 150 units of resource. Identify which of the following values of X and Y the above system will be in safe state?
(a) X = 40, Y = 20 (b) X = 50, Y = 10
(c) X = 30, Y = 20 (d) X = 20, Y = 30

Q.5 Which of the following conditions is not involved in Preventing the deadlock?
(a) No "mutual exclusion"
(b) No "hold and wait"
(c) No preemption
(d) No circular wait

Q.6 Consider the following vector and matrices for 3 processes and 4 resources. Resources available vector (RAV) = (R_1, R_2, R_3, R_4) = (2, 0, 1, 0)

$$\text{Current Allocation Matrix} = \begin{array}{c} \\ P_1 \\ P_2 \\ P_3 \end{array} \begin{array}{c} \begin{array}{cccc} R_1 & R_2 & R_3 & R_4 \end{array} \\ \begin{bmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 0 \\ 0 & 1 & 2 & 0 \end{bmatrix} \end{array}$$

$$\text{Resource Request Matrix} = \begin{array}{c} \\ P_1 \\ P_2 \\ P_3 \end{array} \begin{array}{c} \begin{array}{cccc} R_1 & R_2 & R_3 & R_4 \end{array} \\ \begin{bmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{bmatrix} \end{array}$$

Resource available vector (RV) shows the number of units available currently for each resource in the system. CAM shows each process allocation for every resource type. RRM shows the each process request. Which of the following is correct safe sequence for the above system?
(a) $<P_1, P_2, P_3>$
(b) $<P_3, P_2, P_1>$
(c) $<P_2, P_3, P_1>$
(d) None of these

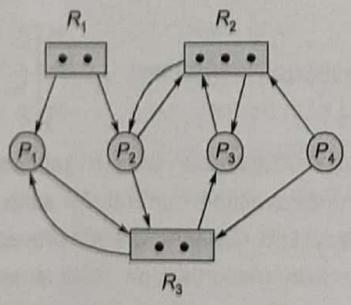
Q.7 Which of the following statement is correct for deadlock avoidance?
(a) Every safe sequence lead to deadlock in the system
(b) Every unsafe system lead to deadlock
(c) Every deadlock has safe sequence
(d) Every deadlock is unsafe

Q.8 Deadlock may be recovered after the detection of deadlock. Which of the following can be used to recover the deadlock?

(a) Resource preemption
(b) Rollback of processes
(c) Killing processes
(d) All of these

Q.9 Identify the correct statement for deadlock condition.
(a) If there are no cycles in resource allocation graph then there is no deadlock.
(b) If there is only one instance per resource type and resource allocation graph has cycle then there is a deadlock.
(c) If there is more than one instance for some resource type and resource allocation graph has cycle, there may or may not be a deadlock.
(d) All of these

Q.10 Consider the following resource allocation graph with 4 processes and 3 resources.



Processes are P_1, P_2, P_3 and P_4. Resources are R_1, R_2 and R_3. Find the correct statement for the above system.
(a) Exactly one safe sequence exist
(b) More than one safe sequence exist
(c) No safe sequence exist
(d) None of these

Q.11 Which of the following statements is/are false for deadlock avoidance scheme?
(a) The request for resources is always granted if the resulting state is safe
(b) It requires a prior information of resources
(c) Both (a) and (b)
(d) Neither (a) nor (b)

Q.12 Consider the following snapshot of a system with 5 processes (P_1, P_2, P_3, P_4 and P_5) and 3 sources (R_1, R_2 and R_3).

Process	Current Allocated			Maximum Demand		
	R_1	R_2	R_3	R_1	R_2	R_3
P_1	0	0	1	1	1	1
P_2	0	1	0	1	1	0
P_3	1	2	3	2	2	4
P_4	0	1	1	1	2	1
P_5	1	0	1	3	0	1

If the system has 1 unit of R_1 and 1 unit of R_3 available then how many minimum resource units available for R_2 to guarantee deadlock free?
(a) 0 (b) 1
(c) 2 (d) 3

Q.13 Consider the following requests and grants of four processes in the system as given order.
(*i*) P_1 requests 2 units of R_1
(*ii*) P_2 requests 3 units of R_2
(*iii*) P_3 requests 2 units of R_3
(*iv*) 1 unit of R_1 is granted to P_1
(*v*) 1 unit of R_2 is granted to P_2
(*vi*) 1 unit of R_3 is granted to P_3
(*vii*) P_2 requests 1 units of R_1
(*viii*) P_1 requests 1 units of R_2
(*ix*) P_2 requests 1 units of R_3
(*x*) P_2 requests 1 units of R_3 and 1 unit of R_3 is granted to P_2.

After the above allocations, system has R_1, R_2 and R_3 has 2, 3 and 2 instances available respectively. Which one of the process can execute last in the system? (Granting a request is nothing but allocation of a request)
(a) P_1 (b) P_2
(c) P_3 (d) Any of these

Q.14 Consider the following sequential code which is executed in a multiprogramming mode by assuming that each statement can execute independently to achieve the concurrency. If any statement dependent on other statements then those statements will be executed in the order.

$S_1: a = b + c$
$S_2: x = y + z$
$S_3: y = a + c$
$S_4: q = y + z$

Which of the above statements can execute concurrently at the beginning of execution?

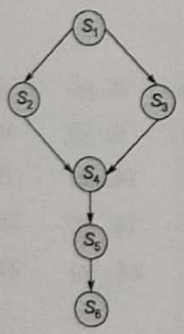
(a) S_3 and S_4
(b) S_2 and S_3
(c) S_1 and S_2
(d) S_2 and S_4

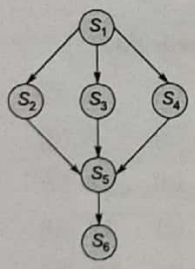
Q.15 Let fork (X) be a function that creates a new process and execution of new process starts from the label X. (i.e., PC register set to X address for new process). Join ( ) will combine all the processes those can execute join ( ).

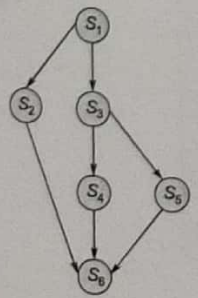
Consider the following code used for concurrency with the statements S_1, S_2, S_3, S_4 and S_5.

```
    S1;
    fork (ABC);
    S2;
    goto DEF;
ABC: S3;
    fork (XYZ);
    S4;
    goto DEF;
XYZ: S5;
DEF: Join ( );
    S6;
```

Which of the following is correct precedence graph for the above code?

(a) 

(b) 

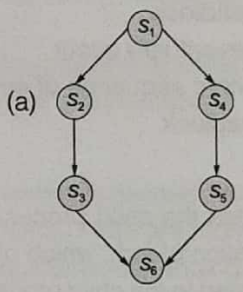
(c) 

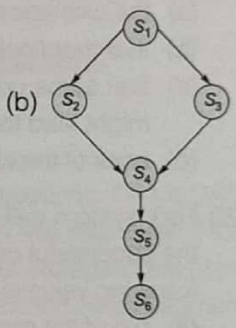
(d) None of these

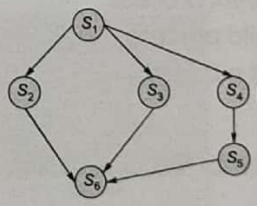
Q.16 Parallelism can be supported by parbegin-parend block. All statements in parbegin-parend block are executed parallelly and all statements within that block are independent. Sequential execution can be supported by begin-end block. All statements in begin-end block are executed sequentially. Consider the following code

```
begin
    S1;
    Parbegin
    S2;
    S3;
        begin S4; S5; end
    Parend
    S6;
end;
```

Which of the following is correct precedence graph for the above code?

(a) 

(b) 

(c) 

(d) None of these

Q.17 Consider the following code with 2-processes.

Process P_1	Process P_2
A;	D;
B;	E;
C;	

If two processes are executing concurrently, which of the following execution order never arise by them?

(a) A, B, C, D, E
(b) D, E, A, B, C
(c) A, D, B, C, E
(d) A, E, B, D, C

Q.18 Consider the following system state:

Proces	Max	Allocated
P1	7	2
P2	6	2
P3	7	4

Total resources are 11. The system will be in a safe state if

(a) process P1 is allocated one additional resource
(b) process P2 is allocated two additional resources
(c) process P3 is allocated three additional resources
(d) process P2 is allocated one additional resource

Q.19 An unsafe state implies

(a) the existence of deadlock
(b) that deadlock will eventually occur
(c) that some unfortunate sequence of events might lead to a deadlock
(d) none of the above

Q.20 Fork system call creates the child process on the successful completion of fork, which of the following value is returned in the child process?

(a) pid of the parent process
(b) pid of the child process
(c) pid of the init process
(d) zero

Q.21 Banker's algorithm for resource allocation deals with

(a) deadlock prevention
(b) deadlock avoidance
(c) deadlock recovery
(d) mutual exclusion

Q.22 Which of the following resources can cause deadlock?

(a) files (b) printers
(c) shared programs (d) all of the above

Q.23 Minimum number of processes required for deadlock is

(a) 0 (b) 1
(c) 2 (d) None of these

Q.24 Consider the following pseudo code fragment.

```
print('hello');
if(fork( )== 0)
print('world');
```

Which of the following statement best explains the outcome when the code is executed?

(a) prints the word 'hello' only
(b) prints the words 'hello' and 'world' in any order
(c) prints the words 'hello' followed by 'world' in that order
(d) prints the words 'hello' followed by two words of 'world' in that order

Q.25 The time complexity of Banker's algorithm to avoid deadlock having n processes and m resources is

(a) $O(m \times n)$ (b) $O(m + n)$
(c) $O(m^2 \times n)$ (d) $O(n^2 \times m)$

Answer Key:

1. (d)	**2.** (c)	**3.** (b)	**4.** (d)	**5.** (c)
6. (d)	**7.** (d)	**8.** (d)	**9.** (d)	**10.** (b)
11. (d)	**12.** (a)	**13.** (d)	**14.** (c)	**15.** (c)
16. (c)	**17.** (d)	**18.** (c)	**19.** (c)	**20.** (d)
21. (b)	**22.** (b)	**23.** (c)	**24.** (c)	**25.** (d)

CHAPTER 06

Memory Management

6.1 Introduction

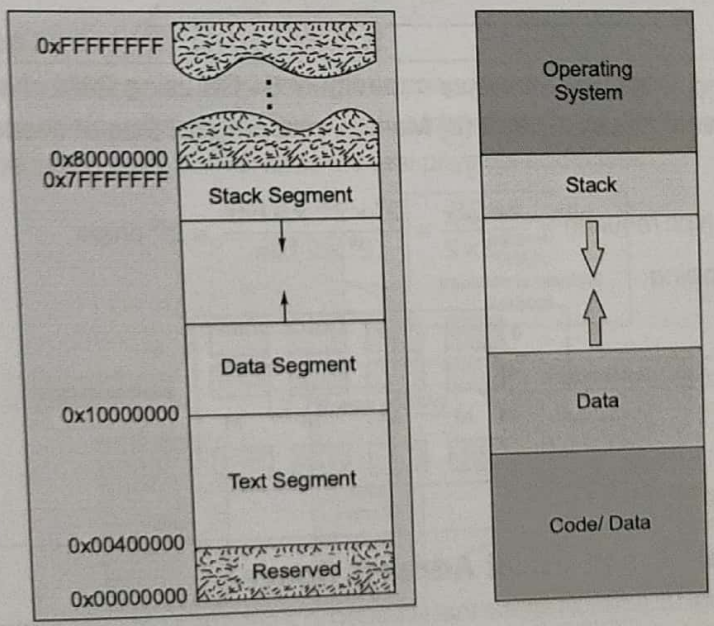
In a multiprogramming computer the operating system resides in part of memory and the rest is used by multiple processes. The task of subdividing the memory (not used by the operating system) among the various processes is called memory management.

Goal: The efficient utilization of memory by minimizing internal and external fragmentation.

Functionality: Allocating and deallocating the memory to the processes.

Memory Layout

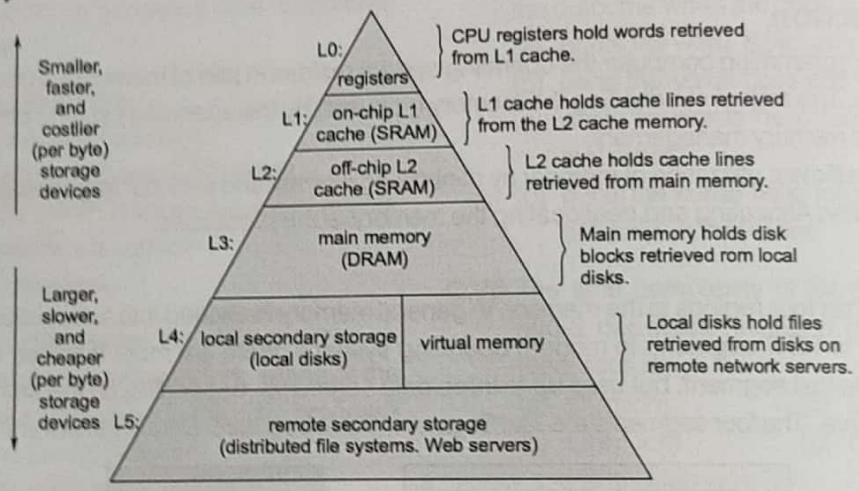
A process has four regions in the memory. In general memory is divided into two sections. One is DATA segment and other is TEXT segment. In modern operating systems, there are more than two segments. Most systems use a single text segment, but uses up to three data segments, depending on the *storage class* of the data being stored there. The four segments are *Text* Segment, *Stack* Segment, *Data* Segment and *Heap* Segment.



- **Code/Text Segment:** This segment contains the machine instructions of the application (program) being executed. It is read-only, so that programs cannot modify their code during execution. It allows sharing among a number of processes that all execute the same program.
- **Data Segment:** The *data segment* contains the *static* data of the program, i.e. the variables that exist throughout program execution. Global variables and static variables of the program are stored in this segment.
- **Heap:** The *heap segment* is a pool of memory used for dynamically allocated memory.
  Example: malloc() in C , *new* in C++ and Java, brk() in UNIX, etc.
- **Stack:** The *stack segment* contains the system stack, which is used as temporary storage. It stores the execution frames (activation record) of functions called by the program.

NOTE: The size of the text and data segments are known as soon as compilation/assembly is completed. (Static memory). The stack and heap segments can grow and shrink during program execution. (Dynamic memory)

Memory Hierarchy

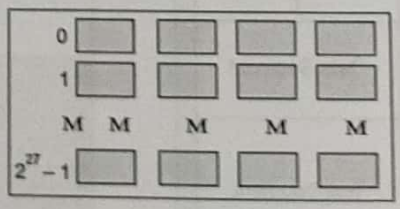


***Example*-6.1** Organize a memory capacity of 64 GB using RAM chips of size 512 x 512 bits than find. (a) Number of chips required (b) Memory mapping (c) Size of decoder required.

Solution:

(a) Number of chips required $= \frac{64\text{ GB}}{512\times 2} = \frac{2^6\times 2^{30}\times 8\text{ bits}}{2^9\times 2\text{ bits}} = 2^{29}$ chips

(b) Memory mapping:



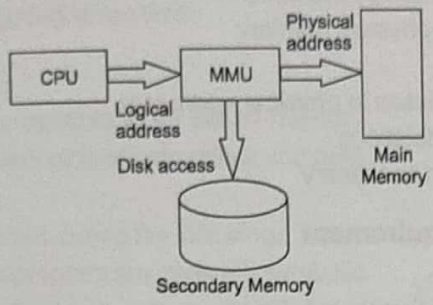
(c) Decoder 27 to 2^{27}

6.2 Logical (Virtual) Vs Physical Address Space

The concept of a logical address space that is bound to a separate physical address space is central to proper memory management.

- **Logical address:** Generated by the CPU; also referred to as virtual address.
- **Physical address:** Address seen by the memory unit.

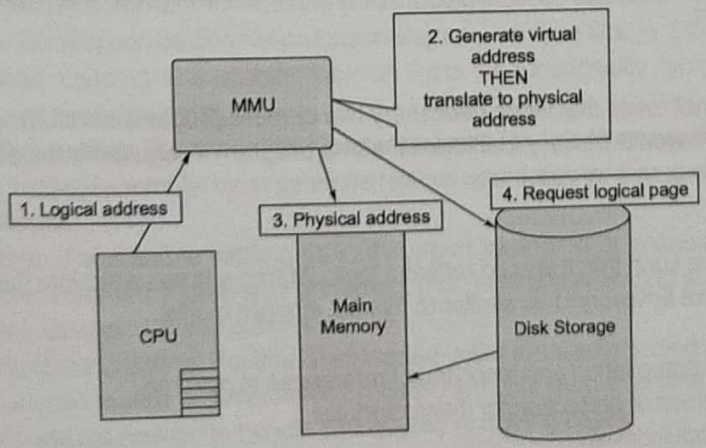
Logical and physical addresses are the same in compile-time and load-time address-binding schemes; logical (virtual) and physical addresses differ in execution-time address-binding scheme.



Logical Address	Physical Address
• Logical address is the address generated by the CPU. • Application programs that are running on the computer do not see the physical addresses. They always work using the logical address. • The logical address space is the set of logical addresses generated by a program. • Logical addresses need to be mapped to physical addresses before they are used and this mapping is handled using a hardware device called the Memory Management Unit (MMU). • Address binding (i.e. allocating instructions and data in to memory addresses) can happen in three different times. • Address binding can happen in compile time if the actual memory locations are known in advance and this would generated the absolute code in compile time.	• Physical address or the real address is the address seen by the memory unit and it allows the data bus to access a particular memory cell in the main memory. • Logical addresses generated by the CPU when executing a program are mapped in to physical address using the MMU. For *example*, using the simplest mapping scheme, which adds the relocation register (assume that the value in the register is y) value to the logical address, a logical address range from 0 to x would map to a physical address range y to x + y this is also called the physical address space of that program. • All the logical addresses need to be mapped in to physical addresses before they can be used.

6.3 Memory-Management Unit (MMU)

Hardware device that maps virtual to physical address. In MMU scheme, the value in the relocation register is added to every address generated by a user process at the time it is sent to memory. The user program deals with logical addresses; it never sees the real physical addresses.



6.3.1 Operating System Vs MMU

- Operating system:
  (a) save/restore MMU state on context switches
  (b) handle exceptions raised by the MMU
  (c) manage and allocate physical memory
- MMU (hardware):
  (a) translate virtual addresses to physical addresses
  (b) check for protection violations
  (c) raise exceptions when necessary

6.3.2 Memory Management Requirement

Relocation

It is necessary to create space/vacate to bring some other process in; the memory management should make sure that the execution of references are maintained.

Relocation: (1) the process of moving a program from one area of memory to another; or (2) the process of adjusting address references in a program, by either software or hardware means, to allow the program to execute correctly when loaded in different sections of memory.

Relocation register: A register that contains the value that must be added to each address referenced in the program so that it will be able to access the correct memory addresses after relocation. Relocation can be done in two ways: Static relocation and dynamic relocation.

- Static Relocation:
  (a) A process executes at a specific (static) address.
  (b) Addresses are assigned at **load time.**
  (c) **Relocation** modifies instructions that address locations in memory.
  (d) Static relocation does not solve protection.
- Dynamic Relocation:
  (a) Addresses can be changed at execution time.
  (b) Facilitates Protection using address translations (with Base & Limit Register).
  (c) Program may be moved during execution (Flexible).

Protection

Data integrity is essential, therefore, each process area should be protected against any undesired intervention. Processes should not be able to reference memory locations in another process without permission. Impossible to check absolute addresses in programs since the program could be relocated, hence must be checked during execution.

Sharing

Protection does not mean that when necessary two or more processes should not be able to access a common area. Because it would be very inefficient if same program if required should not be shared among different processes.

Logical Organization

Division should be such that it should reflect a logic. Memory is just a straight flat portion, construction of programs is typically not envisioned as similar to this flat straight model.

Advantages

- Independent compilation and resolving of references at run time.
- Different degrees of protection for these modules.
- Sharing at module levels

Physical Organization

- Two levels as main memory and secondary memory.
- Available space may not be enough for a process; overlaying.
- At the time of construction of programs it is not possible to predict the amount of space required at run time.

6.3.3 Address Binding

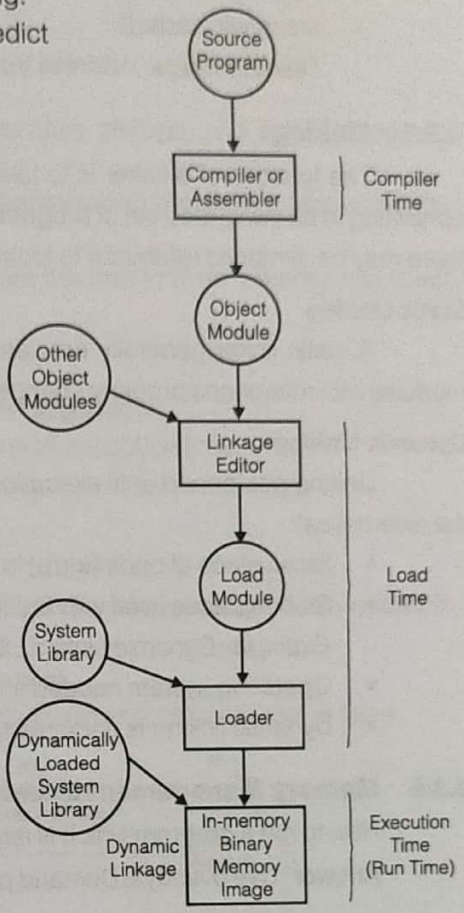
Assignment of instruction or variable to an actual memory address is called as binding. The association of instructions and data to memory addresses.

Addresses may be represented during the following:

- Addresses in the source program are generally symbolic (variable name).
- A compiler/assembler typically binds symbolic addresses to relocatable addresses (in terms of offsets).
- The linkage editor or loader binds relocatable addresses to absolute addresses (physical addresses).

Address binding can occur at any of the following:

- **Compile time:** If memory location known a priori, absolute code can be generated. It must recompile the code if starting location changes.
- **Linkage time:** Program is combined with other modules (resolves all references).
- **Load time:** Program is loaded into memory.
- **Execution time:** Binding delayed until run time if the process can be moved during its execution from one memory segment to another.



6.3.4 Loading

A loader is responsible to place a load module in main memory at some starting address (called loading).

Types of Loading

(a) **Absolute Loading:** A given module/program is always loaded into the same memory location whenever loaded for execution. All references in the load module must be absolute memory addresses (fixed). The address binding can be done at programming time, compile time or assembly time.
Disadvantage: Loading to a specific location limits the functionality, typically required in having multiple processes in the memory.

(b) **Relocatable Loading:** The relocatable loader places a module in any desired location of the main memory. Compiler/Assembler must generate relative addresses for a program. Job of the this loader becomes simple.
Disadvantage: Relocatable loading cannot support swapping of images, as the addresses are bounded absolutely at the initial time of loading. (For multi-programming schemes process images are frequently swapped in and out of memory).

(c) **Dynamic/Runtime Loading:** Routine is not loaded until it is called resulting in better memory-space utilization (unused routine is never loaded, postponed until execution time). It is useful when large amounts of code are needed to handle infrequently occurring cases.

Advantages: No special support from the OS is required implemented through program design. Dynamic loading provides swapping of processes. Better memory-space utilization (unused routines are never loaded).

Disadvantage: Address translation is performed through the hardware.

6.3.5 Linking

The function of a linker is to take as input a collection of object modules and produce a load module consisting of an integrated set of programs and data modules to be passed to the loader. In each object module, there may be symbolic reference to location in other modules. The linking can be done in following two ways.

Static Linking

A static linker generally creates a single load module that is the contiguous joining of all the object modules with references properly changed.

Dynamic Linking

Linking postponed until execution time. All external references are not resolved until the CPU executes the external call.

- Small piece of code (**stub**) is used to locate the appropriate memory-resident library routine.
- Stub replaces itself with the address of the routine, and executes the routine.
  Example: Dynamic Linked Libraries (DLL)
- Operating system needed to check if routine is in processes memory address.
- Dynamic linking is particularly useful for libraries.

6.3.6 Memory Management Issues

How to run a program which is larger than the physical memory?

Answer: Use overlays, Demand paging, or Demand segmentation.

How to move blocked processes onto disk to execute other processes?

Answer: Use swapping.

How to allocate the memory space to the processes?

Answer: Use contiguous or non-contiguous memory allocations.

Is entire memory used in the memory allocation?

Answer: No, It may suffer from internal or external fragmentation.

What is Internal Fragmentation?

Answer: Internal Fragmentation: Allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used.

What is External Fragmentation?

Answer: External Fragmentation: Total memory space exists to satisfy a request, but it is not contiguous.

Overlays

Overlay is a technique to run a program that is bigger than the size of the physical memory by keeping only those instructions and data that are needed at any given time. Divide the program into modules in such a way that not all modules need to be in the memory at the same time.

Advantage: Reduced memory requirements.

Disadvantages:
- Overlap map must be specified by programmer
- Programmer must know memory requirements
- Overlapped modules must be completely disjoint

Swapping

A process can be *swapped* temporarily out of memory to secondary *storage*, and then loaded into memory again to resume execution.

Roll-out/Swap-out: Lower priority process from memory can be swapped out so that higher priority processes can be executed.

Roll-in/Swap-in: Higher priority process can be swapped in from the disk to main memory with lower priority process.

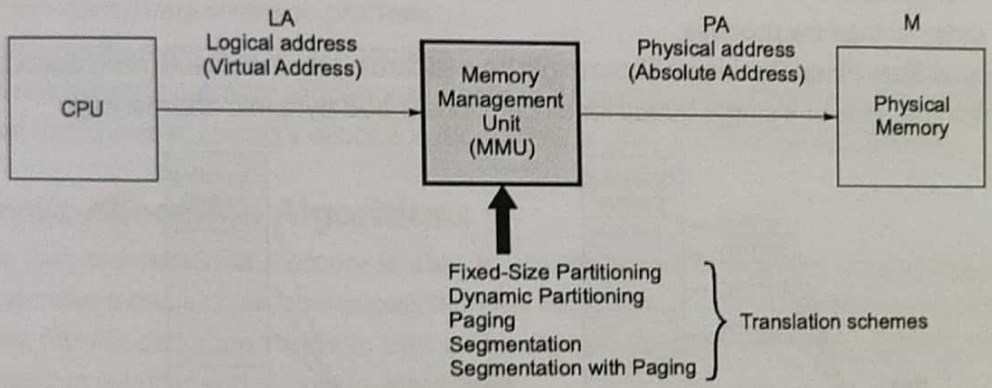
Compaction

Shuffle memory contents to place all free memory together in one large block.
Compaction is possible only if relocation is dynamic, and is done at execution time.
Reduce external fragmentation by compaction.

6.4 Memory Allocation Techniques

A single process is loaded into the memory at a time. Only OS and one user process are loaded in memory.

Advantages: Simple and no special hardware needed to implement.
Disadvantages: Wastage of CPU time, main memory not utilized fully, and program size is fixed.



Partition main memory into a set of non-overlapping memory regions called partitions. Main memory is divided into a set of non-overlapping memory regions called partitions. Memory can be allocated in contiguous or non-contiguous partitions.

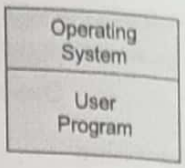
Contiguous Partition

Contiguous Partition categorized into two ways.
1. Single Partitioning (Single User, Single process)
2. Multiple Partitioning (Fixed/Static and Variable/Dynamic)

Non-Contiguous Memory Allocation

Non-Contiguous Partitioning (Static) can be done many ways.
1. Paging
2. Segmentation
3. Segmented paging

6.4.1 Single-partition Allocation



Only one user present and one process is running at any time in the system. Main memory is divided into only two parts as Operating system and user program currently being executed. A single process is loaded into the memory at a time.

Advantages: Simple and no special hardware needed to implement.

Disadvantages: Wastage of CPU time, main memory not utilized fully, and program size is fixed.

6.4.2 Multiple-partition Allocation

If user memory is divided into more than one partition then such partition is called as multiple partition. Multiple partition allocation can be done in the following two ways.

1. **Fixed partitioning (Static):** Size of each partition is static.
2. **Variable partitioning (Dynamic):** Partition done based on process size.

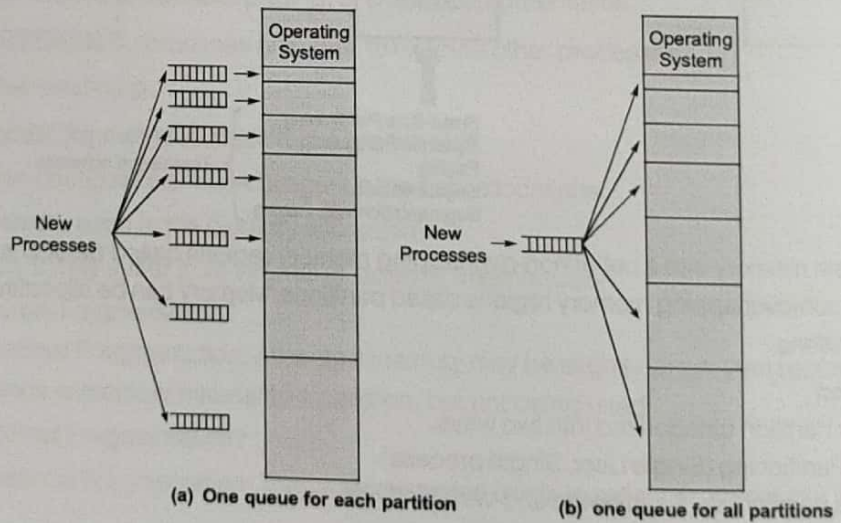
1. Fixed Partitions/Static Partitioning

Memory is divided into fixed-sized partitions. All partitions either equal (same) size or unequal size partitions. One partition is allocated to each active process in the multiprogramming set. In this scheme, the degree of multiprogramming is bounded by the number of partitions.

(a) **Equal Size, Fixed Partitions:**

- Every partition of user memory is having same size.
- If there is an available partition, a process can be loaded into any free partition. Because all partitions are of equal size, it does not matter which partition is used.
- If all partitions are occupied by blocked processes, choose one process to swap out to make room for the new process.

(b) **Unequal Size, Fixed Partitions:** There might be a separate process queue (need absolute addressing) for each partition or a single queue for all partitions (need dynamic addressing).



(a) One queue for each partition (b) one queue for all partitions

- Unequal-size partitions, use of multiple queues:
  (*i*) Assign each process to the smallest partition within which it will fit.
  (*ii*) Separate Queue exists for each partition size.
  (*iii*) It tries to minimize internal fragmentation.
  (*iv*) **Problem:** some queues might be empty while some might be loaded.

- Unequal-size partitions, use of a single queue:
  (i) When its time to load a process into memory, the smallest available partition that will hold the process is selected.
  (ii) Increases the level of multiprogramming at the expense of internal fragmentation.

Disadvantages of Fixed Partitioning:

- It wastes space through internal fragmentation
- Partition size may not large enough for any waiting progress.
- It supports only fixed Number of processes (limits number of active processes)
- It wastes space through internal fragmentation

2. Variable/Dynamic Partitioning

The partitions are created dynamically. When a process arrives, it allocates memory from a hole large enough to accommodate it. Operating system maintains information about allocated partitions and free partitions.

Memory is allocated contiguously to processes until there is no available block of memory large enough. It needs compaction to get all free partitions into one side to get a larger free partition.

Hole: It is a block of available memory (free partition). All holes of various sizes are scattered throughout memory.

Advantages of Dynamic partitioning:

- Memory utilization is generally better for variable-partition schemes.
- There is no internal fragmentation.

Disadvantages of Dynamic partitioning:

- It is more complex to maintain
- It has external fragmentation problem
- It requires the overhead of compaction

How to use dynamic memory allocations effectively to minimize external fragmentation?

Answer: Use dynamic memory allocation algorithms to allocate memory dynamically.

6.5 Dynamic Allocation Algorithms

If more than one region of memory is able to accommodate a segment (or process), the operating system needs to make a decision on how to pick the best segment. Several approaches are possible.

1. **First Fit:** Allocates the first hole that is big enough, starting from the beginning of memory. This algorithm is faster and simple to implement.
2. **Next Fit:** This algorithm is just like First Fit except that it searches from where it left off the previous time (and then wraps around). In practice it exhibits worse performance than First Fit.
3. **Best Fit:** The Best Fit technique searches the list for the hole that best fits the process. It is slower than First Fit since the entire list must be searched. It also produces tiny holes that are often useless.
4. **Worst fit:** The **Worst Fit** algorithm searches for the largest hole on the assumption that the hole remaining after allocation will be big enough to be useful. It also doesn't yield good performance.
5. **Quick Fit:** The Quick Fit algorithm maintains separate lists for common sizes. It's fast but has the drawback that finding neighbours for merging is difficult.

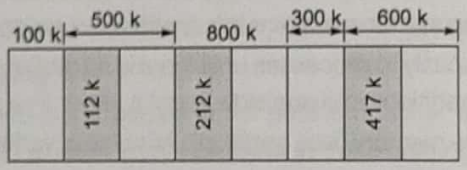
Deciding which placement algorithm is the best depends on the size of the processes and the exact sequence of process swapping that takes place. In general, the First-fit usually results in the best and fastest allocation. Best-fit results in more frequent compaction that others.

How to satisfy a process request of size *n* from a list of free holes?

- First-fit: Allocate the *first* hole that is big enough.
- Best-fit: Allocate the *smallest* hole that is big enough; must search entire list, unless ordered by size. Produces the smallest leftover hole.
- Worst-fit: Allocate the *largest* hole; must also search entire list. Produces the largest leftover hole.

Example-6.2 Given memory partitions in the order below: P_1: 100 k, P_2: 500 k, P_3: 800 k, P_4: 300 k, P_5: 600 k. How would WORST FIT algorithms place processes: 212 k, 417k, 112 k, and 426 k (in order)

Solution:



Principle of worst fit allocation: Allocate the largest hole that is available. We must search the entire list, unless the list in kept ordered by size.

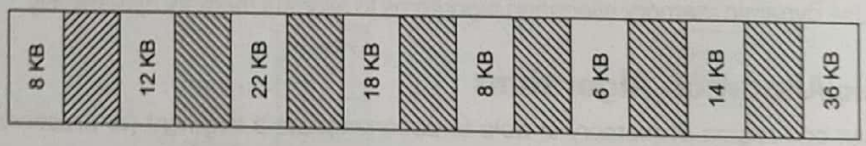
212 k is put in 800 k partition

417 k is put in 600 k partition

112 k is put in 500 k partition

426 k have to wait because there is no partition free that is big enough for 426 k.

Example-6.3 Consider the following memory schema at a point having variable partition.



How many successive requirement of 6 KB will be satisfied by using first fit?

Solution:

19 (1 + 2 + 3 + 3 + 1 + 1 + 2 + 6) successive requests satisfied by using first fit.

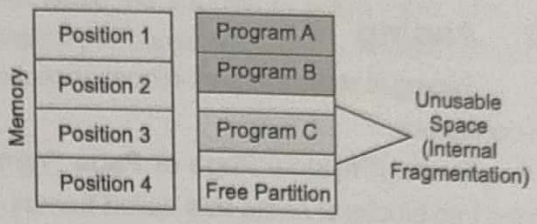
6.6 Internal Fragmentation and External Fragmentation

Both internal fragmentation and external fragmentation are phenomena where memory is wasted. Internal fragmentation occurs in fixed size memory allocation while external fragmentation occurs in dynamic memory allocation. When an allocated partition is occupied by a program that is lesser than the partition, remaining space goes wasted causing internal fragmentation. When enough adjacent space cannot be found after loading and unloading of programs, due to fact that free space is distributed here and there, this causes external fragmentation. Fragmentation can occur in any memory device such as RAM, Hard disk and Flash drives.

6.6.1 Internal Fragmentation

- Internal fragmentation occurs when a fixed size memory allocation technique is used. External fragmentation occurs when a dynamic memory allocation technique is used.

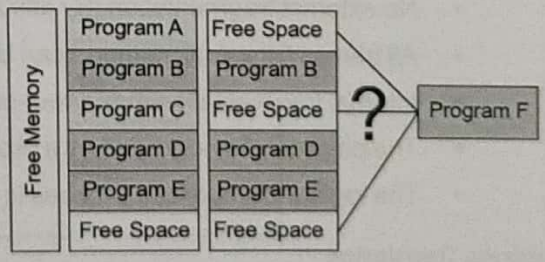
- Internal fragmentation occurs when a fixed size partition is assigned to a program / file with less size than the partition making the rest of the space in that partition unusable. External fragmentation is due to the lack of enough adjacent space after loading and unloading of programs or files for some time because then all free space is distributed here and there.



- Internal fragmentation can be maimed by having partitions of several sizes and assigning a program based on the best fit. However, still internal fragmentation is not fully eliminated.
- Consider the figure above where a fixed sized memory allocation mechanism is being followed. Initially, the memory is empty and the allocator has divided the memory into fixed size **partitions**. Then later three programs named A, B, C have been loaded to the first three partitions while the 4th partition is still free. Program A matches the size of the partition, so there is no wastage in that partition, but Program B and Program C are smaller than the partition size. So in partition 2 and partition 3 there is remaining free space. However, this free space is unusable as the memory allocator only assigns full partitions to programs but not parts of it. This wastage of free space is called internal fragmentation.

6.6.2 External Fragmentation

- External fragmentation can be mined by compaction where the assigned blocks are moved to one side, so that contiguous space is gained. However, this operation takes time and also certain critical assigned areas for example system services cannot be moved safely. We can observe this compaction step done on hard disks when running the disk defragmenter in Windows.
- External fragmentation can be prevented by mechanism such as segmentation and paging. Here a logical contiguous virtual memory space is given while in reality the files/programs are splitted into parts and placed here and there.
- Consider the figure above where memory allocation is done dynamically. In dynamic memory allocation, the allocator allocates only the exact needed size for that program. First memory is completely free. Then the Programs A, B, C, D and E of different sizes are loaded one after the other and they are placed in memory contiguously in that order.



Then later, Program A and Program C closes and they are unloaded from memory. Now there are three free space areas in the memory, but they are not adjacent. Now a large program called Program F is going to be loaded but neither of the free space block of all the free spaces is definitely enough for Program F, but due to the lack of adjacency that space is unusable for Program F. This is called external Fragmentation.

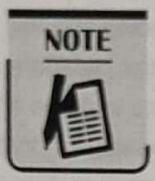
6.7 Paging

Paging is static memory allocation technique that permits the physical address space of a process to be non-contiguous.

Frame (Physical Page or Page Frame): A *frame* is a unit of physical memory. Physical memory is divided into blocks of same size called frames.

Page (Virtual page or Logical page): A *page* is a unit of logical memory of a program. Logical memory is divided into blocks of same size called pages.

When a new process arrives, its size (in pages) is determined. If the process has *n* pages in logical memory then *n* frames must be available in the physical memory.



- All pages are of the same size
- All frames are of the same size
- Size of frame is equal to page size
- Page size is defined by Hardware

To support paging, OS determines:

- The number of pages in the program
- Locates enough empty page frames to facilitate
- Loads all of the pages into memory, pages need not be contiguous.

OS maintains the following tables to support the paging.

- **Job Table:** For each job, it holds the size of the job and the memory location of the Page table.
- **Page Table:** For each page of the process, it holds corresponding frame number with protected information for page.
- **Frame table:** For each frame of physical memory it maintains the information related to the availability (free or busy) and if it is busy which process using it.

Paging Characteristics

- No external fragmentation (By assuming size of main memory is in powers of 2)
- All frames (physical memory) can be used by processes
- Internal fragmentation may occur only on the last page of a process.
- The physical memory used by a process is no longer contiguous (non-contiguous)
- The logical memory of a process is still contiguous.

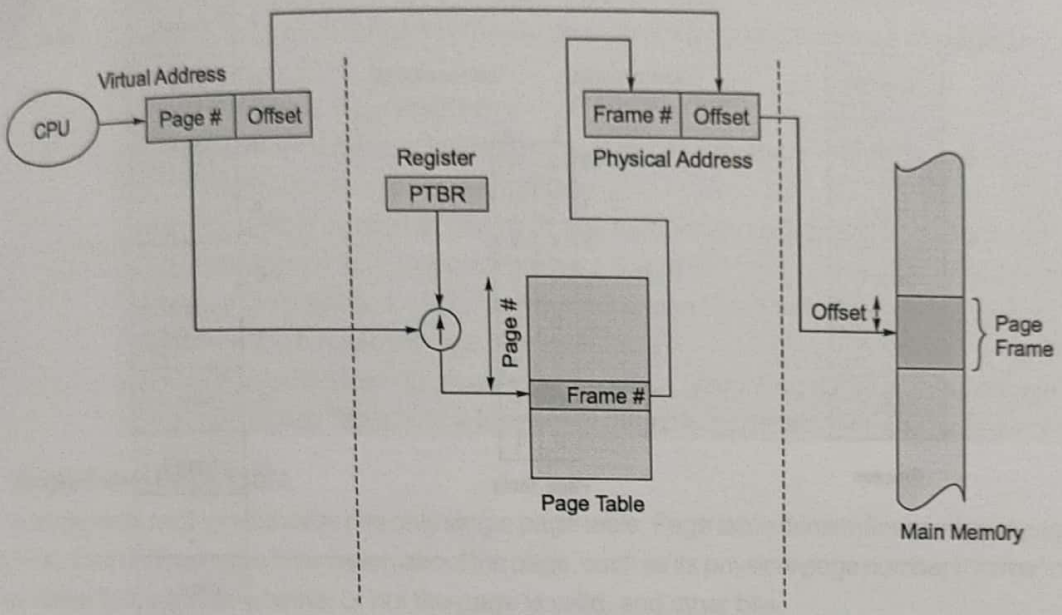
Address Translation

The logical and physical addresses are separated in paging. So it needs address translation mechanism (hardware) to map the logical address into physical address.

- Logical address consisting of a page number and offset.
- Physical address consisting of a frame number and offset.

Page-table base register (PTBR) points to the page table of a process. A processor that supports virtual memory must have a page table base register that is accessible by OS. *Page-table length register* (PTLR) indicates size of the page table.

This address translation from logical to physical address takes the support of page table. Memory address translation takes place at run time. Reading a word from memory involves translating a virtual to physical address.



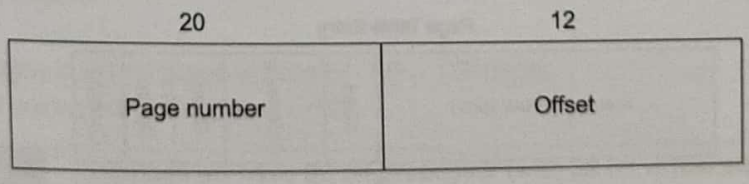
Address translation using Page Table

Example - 6.4 A computer uses 32-bit byte addressing. The computer uses paged virtual memory with 4KB pages. Calculate the number of bits in the page number and offset fields of a logical address.

Solution:

Since there are 4K bytes in a cache block, the offset field must contain 12 bits (2^{12} = 4K). The remaining 20 bits are page number bits.

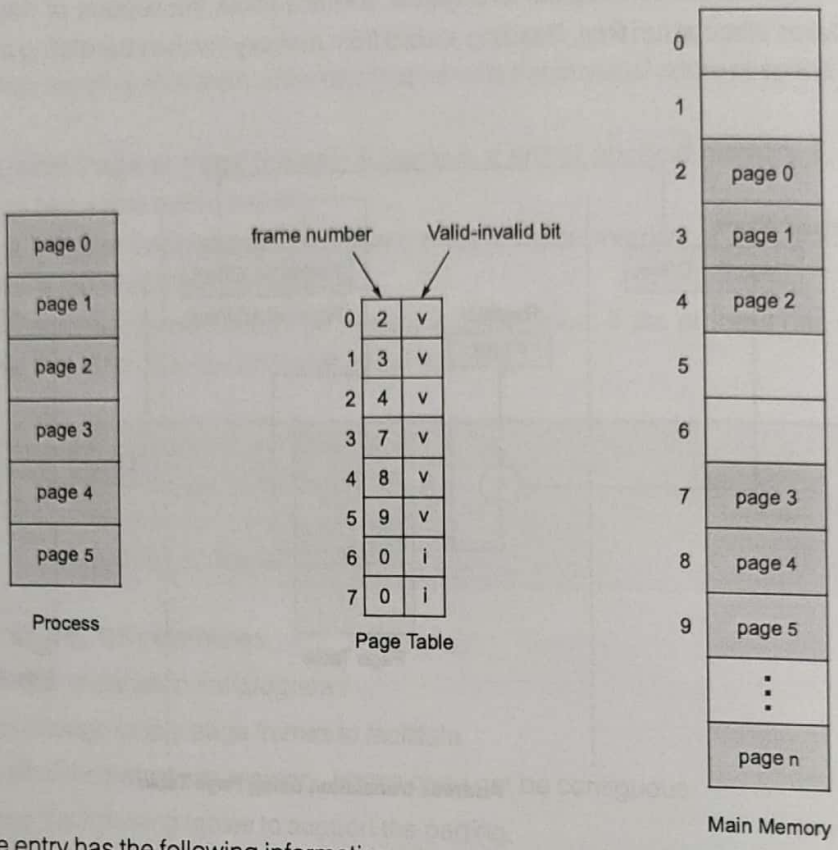
Thus a logical address is decomposed as shown below.



Page Table (PT)

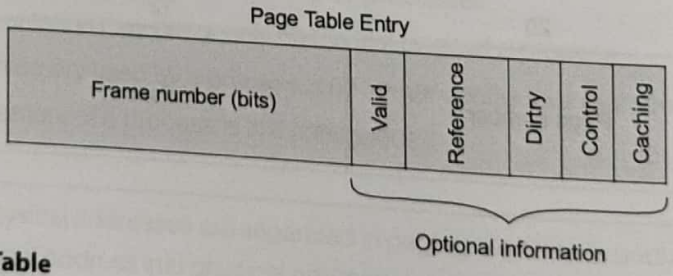
Every new process creates a separate page table. Page table is stored in physical memory.

Page Table Entry (PTE): Page table has page table entries where each page table entry stores a frame number and optional status (Protection) bits. Many of status bits used in virtual memory system.



Page table entry has the following information:

- **Frame number:** Number of bits required depends on number of frames.
  Number of bits for frame = Size of physical memory / Frame size
- **Present/Absent (valid) bit:** It set to 0 if the corresponding page is not in memory. Used to control page faults by OS to support virtual memory.
- **Control bits:** These bits for the protection of the page frame (read, write, etc.)
- **Referenced bit:** It is set to 1 by hardware when the page is accessed.
- **Caching enabled/disabled:** Enables or disables caching of the page.
- **Modified bit (dirty bit):** set to 1 by hardware on write-access to page which is used to avoid writing when swapped out.



Implementation of Page Table

Page table can be implemented using:

- Single level page table stored in Main Memory
- Multi Level page tables stored in Main Memory
- Associative memory (Registers or TLB) and single or multi level page tables
- Inverted page table

- Whenever a process is created paging will be applied on process and page table is created and the base address the page table will be stored in the PCB.
- The Paging is w.r.t. every process and every process will have its own page table.
- The page table of the process will be stored in the main memory.
- There is no external fragmentation in the paging because **size of page and frame is same.**
- The internal fragmentation exists in last page.
- The internal fragmentation is considered as P/2 where P is page size (last).

Q.1 How to know page table entry size?

Ans: Page table entry must contain frame number and optional status information.

Q.2 How many pages in logical address space?

Ans: Number of pages = Logical Address Space Size/Page Size

Q.3 How to compute the size of a page table?

Ans: PageTable Size = Logical Address Space Size × Pagetable entry size/Page size.
(OR) Page Table Size = Number of pages in page table × Page table entry size.

6.7.1 Single-Level Page Table

In single level paging a process has only single page table. Page table contain linear array of page-table entries (PTEs). Each PTE contains information about the page, such as its physical page number ("frame" number) as well as status bits, such as whether or not the page is valid, and other bits.

Address translation (from logical to physical address) must take place on **every** memory reference. Depending on the page size, the page table size can become very large. Larger page table takes time to load.

Example: 32 bit virtual address space, 4K page, the 32 bit address space has 1 million pages. Each process needs its own page table with 1 million entries.

Disadvantage: It is impractical to manage these large page tables as one entity. (Using multilevel page tables, size of page tables can be reduced). The mapping from virtual to physical address is slow due to extra memory reference for page table.

Example-6.5 An OS uses a paging system with 1Kbyte pages. A given process uses a virtual address space of 128Kbytes and is assigned 16Kbytes of physical memory. How many entries does its page table contain?

Solution:

Number of pages in virtual space = 128 KB/1 KB = 128 pages.

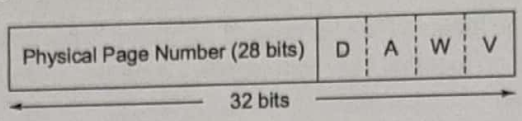
So number of entries in the page table = 128

Example-6.6 Suppose we have 32 bit processor (with 32 bit virtual addresses) and 8 KB pages. Assume that it can address up to 2 TB (terabytes) of DRAM: 1 TB = 1024 GB = $(1024)^2$ MB. Assume that we need 4 permissions bits in each page table entry (PTE) namely Valid (V), Writable (W), Accessed (A), and Dirty (D). Show the format of a PTE, assuming that each page should be able to hold an integer number of PTEs.

Solution:

Since the PTE must have sufficient bits to address all of the physical pages, we must ask how many physical pages there are. 2 TB = $2 \times (1024)^4 = 2 \times 2^{40} = 2^{41}$.

Total number of pages = 2 TB / 8KB = $2^{41}/2^{13} = 2^{28}$. Consequently, we need 28 bits in the PTE for the physical page number. With the 4 other bits, this leads us to a 32 bit PTE (which fits an integral number of times in an 8 KB page (with no wasted bits). Our PTE looks like (bits in any order):



Example-6.7 Assume that you have a small virtual address space of size 64 KB. Further assume that this is a system that uses paging and that each page is of size 8 KB.

(a) How many bits are in a virtual address in this system?

(b) Recall that with paging, a virtual address is usually split into two components: a virtual page number (VPN) and an offset. How many bits are in the VPN?

(c) How many bits are in the offset?

(d) Now assume that the OS is using a linear page table. How many entries does this linear page table contain?

Solution:

(a) 16 (1 KB of address space needs 10 bits, and 64 needs 6; thus 16).

(b) Only eight 8-KB pages in a 64-KB address space.

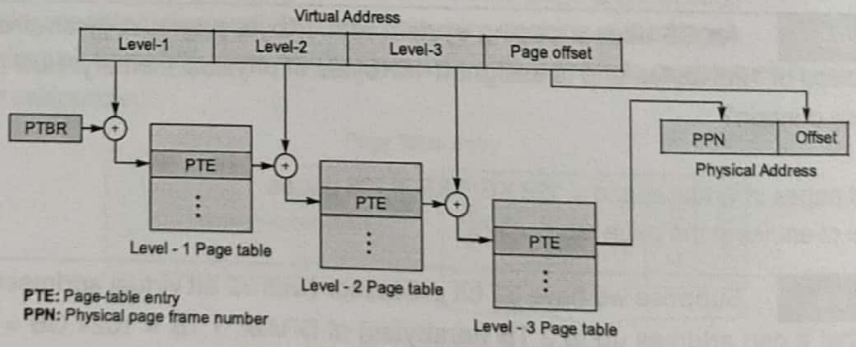
(c) 16 (VA) –3 (VPN) = 13.

(d) One entry per virtual page. Thus, 8.

6.7.2 Multi-Level Page Tables (Hierarchical)

Multilevel page tables are split into two or more levels. Multi-level page tables are tree-like structures to hold page tables.

- The entries of the level-0 page table are pointers to a level-1 page table.
- The entries of the level-1 page table are pointers to a level-2 page table.
- The entries of the last level page table are stores actual page frame information.



3-Level Paging System

- In multilevel paging when paging is applied on page table. The first level page level page entry will have base address of second level page entry and second level page entry will have base address of third page table and third level page entry will have BA of fourth level page table and so on.
- The last level page table entry will have frame number of actual page.
- The multilevel paging whatever may be levels of paging all the page tables (pages of PT) will be stored in the main memory.

- In multi level paging whatever may be the level of paging all the PT entries contain frame number only.
- If page size is not mention in the problem, generally the page size will be same in all the processes.

All page tables are stored in memory. So it requires more than one memory access to get the physical address of page frame. One accesses for each level needed.

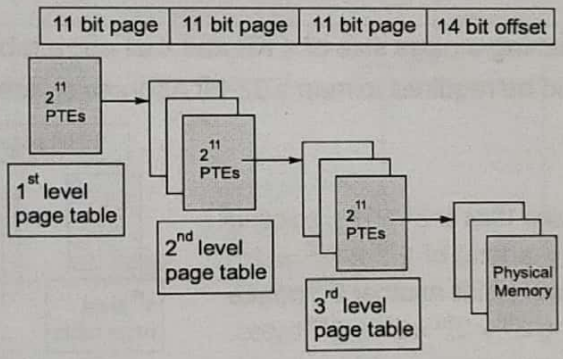
Simple address translation through a page table can cut instruction execution rate in half. More complex translation schemes (e.g., multi-level paging) are even more expensive.

Disadvantage: Extra memory references to access address translation tables can slow programs down by a factor of two or more. (Use translation look-aside buffers (TLB) to speed up address translation by storing page table entries).

Example-6.8 Suppose you have a 47-bit virtual address space with a page size of 16 KB and that page table entry takes 8 bytes. How many levels of page tables would be required to map the virtual address space if every page table is required to fit into a single page?

Solution:

In First level, one page table contains 2,048 or 2^{11} PTEs ($2^3 \times 2^{11} = 2^{14}$ bytes),
Pointing to 2^{11} pages (addressing a total of $2^{11} \times 2^{14} = 2^{25}$ bytes).
Adding a second level yields another 2^{11} pages of page tables,
Addressing $2^{11} \times 2^{11} \times 2^{14} = 2^{36}$ bytes.
Adding a third level yields another 2^{11} pages of page tables,
Addressing $2^{11} \times 2^{11} \times 2^{11} \times 2^{14} = 2^{47}$ bytes. So, we need 3 levels.



Example-6.9 Consider a virtual memory architecture with the following parameters:

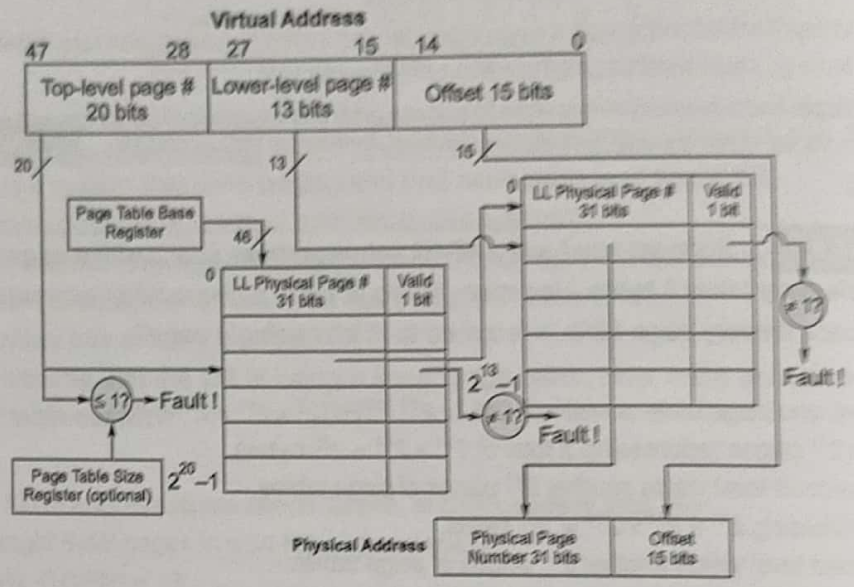
- Virtual addresses are 48 bits. The page size is 2K byte.
- The architecture allows a maximum 64 Terabytes (TB) of real memory (RAM).
- The first- and second-level page tables are stored in real memory.
- All page tables can start only on a page boundary.
- Each second-level page table fits exactly in a single page frame.
- There are only valid bits and no other extra permission, or dirty bits.

Show how the various fields of each address are interpreted, including the size in bits of each field, the maximum possible number of entries each table holds, and the maximum possible size in bytes for each table (in bytes).

Solution:

The first-level page table has 2^{20} (1 million) entries, each of which is 32 bits (31- bit PPN plus valid bit, or 4 bytes), so it has a maximum size of 4 MBytes.

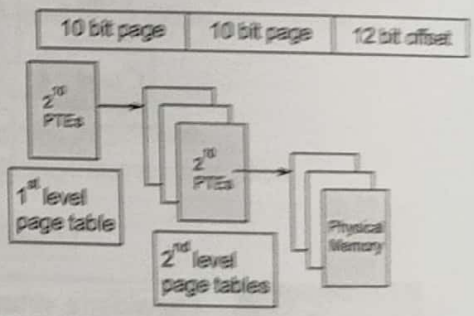
Each second-level page table has 8K entries. The page size is 32KB and the second-level PTE has a PPN of 31 bits plus a valid bit, which fits nicely in a 32 bit word or 4 bytes. A 32 KB page can thus hold 8K 4-byte entries (4 bytes times 8K is 32 KB).



Example-6.10 Assuming a page size of 4 KB and that page table entry takes 4 bytes, how many levels of page tables would be required to map a 32-bit address space if every page table fits into a single page?

Solution:

A 1-page page table contains 1024 or 2^{10} PTEs, pointing to 2^{10} pages (addressing a total of $2^{10} \times 2^{12} = 2^{22}$ bytes). Adding another level yields another 2^{10} pages of page tables, addressing $2^{10} \times 2^{10} \times 2^{12} = 2^{32}$ bytes. So, we need 2 levels.



Remember

Speed of Address Translation

Problem: Execution of each machine instruction may involve one or more memory operations (accesses):

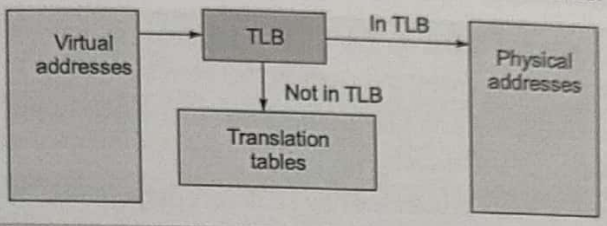
- One operation to fetch instruction.
- One operation or more for instruction operands.
- One extra memory operation for address translation through a page table for page table entry lookup.

Solution: Include a Translation Look-aside Buffer (TLB) in the MMU

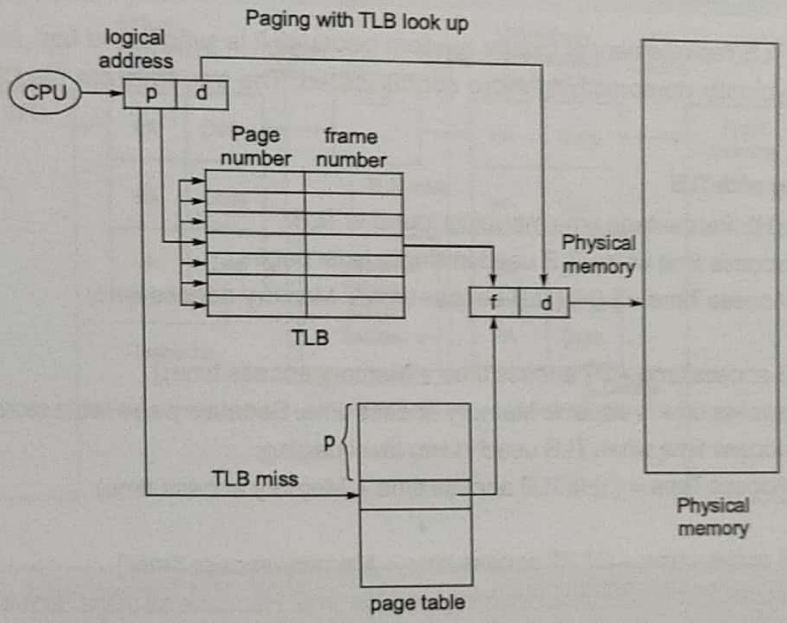
- TLB is a fast, fully associative address translation cache
- TLB hit avoids page table lookup

6.7.3 Paging with TLB (Translation Look-aside Buffer)

A TLB is on-chip hardware special cache which is part of memory-management unit. TLB is used in virtual-to-physical address translations so it is also called as address-translation cache. It is a cache that holds recently accessed page table entries. If address translation uses a TLB entry, access to the page table is avoided. TLB can cache only a few of page table entries. TLB is fully associative memory, so page numbers of all TLB entries are checked simultaneously for a match. TLB lookup is much faster than a memory access.

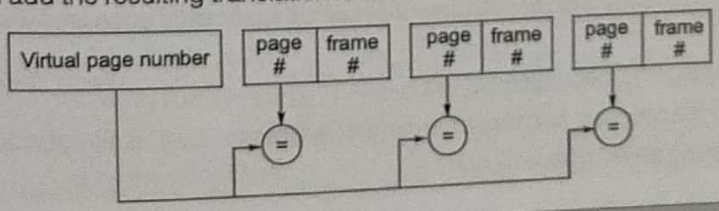


NOTE: Fully associative cache allows looking up all TLB entries in parallel. TLB used in address translation so TLB also called as **address translation cache.**



How TLB lookup helps in paging?

- When a logical address is generated by CPU, its page number searched in fully associative TLB.
- If page number is found in the TLB (hit), its frame number is accessed from TLB (without memory reference for address translation) to get physical address of actual frame.
- If page number is not found in the TLB(miss), one extra memory reference is needed to access the frame number (address translation needed) from page table to get the physical address of actual frame, and add the resulting translation to the TLB, replacing an existing entry if necessary.



Each entry in the TLB contains a (page number, frame number) pair, plus copies of some or all of the page's protection bits, use bit, and dirty bit.

Replacements of TLB Entries: For direct mapping, the entry being mapped is replaced whenever there is a mismatch of the virtual page number. For set associative or fully associative cache, it is possible to replace a random entry, the least recently used entry, or the most recently used, depending on the reference patterns.

- Typically, the TLB is on the CPU chip (Hardware controlled TLB), so the lookup time is significantly faster than looking up from the memory.
- TLBs are small and fully associative. Hardware caches are larger, and are direct mapped or set associative.
- Since different processes have different page tables, the on-chip TLB entries have to be entirely invalidated on context switches. An alternative is to include process IDs in TLB, at the additional cost of hardware and an additional comparison per lookup.

In hardware, TLB replacement is mostly random because it is simple and fast. In software, memory page replacements typically do something more sophisticated. The tradeoffs are the CPU cycles and the cache hit rate.

Performance of Paging with TLB

TLB Hit Rate (H): Percentage time mapping found in TLB.

1. Effective access time when TLB used in single level paging:
   Effective Access Time = [(H) (TLB access time + Memory access time)
   +
   (1-H) (TLB access time + PT access time + Memory access time)]
   Note: PT access time is equal to Memory access time. Because page table stores in main memory.
2. Effective access time when TLB used in two level paging:
   Effective Access Time = [(H)(TLB access time + Memory access time)
   +
   (1-H) (TLB access time + 2 * PT access time + Memory access time)]
   Note: PT access time is equal to Memory access time. Because all page tables are stored in main memory.

6.7.4 Paging with TLB and Hardware Memory Caches

Hardware memory cache holds data rather than frame number. So it may be possible to access the data on cache without accessing the TLB or main memory.

Hardware caches is to take advantage of **locality** in instruction and data references. Types of Locality: **temporal locality** and **spatial locality**.

Temporal Locality: An instruction or data item that has been recently accessed will likely be re-accessed soon in the future.

Examples: Variables and instructions of loop.

Spatial Locality: If a program accesses memory at address x, it will likely soon access memory near x.

Example: Accessing an array one by one.

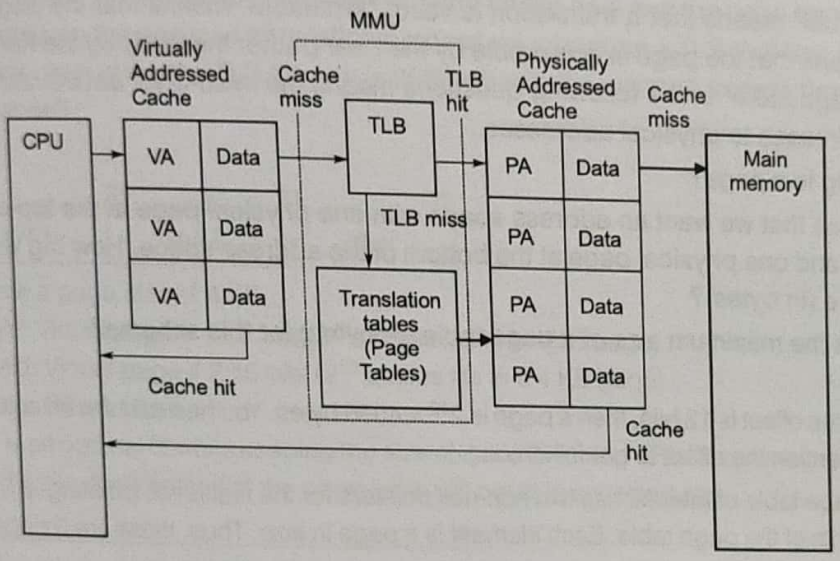
Hardware Memory caches are two types:

1. **Virtually Addressed Cache:** The cache between the CPU and MMU(the translation tables) is called the *virtually addressed cache*. If the virtually addressed cache gets a cache hit, it returns the data immediately without translating the virtual address.

   Advantage:
   - Without address translation virtually addressed cache is accessed.
   - It no need to access TLB and Main memory when cache hit occurs.
   - Data can be retrieved faster on cache hit.

2. **Physically Addressed Cache:** The cache between the MMU (translation tables) and the main memory is called the *physically addressed cache*. If the physically addressed cache gets a cache hit, it returns the data without consulting the main memory. Physically Addressed Cache accessed after address translation.

   Advantage: It is similar to physical memory but holds less data and faster than main memory.



Remember

Advantage of TLB: Since a process often references the same page repeatedly, translating each virtual address to physical address through multi-level translation is wasteful. Therefore, modern hardware provides a translation lookaside buffer (TLB) to track frequently used translations, to avoid going through translation in the common case.

How to update the data in physical memory when cache updates it?

Answer: If the cache data is modified, there are two ways to propagate the data back to the main memory.

1. The write-through approach immediately propagates update through various levels of caching, so the cached values are more consistent across different levels of memory hierarchy.
2. The write-back approach delays the propagation until the cached item is replaced from cache. The goal is to amortize the cost of update propagation across several cache modifications.

Example-6.11 Consider a multi-level memory management scheme with the following format for virtual addresses:

Virtual Page # (10 bits)	Virtual Page # (20 bits)	Offset (12 bits)

Virtual addresses are translated into physical addresses of the following form:

Physical Page # (20 bits)	Offset (12 bits)

Page table entries (PTE) are 32 bits in the following format, *stored in big-endian form* in memory (i.e., the MSB is first byte in memory):

Physical Page# (20 bits)	OS defined (3 bits)	0	Large Page	Dirty	Accessed	Nocache	Write Through	User	Writeable	Valid

Here, "Virtual" means that a translation is valid, "Writeable" means that the page is writeable, "User" means that the page is accessible by the User (rather than only by the Kernel). *Note:* the phrase "page table" in the following questions means the multi-level data structure that maps virtual addresses to physical addresses.

(a) How big is a page?

(b) Suppose that we want an address space with one physical page at the top of the address space and one physical page at the bottom of the address space. How big would the page table be (in bytes)?

(c) What is the maximum size of a page table (in bytes) for this scheme?

Solution:

(a) Since the offset is 12 bits, then a page is $2^{12} = 4096$ bytes. You had to show an actual calculation and mention the offset to get full credit.

(b) The page table of interest has two non-null pointers for the first level, pointing at 2 second-level elements of the page table. Each element is a page in size. Thus, there are 3 pages = 3×4096 = 12288.

(c) The maximum page table size has an entry for every virtual address. Thus, all pointers non-null at the top level, each of which points at a second-level element of the page table. Thus, the total size of the page table has 1 + 1024 = 1025 page table elements = $1025 \times 4096 = 4198400$ bytes.

Example-6.12 Compute effective access time when TLB with single page table is used for paging with following parameters. Memory access time = 100 ns, TLB access time = 20 ns, hit rate = 80%.

Solution:

Effective Access Time = (0.8) (20 +100) + (0.2) (20 + 100 + 100) = 140 ns

Example-6.13 Compute effective access time when TLB used in two level paging with following parameters. Memory access time = 100 ns, TLB access time = 20 ns, hit rate = 80%.

Solution:

Effective Access Time = (0.8) (20 +100) + (0.2) (20 + 100 + 100 + 100)
= 96 ns + 64 ns = 160 ns

Example-6.14 For a byte addressable system, the virtual memory address space is 32 bits and the physical memory address space is 16 bits.

(a) Assume the system uses a two level page table to translate a virtual address to a physical address. Show the format of the virtual address, specify the page size (pick one size if multiple sizes are feasible), and specify the length of each field in the virtual address. Make sure that each translation table fits in a page.

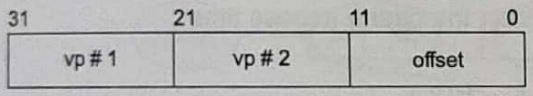
(b) Assume you add to your system a 4 way set-associative data cache with 16 cache blocks. Each block in the cache holds 8 bytes of data. In order to address a specific byte of data, you will have to split the address into the cache tag, cache index and byte select. Which parts of the address would you associate with each component, how long will each component be (in bits) and why? (*Note*: Assume there are no modifiers bits in the table).

(c) The main memory access time is 100 ns, and the cache lookup time is 50 ns. Assuming a cache hit rate of 90%, what is the average time to read a location from main memory? (*Note*: Assume the cache hit rate is the same for the data and the page translation tables).

(d) To speed up the address translation process we introduce a TLB that has an access time of 20ns. Assuming the TLB hit rate is 95%, what is the average access time for a memory operation?

Solution:

(a)



Assume a page size of 4 KB

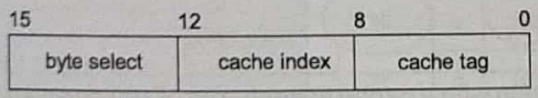
1st field: Virtual page # 1:10 bits (2^{10} entries fits in a 4 KB page)

2nd field: Virtual page # 2:10 bits (2^{10} entries fits in a 4 KB page)

3rd field: Offset 12 bit

4 KB is an optimal fit while keeping the size of the VPN#1 and VPN#2 constant. If the offset is less than 12 bits, then entries of the page table will not fit in a single page

(b)



1st field: byte select 3 bits (8 bytes of data = 2^3)

2nd field: cache index 4 bits (16 cache blocks = 2^4)

3rd field: cache tag 9 bits (16 – 2 – 3) = 11

(c) Each access requires three memory reads, one for 1st level page table, one for the 2nd level page tables, and one for accessing that memory data. For each memory access, the hit rate is 90%, and thus the

$$\text{Average access time} = 0.9 \times 50 \text{ ns} + 0.1 \times (50 + 100) \text{ ns} = 60 \text{ ns}$$

Since each read requires three accesses to the memory.

$$\text{Average read time} = 3 \times 60 \text{ ns} = 180 \text{ ns}$$

If the cache lookup and memory reads were assumed to be parallel, then the average access time would be 55ns.

(d) If we have a TLB hit, then

$$\text{Average read time} = 0.95 \times (20 \text{ ns} + 60 \text{ ns}) + 0.05 \times (20 \text{ ns} + 180 \text{ ns})$$
$$= 86 \text{ ns}$$

Example-6.15 Consider the following parameters of virtual memory system

Measurement	Value
Pt = Probability of a TLB miss	0.1
Pp = Probability of a page fault when a TLB miss occurs	0.0002
Tt = Time to access TLB	0
Tm = Time to access memory	1 microsecond
Td = Time to transfer a page to / from disk	10 milliseconds = 10000 microseconds
Pd = Probability page is dirty when replaced	0.5

Find the average access time.

Solution:

$$\text{Time} = P_t \times (T_m + P_p \times (T_d + P_d \times T_d)$$

$$\text{Time} = 0 + 1\,\mu s / 0.1 \times (1\,\mu s + 0.0002 \times (10000 + 0.5 \times 10000))$$

$$= 1 + 0.1 \times 1(1 + 0.0002 \times 15000)$$

$$= 1 + 0.1 \times 4 = 1.4\,\mu s$$

Example-6.16 Consider a memory system with a cache access time of 10 ns and a memory access time of 200 ns, including the time to check the cache. What hit rate H would we need in order to achieve an effective access time 10% greater than the cache access time?

Solution:

Effective Access Time: $T_e = H \times T_c + (1 - H) \times T_m$

where $T_c = 10$ ns

$T_e = 1.1 \times T_c$

and $T_m = 200$ ns

Thus, $(1.1)(10) = 10H + (1 - H)\,200\;11$

$= 10H + 200 - 200H$

$-189 = -190H$

$H = 189/190$

Example-6.17 Consider a memory system with a cache access time of 100 ns and a memory access time of 1200 ns. If the effective access time is 10% greater than the cache access time, what is the hit ratio H?

Solution:

Effective Access Time: $T_e = H \times T_c + (1 - H)(T_m + T_c)$

where, $T_c = 100$ ns

$T_e = 1.1 \times T_c$

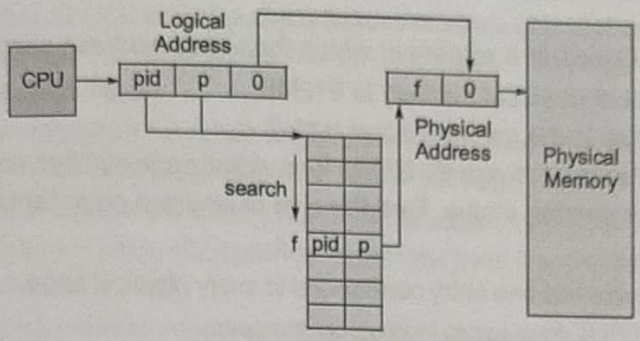
and $T_m = 1200$ ns

Thus, $(1.1)(100) = 100H + (1 - H)(1200 + 100)\;110 = 100H + 1300 - 1300H$

$H = 119/120$

6.7.5 Inverted Page Table

Inverted table is shared among processes. Each entry of inverted table must contain the *process ID* of the page and *virtual page number*. The physical page number is not stored, since the index in the table corresponds to it.



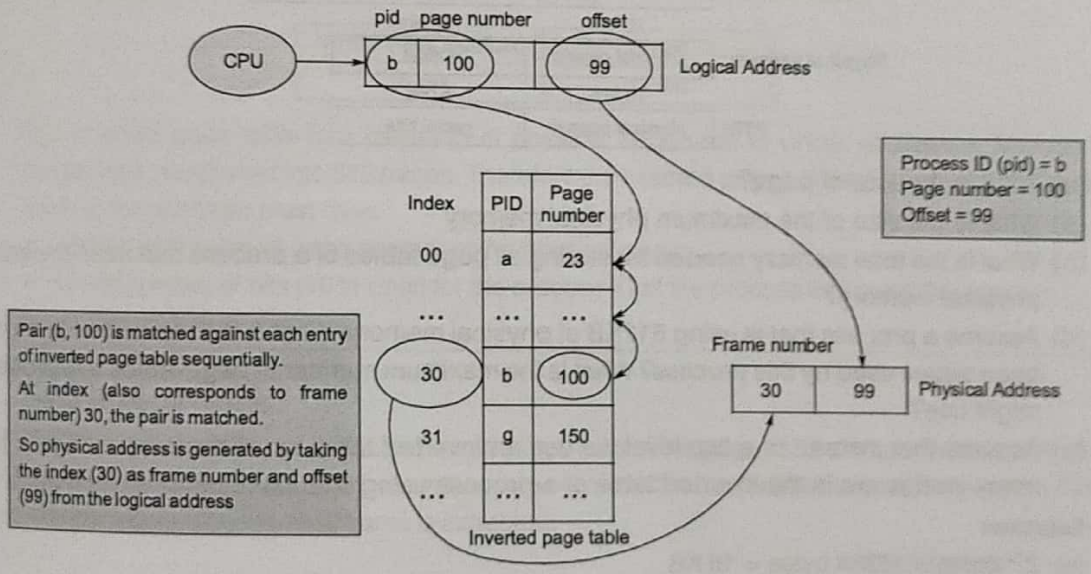
Inverted Page Table

Inverted page table has following differences when compared with normal page tables:

- There is only one inverted page table, not one table per process
- Entries in an inverted page table must include a process identifier
- An inverted page table maps physical frames to virtual pages.

In order to translate a virtual address, the virtual page number and current process ID are compared against each entry of inverted table (linear) sequentially. When a match is found, the **index** of the match replaces the virtual page number in the address to obtain a physical address. If no match is found, a page fault occurs.

Example:



Advantages:

- Size of inverted page table scales with physical but not virtual memory.
- No problem with virtual sparsity
- Only one Inverted Page Table per system

Disadvantages:

- Lookup time is increased because it requires a search on the inverted table. (Hashed inverted page table would help to reduce the lookup time instead of linear inverted page table).
- An inverted page table only specifies the location of virtual pages that are located in memory. (no information about the pages that are not in memory).
- Hard to implement shared memory.

Example-6.18 Consider a system in which the virtual address space is 64 bits, the page size is 4KB, and the amount of physical memory is 512MB.

(a) How many entries in the inverted page table?

(b) Assuming 16 bits for a process ID, 52 bits for a virtual page number, and 12 bits of information for protection or sharing status, Find the size of inverted page table.

Solution:

(a) Inverted page table has one entry correspond to every physical page.

512MB/4KB = 128 K frames are in Physical memory.

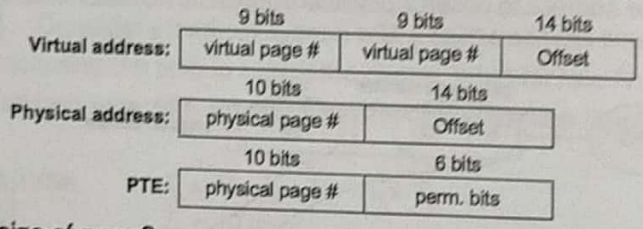
So Inverted table has 128 K entries.

(b) Each entry takes 16 + 52 + 12 = 80 bits or 10 bytes.

128 K frames are in physical memory.

So the size of inverted page table is 10B × 128K = 1.3 MB.

Example-6.19 Consider a memory architecture using two-level paging or address translation. The format of the virtual address, physical address, and PTE (page table entry) are below:



(a) What is the size of page?

(b) What is the size of the maximum physical memory

(c) What is the total memory needed for storing all page tables of a process that uses the entire physical memory?

(d) Assume a process that is using 512KB of physical memory. What is the minimum number of page tables used by this process? What is the maximum number of page tables this process might use?

(e) Assume that instead of a two-level we use an inverted table for address translation. How many entries are in the inverted table of a process using 512 KB of physical memory?

Solution:

(a) 2^{14} bytes = 16384 bytes = 16 KB

(b) 2^{24} bytes = 16 MB

(c) There are 2^{10} = 1024 physical pages. There is one page table at the first level, and up to 2^9 = 512 page tables at the second level. Since the physical address is 3 bytes, the size of the first level page is 2^9 3 bytes = 1536 bytes. Furthermore, the PTE is 2 bytes, so the size of a second level table is 2^9 2 bytes = 1024 bytes. All in all, the page tables use 1536 bytes * 512 *1024 bytes = 528834 bytes of memory.

(*Note:* We gave full credit to answers assuming that the entries at the first level page are 2 bytes, as well. Indeed, the last 9 bits of an address to a page table are typically 0, and don't need to be stored)

(d) The process uses 512 KB/16 KB = 32 physical pages. Since a second level page can hold up to 512 PTEs, in the best case scenario we use only 2 page tables: 1st level page + a 2nd level page.

In the worst case, the process may use a little bit of every physical page (e.g., 0.5 KB of each physical page), and all page tables will be populated. Thus, the process ends up using **1 + 512 = 513 page tables.**

Note: We have also given full credit to people who assumed that the process fully uses each physical page. In this case the answer is **1 + 32 = 33 page tables.**

(e) The inverted table maintains one entry per physical page. In the worst case, the process uses all physical pages, which yields **1024 entries.** In the best case, the process fully uses each physical page which yields **32 entries.**

Note: We gave full credit to people who only answered **32 entries.**

Example-6.20 Suppose that you have a system with 32 bit pointers and 4 megabytes of physical memory that is partitioned into 8192 byte pages. The system uses an Inverted Page Table (IPT). Assume that there is no page sharing between processes. How many bits should be in each page table entry and what are they for? Also, how many page table entries should there be in the page table?

Solution:

Virtual addresses are 32 bits, and split into two parts. The page number is the first 19 bits, and the offset within the page is the last 13 bits ($2^{13} = 8192$)

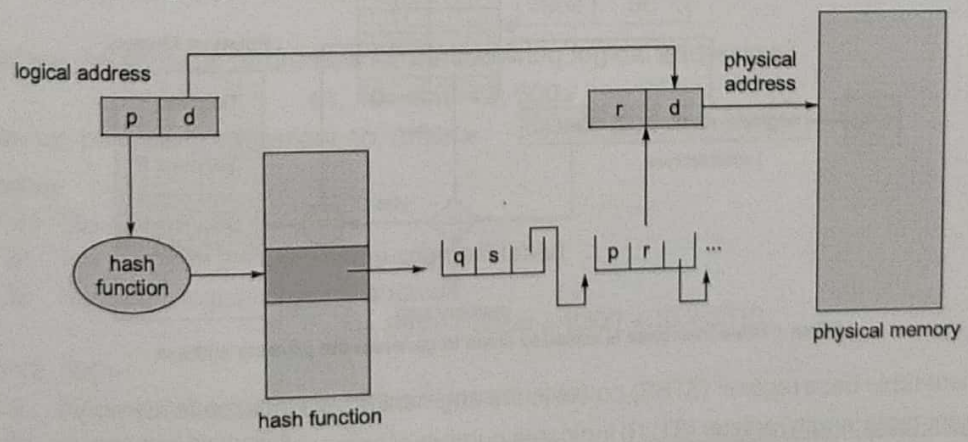
Virtual Page Number	Offset
19 bits	13 bits

The inverted page table is a mapping of physical addresses to virtual addresses. Memory is 4 megabytes, partitioned into 512 pages. Therefore the inverted page table will consists of 512 entries. Each of these entries must have:

- 19 bits for the virtual page number of the physical page.
- Some number of bits (16 in Unix) for the process ID of the process that owns the page.
- Protection bits (r/w/x)

6.7.6 Hashed Page Tables

The virtual page number is hashed into a page table. This page table contains a chain of elements hashing to the same location. Virtual page numbers are compared in this chain searching for a match. If a match is found, the corresponding physical frame is extracted.



Advantage: Simple.

Disadvantages: Need to handle collisions and need one hash table per address space.

Advantages	Disadvantages
• Efficient memory use • Simple partition management (non-contiguous memory allocation) • Allocating memory is easier and cheap • No compaction is necessary • Easy to share pages • No external fragmentation • More efficient swapping	• Internal fragmentation (only at last page of process) • Need special hardware for address translation • Main memory is used for page table (Consumes memory) • Address translation lengthens memory cycle times • Multi-level paging leads to memory reference overhead • Longer memory access time (page table lookup)

6.8 Segmentation

Instead of dividing memory into equal-sized pages, virtual address space is divided into *variable-length* segments. Segments are variable-length modules of a program that correspond to logical units of the program. Segments represent the modular structure of how a program is organized. Before a program can execute, all its segments need to be loaded into memory. For every segment, the operating system needs to find a contiguous block of available memory to allocate to the segment.

6.8.1 Address Mapping in Segmentation

Logical address consists of two parts: segment number and offset.

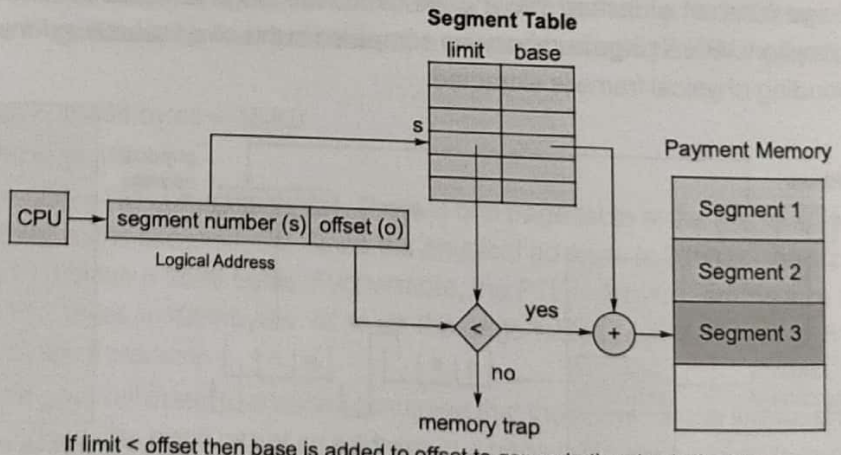
<segment-number, offset>

Mapping from logical address to physical address is done with the help of a *segment table.*

Segment table: It maintains an entry for each segment in logical address space. Each entry of segment table contain *base* and *limit.*

Limit (Bound): It specifies the length of the segment.

Base: It is the starting address where the segment reside in memory.

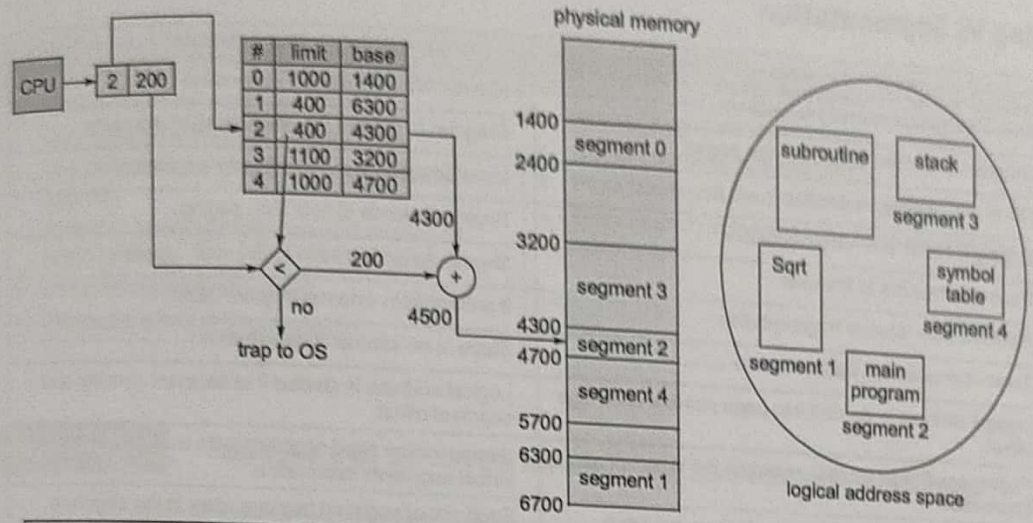


If limit < offset then base is added to offset to generate the physical address

Segment-table base register (STBR) points to the segment table's location in memory.

Segment-table length register (STLR) indicates number of segments used by a program.

Example:



Advantages	Disadvantages
• No internal fragmentation • Segment tables have only one entry per actual segment • Average segment size is larger than average page size • Less overhead • Efficient translation (Segment table is small, so it fits in MMU) • Different protection for different segments • Enables sharing of selected segments • Easier to relocate segments than entire address space • Enables sparse allocation of address space	• It has external fragmentation • Costly memory management algorithms • Segmentation: find free memory area big enough • Still expensive/difficult to allocate contiguous memory to variable sized segments

Example-6.21 Consider the following segment table:

Segment	Base	Length
0	219	600
1	2300	14
2	90	100
3	1327	580
4	1952	96

What are the physical addresses for the following logical addresses?

<1, 10> and <2, 500>

Note: <a, b> means <segment no, offset>

Solution:

For <1, 10>:

a: 10 < 14 (from 2nd entry of the segment table)

b: Hence the logical physical address

= Base + offset = 2300 + 10 = 2310

For <2, 500>:

a: 500 > 100 (from 3rd entry of the segment table)

b: Hence this is invalid

6.8.2 Paging Vs Segmentation

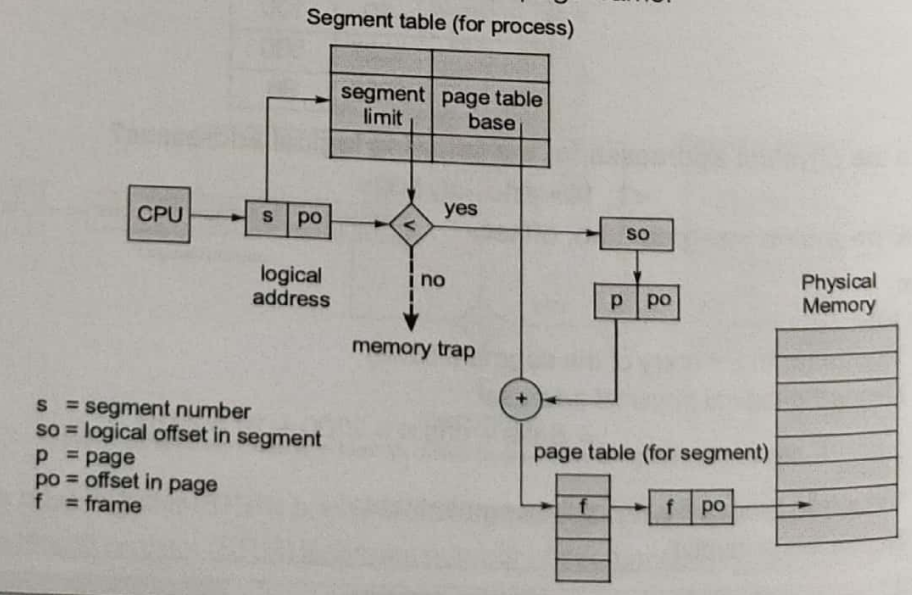
Paging	Segmentation
Non-Contiguous memory allocation	Non-Contiguous memory allocation
Program is divided into fixed size pages	Program is divided into variable size segments
OS is responsible for paging	User/Compiler is responsible for segmentation
Paging is faster than Segmentation	Segmentation is slower than paging
Paging is invisible to the user	Segmentation is visible to the user
It suffers from internal fragmentation	It suffers from external fragmentation
There is no external fragmentation	There is no internal fragmentation
Logical address is divided into page number and page offset	Logical address is divided into segment number and segment offset
Paging need page table to maintain the virtual pages information	Segmentation need segment table to maintain the virtual segments information
Each virtual page has one entry in the page table	Each virtual segment has one entry in the segment table
Page table entry has frame number and additional protected bits for pages	Segment table entry has limit, base and may contain some bits for protection of segments
Operating system must maintain a free frame list.	Operating system must maintain a list of free holes in main memory

6.9 Segmented Paging

Paging and segmentation can be combined to take advantage of the best features of both by assigning fixed-size pages within variable-sized segments.

- Pages are typically smaller than segments.
- Each segment has a page table, which means every program has multiple page tables.
- The virtual address is represented as a pair (segment number, offset), but the offset consists of a pair (page number, page offset). **or** The virtual address is represented as (segment number, page number , page offset)

Segment number: Which points the system to the appropriate page table.
Page number: Which is used as an offset into the page table of the segment.
Page offset: Which is used as an offset within the page frame.



Advantages and Disadvantages

Advantages	Disadvantages
• Reduces memory usage • Page table size limited by segment size • Segment table has only one entry per actual segment • Simplifies handling protection and sharing of larger modules • Simplifies memory allocation • Eliminates external fragmentation	• Internal fragmentation • Higher overhead and complexity • Page tables still need to be contiguous • Each memory reference now takes two lookups

Example-6.22 Suppose that we have a 64-bit virtual address split as follows:

6 Bits [Segment ID]	11 Bits [Table ID]	11 Bits [Table ID]	11 Bits [Table ID]	11 Bits [Page ID]	14 Bits [Offset]

(a) How big is a page in this system?

(b) How many segments are in this system?

(c) Assume that the page tables are divided into page-sized chunks (so that they can be paged to disk). How much space have we allowed for a PTE in this system?

(d) Assume that a particular user is given a maximum-sized segment full of data. How much space is taken up by the page tables for this segment?

Solution:

(a) Since offset is 14 bits, the page is 2^{14} = 16384 bytes or 16KB

(b) There is a 6-bit segment ID, So there are 2^6 = 64 possible segments.

(c) Since leaves of page table contain 11 bits to point at pages (the field marked "Page ID"), a 2^{14} bit page must contain 2^{11} PTEs, which means that a PTE is simply $2^{14} - 11 = 8$ bytes in size.

(d) Full page table will be a tree-like structure.

Top-page entry (1) will point at 2^{11} page entries, each of which will point at 2^{11} page entries, each of which will point at 2^{11} page entries, each of which finally points at 2^{11} pages.

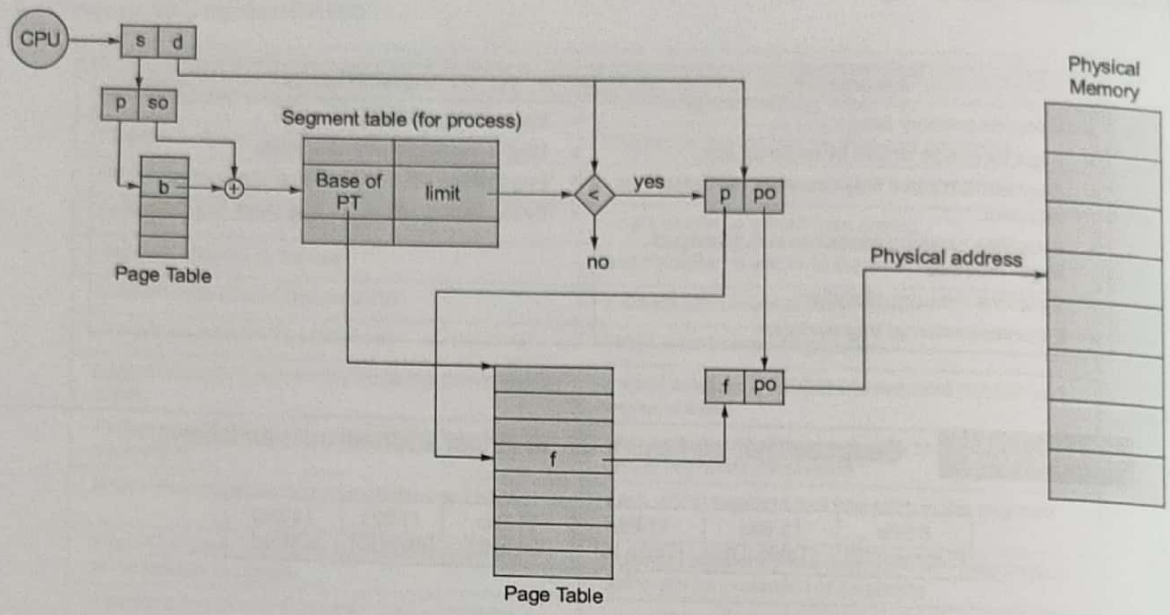
All of these are a full page in size (2^{14}).

So, Answer (just page tables) = $2^{14} \times (1 + 2^{11} + 2^{11} \times 2^{11} + 2^{11} \times 2^{11} \times 2^{11})$

$= 214 \times (1 + 211 + 222 + 233)$

Paging on Segment

Paging on segment is needed when segment size increases and also paging on segment table, when ST size increases.



Paging on Segmentation

- To avoid overhead of bringing large size segment table into memory hence segmented paging is implemented and paging is applied on segment table.
- Instead of bringing entire segment table into memory the pages of segment table will be brought into memory.

  Let P_2 : Number of bits required to represent pages of segment table or page number of segment table. d_2 : Number of bits required to replacement size of page of segment table.

Note: In above figure, ***so*** represent segment offset and ***po*** represent page offset.

6.10 Buddy System

Static(Fixed) partition schemes suffers from the limitation of having fixed number of active (non suspended) processes and the usage of space may also not be optimal. The buddy system is a memory allocation and management algorithm that manages memory in power of two increments. Assume the memory size is 2U. Suppose a size of s is requested,

- If $2^{U-1} < S \leq 2^U$: Allocate the whole block.
- **Else**: Recursively divide the block equally and test the condition at each time; when it satisfies, allocate the block and get out of the loop.

System also keep the record of all the unallocated blocks (holes) each and can merge these different size blocks to make one big chunk.

Advantage:

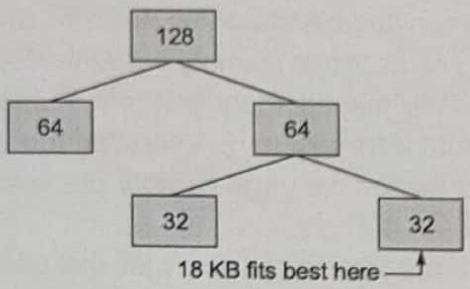
- Easy to implement a buddy system
- Allocates block of correct size
- It is easy to merge adjacent holes
- Fast to allocate memory and de-allocating memory

Disadvantages:

- It requires all allocation units to be powers of two.
- It leads to internal fragmentation.

Example-6.23 Consider a system having buddy system with physical address space (PAS) 128 KB. Calculate the size of partition for a 18 KB process.

Solution:



Size of partition for 18 KB process = 32 KB.

Summary

- **Compaction:** The process of collecting fragments of available memory space into contiguous blocks by moving programs and data in a computer's memory or disk. Also called *garbage collection.*
- **Deallocation:** The process of freeing an allocated resource, whether memory space, a device, a file, or a CPU.
- **Address resolution:** The process of changing the address of an instruction or data item to the address in main memory at which it is to be loaded or relocated.
- **Associative memory:** The name given to several registers, allocated to each active process, whose contents associate several of the process segments and page numbers with their main memory addresses.
- **Cache memory:** A small, fast memory used to hold selected data and to provide faster access than would otherwise be possible.
- **Job Table:** A table in main memory that contains two values for each active job the size of the job and the memory location where its page map table is stored.
- **Locality of reference:** The behaviour observed in many executing programs in which memory locations recently referenced, and those near them, are likely to be referenced in the near future.
- **Fixed partitions:** A memory allocation scheme in which main memory is sectioned off, with portions assigned to each job.
- **Dynamic partitions:** A memory allocation scheme in which jobs are given as much memory as they request when they are loaded for processing, thus creating their own partitions in main memory.
- **Relocatable dynamic partitions:** A memory allocation scheme in which the system relocates programs in memory to gather together all of the empty blocks and compact them to make one block of memory that's large enough to accommodate some or all of the jobs waiting for memory.
- **First-fit**: Allocate the *first* hole that is big enough.
- **Best-fit**: Allocate the *smallest* hole that is big enough; must search entire list, unless ordered by size. Produces the smallest leftover hole.
- **Worst-fit**: Allocate the *largest* hole; must also search entire list. Produces the largest leftover hole.

- **Internal fragmentation:** A situation in which a fixed partition is only partially used by the program; the remaining space within the partition is unavailable to any other job and is therefore wasted.
- **External fragmentation:** A situation in which the dynamic allocation of memory creates unusable fragments of free memory between blocks of busy, or allocated, memory.
- **MMU:** A memory management unit that supports paging causes every **logical address (virtual address)** to be translated to a **physical address (real address)** by translating the logical page number of the address to a physical page frame number.
- **Page:** A fixed-size section of a user's job that corresponds in size to page frames in main memory.
- **Paged memory allocation:** A memory allocation scheme based on the concept of dividing a user's job into sections of equal size to allow for non-contiguous program storage during execution.
- **Page frame:** An individual section of main memory of uniform size into which a single page may be loaded without causing external fragmentation.
- **Page Table :** a table in main memory with the vital information for each page including the page number and its corresponding page frame memory address.
- **Re-entrant code:** Code that can be used by two or more processes at the same time; each shares the same copy of the executable code but has separate data areas.
- **Segment:** A variable-size section of a user's job that contains a logical grouping of code.
- **Segment Table:** A table in main memory with the vital information for each segment including the segment number and its corresponding memory address.
- **Segmented memory allocation:** A memory allocation scheme based on the concept of dividing a user's job into logical groupings of code to allow for non-contiguous program storage during execution.

Student's Assignment

Q.1 A system uses virtual address space is of size 1 GB, and page size is 1 KB. The system has maximum 32 K physical pages. If each page tables entry holds only a valid bit and the page frame bits. How much memory is used for page table when virtual memory uses single level paging?

(a) 1 MB (b) 2 MB
(c) 3 MB (d) 4 MB

Q.2 Which of the following elements is not applicable in a page table entry of page table?

(a) Frame number (b) Present bit
(c) Modify bit (d) None of these

Q.3 Consider the following sequence.

Process P_1 of size 7 K loaded.

Process P_2 of size 4 K loaded.

Process P_1 is terminated and space is returned (available for other processes)

Process P_3 of size 3 K loaded

Process P_4 of size 6 K loaded

If a system has 16 KB memory and buddy system is used to allocate the memory for process during runtime.

How much space is wasted due to internal fragmentation? (Assume frame size is 1 K)

(a) 1 K (b) 2 K
(c) 3 K (d) None of these

Q.4 Consider a system uses level-1 cache. (When a memory is accessed from main memory, the cache access will not burden) cache access time is 10 ns and memory access time is 110 ns. If the effective access time is 10% greater than the cache access time then what is the hit ratio?
(a) 90% (b) 95%
(c) 99% (d) 100%

Q.5 Consider a virtual memory system with physical memory of 8 GB, a page size of 8 KB and 46-bit virtual address. Assume every page table exactly fits into a single page. If page table entry size is 4 bytes then how many levels of page tables would be required?
(a) 1 (b) 2
(c) 3 (d) 4

Q.6 Suppose that TLB is used in one level paging system with each look up time of TLB takes 20 ns. Memory reference takes 100 ns. What is the effective memory reference time if 60 percent of page table references are found in the TLB?
(a) 160 (b) 180
(c) 200 (d) None of these

Q.7 Consider a system has 32-bit virtual address space, and each page size of 16 KB. Assume memory is byte addressable and size of main memory is 512 MB. What is the minimal size of the page table when page table entry includes a valid bit and frame bits?
(a) 0.5 Mbytes (b) 10 M Bytes
(c) 1.5 M Bytes (d) 2 M Bytes

Q.8 Consider the following memories
1. Main memory 2. Hard disk
3. Level-2 cache
Which of the above memories can be used to store the page tables?
(a) Only 1 and 2 (b) Only 2 and 3
(c) Only 1 and 3 (d) 1, 2 and 3

Q.9 Consider a virtual memory with following parameters.
- Each page size of 2 KB
- 32-bit virtual address and 32-bit physical address
- single level paging is used
- For each page it also stores additional 3 bits for protection.

What is size of page table for each process?
(a) 1 MB (b) 3 MB
(c) 6 MB (d) 9 MB

Q.10 Consider a virtual system with 2-level paging. 32-bit virtual address is used to support 32-bit physical address. Page size is 4096 byte pages and page table has 4 bytes of page table entry. How many memory accesses required to read or write a single word?
(a) 2 (b) 3
(c) 4 (d) 5

Q.11 Consider a 32-bit virtual address is used for paging with page size of 1024 bytes. Two-level paging is implemented with equal number of entries in every page table of the system. If page table entry size is 2 bytes. What is the maximum size of main memory supported by the above system?
(a) 16 MB (b) 32 MB
(c) 64 MB (d) None of these

Q.12 Consider a virtual memory system with 34-bit address and page size is 2 KB.How many entries does an inverted page table need for the system with 64 MB physical memory?
(a) 32768 (b) 65536
(c) 131072 (d) None of these

Q.13 Operating systems can divide main memory into portions called _____ so that one process cannot overwrite another's memory, either inadvertently or maliciously.
(a) sections (b) partitions
(c) modules (d) regions

Q.14 A memory manager's primary function is
(a) to determine how to allocate available memory
(b) to determine how to respond to changes in a process's memory usage
(c) to interact with special memory management hardware
(d) all of the above

Q.15 Noncontiguous memory allocation splits programs into blocks of memory called _____ that can be loaded into nonadjacent "holes" in main memory.
(a) pages (b) partitions
(c) frames (d) segments

Q.16 A _____ can be used in single-user contiguous memory allocation systems to prevent programs from damaging the operating system's memory.
(a) lock (b) protection bit
(c) boundary register (d) semaphore

Q.17 In a paging system
(a) a page frame is always larger than an incoming page
(b) a page frame is always smaller than the incoming page
(c) given the virtual memory address $v = (p, d)$, where page p is in page frame p' and the fixed page size is p_s, then the real memory address of v is $(p' + d) \times p_s$
(d) when a process references a page that is not in main memory, the operating system loads the nonresident page into memory from secondary storage

Q.18 When a page references a page that is not in main memory, the processor generates a
(a) TLB miss (b) page fault
(c) protection fault (d) segment fault

Q.19 Page sharing
(a) increases the amount of main memory required for a group of processes to run efficiently
(b) decreases system's degree of multi-programming
(c) does not allow modifiable procedures from being used by more than one process at the same time
(d) none of the above

Q.20 Scheduling strategies are commonly evaluated by the system's
(a) throughput
(b) mean response time
(c) variance of response times
(d) all of these

Q.21 Solution of external fragmentation is
(a) compaction (b) segmentation
(c) both (a) and (b) (d) none of these

Q.22 After a user has saved and deleted many files, many scattered areas of stored data remain that are too small to be used efficiently, causing
(a) disarray (b) fragmentation
(c) disorder (d) none of these

Q.23 Dirty bit is used to show the
(a) page with corrupted data
(b) the wrong page in the memory
(c) page that is modified after being loaded into cache memory
(d) page that is less frequently used

Q.24 In which of the storage placement strategies a program is placed in the smallest available hole in the main memory?
(a) best fit (b) first fit
(c) worst fit (d) All of these

Q.25 Moving process from main memory to disk is called
(a) scheduling (b) caching
(c) swapping (d) spooling

Answer Key:

1. (b)	**2.** (d)	**3.** (c)	**4.** (c)	**5.** (c)
6. (a)	**7.** (a)	**8.** (d)	**9.** (c)	**10.** (b)
11. (c)	**12.** (a)	**13.** (b)	**14.** (d)	**15.** (d)
16. (c)	**17.** (d)	**18.** (b)	**19.** (c)	**20.** (d)
21. (c)	**22.** (b)	**23.** (c)	**24.** (a)	**25.** (a)

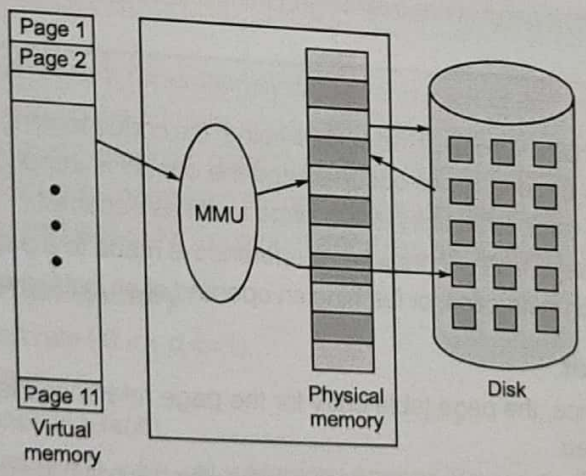
CHAPTER 07

Virtual Memory

7.1 Introduction

Virtual memory is the separation of user logical memory from physical memory. This technique provides larger memory to the user by creating virtual memory space. It facilitates the user to create a process which is larger than the physical memory space. We can have more processes executing in memory at a time.

It increases degree of multiprogramming. With the virtual memory technique, we can execute a process which is only partially loaded in memory.



Virtual Memory vs Physical Memory

Advantages of virtual memory:

- A process can execute without having all its pages in physical memory.
- A user process can be larger than physical memory
- Higher degree of multiprogramming
- Less I/O for loading and unloading for individual user processes
- Higher CPU utilization and throughput.
- Allows address spaces to be shared by several processes.

Virtual memory techniques:

- Overlays
- Paged virtual memory
- Segmented virtual memory

7.1.1 Paged Virtual Memory

The memory space of a process is divided into pages. Those pages in virtual memory that do not currently reside in main memory may reside on a disk or other secondary memory.

When to swap a page?

1. **Demand paging:** A page is not swapped in until it is referenced.
2. **Pre-paging:** A page is swapped in before it is referenced.

7.1.2 Demand Paging

Load a page into memory only when it is needed (when it is referenced). The virtual memory manager swaps in a page of an executing process whenever the execution of a process references a page that is not in physical memory. Any unused page in physical memory will normally be swapped out to disk.

7.1.3 Valid/Invalid Bit in Demand Paging

When the operating system decides to start a new process, it swaps only a small part of this new process (a few pages) into memory. The page table of this new process is prepared and loaded into memory, and the valid/invalid bits of all pages that are brought into memory are set to "valid". All the other valid/invalid bits are set to "invalid" showing that those pages are not brought into memory.

- Valid/Invalid bit = 0, if page is available in memory
- Valid/Invalid bit = 1, if page is not available in memory

7.2 Page Fault

Each page table entry has a valid bit, it indicates whether the corresponding page is currently in memory. If the page is not in memory, a page fault has occurred and the control is trapped to the OS. During address translation, if valid/invalid bit in page table entry is 0, a page fault has occurred.

In demand paging, a page fault occurs when a reference is made to a page not in memory. The page fault may occur while fetching an instruction, or fetching an operand of an instruction.

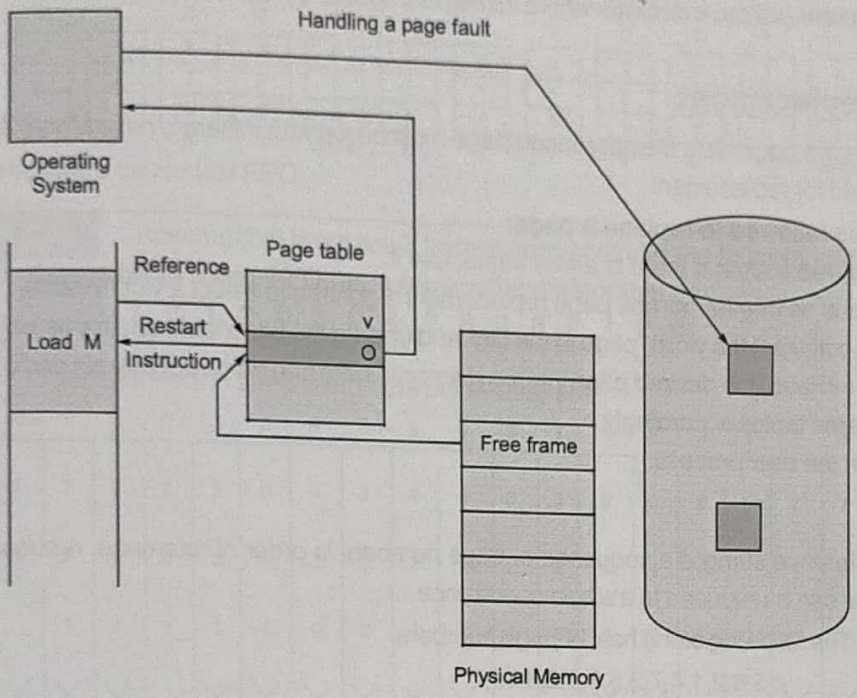
7.2.1 Handling a Page Fault

For every page reference, the page table entry for the page referenced is examined. If the access is invalid, the process is terminated.

If the process currently executing tries to access a page which is not in the memory, a page fault occurs, and the OS brings that page into the memory. If a process causes page fault then the following procedure is applied:

1. Trap the OS
2. Save registers and process state for the current process
3. Check if the trap was caused because of a page fault and whether the page reference is legal
4. If yes, determine the location of the required page on Disk
5. Find a free frame
6. Read (swap in) the required page from the Disk into the free frame. (During this I/O, the processor may be scheduled to some other process)

7. When I/O is completed, restore registers and process state for the process which caused the page fault and save state of the currently executing process
8. Modify the corresponding page table entry to show that the recently copied page is now in memory.
9. Resume execution with the instruction that caused the page fault



NOTE: Use modify (dirty) bit to reduce overhead of page transfers (only modified pages are written to disk)

Remember

Main operations by operating system during page fault:
- Checking the address and finding a free frame or victim page (fast)
- Swap out the victim page to secondary storage (slow)
- Swap in the page from secondary storage (slow)
- Context switch for the process and resume its execution (fast)

7.2.2 Performance in Virtual Memory

Let p be the page fault rate ($0 <= p <= 1$).

If $p = 0$ no page faults

If $p = 1$, every reference is a fault

Effective access time (EAT) = $(1 - p)$ × Memory Access Time + p × Page fault time.

NOTE: Page fault time = page fault overhead + swap-out + swap-in + restart overhead

The performance of a virtual memory management system depends on the total number of page faults, which depend on "*paging policies*" and "*frame allocation*".

Frame Allocation

Number of frames allocating to each process is either static or dynamic.

- **Static allocation:** The number of frames allocated to a process is fixed.
- **Dynamic allocation:** The number of frames allocated to a process changes.

Paging Policies

- **Fetch policy:** It decides when a page should be loaded into memory.
- **Replacement policy:** It decides which page in memory should be replaced.
- **Placement policy:** It decides where in memory should a page be loaded.

7.3 Page Replacement

When there is a page fault, the referenced page must be loaded. If there is no available frame in memory one page is selected for replacement.

Steps to be followed to replace a page:

- **Find a free frame:** If there is a free frame, use it.
- If there is no free frame, use page replacement algorithm to select a victim frame.
- Swap-out(write) the victim page to the disk and update the page table and frame tables accordingly.
- Swap-in(Read) the desired page into the memory where free frame is available and change the page and frame tables accordingly.
- Restart the user process.

Page Reference

A page reference string is a sequence of page numbers in order of reference. A subsequence of the same page number can be reduced to a single occurrence.

Example: The following string has 14 page numbers.

< 3,6,2,1,4,7,3,5,8,9,2,8,10,7 >

7.4 Page Replacement Algorithms

A page replacement algorithm determines how the victim page (the page to be replaced) is selected when a page fault occurs. The aim is to minimize the page fault rate.

The efficiency of a page replacement algorithm is evaluated by running it on a particular string of memory references and computing the number of page faults. The static paging algorithms implement the replacement policy when the frame allocation to a process is fixed.

Static page replacement algorithms are:

- First-in-first-out (FIFO) replacement
- Optimal replacement
- Least recently used (LRU) replacement
- Second chance (Clock)
- Counting (LFU/MFU)

7.4.1 FIFO Algorithm

When a page fault occurs and there are no empty frames for the process, the page selected to be replaced is the one that has been in memory the longest time(the oldest page).

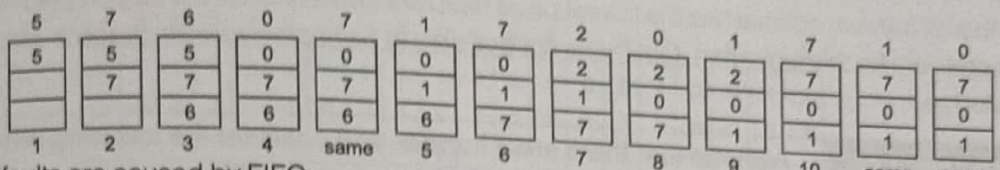
This selection of the page to replace is completely independent of the locality of the process. The oldest page may be needed again soon, some page may be important. It will get old, but replacing it will cause an immediate Page Fault.

FIFO can be implemented using either Queue or a time-stamp on pages.

Example-7.1 Assume that there are 3 frames, and consider the following reference string. Find the number of page faults using FIFO page replacement algorithm.

5, 7, 6, 0, 7, 1, 7, 2, 0, 1, 7, 1, 0

Solution:



10 page faults are caused by FIFO

Example-7.2 Assume that there are 3 frames, and consider the following reference string. Find the number of page faults using FIFO page replacement algorithm.

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

Solution:

There are 15 page faults and are shown in the below diagram.

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
7	7	7	2	2	2	2	4	4	4	0	0	0	0	0	0	0	7	7	7
	0	0	0	0	3	3	3	2	2	2	2	2	1	1	1	1	1	0	0
		1	1	1	1	0	0	0	3	3	3	3	3	2	2	2	2	2	1
X	X	X	X		X	X	X	X	X	X			X	X			X	X	X

Example-7.3 Assume that there are 3 frames, and consider the following reference string. Find the number of page faults using FIFO page replacement algorithm.

A, B, C, D, A, B, E, A, , B, C, D, E

Solution:

	A	B	C	D	A	B	E	A	B	C	D	E	
FIFO	A	A	A	D	D	D	E	E	E	E	E	E	9 faults
		B	B	B	A	A	A	A	A	C	C	C	
			C	C	C	B	B	B	B	B	D	D	

Belady's Anomaly

If number of frames increasing, the number of page faults may also increase. This anomaly is called as belady's anomaly. FCFS suffers with belady's anomaly.

FIFO with 3 page frames

0	1	2	3	0	1	4	0	1	2	3	4
0	1	2	3	0	1	4	4	4	2	3	3
	0	1	2	3	0	1	1	1	4	2	2
		0	1	2	3	0	0	0	1	4	4
P	P	P	P	P	P	P			P	P	

9 Page faults

FIFO with 4 page frames

0	1	2	3	0	1	4	0	1	2	3	4
0	1	2	3	3	3	4	0	1	2	3	4
	0	1	2	2	2	3	4	0	1	2	3
		0	1	1	1	2	3	4	0	1	2
			0	0	0	1	2	3	4	0	1
P	P	P	P			P	P	P	P	P	P

10 Page faults

In above example, when number of frames are increased the number of page faults are also increased.

7.4.2 Optimal (MIN) Algorithm

The approach used is to replace the page in memory that will not be used for the longest period. The optimal algorithm for page replacement requires knowledge of the entire page reference stream in advance. For a fixed number of frames, optimal has the lowest page fault rate between all of the page replacement algorithms, but there is problem for this algorithm. Optimal is not possible to be implemented in practice. Because it requires future knowledge.

Example-7.4 Assume that there are 3 frames, and consider the following reference string. Find the number of page faults using optimal page replacement algorithm.

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

Solution:

There are 9 page faults and are shown in the below diagram.

	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
	7	7	7	2	2	2	2	2	2	2	2	2	2	2	2	2	2	7	7	7
		0	0	0	0	0	0	4	4	4	0	0	0	0	0	0	0	0	0	0
			1	1	1	3	3	3	3	3	3	3	3	1	1	1	1	1	1	1
	X	X	X	X		X		X			X			X				X		

Example-7.5 Assume that there are 3 frames, and consider the following reference string. Find the number of page faults using optimal page replacement algorithm.

1, 2, 3, 2, 1, 5, 2, 1, 6, 2, 5, 6, 3, 1, 3, 6, 1, 2, 4, 3

Solution:

	1	2	3	2	1	5	2	1	6	2	5	6	3	1	3	6	1	2	4	3
	1	1	1	1	1	1	1	1	6	6	6	6	6	6	6	6	6	2	2	2
		2	2	2	2	2	2	2	2	2	2	2	2	1	1	1	1	1	4	4
			3	3	3	5	5	5	5	5	5	5	3	3	3	3	3	3	3	3
	x	x	x			x			x				x	x				x	x	

9 Page faults with Optimal

7.4.3 LRU Algorithm

The least recently used (LRU) replacement algorithm replaces the page that has not been used for the longest period (least recently referenced page). The assumption is that recent page references give a good estimation of page references in the near future. LRU can be implemented using Stack or Counter.

Example-7.6 Assume there are 3 frames, and consider the following reference string. Find the number of page faults using LRU page replacement algorithm.

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

Solution:

	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
	7	7	7	2	2	2	2	4	4	4	0	0	0	1	1	1	1	1	1	1
		0	0	0	0	0	0	0	0	3	3	3	3	3	3	0	0	0	0	0
			1	1	1	3	3	3	2	2	2	2	2	2	2	2	2	7	7	7
	X	X	X	X		X		X	X	X	X			X		X		X		

12 page faults occurred.

Example-7.7 Assume that there are 3 frames, and consider the following reference string. Find the number of page faults using LRU page replacement algorithm.

1, 2, 3, 2, 1, 5, 2, 1, 6, 2, 5, 6, 3, 1, 3, 6, 1, 2, 4, 3

Solution:

1	2	3	2	1	5	2	1	6	2	5	6	3	1	3	6	1	2	4	3
1	1	1	1	3	2	1	5	2	1	6	2	5	6	6	1	3	6	1	2
	2	2	3	2	1	5	2	1	6	2	5	6	3	1	3	6	1	2	4
		3	2	1	5	2	1	6	2	5	6	3	1	3	6	1	2	4	3
x	x	x			x			x		x		x	x				x	x	x

Number of page faults = 11.

7.4.4 LRU Approximations

1. Not Recently Used(NRU)

- It is an approximation to LRU.
- Select one of the pages that has not been used recently.
- It uses "used bit" or "reference bit".

Each page is associated with reference/used bit. When a page is brought into memory, its reference bit is set to 0. whenever a page is referenced, its reference bit is set to 1.

NRU algorithm:

```
While(page is not stored)
{
     if scan pointer is at frame whose used bit is 0
     {    select it
          "store new page "
          set used bit to 1
     }
     else
        set used bit to 0
        move scan pointer to the next frame (wrap around
        to first frame if at the last frame)
}
```

NOTE: Many reference bits also can be used for NRU algorithms

Example-7.8 Assume there are 5 frames, and consider the following reference string. Find the number of page faults using NRU page replacement algorithm.

3, 2, 3, 0, 8, 4, 2, 5, 0, 9, 8, 3, 2

Solution:

3		2		3		0		8		4		2		5		0		9		8		3		2			
P	U	P	U	P	U	P	U	P	U	P	U	P	U	P	U	P	U	P	U	P	U	P	U	P	U	P	U
0	•	3	1	3	1	3	1	3	1	3	1	3	1	3	1 •	3	1 •	5	1	5	1	5	1	5	1 •	5	0
0			0 •	2	1	2	1	2	1	2	1	2	1	2	1	2	1	2	0 •	2	0 •	9	1	9	1	9	0
0			0		0 •		0 •	0	1	0	1	0	1	0	1	0	0	0	1	0	1 •	0	1 •	0	0	2	1
0			0		0		0		0 •	8	1	8	1	8	1	8	0	8	0	8	0	8	1	8	0	8	0 •
0			0		0		0		0		0 •	4	1	4	1	4	0	4	0	4	0	4	0	3	1	3	1

P is page and U is used bit. Number of page faults = 9.

2. Second Chance Replacement Algorithm (CLOCK)

First time when page is referenced, its reference bit set to 1. When a page gets a second chance, its reference bit is cleared. It works based circular queue of page frames in FIFO order.

The frame table is viewed as a circular queue with a pointer. When a page faults occur, pointer advances until it finds a frame whose reference bit is 0. As it advances, it resets each reference bit to 0. A frame with reference bit 1 is given a second chance.

CLOCK algorithm:

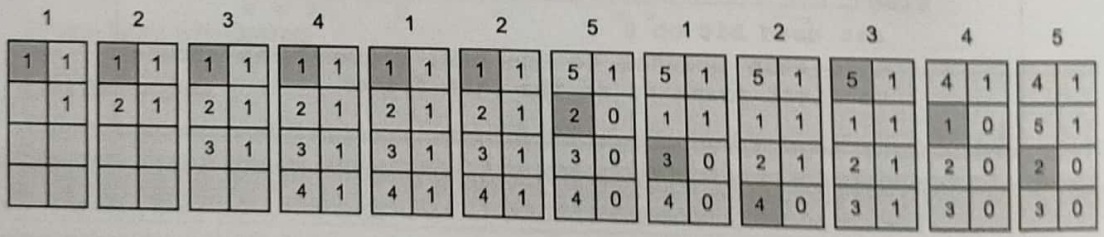
```
While(page is not stored)
{
   if reference bit is 0
   { "store new page " (replace)
     set reference bit to 1
   }
      Else if reference bit is 1
   {
      set used bit to 0 // Give second chance to this page
   }
      Increment the pointer in circular queue
}
```

NOTE: The clock algorithm uses only one bit of information: whether the page was accessed since the last time the operating system looked at it.

Example-7.9 Consider the following page reference string with 4 available empty frames of main memory. Find the number of page faults using CLOCK algorithm.

1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

Solution:



10 Page faults

3. Enhanced Second Chance Algorithm

The second-chance algorithm can be enhanced by considering both the reference bit and the modify bit. Let the pair (reference R, modify M) is used with two bits to indicate reference and modify bits.

(R, M) = (0, 0), neither recently used nor modified
= (0, 1), not recently used but modified
= (1, 0), recently used but not modified
= (1, 1), recently used and modified

Perform up to four passes over the circular queue, considering pages in each class at a time (first to fourth choices). Choose the first page encountered in the lowest nonempty class.

- If (R, M) = (0, 0), best to replace: first choice
- (R, M) = (0, 1), second choice
- (R, M) = (1, 0), third choice
- (R, M) = (1, 1), fourth choice

If victim page is not found then swap out the unreferenced dirty page onto the disk and replace this page.

4. Nth Chance Replacement

This approach is similar to the Second Chance algorithm. The variation is that we maintain a counter along with a "referenced" bit.

On a page fault, we advance our pointer (clock hand) to point to the next page frame. We then check the "referenced" bit for that page. If it is 1, we clear the bit and set the counter to 0. If it is 0, we increment the counter. If counter < N, go on. Otherwise, we replace this page.

It is counting the number of times that we examined this page and it has *not* been referenced. The value of *N* has to be tuned to a specific environment and serves as a threshold for determining whether a page should be evicted because it has not been referenced after a number of page faults. The Nth Chance replacement algorithm gives us a better approximation of LRU Than second chance.

7.4.5 Counting Algorithms

There are many other algorithms that can be used for page replacement.

LFU Algorithm

The least frequently used (LFU) page-replacement algorithm requires that the page with the smallest count be replaced. This algorithm suffers from the situation in which a page is used heavily during the initial phase of a process, but then is never used again.

MFU Algorithm

The most frequently used (MFU) page-replacement algorithm is based on the argument that the page with the smallest count was probably just brought in and has yet to be used.

7.5 Frame Allocation

	Local Replacement	Global Replacement
Fixed Allocation	• Fixed number of frames allocated to a process • Page is replaced from set of its allocated frames • If allocation is too small, there will be a high page fault rate • If allocation is too large there will be too few programs in main memory • One replacement clock or queue per process	Not possible
Variable Allocation	• Number of frames allocated changes from time to time • Page is replaced from set of its allocated frames • One global clock or queue for all processes • When new process added, allocate number of page frames based on application type, program request, or other criteria • Re-evaluate allocation from time to time	• Page is replaced from all available frames in main memory. • Easiest to implement • Adopted by many operating systems • Operating system keeps list of free frames • Free frame is added to resident set of process when a page fault occurs • If no free frame, replaces one from another process

- Fixed-allocation: Gives a process a fixed number of pages within which to execute. When a page fault occurs, one of the pages of that process must be replaced.
- Variable-allocation: Number of pages allocated to a process varies over the lifetime of the process.
- Local Replacement: Replace the page only from its own set of allocated frames.
- Global Replacement: Replace the page from the set of all frames.

7.5.1 Frame Allocation Problem

A process should allocate enough frames for its current locality. If more frames are allocated, there is little improvement. If less frames are allocated, there is very high page fault rate. So low CPU utilization. The OS attempts to increase the degree of multiprogramming. Thrashing may occur (a process spends most or all of its time swapping pages)

Allocation schemes are mainly two types:

1. Equal allocation: If there are n frames and p processes, n/p frames are allocated to each process.
2. Proportional allocation (based on size, priority, etc.):
   (a) Sized Allocation: Let the virtual memory size for process p be $v(p)$. Let there are m processes and n frames. Then the total virtual memory size will be: $V = S \times v(p)$.
   Allocate $(v(p) / V) \times n$ frames to process p.
   (b) Priority Allocation: Use a proportional allocation scheme using priorities rather than sizes. If process P_i generates a page fault, select for replacement one of its frames and the frame from a process with lower priority.

Example-7.10 Consider a system having 64 frames and there are four processes with the following virtual memory sizes: $v(1) = 16$, $v(2) = 128$, $v(3) = 64$ and $v(4) = 48$. Find the frame allocation using equal and proportional allocations.

Solution:

(a) Equal Allocation: Assume that there are n frames, and p processes, then n/p frames are allocated to each process allocates 64/4 = 16 frames to each process.

(b) Proportional Allocation:

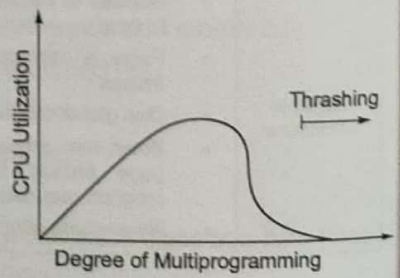
$$V = 16 + 128 + 64 + 48 = 256$$

It allocates:
$(16 / 256) \times 64 = 4$ frames to process 1
$(128 / 256) \times 64 = 32$ frames to process 2
$(64 / 256) \times 64 = 16$ frames to process 3
$(48 / 256) \times 64 = 12$ frames to process 4

7.5.2 Thrashing

Thrashing is a situation which occur when a process is spending more time in paging than executing. CPU has little to do but there is heavy disk traffic moving pages to and from memory with little use of those pages.

When CPU utilization is low, the OS may increase the degree of multiprogramming and cause other processes to thrash. A thrashing process can cause other processes to thrash if a global page replacement strategy is used.



Causes of Thrashing

(*i*) High degree of multi programming
(*ii*) Lack of frames

Recovering of Thrashing

(*i*) Instruct the long-term scheduler not be bring the processes in the system after point of maxima.
(*ii*) If the system is already in thrashing then instruct the mid-term scheduler to suspend some of the process, so that we can recover the system from thrashing.

Example-7.11 Consider the system with following parameters (i) CPU utilization 20% (ii) Paging disk 80%. Which of the following will improve CPU utilization?

(i) Install faster CPU
(ii) Install bigger disk
(iii) Increase degree of multiprogram
(iv) Decrease degree of multiprogram
(v) Install more main memory
(vi) Decrease page size

Solution:

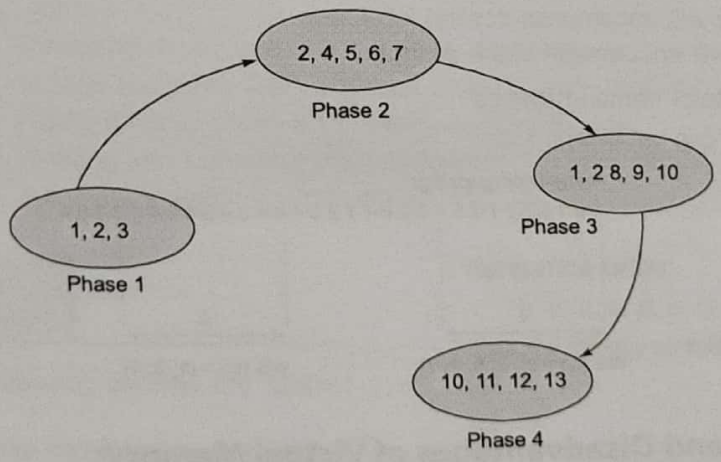
With given parameters system is in thrashing. Hence only (iv) and (v) will improve CPU utilization.

7.5.3 Locality

Locality is a property of the page reference string (property of programs), A set of pages that are actively used together.

- **Temporal locality:** Pages that have been used recently are likely to be used again.
- **Spatial locality:** Pages near to those that have been used are likely to be used next. That is once a location is referenced, a nearby location is often referenced soon.

Example of locality:



Remember

Why does Paging Work?

Answer: The locality model is used as the set of pages used for that particular phase of computation. A process migrates from one locality to the next, as the process moves to a different phase of computation. Localities may overlap.

Why does thrashing occur?

Answer: The size of a locality is greater than the allocated memory (in frames). To stop thrashing, each active process should be allocated enough frames for its current locality.

7.6 Dynamic Paging Algorithms

The Static paging algorithms assume that a process is allocated a fixed amount of frames from the start of execution. Dynamic paging algorithms attempt to adjust the memory allocation to match the process' needs as it executes.

The *Working Set algorithm* is the best known algorithm for dynamic paging.

7.6.1 Working Set

The working set for each model is the *set of pages referenced by the process during the most recent w page references*. It directly address the thrashing problem. Before starting a process, make sure it's working set is in main memory. Restrict the number of processes in the ready list so that all can have their working set of pages in memory. Let Δ be the sum of the sizes of the working sets of all active processes.

- If $\Delta <$ (total number of frames) then the OS can allow more active processes.
- If $\Delta >$ (total number of frames) then the OS must suspend some active processes, otherwise thrashing occurs.

$\Delta \equiv$ working-set window $\equiv$ a fixed number of page references.

A Strategy to Control Thrashing

- Set the lower and upper bounds of page fault rate for each process.
- Establish "acceptable" page fault rate:
  (a) If actual rate is lower than lower bound, decrease the number of frames.
  (b) If actual rate is larger than the upper bound, increase the number of frames.

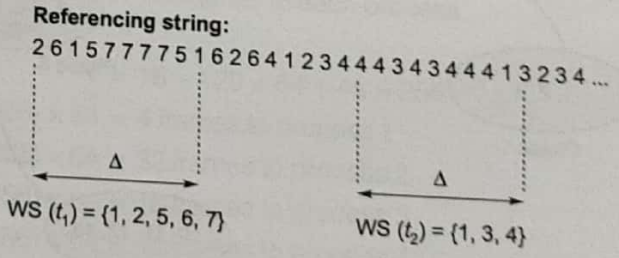
Example

WSS_i (working set of Process P_i) = total number of pages referenced in the most recent Δ (varies in time)

- If Δ too small will not encompass entire locality
- If Δ too large will encompass several localities
- If $\Delta = \infty \Rightarrow$ will encompass entire program

$\Delta = \Sigma\ WSS_i$ total demand frames

WS (t_i) is working set at time t_i.



7.7 Advantages and Disadvantages of Virtual Memory

Advantages of Virtual memory

- A job's size is no longer restricted to the size of main memory (or the free space within main memory).
- Memory is used more efficiently because the only sections of a job stored in memory are those needed immediately while those not needed remain in secondary storage.
- It allows unlimited amounts of multiprogramming.
- It eliminates external fragmentation and minimizes internal fragmentation by combined segmentation and paging.
- It allows for sharing of code and data.

Disadvantages of Virtual memory

- Increased hardware costs
- Increased overheads for handling paging interrupts.
- Increased software complexity to prevent thrashing.

Summary

- **Virtual memory:** A technique that allows programs to be executed even though they are not stored entirely in memory.
- **Demand paging:** A memory allocation scheme that loads a program's page into memory at the time it is needed for processing.
- **Segmented/demand paged memory allocation:** A memory allocation scheme based on the concept of dividing a user's job into logical groupings of code and loading them into memory as needed to minimize fragmentation.
- **Page fault:** A type of hardware interrupt caused by a reference to a page not residing in memory. The effect is to move a page out of main memory and into secondary storage so another page can be moved into memory.
- **Page replacement policy:** An algorithm used by virtual memory systems to decide which page or segment to remove from main memory when a page frame is needed and memory is full.
- **First-in first-out (FIFO) policy:** A page replacement policy that removes from main memory the pages that were brought in first.
- **least recently used (LRU) policy:** A page-replacement policy that removes from main memory the pages that show the least amount of recent activity.
- **Clock page replacement policy:** A variation of the LRU policy that removes from main memory the pages that show the least amount of activity during recent clock cycles.
- **Thrashing:** A phenomenon in a virtual memory system where an excessive amount of page swapping back and forth between main memory and secondary storage results in higher overhead and little useful work.
- **Working set:** A collection of pages to be kept in main memory for each active process in a virtual memory environment.

Student's Assignment

Q.1 Which of the following defines the spatial locality?

(a) If any memory location is accessed, will also accesses the memory near to it.
(b) If any memory location is accessed, the same location will be accessed again soon.
(c) Both (a) and (b)
(d) Neither (a) nor (b)

Q.2 If a system has 3 frames and following references are given in demand paging, find the number of page faults using FIFO.

Reference order:

e, d, h, b, d, e, d, a ,e, b, e, d, e, b, g

Assume initially all three frames are empty.

(a) 7 (b) 9
(c) 11 (d) 13

Q.3 If a system has 3 frames (empty) and following references are given in demand paging, find the total number of pages for which no page fault occurs using LRU page replacement policy.

Reference order:

e, d, h, b, d, e, d, a, e, b, e, d, e, b, g

(a) 5 (b) 6
(c) 7 (d) 8

Q.4 Assume that the reference stream of virtual pages 1, 2, 3, 4, 5, 3, 1, 2, 3, 4, 6. If LRU replacement policy is used then how many minimum number of memory frames required in the system such that the number of page faults to be no larger than 6?

(a) 4 (b) 5
(c) 6 (d) None of these

Q.5 Assume that we have a demand-paged memory. The page table is held in registers. It takes 8 milliseconds to service a page fault if an empty page is available or the replaced page is not modified, and 20 milliseonds if the replaced page is modified. Memory access time is 100 ns. Assume that the page to be replaced is modified 70 percent of the time. What is the maximum acceptable page-fault rate for an effective access time of no more than 200 ns?

(a) 6.1×10^{-6} (b) 7.3×10^{-6}
(c) 3.4×10^{-4} (d) None of these

Q.6 Demand paging uses a second chance page replacement policy (clock). It uses one use bit to give every page one more chance in FIFO replacement. Whenever a page is referenced its use bit set to 0. Whenever it is need to replace by other page first time its use bit is changed to 1 and next page will be searched for replacement. If already given the chance then it will be replaced. Assume system has 3 page frames. Consider the following page reference stream.

7, 0, 1, 2, 0, 3, 0, 4, 2, 3 (in order)

How many page faults occur using clock algorithm?

(a) 4 (b) 5
(c) 6 (d) 7

Q.7 In a virtual memory system

(a) each virtual address space is much smaller than the real address space
(b) determining how to map virtual addresses to physical addresses is of great importance
(c) MMUs convert physical addresses to virtual addresses during execution
(d) when the system is ready to run a process, the system loads the process's code and data from main memory into cache

Q.8 Virtual memory fetch strategies determine when a page or segment should be moved from _____ to _____

(a) main memory, the TLB
(b) secondary storage, main memory
(c) main memory, secondary storage
(d) the TLB, registers

Q.9 Virtual memory replacement strategies determine

(a) when the system should update page or segment table entries
(b) how many pages should be added to main memory
(c) which pages should be brought into memory because a process is likely to reference them soon
(d) which page or segment to remove to provide space for an incoming page or segment

Q.10 LRU replaces the page that has spent the

(a) longest time in memory
(b) longest time in memory without being referenced
(c) shortest time in memory
(d) None of these

Q.11 The optimal page replacement algorithm will select the page that

(a) has not been used for the longest time in the past
(b) will not be used for longest time in the future
(c) has been used least number of times
(d) has been used most number of times

Q.12 A memory page containing a heavily used variables that was initialized very early and in constant used is removed when

(a) LRU page replacement algorithm is used
(b) FIFO page replacement algorithm is used
(c) LFU page replacement algorithm is used
(d) none of the above

Q.13 Which of the following is an advantage of virtual memory?

(a) faster access to memory on an average
(b) process can be given protected address spaces
(c) linker can assign address independent of where program will be loaded in memory.
(d) none of these

Q.14 Choose the correct statement

(a) paging is virtual storage and segmentation is real storage.
(b) both paging and segmentation are forms of contiguous storage allocation.
(c) both paging and segmentation are popular schemes for virtual storage.
(d) None of these

Q.15 Consider a logical address space of eight pages of 1024 words each mapped on to a physical memory of 32 frames. How many bits are there in physical address?

(a) 13 (b) 15
(c) 10 (d) 14

Q.16 Assuming pages of size 128 words each and array is stored in row major. How many page faults will be generated by the following C program:

```
int A [128] [128];
for (int j = 0; j < 128; j++)
for (int i = 0; i < 128; i++)
A [i] [j] = 0;
```

(a) 128 (b) 16384
(c) 0 (d) None of these

Answer Key:

1. (a)	**2.** (c)	**3.** (b)	**4.** (b)	**5.** (a)
6. (d)	**7.** (b)	**8.** (b)	**9.** (d)	**10.** (b)
11. (b)	**12.** (b)	**13.** (b)	**14.** (c)	**15.** (b)
16. (b)				

CHAPTER 08

File System

8.1 Introduction

Basic concepts of file:

- File is a sequence of records.
- A block contains one or more records specific to one file only.
- In Spanned organization, records can cross block boundaries.
- In Unspanned organization, records can't cross block boundaries.
- Blocking Factor = Number of records per block.
- Files can be data or programs.
- File can be simple or complex (plain text, or a specially-formatted file).
- The structure of a file is determined by both the OS and the program that creates it.
- Files are stored in a *file system*, which may exist on a disk, tape or in main memory.

Attributes of File

- filename
- file type (maybe)
- location – where is it on the device
- size
- protection/permissions (maybe)
- timestamp, ownership
- directory information

Operations on File

- create
- write/append
- read
- seek (reposition within file)
- delete
- truncate
- open, close

8.2 Directories

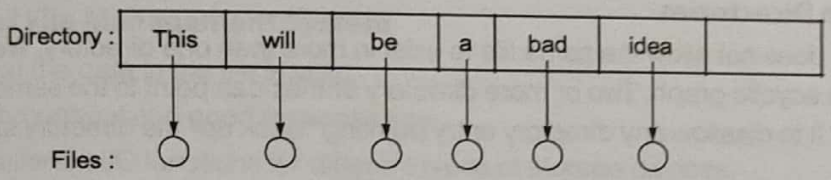
A listing of the files on a disk is a *directory*. Both the directory structure and the files reside on the disk. The directory may store some or all of the file attributes we discussed. A directory should be able to support a number of common operations:

- search for a file
- create a file
- delete a file
- rename a file
- listing of files
- file system traversal (cd)

There are many ways to organize a directory, with different levels of complexity, flexibility, and efficiency.

8.2.1 Single Level Directory

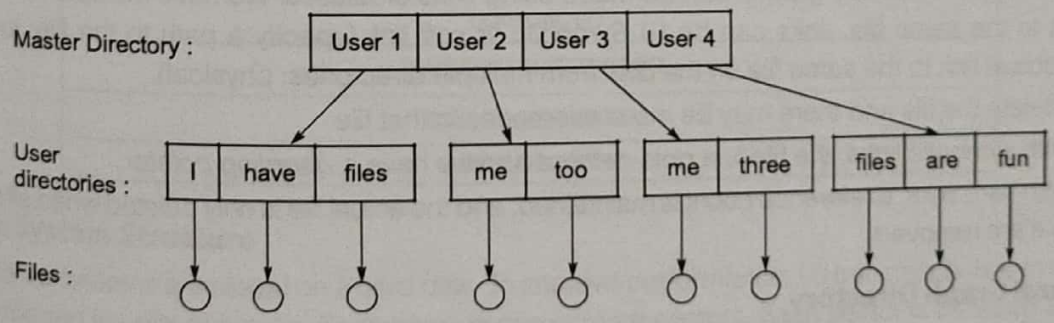
The simplest method is to have one big list of all files on a disk.



This can be used for a simple system.

- cannot have two files with the same name
- could be necessary for multiple users/programs on a disk
- no way to group files
- just one big list
- searches need to look through the entire directory

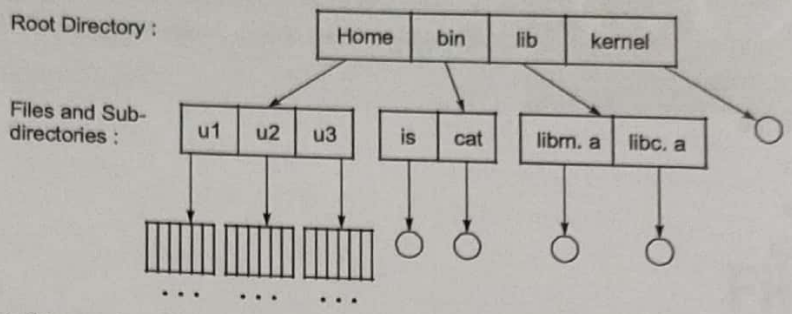
8.2.2 Two Level Directory



We can create a separate directory for each user.

- files now have a path name /user1/have
- different users can have the same file name (/user2/me and /user3/me)
- searching is more efficient, as only one user's list needs to be searched
- but still no grouping capability for a user's files

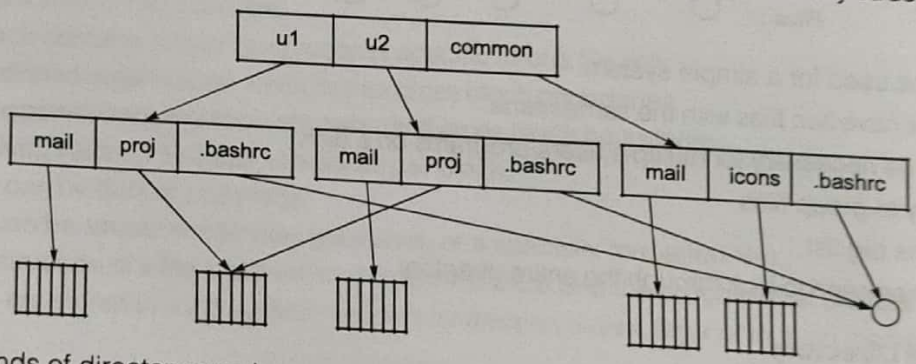
8.2.3 Tree structured Directory



Any directory entry can be either a file or a *subdirectory*. Files can be grouped appropriately. Searching is more efficient in this directory structure and concept of a *current working directory* is used. It operates on files in current directory by default or specify path, either *relative* or *absolute*. Need to be able to create and remove directories as well as files.

8.2.4 Acyclic-Graph Directories

The tree model does not allow the same file to exist in more than one directory. We can provide this by making the directory an acyclic graph. Two or more directory entries can point to the same subdirectory or file, but (for now) we restrict it to disallow any directory entry pointing "back up" the directory structure.



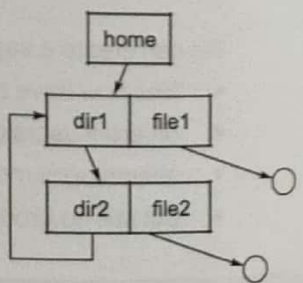
These kinds of directory graphs can be made using *links* or *aliases*. We have multiple names for and multiple paths to the same file. links can be (*i*) *Symbolic* or *soft link* (specify a path to the file: logical) and (*ii*) *Hard link* (actual link to the same file on the disk from multiple directories: physical).

If we delete the file and there may be more references to that file

- With symbolic links, the file just gets deleted and we have a *dangling pointer*.
- With hard links, a reference count is maintained, and the actual file is only deleted when all references to it are removed.

8.2.5 General Graph Directory

All directories except the system's root directory have a special entry , that indicates the parent directory, and an entry . that indicates the current directory. But they also allow links to be created back up the chain of the directory structures, potentially introducing cycles in the directory graph.



When general graph directories are allowed, we need to be careful with command like search a directory and its subdirectories for something. The search is infinite if cycles are followed.

8.2.6 Directory Implementation

An individual subdirectory will typically contain a list of files. It can be implemented with following approaches:

Linear list: It contain list of names, each of which has a pointer to the file's data blocks. It requires a costly search on large directories.

Hash Table: It is hashed linear list, decrease search time, but more complex to implement.

8.3 File Management System

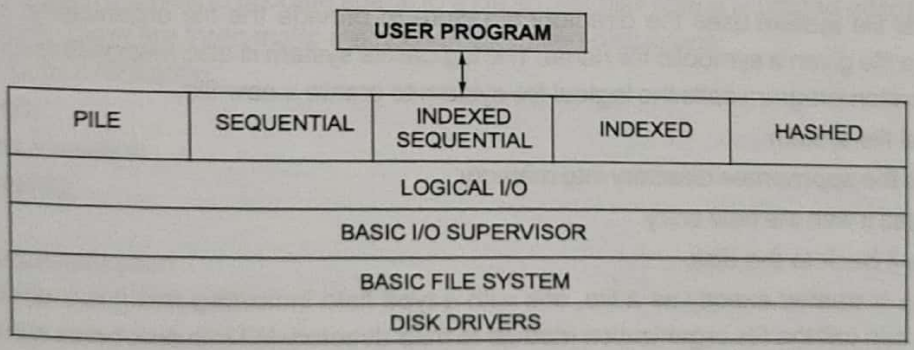
File management system is a set of software modules. It provides services to the user and application programs concerning files. Typically file management system is the only medium through which user or application program can access files.

The supreme achievement of the file management is that it has relieved the burden of the user or application, because otherwise user or application program has to developed programs for different applications to access files.

8.3.1 Objectives of File Management System

- Ensure that the data in the file is valid.
- Achieve throughput and good response time.
- Support different I/O functions for different types of storage devices.
- Eliminate /minimize to the extent possible the potential for lost/destroyed data.
- Provide a standardized set of I/O interface routines.
- Provide I/O support for multiple users in a multi user system

8.3.2 File System Architecture



8.3.3 File System Structure

Most file systems are stored on a hard disk. To improve performance, I/0 transfers between memory and disk are performed in units of blocks. Each block is one or more sectors. Each sector is between 32 bytes and 4096 bytes; usually 512 bytes.

Why are disks the primary medium for storing multiple files?

- They can be rewritten in place; it is possible to read a block from the disk, to modify the block and to write it back into the same place.
- We can access directly any given block of information on the disk; thus it is simple to access any file either sequentially or randomly.

8.3.4 File System Implementation

The file system provides the mechanism for on-line storage and access to both data and programs. The file system resides permanently on secondary storage The main requirement of a file system is that it must be able to hold a large amount of data permanently. Issues associated with file system implementation include:

- Disk space allocation
- Free space recovery
- Tracking the location of data
- Performance

8.4 File System Organization

To provide an efficient and convenient access to the disk, the operating system imposes a file system to allow data to be stored, located and retrieved easily. A file system poses two quite different design problems:

- How should the file system look to the user?
- What algorithms and data structures must be created to map the logical file system onto the physical secondary-storage devices?

The file system is composed of many different levels:

- Application programs
- Logical file system
- File-organization module
- Basic file system
- I/O Control and Devices

Logical File System

The *logical file system* uses the directory structure to provide the file organization module with the information about a file given a symbolic file name. The logical file system is also responsible for protection and security. An application program calls the logical file system to create a new file.

The logical file system:

- Reads the appropriate directory into memory
- Updates it with the new entry
- Writes it back to the disk

A directory is treated exactly as a file, one with a type field indicating that it is a directory. Thus, the logical file system can call the file organization module to map directory I/O into disk-block numbers.

These get passed to the basic file system and onto the I/O control system. Once the directory has been updated, the logical file system can use it to perform I/O. When a file is opened, the directory is searched for the desired file entry.

This file entry is saved in an in-memory table called the open file table. The index into this table is returned to the user and all subsequent references are made through this index (called a file descriptor). The directory could be search each time any I/O request was made (read, write etc) but that would less efficient. All changes to the directory entry are made to the open-file table in memory.

When the file is closed by all users that have opened it, the updated entry is copied back to the disk-based directory structure. Under UNIX, information about an open file is stored in multiple levels of in-memory tables.

I/O Control and Devices

The *I/O control* consists of device drivers and interrupt handlers. A device driver translates commands such as "retrieve block 123" into low-level hardware specific instructions used by the hardware controller which interfaces the I/0 device to the system.

File Organization Module

The *file-organization module* translates logical block addresses to physical block addresses for the basic file system to transfer. Each file's logical blocks are numbered from 0 to N and are mapped to physical blocks allocated from a global pool.

Free Space Manager

The *free-space manager* is also part of the file-organization module. The free-space manager tracks unallocated blocks and provides these blocks to other parts of the file-organization module when requested.

File System Mounting

A file system must be mounted before it can made available to processes on the system.

- The OS is given the name of the device and the location within the file system at which to attach the file system (called the mount point).
- The OS verifies that the device contains a valid file system.
- This is done asking the device driver to read the device directory and verifying that the directory has the expected format.
- The OS notes in its directory structure that a file system is mounded at the specific mount point.

This scheme enables the OS to traverse its directory structure, switching among file system as appropriate.

8.5 File Allocation Methods

The main problem is how to allocate space to a file so that disk space is utilized effectively and file can be accessed quickly. There are three major methods of allocating disk space:

1. Contiguous Allocation
2. Extents
3. linked Allocation
4. Clustering
5. FAT
6. Indexed Allocation
7. Linked Indexed Allocation
8. Multi Level Indexed allocation
9. Inode

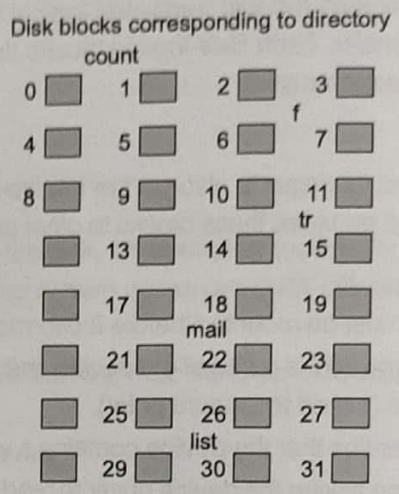
8.5.1 Contiguous Allocation

- Each file is allocated a set of contiguous disk blocks.
- The contiguous allocation method requires each file to occupy a set of contiguous blocks on the disk.
- Disk addresses define a linear ordering on the disk. With this ordering, accessing block $b + 1$ after block b normally requires no head movement.
- Contiguous allocation of a file is defined by the disk address of the first block and length (in block units).

- If the file is n blocks long, and starts at location b, then it occupies blocks $b, b+1, b+2, .. b+n-1$.
- For **sequential access:** seek to b and read and read next.
- For **direct access:** if we want block i of a file that starts at block b, seek to block $b+i$ and read.
- The directory entry for each file indicates the address of the starting block and the length of the area allocated for this file.

Directory

File	Start block	Length
Count	0	2
Tr	14	3
Mail	19	6
List	28	4
F	6	2



What happens we want to allocate more blocks to file?

We could copy the file to a space that would allow growth. If we allocate a maximum blocks to allow growth, we waste blocks. Some operating systems use a modified contiguous allocation scheme.

- A chunk of contiguous space is allocated initially.
- When that isn't enough, another chunk of contiguous space is allocated.

Advantages of Contiguous Allocation

- It is simple, directory entry needs only starting location (block number) and length (number of blocks).
- Accessing a file that has been allocated contiguously is easy.
- Supports random access into files.
- It can easily compute and read the block that contains a certain part of the file.

Disadvantages

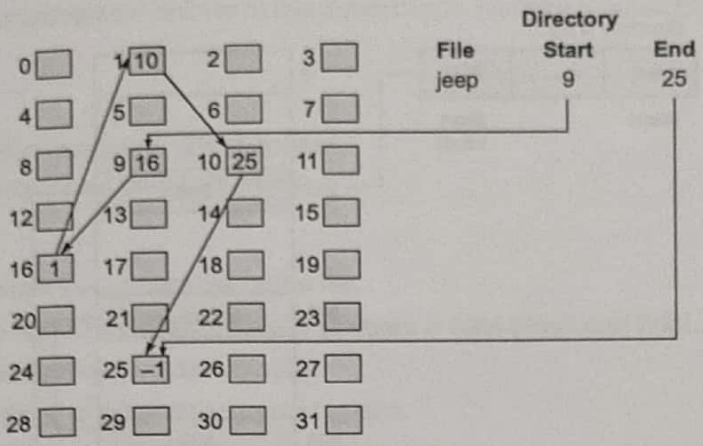
- It can lead to holes (external fragmentation).
- It may be difficult to have a file grow.
- Reading should be very efficient, since consecutive blocks of the file can be stored in consecutive blocks on the disk.

8.5.2 Extents

Files are allocated as a collection of *extents*, which are contiguous chunks of disk blocks. Each has a starting block and a size.

8.5.3 Linked Allocation

Linked allocation solves all problems of contiguous allocation. Each file is a linked list of disk blocks. The disk blocks may be scattered anywhere on the disk. Each disk block has a pointer to the next disk block in the file as well as some file data.



Advantages

- There is no external fragmentation with linked allocation.
- Any free block on the free-space list can be used to satisfy a request for a free block.
- A file can continue to grow as long as there are free blocks.
- There is no need for compaction.
- Directory entry requires only starting block.

Disadvantages

- Linked allocation doesn't support direct access.
- The space required for the pointers to next blocks.
- Need to traverse each block.
- A bad disk block causes the entire file from that block on is lost.

8.5.4 Clustering

A cluster is a collection of blocks. Clusters improve access, since most blocks are consecutive.

Disadvantages

- The cost of this approach is an increase in internal fragmentation, because more space is wasted if a cluster is partially full than when a block is partially full.
- Links on the disk can cause problems.

  What if a pointer is lost or damaged?

  Wrong blocks could be accessed or even overwritten.

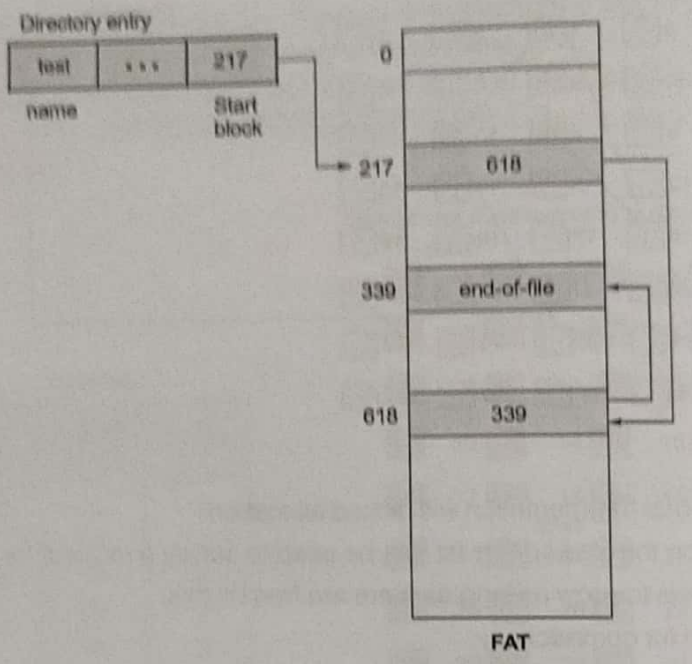
8.5.5 File Allocation Table (FAT)

An important variation on the linked allocation method is the use of a file-allocation table (FAT). This gathers the links into one table.

A section of disk at the beginning of each partition is set aside to contain the table. The table has one entry for each disk block and is indexed by block number.

Note that the FAT allocation scheme can result in a significant number of head seeks, unless the FAT is cached. Random access is optimized, because the disk head can find the location of any block by reading the information in the FAT.

Example: Used by MS-DOS and pre-NT Windows versions.



Advantages

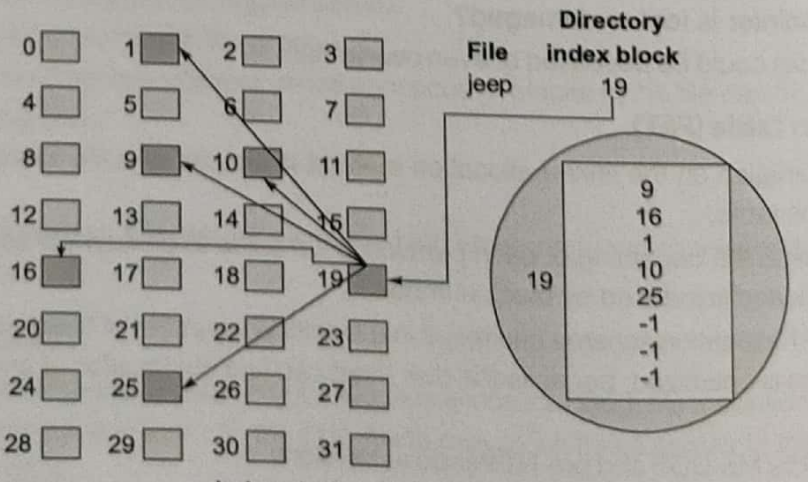
- Uses the whole disk block for data
- A bad disk block causes only that block is lost
- Somewhat better random access
- Traverses the FAT only (read disk blocks only for the data stored there)

Disadvantages

- Each disk block needs a FAT entry
- Total number of blocks, and total size of a partition is limited by the size of the FAT.
- Increased block size means fewer blocks/FAT entries, but more internal fragmentation

8.5.6 Indexed Allocation

Use disk blocks as *index blocks* that don't hold file data, but hold pointers to the disk blocks that hold file data. Indexed allocation solves the direct access problem by bringing all the pointers into one location called as index block.



Indexed allocation of disk space

Directory entry now contains a pointer to the index block. Each file's index block contains pointers to all of its data blocks.

Advantages

- Indexed allocation supports direct access.
- A bad data block costs only that block.

Disadvantages

- A Bad index block could cost the entire file.
- Size of a file is limited by the number of pointers a data block can hold.
- Even small files require two data blocks.
- Extra disk reads, and potentially wasted space.
- Indexed allocation does suffer from wasted space.
- The pointer overhead of the index block is generally greater than the pointer overhead of linked allocation.
- Suppose we have a file of only one or two blocks: (a) With linked allocation, we lose the space of only one pointer per block (one or two blocks). (b) With indexed allocation, an entire block must be allocated, even if only one or two pointers will be non-nil (non-empty).

How large should the index block be?

Since every file must have an index block, we want it to be as small as possible. But, if it is too small, it will not be able to hold enough pointers for a large file.

8.5.7 Linked Indexed Allocation

An index block is normally one disk block. To allow large files, we may link together several index blocks. An index block may contain:

- Small header giving the name of the file.
- Set of the first 100 disk-block addresses.
- Pointer to another index block, when file is larger. (use the last entry in the index block as a pointer to another index block)

Advantage: It removes file size limitations.

Disadvantage: Random access becomes a bit harder.

8.5.8 Multi-Level Index

A variant of the linked representation is to use a separate index block to point to the index blocks, which point to the file blocks themselves. To access a block in two level indexing, the operating system uses:

- The first-level index to find the second-level index and
- The second-level index to find the desired data block.

Advantage: Random access is better.

Disadvantage: All files take at least 3 blocks of space and access time.

Example 1: With 2048-byte blocks (through clustering) we can get 512 4- byte pointers into an index block. Two levels of indexes allow 4,194,304 data blocks, which allows a file of up to 8.5 gigabytes.

Example 2: File is addressed by a 256-entry index block, each of which points to a 256-entry index block, meaning we can store 65536-block or 32 MB files.

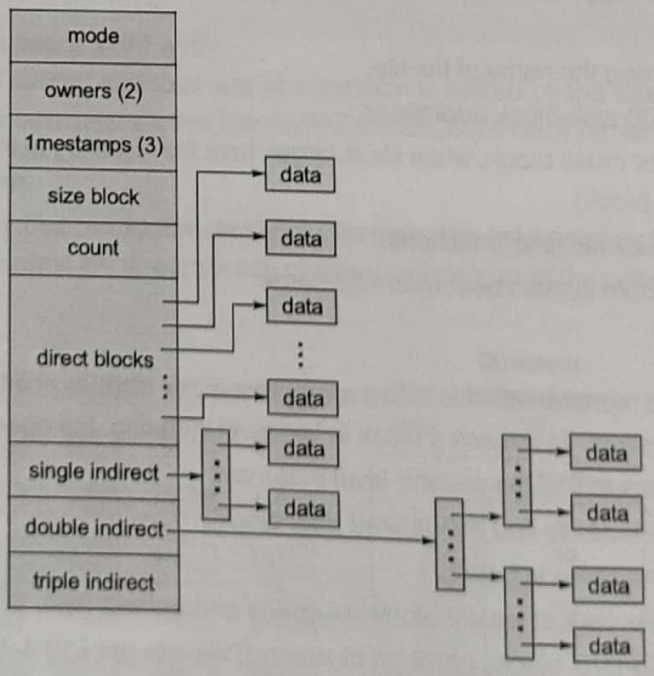
8.5.9 Inode

Each file is indexed by an *inode*. Inodes are special disk blocks they are created when the file system is created. The number of inodes limits the total number of files/directories that can be stored in the file system. The inode contains the following information:

- Administrative information (permissions, timestamps, etc.)
- A number of direct blocks (typically 12) that contain pointers to the first 12 blocks of the file.
- A single indirect pointer that points to a disk block which in turn is used as an index block, if the file is too big to be indexed entirely by the direct blocks.
- A double indirect pointer that points to a disk block which is a collection of pointers to disk blocks which are index blocks, used if the file is too big to be indexed by the direct and single indirect blocks.
- A triple indirect pointer that points to an index block of index blocks of index blocks.

INODE total size:

- Number of disk block address possible to store in 1 disk block = $\boxed{\frac{\text{DB size}}{\text{DBA}}}$ = Size of disk block/ disk block address length.
- Total size of file system = $\left[\text{\# of direct DBA's} + \underbrace{\frac{\text{DB size}}{\text{DBA}}}_{\text{\# of single indirect DBA's}} + \underbrace{\left(\frac{\text{DB size}}{\text{DBA}}\right)^2}_{\text{\# of doble indirect DBA's}} + \ldots\right] \times \text{DB size}$



- Small files need only the direct blocks, so there is little waste in space or extra disk reads in those cases. Medium sized files may use indirect blocks. Only large files make use of (and incur the overhead of) the double or triple indirect blocks, and that is reasonable since those files are large anyway. The disk is now broken into two different types of blocks: *inodes* and *data blocks*.

- There must be some way to determine where the inodes are, and to keep track of free inodes and disk blocks. This is done by a ***superblock***. Superblock is located at a fixed position in the file system. The superblock is usually replicated on the disk to avoid catastrophic failure in case of corruption of the main superblock.
- Index allocation schemes suffer from some of the same performance problems, as does linked allocation. For example, the index blocks can be cached in memory, but the data blocks may be spread all over a partition.

Disk Allocation Considerations:

- Limitations on file size, total partition size
- Internal and External fragmentations
- Overhead to store and access index blocks
- Layout of files, inodes, directories, etc, as they affect performance
- Disk head movement, rotational latency
- May want to reorganize files occasionally to improve layout (disk defragmenting, etc)

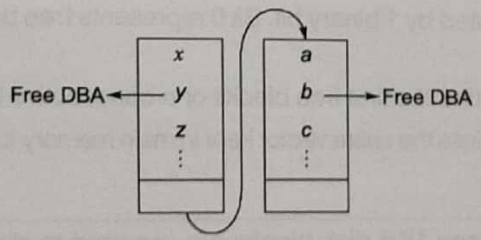
8.6 Free Space Management

Disk space is limited. It is necessary to reuse the space freed when files are deleted for new files. To keep track of free disk space, a free-space list is maintained by the system. The free-space list records all disk blocks that are free. Free blocks are any that are not allocated to some file or directory.

When a new file is created, we search the free-space list for the required amount of space and allocate that space to the new file. Any allocated space is removed from the free-space list. When the file is deleted, its disk space is added to the free-space list. A free-space list is not always implemented as a list.

8.6.1 Free-space list (Free block List)

- All disk blocks that are free maintained in a reserved portion of the disk called free space list.
- Each block is assigned a sequential number.
- List of all numbers corresponding to free blocks are maintained in a free space list.
- There are possibilities to store a part of the table in main memory by using "Stack or Queue".



In free list approach some disk blocks are used just to store the free disk block addresses.

Example-8.1 Size of disk = 20 MB, Disk block size = 1 KB, DBA block size = 16 bits. Find the number of disk blocks required.

Solution:

Step-1: Number of block address possible to store in 1 DB = $\frac{\text{Disk block size}}{\text{DBA}} = \frac{1\text{ KB}}{16\text{ bits}} = 512$

Step-2: 1 disk block can contain 512 disk block addresses.

Number of disk blocks available = $\frac{\text{Size of Disk}}{\text{DB size}}$ = 20K = 10×2^{11} disk blocks are available on disk.

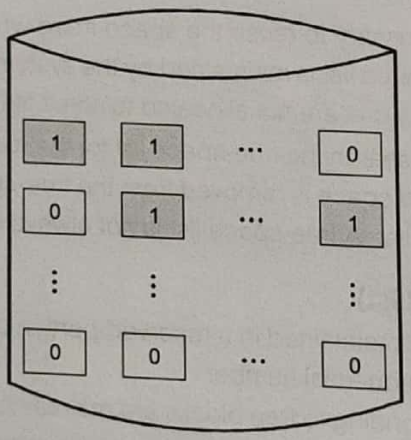
Number of disk blocks required to store all addresses = $\frac{1}{2^9} \times 2^{11} \times 10 = 2^2 \times 10 = 40$

8.6.2 Bit vector

- The free-space list is often implemented as a bit map or bit vector. Keep a vector, one bit per disk block.
- If a block is free, the corresponding bit is 1. Otherwise bit is 0 (block is in use).
- Search for a free block requires search for the first 0 bit.
- Assume the following blocks are allocated, the rest free:

  2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 17, 18, 25, 27.

  The free-space bit map would be: 00111100111111000110000001110000000...



Every disk block is represented by 1 binary bit. Bit 0 represents **free** block and bit 1 represents **used** or **busy** block.

Advantage: Efficiently can find the first free blocks or n consecutive free blocks on the disk.

Disadvantage: Inefficient unless the entire vector kept in main memory for most accesses and occasionally written to disk for recovery.

Example-8.2 **How many 1KB disk blocks are required to store 20 K bit addresses?**

Solution:

Number of disk blocks we can map in 1 disk block

Disk block size = 1 KB = 1K × 8 bits = 8K bits

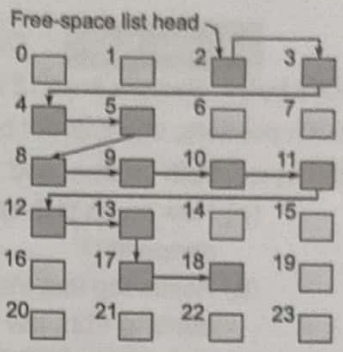
Hence we can map 8K blocks in 1 disk block

8K bits → 1 disk block

20K bit → $\frac{20K}{8K}$ = 2.5 blocks required.

8.6.3 Linked List

With linked allocation, use existing links to form a free list. Link together all the free disk blocks.



Advantage: No wasted space

Disadvantages:

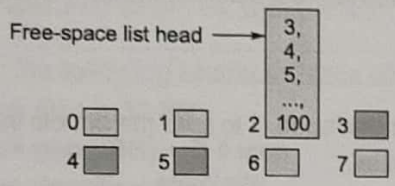
(a) It can be difficult to allocate contiguous blocks.

(b) Not efficient to traverse the list.

8.6.4 Grouping

The addresses of n free blocks are stored in the first free block. The first $(n-1)$ of these blocks are actually free. The last block (n^{th}) contains addresses of another n blocks and so on.

Advantage: Addresses of a large number of free blocks can be found quickly.



8.6.5 Counting

The free-Space list can contain pairs (block number, count). Keep the address of first free block number and the number n of free contiguous blocks that follow.

Advantage: Several contiguous blocks may be allocated or freed.

Example-8.3 Consider a file system that uses an FAT. The state of the directory is

File name	First block	Size in bytes
AA	5	1350
BB	7	2200

(The other information in the directory is omitted). The current state of the FAT is

0	FREE
1	FREE
2	–1
3	4
4	–1
5	2
6	FREE
7	3
8	FREE

The system uses a block size of 1 Kbyte. What is the last block of file BB? How many bytes of that block are used? (Useful fact: 1 K = 1024).

Solution:

The blocks of BB are 7, 3, 4. Since BB has length 2200 and blocks 7 and 3 hold 2048 bytes, there are 152 bytes used in block 4.

Example-8.4 Consider a UNIX filesystem with the following components: Disk blocks are 4096 bytes. Sectors are 512 bytes long. All metadata pointers are 32-bits long. An inode has 12 direct block pointers, one indirect block pointer and one double-indirect block pointer. The total inode size is 256 bytes. Both indirect and double indirect blocks take up an entire disk block.

(a) How much disk space, including metadata and data blocks, is needed to store a 4 GB DVD image file?

(b) Assuming that there is not a buffer cache (i.e., no filesystem structures or data are cached), starting from the inumber, how many disk accesses will be required to read only the last byte in this file? To instead overwrite it?

Solution:

(a) Requires 2^{32} bytes of actual data: 1048576 data blocks. 1024 block ptrs per indirect block.

1,048,564 blocks/1024 = 1024 indirect blocks needed. 1 double-indirect block needed.

256 bytes for inode.

Total: 4,299,165,952 bytes.

(b) To read: 1 read required for the inode, 1 to read the double indirect block, 1 to read indirect, 1 to read the data: 4 disk accesses.

To write: 4 reads, plus 1 write = 5 disk accesses.

Example-8.5 What is the maximum file size on a typical Unix system that uses an inode, an indirect pointer, and a double-indirect pointer? Assume a 4 KB block size, and 12 direct pointers within the inode, and disk addresses that are 32 bits.

Solution:

$$\text{Pointers: } 12 + 1024 + (1024)^2 = 12 \text{ direct} + \underbrace{\left(\frac{4\text{ KB}}{4\text{ bytes}}\right)}_{1024} + \underbrace{(1024 \times 1024)}_{\text{Double indirect}}$$

File size: Pointers × 4 KB

Example-8.6 Suppose that a block is 1 K Byte and that a disk address is 4 bytes. Suppose that files are implemented using *i*-nodes, and that an *i*-node has the following structure: the last address in the inode is a third-level indirect block; the second-to-last address is a second-level indirect block; the third-to-last address is an indirect block; and the remaining addresses are data blocks. Approximately how large a file can this system support?

Solution:

Each block holds 2^8 = 256 addresses (= 1 K Bytes/4 Bytes).

Therefore the third-level indirect block points to 2^8 second-level indirect blocks.

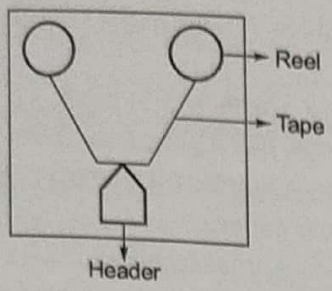
Each of these points to 2^8 indirect blocks.

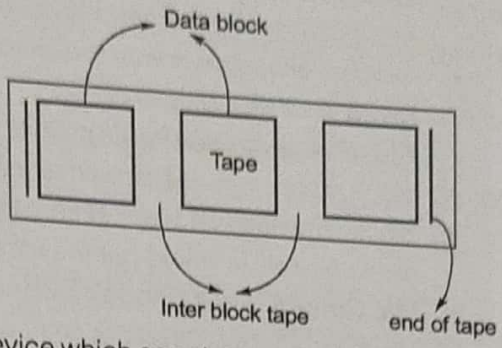
Each of these points to 2^8 data blocks.

Each of these holds 2^{10} bytes.

So the total size of the data indexed under the third-level indirect block is $2^8 \times 2^8 \times 2^8 \times 2^{10} = 2^{34}$ bytes = 16 GBytes.

8.7 Tape





- Tape is cheapest secondary storage device which can store
- Tape is generally used to take large data backup.
- The tape is divided into various data blocks and each data blocks is separated by inter block gap.
- The recording density is measured as BPI i.e. byte per inch.

Example-8.7 Consider the following characteristics of tape.
Data block size = 32 KB
Inter block gap (IBG) = 0.4 inch
Recording density = 6250 BPI
Tape length = 2400 feets
Find possible tape capacity.

Solution:
Tape length require to store 1 data block
6250 bytes can store in 1 inch

For 32 KB data we require = $\frac{32KB}{6250}$ = 5.12 inch

The tape length required to store including gap = 5.12 + 0.4 inch = 5.52 inch

Number of data blocks to store in given tape (possible) = $\frac{2400 \times 12}{5.52}$ = 5218 block

Capacity of tape = Number of blocks × Blocks size = 5218 × 32 KB ≃ 167 MB

Summary

- The structure used to describe where the file is on the disk and the attributes of the file is the file descriptor.
- **Contiguous Allocation:** OS maintains an ordered list of free disk blocks. OS allocates a contiguous chunk of free blocks when it creates a file. Need to store only the start location and size in the file descriptor.
- **Linked files:** Keep a list of all the free sectors/blocks. In the file descriptor, keep a pointer to the first sector/block. In each sector, keep a pointer to the next sector.
- **Indexed files:** OS keeps an array of block pointers for each file. The user or OS must declare the maximum length of the file when it is created. OS allocates an array to hold the pointers to all the blocks when it creates the file, but allocates the blocks only on demand. OS fills in the pointers as it allocates blocks.

- Contiguous allocation is simple, but suffers from external fragmentation, the need for compaction, and the need to move files as they grow.
- Indexed allocation is very similar to page tables. A table maps from logical file blocks to physical disk blocks.
- **FAT (file allocation table):** Directory entry points to first block (i.e. specifies the block number). A FAT is maintained in memory having one (word) entry for each disk block. The entry for block N contains the block number of the next block in the same file as N. This is linked but the links are store separately.
- **File Control Block (inode):** FCB has all the information about the file also called as inode structures. Directory entry points to inode (index-node). Inode points to first few data blocks, often called direct blocks. Inode also points to an indirect block, which points to disk blocks. Inode also may points to a double indirect or triple indirect.
- FAT is used in windows whereas inodes are used in UNIX.
- Free space can be managed using a bitmap or a linked list.
- **Directory:** Directories/folders are used to keep track of files. It is a symbol table that translates file names to directory entries. Usually are themselves files.

Student's Assignment

Q.1 _____ is a unique tag, usually a number, identifies the file within the file system.
(a) File identifier (b) File name
(c) File type (d) None of these

Q.2 To create a file
(a) allocate the space in file system
(b) make an entry for new file in directory
(c) both (a) and (b)
(d) None of these

Q.3 By using the specific system call, we can
(a) open the file (b) read the file
(c) write into the file (d) All of these

Q.4 File type can be represented by
(a) file name (b) file extension
(c) file identifier (d) None of these

Q.5 Which file is a sequence of bytes organized into blocks understandable by the system's linker?
(a) object file (b) source file
(c) executable file (d) text file

Q.6 What is the mounting of file system?
(a) crating of a filesystem
(b) deleting a filesystem
(c) attaching portion of the file system into a directory structure
(d) removing portion of the file system into a directory structure

Q.7 Mapping of file is managed by
(a) file metadata (b) page table
(c) virtual memory (d) file system

Q.8 Mapping of network file system protocol to local file system is done by
(a) network file system
(b) local file system
(c) volume manager
(d) remote mirror

Q.9 Which one of the following explains the sequential file access method?
(a) random access according to the given byte number
(b) read bytes one at a time, in order
(c) read/write sequentially by record
(d) read/write randomly by record

Q.10 File system fragmentation occurs when
(a) unused space or single file are not contiguous
(b) used space is not contiguous
(c) unused space is non-contiguous
(d) multiple files are non-contiguous

Q.11 Data cannot be written to secondary storage unless written within a _____.
(a) file (b) swap space
(c) directory (d) text format

Q.12 File attributes consist of:
(a) name (b) type
(c) identifier (d) All of these

Q.13 The information about all files is kept in :
(a) swap space
(b) operating system
(c) separate directory structure
(d) None of these

Q.14 File is _____ data type.
(a) abstract (b) primitive
(c) public (d) private

Q.15 The operating system keeps a small table containing information about all open files called:
(a) system table
(b) open-file table
(c) file table
(d) directory table

Q.16 In UNIX, the open system call returns:
(a) pointer to the entry in the open file table
(b) pointer to the entry in the system wide table
(c) a file to the process calling it
(d) None of these

Q.17 The open file table has _____ associated with each file.
(a) file content
(b) file permission
(c) open count
(d) close count

Q.18 The file name is generally split into
(a) name (b) extension
(c) both (a) and (b) (d) None of these

Q.19 The three major methods of allocating disk space that are in wide use are:
(a) contiguous (b) linked
(c) indexed (d) all of these

Q.20 In contiguous allocation:
(a) each file must occupy a set of contiguous blocks on the disk
(b) each file is a linked list of disk blocks
(c) all the pointers to scattered blocks are placed together in one location
(d) None of these

Q.21 In linked allocation:
(a) each file must occupy a set of contiguous blocks on the disk
(b) each file is a linked list of disk blocks
(c) all the pointers to scattered blocks are placed together in one location
(d) None of these

Q.22 In indexed allocation:
(a) each file must occupy a set of contiguous blocks on the disk
(b) each file is a linked list of disk blocks
(c) all the pointers to scattered blocks are placed together in one location
(d) None of these

Q.23 All of the following are included in the file path except
(a) the drive
(b) all folders and subfolders
(c) the file name
(d) the file view

Q.24 Each inode in a file system has 5 direct pointers to disk blocks, 3 single-indirect pointers to disk blocks, 2 double-indirect pointers to disk blocks and nothing else. A disk block is 500 bytes. A pointer to a disk block is 10 bytes. The entire disk contains 15,000,000 bytes, at most.

What is the maximum size in bytes, of a file in this file system?

(a) 2552500 bytes
(b) 2575000 bytes
(c) 2575000 bytes
(d) 2577500 bytes

Q.25 A unix style I-node has 8 direct pointers, one double and one triple indirect pointers. Disk block size is 1 KB, disk block address is 32 bits. What is the maximum possible file size?

(a) 2^{24} bytes (b) 2^{32} bytes
(c) 2^{34} bytes (d) 2^{48} bytes

Answer Key:

1. (a)	**2.** (c)	**3.** (d)	**4.** (b)	**5.** (a)
6. (c)	**7.** (a)	**8.** (a)	**9.** (b)	**10.** (a)
11. (a)	**12.** (d)	**13.** (c)	**14.** (a)	**15.** (b)
16. (a)	**17.** (c)	**18.** (c)	**19.** (d)	**20.** (a)
21. (b)	**22.** (c)	**23.** (d)	**24.** (d)	**25.** (c)

CHAPTER 09

Input/Output System

9.1 Introduction

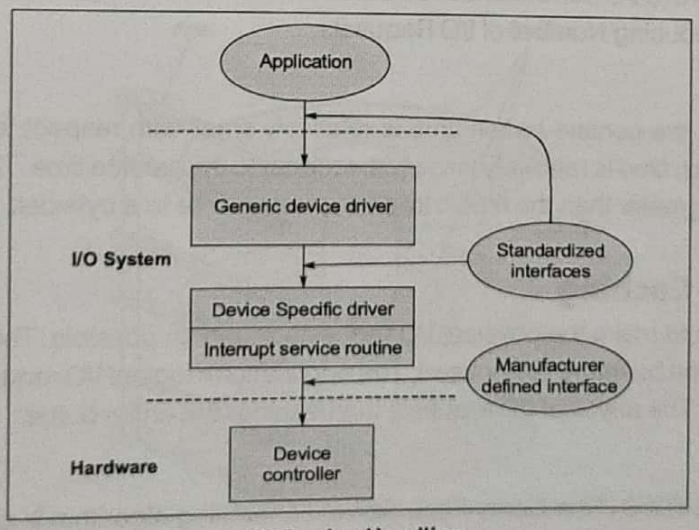
Objectives of I/O system:

- Take an application I/O request and send it to the physical device.
- Take whatever response comes back from the device and send it to the application.
- Optimize the performance of the various I/O requests.

I/O requests: In general, there may be many I/O requests for a device at the same time. These requests may come from multiple processes or the same process.

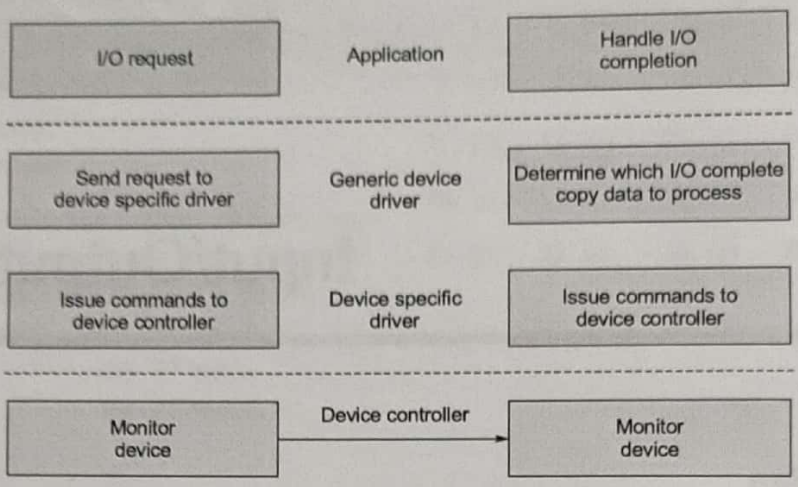
I/O Issues: The operating system is able to improve overall system performance if it can keep the various devices as busy as possible. It is important for the operating system to handle device interrupts as quickly as possible.

- For interactive devices (keyboard, mouse, microphone), this can make the system more responsive.
- For communication devices (modem, Ethernet, etc), this can affect the effective speed of the communications.
- For real-time systems, this can be the difference between the system operating correctly and malfunctioning.



I/O Device Handling

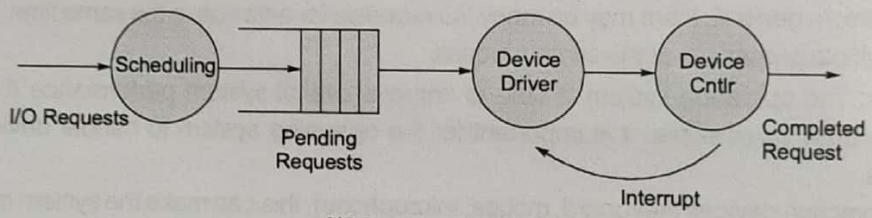
9.2 I/O System Structure



Stages of an I/O Request

9.2.1 A Queue of Pending Requests

Resource Scheduler that determines the next request to execute. A Mechanism to initiate the next request whenever a request completes.



I/O Request Queuing

I/O Performance Optimization

I/O processing is much slower than CPU processing. Every physical disk I/O has a dramatic impact on system performance. To improve I/O performance reduce the number of I/O requests, Carry out buffering and/or caching, I/O Scheduling, Reducing Number of I/O Requests.

Context Switching in I/O

In CPU scheduling, the context-switch time is relatively small with respect to the service time. In I/O scheduling the context-switch time is relatively large with respect to the service time. The time to move the head between cylinders is much greater than the time it takes to read or write to a cylinder.

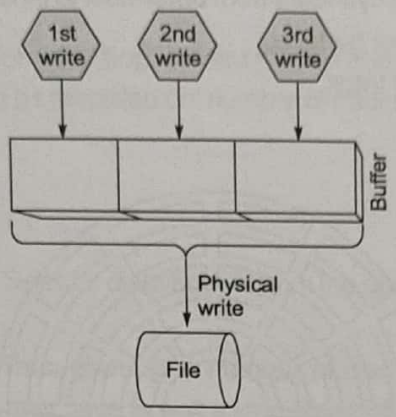
9.3 Buffering and Caching

The I/O system should make the physical I/O requests as big as possible. This will reduce the number of physical I/O requests by the buffering factor used. The application's logical I/O requests should copy data to/from a large memory buffer. The physical I/O requests then transfer the entire buffer.

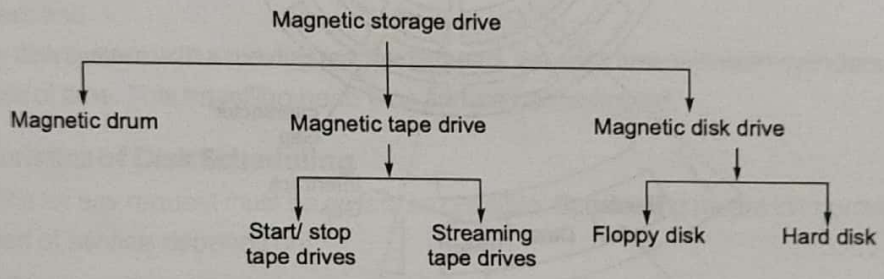
9.3.1 Buffered Write

For most devices, a FCFS (First-Come-First- Serve) scheduling algorithm is appropriate. For example, one wants the segments of a music file to be played in sequential order.

For some devices (disks especially), the order in which requests are processed is not inherently constrained by the device characteristics. On a typical system, there will be pending disk I/O requests from many different processes. The correct functioning of these processes usually does not depend on the order in which the disk I/O operations actually occur. Thus, we will want the Resource Scheduler to attempt to optimize performance for devices such as disks.

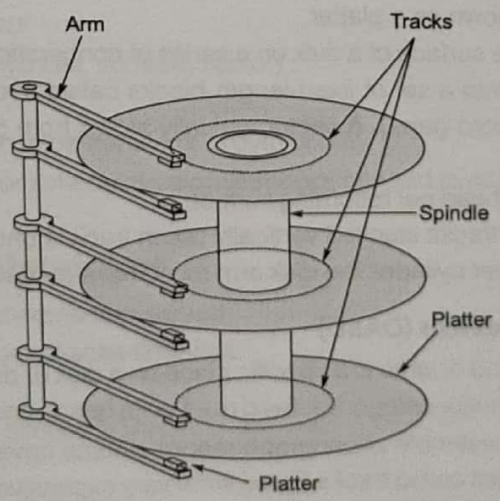


9.4 Magnetic Storage Devices



9.4.1 Disk Drive Mechanism

A disk (for data storage) is coated with magnetic material. Several disks are sometimes mounted on one spindle.

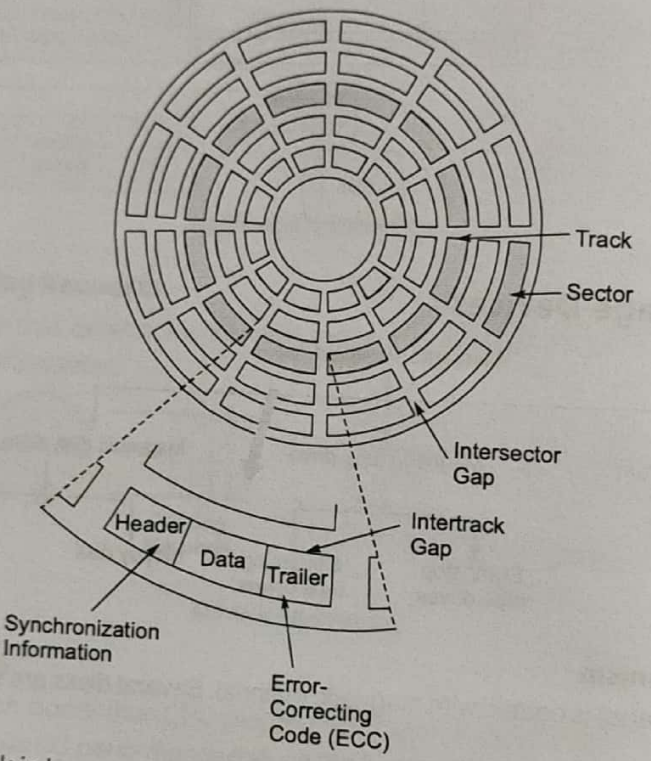


Data on a disk is addressed by: Cylinder, Surface and Sector. Disk drives are addressed as large 1-dimensional arrays of *logical blocks*, where the logical block is the smallest unit of transfer. The 1-dimensional array of logical blocks is mapped into the sectors of the disk sequentially. Sector 0 is the first sector of the first track on the outermost cylinder. Mapping proceeds in order through that track, then the rest of the tracks in that cylinder, and then through the rest of the cylinders from outermost to innermost.

Moving-head disk: One head per surface.

Fixed-head disk: One head per track.

9.4.2 Disk Track Format



- An individual disk is known as a platter.
- Data is recorded on the surface of a disk on a series of concentric **tracks**.
- Each track is divided into a set of fixed-length blocks called **sectors** (which are separated from each other by inter-record gaps). A sector typically stores from 512 bytes to several kilobytes of data.
- There is one read/write head per recording surface.
- A cylinder is the set of tracks stacked vertically (each track is under its own head). Since there is only one set of heads per cylinder, the disk arm must move (or seek) to access cylinders.

9.4.3 Direct Access Storage Devices (DASD)

Devices that can directly read or write to a specific place on a disk or drum. Also called random access storage devices. Grouped into two major categories: fixed read/write heads and movable read/write heads.

Fixed-Head Disks: Disks resemble phonograph record albums covered with magnetic film that has been formatted into concentric circles called tracks.These were very expensive and had less storage space as compared to movable-head disks but were faster.

Movable-Head Disks: The read/write head floats over the surface of the disk: Exist as individual units, as in a PC. It can also be in a disk pack, which is a stack of disks.

Disk Pack: A typical disk pack consists of several platters that are stacked on a common central spindle, with a slight space between them so the read/write heads can move between the pairs of disks.

Architecture of M-Head Disks

Each platter has two surfaces for recording, except the top and bottom. Each surface is formatted with specific numbers of tracks for the data to be recorded on: number of tracks varies depending on the manufacturer, usually range from 200 to 800 tracks.

9.5 Disk Scheduling

Purpose of Disk Scheduling: Select a disk request from the queue of I/O requests and decide when to process this I/O request.

Issues in Disk Scheduling: Throughput (the number of disk requests that are completed in some period), Fairness (some disk requests may have to wait a long time before being served).

Goal of disk scheduling:

- High Throughput
- Fairness and
- In any disk system with a moving read/write head, the seek time between cylinders takes a significant amount of time. This travelling head time should be minimized.

9.5.1 Characteristics of Disk Scheduling

Disk service for any request must be as fast as possible. Scheduling meant to improve the average disk service time. Speed of service depends on:

- Seek time, most dominating in most disks
- Latency time, or rotational delay
- Data transfer time

Each disk drive has a queue of pending requests. Each request made up of:

- Whether input or output
- Disk address (disk, cylinder, surface, sector)
- Memory address
- Amount of information to be transferred (byte count)

For moving-head disk, disk scheduling algorithms are needed to minimize seek time.

9.6 Disk Scheduling Algorithms

- FCFS scheduling: first-come-first-served
- SSTF scheduling: shortest-seek-time-first
- SCAN scheduling
- C-SCAN scheduling: circular SCAN
- LOOK scheduling
- C-LOOK scheduling

9.6.1 FCFS Scheduling (First Cum First Serve)

FCFS scheduling service I/O requests in the order in which they arrive. The simplest thing to do is to service each disk request on the queue in the order that it arrived.

Disadvantages:

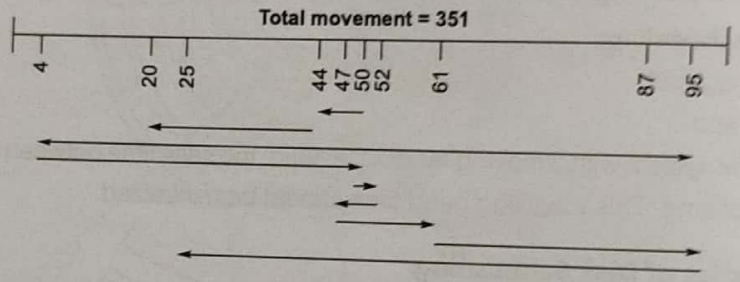
- This scheme does not try to optimize seek time.
- May not provide the best possible service
- Wild swings occur because the requests do not always come from the same process; they are interleaved with requests from other processes.

Example-9.1 Consider the following disk request sequence for a disk with 100 tracks.

44, 20, 95, 4, 50, 52, 47, 61, 87, 25

Head pointer starting at 50 (current position of R/W heads) and moving in left direction. Find the number of head movements in cylinders using FCFS scheduling.

Solution:

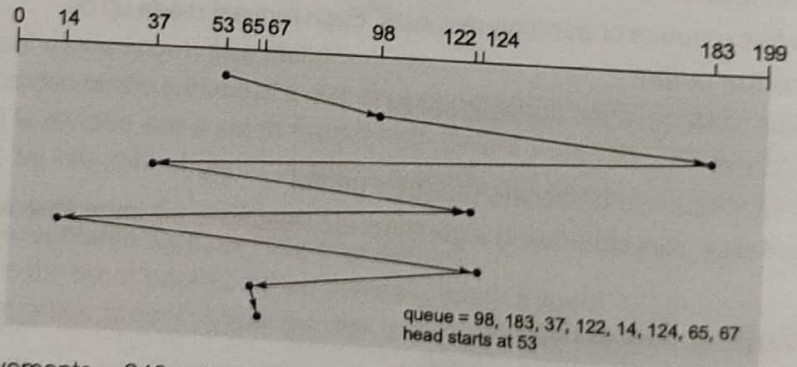


Example-9.2 Consider the following disk request sequence for a disk with 100 tracks.

98, 183, 37, 122, 14, 124, 65, 67

Head pointer starting at 53 (current position of R/W heads) and moving in right direction. Find the number of head movements in cylinders using FCFS scheduling.

Solution:



Total head movements = 640 cylinders.

9.6.2 SSTF (Shortest Seek Time First scheduling)

Service all requests close to the current head position before moving the head far away. Move the head to the closest track in the service queue. Selects the request with the minimum seek time from the current head position.

Disadvantages:

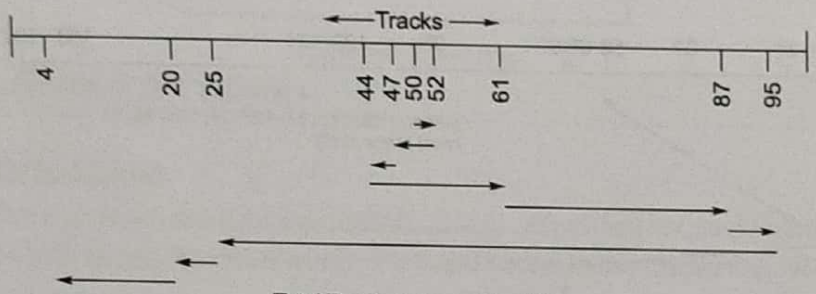
- SSTF scheduling is a form of SJF scheduling; may cause starvation of some requests.
- Requests for data on the ends of the disk may suffer from extreme postponement as "better" requests may keep coming in and be scheduled ahead of the less desirable requests.
- SSTF also may create a lot of disk seeking activity as the head may seek back and forth with ever wider swings as it services the more distant requests.

Example-9.3 Consider the following disk request sequence for a disk with 100 tracks.

44, 20, 95, 4, 50, 52, 47, 61, 87, 25

Head pointer starting at 50 (current position of R/W heads). Find the number of head movements in cylinders using SSTF scheduling.

Solution:



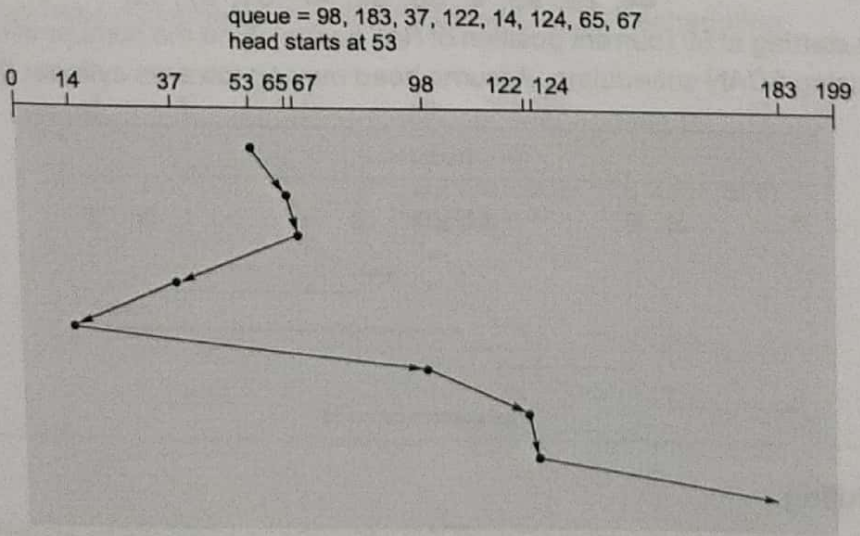
Total Track movement = 152

Example-9.4 Consider the following disk request sequence for a disk with 100 tracks.

98, 183, 37, 122, 14, 124, 65, 67

Head pointer starting at 53 (current position of R/W heads). Find the number of head movements in cylinders using SSTF scheduling.

Solution:



Total head movements = 222 cylinders.

9.6.3 Elevator algorithm (SCAN)

The disk arm starts at one end of the disk, and moves toward the other end, servicing requests until it gets to the other end of the disk, where the head movement is reversed and servicing continues.

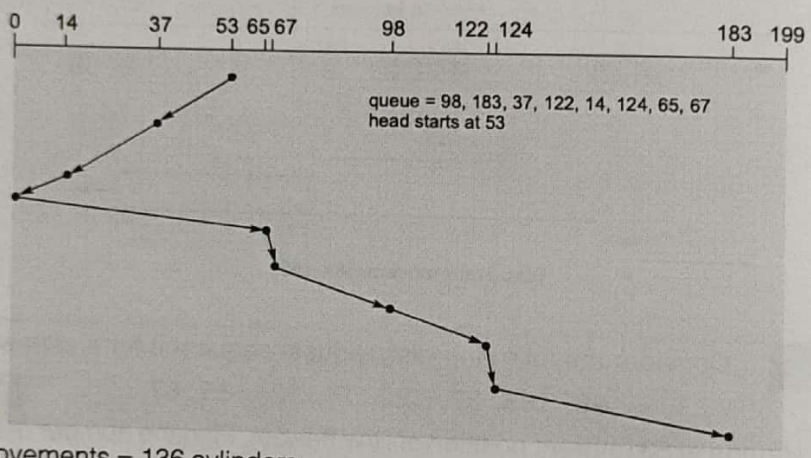
- Head continuously scans the disk from end to end
- Read/write head starts at one end of the disk
- It moves towards the other end, servicing all requests as it reaches each track
- At other end, direction of head movement is reversed and servicing continues

Example-9.5 Consider the following disk request sequence for a disk with 100 tracks.

98, 183, 37, 122, 14, 124, 65, 67

Head pointer starting at 53 (current position of R/W heads) and moving in left direction. Find the number of head movements in cylinders using SCAN scheduling.

Solution:



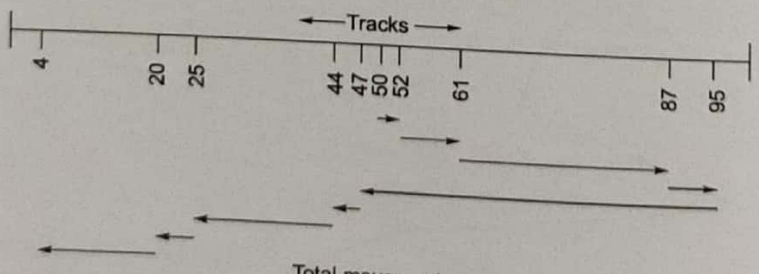
Total head movements = 136 cylinders.

Example-9.6 Consider the following disk request sequence for a disk with 100 tracks.

44, 20, 95, 4, 50, 52, 47, 61, 87, 25

Head pointer starting at 50 (current position of R/W heads). Find the number of head movements in cylinders using SCAN scheduling. Assume head moving towards cylinder 99.

Solution:



Total movement = 136

9.6.4 LOOK Scheduling

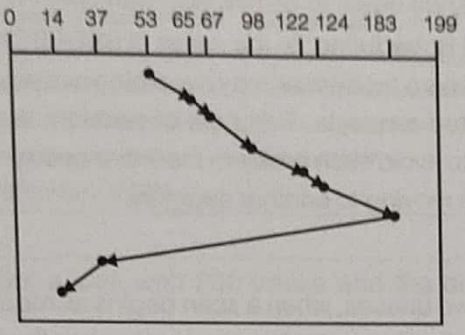
LOOK is a common-sense improvement over SCAN. Move the head only as far as the last request in that direction. If no more requests in the current direction, reverse the head movement. Always look for a request before moving in that direction.

Example-9.7 Consider the following disk request sequence for a disk with 100 tracks.

98, 183, 37, 122, 14, 124, 65, 67

Head pointer starting at 53 (current position of R/W heads) and moving in right direction. Find the number of head movements in cylinders using LOOK scheduling.

Solution:



LOOK disk Scheduling

Total head movements = 299 cylinders.

9.6.5 Circular SCAN (C-SCAN)

It provides a more uniform wait time than SCAN. It only schedules requests when the head is moving in one direction. Once the end of the disk is reached, the head seeks to the beginning without servicing any I/O.

9.6.6 Circular-LOOK (C-LOOK)

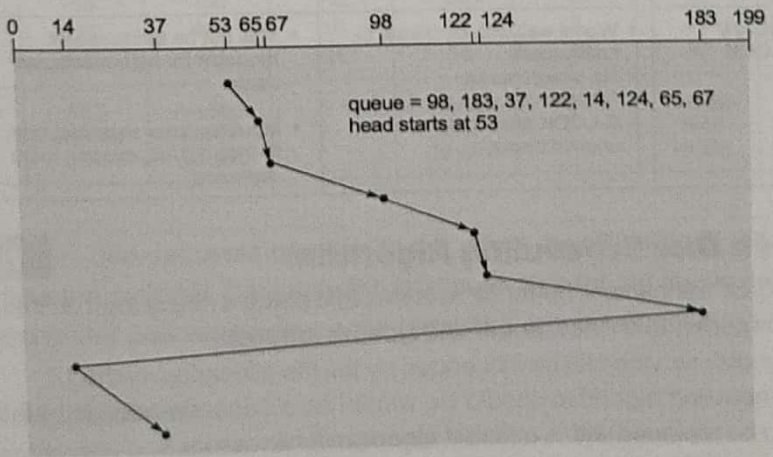
This is like C-SCAN but instead of seeking to the start of the disk, we seek to the lowest track with scheduled I/O. Disk arm only travels as far as the last request in each direction, then reverses direction immediately, without first going all the way to the end of the disk.

Example-9.8 Consider the following disk request sequence for a disk with 100 tracks.

98, 183, 37, 122, 14, 124, 65, 67

Head pointer starting at 53 (current position of R/W heads) and moving in right direction. Find the number of head movements in cylinders using C-LOOK scheduling.

Solution:



Total head movements = 322 cylinders.

9.6.7 N-step SCAN

The N-step-SCAN policy segments the disk request queue into sub queues of length N. Sub queues are processed using SCAN policy, while this processing is on new requests are added to some new queue. If at the end of SCAN fewer than N requests are available, then all of them are processed with the next scan. With large values of N this policy gives performance equal to SCAN, also with N=1 FIFO is adopted.

Advantage of N-step-SCAN Scheduling: In the cases of SSTF, SCAN & CSCAN the arm may not move for a considerable time, that is one or more processes may have high access rate to one particular track, so they monopolize the entire disk by repeated requests. This type of problem is more serious in high density disks rather than low density disks. In order to avoid such problem the entire queue can be segmented, and processing the one segment completely and then moving to another segment.

9.6.8 FSCAN

FSCAN policy has basically two queues, when a scan begins all requests are in one queue and the new requests are added to the other queue, it will process these requests when the first queue requests are completely finished.

9.7 Comparisons of Disk Scheduling Algorithms

Strategy	Advantages	Disadvantages
FCFS	• Easy to implement • Sufficient for light loads	• Doesn't provide best average service • Doesn't maximize throughput
SSTF	• Throughput better than FCFS • Tends to minimize arm movement	• May cause starvation of some requests • Localizes under heavy loads
SCAN/LOOK	• Eliminates starvation • Throughput similar to SSTF • Works well with light to moderate loads	• Needs directional bit • More complex algorithm to implement • Increased overhead
N-Step SCAN	• Easier to implement than SCAN	• The most recent requests wait longer than with SCAN
C-SCAN/ C-LOOK	• Works well with moderate to heavy loads • No directional bit • Small variance in service time • C-LOOK doesn't travel to unused tracks	• May not be fair to recent requests for high-numbered tracks • More complex algorithm than N-Step SCAN, causing more overhead

9.8 Selection of a Disk Scheduling Algorithm

- SCAN and C-SCAN perform better for systems that place a heavy load on the disk.
- Performance depends on the number and types of requests.
- Requests for disk service can be influenced by the file-allocation method.
- The disk-scheduling algorithm should be written as a separate module of the operating system, allowing it to be replaced with a different algorithm if necessary.
- Either SSTF or LOOK is a reasonable choice for the default algorithm.
- Comparisons of disk scheduling algorithms based on selection criteria:

Name	Description	Remarks
Selection according to requester		
FIFO	First in First out	Fairest of all
LIFO	Last in Last out	Maximize locality & resource utilization
Selection according to requested items		
SSTF	Shortest service time first	High utilization, small queues
SCAN	Back & forth over disk	Better service distribution
CSCAN	One way with fast return	Lower service variability
N-step-scan	Scan of N records at a time	Service guarantee
FSCAN	N=queue size at the beginning of SCAN	Load sensitive

Example-9.9 Consider a disk with 200 tracks and the queue has random requests from different processes in the order:

55, 58, 39, 18, 90, 160, 150, 38, 184.

Initially arm is at 100. Find average seek length using FIFO, SSTF, SCAN and C-SCAN scheduling algorithms.

Solution:

FIFO (Starting at Track 100)		SSTF (Starting at Track 100)		SCAN (Starting at Track 100 in the Direction of Increasing Track Number)		CSCAN (Starting at Track 100 in the Direction of Increasing Track Number)	
Next Track Access	No. of Tracks Traversed	Next Track Access	No. of Tracks Traversed	Next Track Access	No. of Tracks Traversed	Next Track Access	No. of Tracks Traversed
55	45	90	10	150	50	150	50
58	3	58	32	160	10	160	10
39	19	55	3	184	24	184	24
18	21	39	16	90	94	18	166
90	72	38	1	58	32	38	20
160	70	18	20	55	3	39	1
150	10	150	132	39	16	55	16
38	112	160	10	38	1	58	3
184	146	184	24	18	20	90	32
Average seek length	55.3 (498/9= 55.3)	Average seek length	27.5	Average seek length	27.8	Average seek length	35.8

Example-9.10 Disk requests come in to the driver for cylinders 8, 24, 20, 5, 41, 8 in that order. A seek takes 6ms per cylinder. Calculate the total seek time for the above sequence of requests using FCFS, SSTF and SCAN disk scheduling policy (the disk arm is initially at cylinder 20).

Solution:

(a) First-come first-server:

Sequence is: 20 [starting], 8, 24, 20, 5, 41, 8

Time is: $[(20-8)+(24-8)+(24-20)+(20-5)+(41-5)+(41-8)] \times 6\text{ms} = 696\text{ms}$

(b) Closest cylinder next:
Sequence is: 20 [starting], 20, 24, 8, 8, 5, 41
Time is: $[(20-20)+(24-20)+(24-8)+(8-8)+(8-5)+(41-5)] \times 6$ ms = 354 ms

(c) Elevator algorithm (initially moving upward in cylinder value):
Sequence is: 20 [starting], 20, 24, 41, 8, 8, 5
Time is: $[(20-20)+(24-20)+(41-24)+(41-8)+(8-8)+(8-5)] \times 6$ ms = 342 ms

Summary

- **Blocking:** A storage-saving and I/O-saving technique that groups individual records into a block that's stored and retrieved as a unit.
- **Buffers:** Temporary storage areas residing in main memory, channels, and control units.
- **Direct Memory Access (DMA):** An I/O technique that allows a control unit to access main memory directly and transfer data without the intervention of the CPU.
- **Flash Memory:** A type of nonvolatile memory used as a secondary storage device that can be erased and reprogrammed in blocks of data.
- **Inter Record Gap (IRG):** An unused space between records on a magnetic tape. It facilitates the tape's start/stop operations.
- **Interrupt:** A hardware signal that suspends execution of a program and activates the execution of a special program known as the interrupt handler.
- **RAID:** Acronym for redundant array of independent disks; a group of hard disks controlled in such a way that they speed read access of data on secondary storage devices and aid data recovery.
- **First-Come, First-Served (FCFS):** The simplest scheduling algorithm for direct access storage devices that satisfies track requests in the order in which they are received.
- **C-LOOK:** A scheduling strategy for direct access storage devices that's an optimization of C-SCAN.
- **C-SCAN:** A scheduling strategy for direct access storage devices that's used to optimize seek time. It's an abbreviation for circular-SCAN.
- **LOOK:** A scheduling strategy for direct access storage devices that's used to optimize seek time. Sometimes known as the elevator algorithm.
- **N-step SCAN:** A variation of the SCAN scheduling strategy for direct access storage devices that's used to optimize seek times.
- **SCAN:** A scheduling strategy for direct access storage devices that's used to optimize seek time. The most common variations are N-step SCAN and C-SCAN.
- **Shortest seek time first (SSTF):** A scheduling strategy for direct access storage devices that's used to optimize seek time. The track requests are ordered so the one closest to the currently active track is satisfied first and the ones farthest away are made to wait.
- **Track:** A path on a storage medium along which data is recorded.
- **Cylinder:** A concept that describes a virtual tube that is formed when two or more read/write heads are positioned at the same track, at the same relative position, on their respective surfaces.

- **Access Time:** The total time required to access data in secondary storage.
- **Search Time:** The time it takes to rotate the disk from the moment an I/O command is issued until the requested record is moved under the read/write head. Also known as rotational delay.
- **Seek Time:** The time required to position the read/write head on the proper track from the time the I/O request is issued.
- **Transfer Rate:** The rate at which data is transferred from sequential access media.
- **Transfer time:** The time required for data to be transferred between secondary storage and main memory.

Student's Assignment

Q.1 Consider the following parameters for disk system.

- Seek time is 6 ms per cylinder
- Disk request for cylinders 8, 24, 20, 5, 41, 8 in that order come into the driver
- Initially disk arm is at cylinder 20.

What is the total seek time for the above requests using first come first serve scheduling?

(a) 342 ms (b) 57 ms
(c) 116 ms (d) 696 ms

Q.2 Consider the following cylinder requests of disk coming in the given order.

10, 22, 20, 2, 40, 6, 38

Assume the disk head is currently positioned at cylinder 20. Seek time takes 6 ms. How many number of cylinder moves are taken for the given requests using the SCAN scheduling? (Assume initially head is moving upwards)

(a) 58 (b) 60
(c) 62 (d) 64

Q.3 Match the following groups.

Group-I

A. Access time
B. Seek time
C. Rotational latency
D. Data transfer time

Group-II

1. Desired cylinder
2. Desired sector
3. Actual data
4. Total time from start to end

Codes:

	A	B	C	D
(a)	3	1	2	4
(b)	3	2	1	4
(c)	4	1	2	3
(d)	4	2	1	3

Q.4 Suppose that the head of a moving-head disk with 200 tracks, numbered 0 to 199, is currently serving a request at track 53. Ordered disk queues with requests involving tracks are shown below.

98, 183, 37, 122, 14, 124, 65 and 67.

What is the total number of head movements needed to satisfy these requests for the FCFS and SSTF disk-scheduling algorithms?

(a) 638, 238 (b) 640, 236
(c) 640, 238 (d) 638, 236

Q.5 Match **List-I** with **List-II** and select the correct answer using the codes given below the lists:

List-I	List-II
A. Disk scheduling	1. Round robin
B. Time sharing	2. LIFO
C. page replacement	3. SCAN
D. Interrupt processing	4. LRU

Codes:

	A	B	C	D
(a)	4	1	3	2
(b)	4	3	1	2
(c)	3	1	4	2
(d)	3	4	1	2

Q.6 Suppose that a disk drive has 5000 cylinders, numbered 0 to 4999. The drive is currently serving a request at cylinder 143, and the previous request was at cylinder 125. The queue of pending request, in FIFO order is 86, 1470, 913, 1774, 948, 1509, 1022, 1750, 130 starting from the current head position, what is the total distance (in cylinders) that the disk arm moves to satisfy all the pending requests, for FCFS, SSTF and SCAN disk scheduling algorithms respectively?

(a) 7081, 1745, 9769 (b) 7081, 1745, 7081
(c) 1745, 1004, 9600 (d) 9769, 1745, 7081

Q.7 Suppose that the disk drive has 200 cylinders. The drive is currently serving the request at cylinder 53. The previous request was at cylinder 55. A queue of pending requests in FIFO order is 98, 183, 37, 122, 14, 124, 65, 67 starting from the current head position. What is the total distance traveled by the disk arm in SCAN scheme?

(a) 640 (b) 208
(c) 384 (d) 236

Q.8 Requests are not usual uniformly distributed. For example, a cylinder containing the file system FAT or inodes can be expected to be accessed more frequently than a cylinder that only contains files. Suppose you know that 50 percent of the requests are for a small, fixed number of cylinders.

Which of the following scheme results in the lowest arm movement?

(a) SSTF (b) LOOK
(c) C-SCAN (d) FCFS

Answer Key:

1. (d) **2.** (a) **3.** (c) **4.** (b) **5.** (c)
6. (a) **7.** (b) **8.** (a)