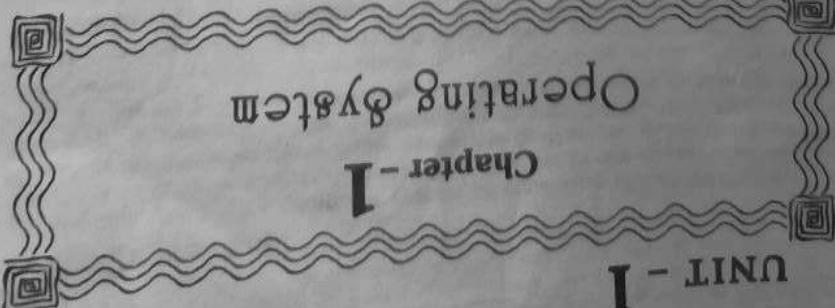


# UNIT - I

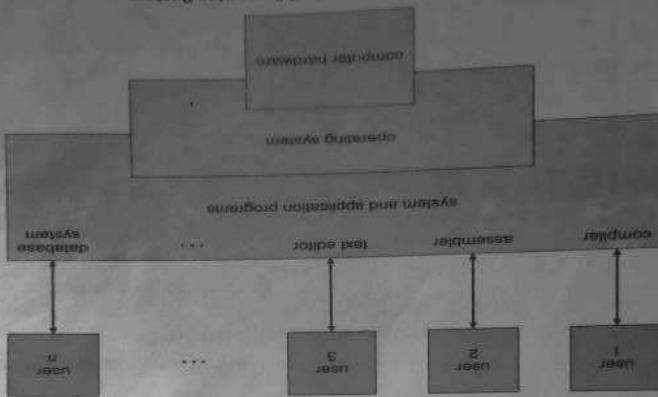
## Operating System

### Chapter - I



The hardware Central Processing Unit, the memory and the input/Output(I/O) devices provides the basic computing resources for the system. The application program such as word processors, spreadsheets, and Web browsers define the ways in which these resources are used.

Figure: The Components of Computer System



The four components are, the hardware / the operating system, the application programs / and the users.

**Component of Computer System :**

A computer system is an important part of almost every computer system. An operating system is an interface between the user and the hardware. The main purpose of an operating system is to provide an environment in which a user can easily execute their programs and can interface with hardware. The main purpose of an operating system is to provide an environment in which a user can easily execute their programs and share this information and resources through other computer system in network which is accessed by other computer users.

Operating system is the program that manages computer hardware program and acts as an intermediary between the computer user and the computer.

#### INTRODUCTION :

An operating system is an interface between the user and the hardware.

A computer system can be divided roughly into four components: the hardware, the operating system, the application programs, and the users.

In the hardware, the operating system controls the hardware and coordinates its use among the various application programs for the various resources in the operation of the computer system. An operating system is similar to a government. Like a government, it performs no useful function by itself. It simply provides an environment within which other programs can do useful work.

The operating system role can be seen from user view and system view

a) The user's view of the computer varies according to the interface being used.

b) Provides Graphical User Interface(GUI)

c) Here Computer system can process many programs and share this information and resources through other computer system in network which is accessed by other computer users.

d) A computer system has many resources that may be required to solve a problem: CPU time, memory space, file-storage space, I/O devices, and so on. The operating system acts as the manager of these resources.

- 115

- 149

- 115

- 152

- 25 - 60

01 - 05  
12 - 19  
20 - 24

Operating System (S.Y.B.Sc. C.S.) Sem. 02

154

155

156

157

158

159

160

161

162

163

164

165

166

167

168

169

170

171

172

173

174

175

176

177

178

179

180

181

182

183

184

185

186

187

188

189

190

191

192

193

194

195

196

197

198

199

200

201

202

203

204

205

206

207

208

209

210

211

212

213

214

215

216

217

218

219

220

221

222

223

224

225

226

227

228

229

230

231

232

233

234

235

236

237

238

239

240

241

242

243

244

245

246

247

248

249

250

251

252

253

254

255

256

257

258

259

260

261

262

263

264

265

266

267

268

269

270

271

272

273

274

275

276

277

278

279

280

281

282

283

284

285

286

287

288

289

290

291

292

293

294

295

296

297

298

299

300

301

302

303

304

305

306

307

308

309

310

311

312

313

314

315

316

317

318

319

320

321

322

323

324

325

326

327

328

329

330

331

332

333

334

335

336

337

338

339

340

341

342

343

344

345

346

347

348

349

350

351

352

353

354

355

356

357

358

359

360

361

362

363

364

365

366

367

368

369

370

371

372

373

374

375

376

377

378

379

380

381

382

383

384

385

386

387

388

389

390

391

392

393

394

395

396

397

398

399

400

401

402

403

404

405

**Function of Operating system**

- Process Management :** According to the priority and burst time of the process operating system decides which process need to be execute first and then only allocation and de-allocation of the process is done at processor for processing.

- File Management :** Creating file, removing the file, storing the file and accepting track of location.

- Memory Management :** According to the size of the process, memory will be allocated for the specific process this will be decided by the operating system it is the main function of the operating system.

- Device Management :** Maximum utilization of the resources is done by the operating system. Here hardware devices are controlled with the help of software device drivers.

- Interface :** Graphical user interface and Command line interface provided by the operating system.

- Security :** Operating system protect all type of data which is stored on the computer from unauthorized users.

- Caching :** Here we have to select the information or data carefully as limited space is provided by the operating system for quick access of the information.

**Computing Environment**

A computing environment contains computer systems interface and the services which are given by operating system for users. Different types of computing environments are:

**1. Traditional Computing**

Office environment uses traditional computing and who technology also uses traditional computing. Today, traditional time-sharing systems are uncommon. For a period of time, systems were either batch or interactive.

A timer and scheduling algorithms uses time-sharing system to rapidly cycle processes through the CPU, giving each user a share of the resources. The scheduling are all owned by the same user.

**2. Client-Server Computing**

PCs Specialized form of distributed system which is called a client server system, where systems act as server system to fulfill the requirements of client systems.



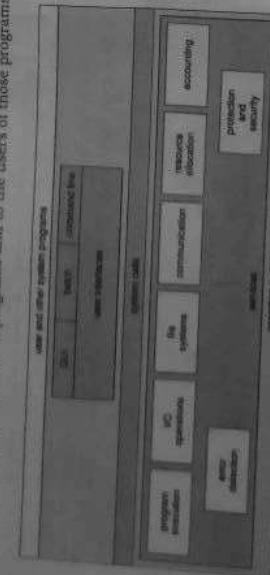
Server systems can be categorized into:

The compute-server system acts as an interface where client sends a request to perform some action and response, the server executes action and sends back the results to client. For example, server running a database which gives response to client requests for data.

An operating system provides the environment within which programs are executed.

**Operating System Services**

An operating system provides an environment for the execution of programs. These operating system services are provided for the convenience of the programmer, to facilitate the programming task easier. It provides certain services to programs and to the users of those programs.



An Operating System supplies different kinds of services to both the users and to the programs as well. It also provides application programs (that run within an Operating system) an environment to execute it freely.

**1. User Interface of Operating System**

Usually Operating System comes in three forms or types. Depending on the interface their types have been further subdivided. These are:

- Command line interface
- Batch based interface
- Graphical User Interface

**2. Program Execution In operating system**

The operating system must have the capability to load a program into memory and execute that program. Furthermore, the program must be able to end its execution, either normally or abnormally / forcefully.

The file-server system acts as an interface where the clients can create, update, read, and delete file. For eg. Web browsers can receive the file from web server.

**3. Peer-to-peer Computing**

In peer to peer computing, client and server are not different from one another, all nodes within the system are considered peers and all of them can act as client or a server, depending on request or providing the service.

**4. Web-based Computing**

Web computing has increased the emphasis on networking. Devices that were not previously networked now include wired or wireless access. Devices that were networked now have faster network connectivity provided by either improved networking technology optimized network implementation code / or both.

The implementation of Web-based computing has given rise to new categories of devices / such as load balancer.

**QUESTIONS**

1. Explain Operating System in detail.
2. What are the different function of operating system.
3. Define operating system, explain the roles of operating system.
4. Write a short note on peer-peer system and client-server system.

**Spring board (1) / 2.1****Operating System Structure****3. File system Manipulation**

Programs need has to be read and then write them as files and directories. File handling portion of operating system also allows users to create and delete files by specific name along with extension, search for a given file and / or list file or denying access to files or directories based on file ownership.

**4. I/O operation**

A program which is currently executing may require I/O which may involve file or other I/O device. For efficiency and protection, users cannot directly operate which means read or write operation with any file.

**5. Communication**

Process needs to swap over information with other process. Processes communicating on same computer system or on different computer systems can communicate using operating system support. Communication between two processes can be done using shared memory or via message passing.

**6. Resources Allocation**

When multiple jobs running concurrently, resources must need to be allocated to each of them. Resources can be CPU cycles, main memory storage, file storage and I/O devices. CPU scheduling routines are used here to establish how best the CPU can be used.

**7. Error detection**

Errors may occur within CPU, memory hardware, I/O devices and in the user program. For each type of error, the OS takes adequate action for ensuring correct and consistent computing.

**8. Accounting**

This service of the operating system keeps track of which users are using how much and what kinds of computer resources have been used for accounting or simply to accumulate usage statistics.

**9. Protection and Security**

Protection includes in ensuring all access to system resources in a controlled manner. For making a system secure, the user needs to authenticate him or her to the system before using.

**User and Operating System Interface**

There are two fundamental approaches for users to interface with the operating system. One technique is to provide a command-line interface or command interpreter that allows users to directly enter commands that are to be performed by the operating system. The second approach allows the user to interface with the operating system via a graphical user interface or GUI.

**Command Interpreter**

- Some operating systems include the command interpreter in the kernel. Others, such as Windows XP and UNIX, treat the command interpreter as a special program that is running when a job is initiated or when a user first logs on (on interactive systems).
- On systems with multiple command interpreters to choose from, the interpreters are known as shells. For example, on UNIX and Linux systems, there are several different shells a user may choose from including the Bourne shell, C shell, Bourne Again shell, sh, ksh, tcsh,

- Most shells provide similar functionality with only minor differences; most users choose a shell based upon personal preference.
- The main function of the command interpreter is to get and execute the next user-specified command.
- Many of the commands given at this level manipulate files: create, delete, list, print, copy, execute, and so on. The MS-DOS and UNIX shells operate in this way. There are two general ways in which these commands can be implemented. In one approach, the command interpreter itself contains the code to execute the command.

#### **Graphical User Interface**

- A second strategy for interfacing with the operating system is through a user-friendly graphical user interface or GUI.
- Rather than having users directly enter commands via a command-line interface, a GUI allows provides a mouse-based window-and-menu system as an interface.
- A GUI provides a desktop metaphor where the mouse is moved to position its pointer on images or icons, on the screen (the desktop) that represent programs, files, directories, and system functions.
- Depending on the mouse pointer's location, clicking a button on the mouse can invoke a program, select a file or directory—known as a folder—or pull down a menu that contains commands.

#### **System Calls**

To understand system calls, first one needs to understand the difference between **kernel mode** and **user mode** of a CPU.

##### **Kernel Mode**

- When CPU is in **kernel mode**, the code being executed can access any memory address and any hardware resource.
- Hence kernel mode is a very privileged and powerful mode.
- If a program crashes in kernel mode, the entire system will be halted.

##### **User Mode**

- When CPU is in **user mode**, the programs don't have direct access to memory and hardware resources.
- In user mode, if any program crashes, only that particular program is halted.
- That means the system will be in a safe state even if a program in user mode crashes.
- Hence, most programs in an OS run in user mode.

When a program in user mode requires access to RAM or a hardware resource, it must ask the kernel to provide access to that resource. This is done via something called a **system call**.

When a program makes a system call, the mode is switched from user mode to kernel mode. This is called a **context switch**.

Then the kernel provides the resource which the program requested. After that, another context switch happens which results in change of mode from kernel mode back to user mode.

Generally, system calls are made by the user level programs in the following situations:

#### **Operating System Structure**

- Creating, opening, closing and deleting files in the file system.
- Creating and managing new processes.
- Creating a connection in the network, sending and receiving packets.
- Requesting access to a hardware device, like a mouse or a printer.

#### **Type of System Call**

There are mainly five types of system calls. These are explained in detail as follows:

- **Process Control:** These system calls deal with processes such as process creation, process termination etc.

- **File Management:** These system calls are responsible for file manipulations such as creating a file, reading a file, writing into a file etc.

- **Device Management:** These system calls are responsible for device manipulation such as reading from device buffers, writing to device buffers etc.

- **Information Maintenance:** These system calls handle information and its transfer between the operating system and the user program.

- **Communication:** These system calls are useful for inter-process communication. They also deal with creating and detecting a communication connection.

#### **Operating System Structure**

A system as large and complex as a modern operating system must be engineered carefully if it is to function properly and be modified easily. A common approach is to partition the task into small components rather than have one monolithic system.

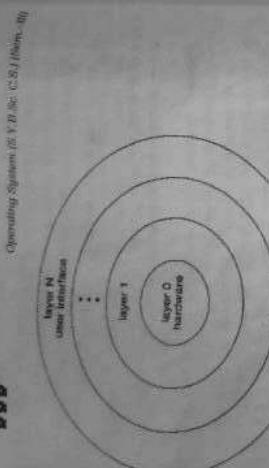
##### **Simple Structure :**

Many commercial operating systems do not have well-defined structures. Frequently, such systems started as small, simple, and limited systems and then grew beyond their original scope. MS-DOS is an example of such a system.



**Layered Structure:** Operating system can be broken into pieces that are smaller and more appropriate with proper hardware support.

#### **Operating System Structure**



#### **Questions**

1. What are the different services provided by Operating system?
2. Explain system call in detail
3. Explain Operating system structure in detail
4. What is the purpose of the command interpreter? Why it is separate from the kernel?

#### **Microkernel**

- Moves as much from the kernel into "user" space
- Communication takes place between user modules using message passing

##### **Benefits :**

- Easier to extend a microkernel

- Easier to port the operating system to new architectures

- More reliable (less code is running in kernel mode)

##### **Drawbacks :**

- Performance overhead of user space to kernel space communication

#### **Modules**

- Most modern operating systems implement kernel modules

- Uses object-oriented approach

- Each core component is separate

- Each talks to the others over known interfaces

- Each is loadable as needed within the kernel

- Overall similar to layers but with more flexible

# Chapter - 3

## Process Concept

### INTRODUCTION

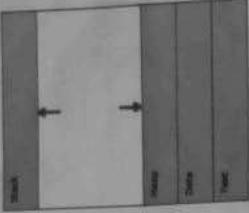
- Now-a-days computer systems allow multiple programs to be loaded into memory and executed concurrently.
- The needs such as former control and more compartmentalization of the various programs results in the notion of process, which is a program in execution.
- A system consists of a collection of processes where operating system processes executing system code and user processes executing user code.

### Process Concept

- A process is an instance of a program running in a computer. It is close in meaning to task, a term used in some operating systems.... Like a task, a process is a running program with which a set of data is associated so that the **process** can be kept track of.
- In a single user system, a user may be able to run several programs at one time; for example, a word processor, a web processor, and an email package.
- If the user can execute only one program at a time, the operating system supports its own internal activities, such as memory management which we call all of them as processes

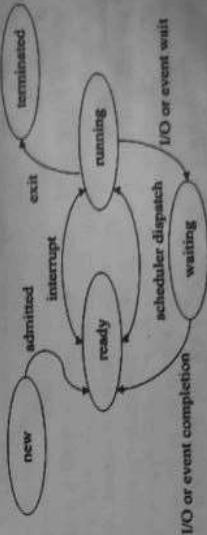
### The Process

- A process includes the current activity represented by the value of the program counter and the contents of the processor's registers. It also includes the process stack, a data section and a heap.
- A program is a passive entity, such as a file containing a list of instructions stored on disk whereas a process is an active entity, with a program counter specifying the next instruction to execute and a set of associated resources.
- When an executable file is loaded into memory, a program becomes a process. For loading executable files there are two techniques: double-clicking an icon representing the executable file and entering the name of the executable file on the command line.



### Process State

- The state of a process which may be defined in part by the current activity of that process. Each process can be in one of the following states:
- NEW-The process is being created.
- READY-The process is waiting to be assigned to a processor.
- RUNNING-Instructions are being executed.
- WAITING-The process is waiting for some event to occur such as an I/O completion or reception of a signal.
- TERMINATED-The process has finished execution.
- One process can be running on any processor and others may be in ready or waiting.



### Process Control Block

- Process control block also called as task control block which contains many pieces of information.

| Process control block  |
|------------------------|
| Process Id             |
| State                  |
| Pointer                |
| Priority               |
| Program counter        |
| CPU registers          |
| I/O information        |
| Accounting Information |
| etc...                 |

### Operating System [B. K. Bala, C. A. Rose, 4e]

- o **Process state :** The states are new, ready, running, waiting, halted, etc.
- o **Program counter :** It indicates the address of the next instruction to be executed for this process.
- o **CPU registers :** The registers may be different in number and type, depending on the computer architecture.
- o **CPU scheduling information :** Process priority, pointers to scheduling queues and other scheduling parameters.
- o **Accounting information :** Amount of CPU used, time limits, account numbers, job or process numbers are included in accounting information.
- o **I/O status information :** It consist list of I/O devices allocated to the process, a list of open files.
- o **Memory management information :** It contains value of the base and limit registers, the page tables or the segment tables.

### Threads

- A thread is a single sequence stream within a process. When a process is running a word processor program, a single thread of instructions is being executed. This single thread of control allows the process to perform only one task at one time.

### Process Scheduling

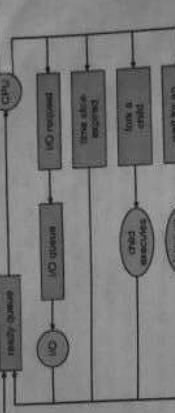
- To meet the objective of multiprogramming and time sharing the process scheduler selects an available process for program execution on the CPU.

### Scheduling Queues

- Processes entering the system are kept in job queue, whereas processes residing in main memory which are ready to execute are kept in ready queue.
- When a process is allocated the CPU, it executes for a while and quits, is interrupted or waits for the occurrence of a particular event, such as the completion of an I/O request.
- The list of processes waiting for a particular I/O device is called a device queue. Each device has its own device queue.

### 13

Process Concept  
A common representation of process scheduling is a queuing as shown in Fig. 3.2. i. Each rectangular box represents a queue. Two types of queues are present, the ready queue and a set of device queues. The circles indicate resources and arrows indicate the flow of processes in the system.



Queuing representation of process scheduling

### Schedulers

- The operating system must select processes from the queues for scheduling purposes which is carried by the scheduler. Processes are loaded into memory for execution.
- The operating system must select the processes that are ready to execute are selected by the **short-term scheduler** or CPU scheduler and allocates the CPU to one of them.
- The short-term scheduler executes at least once every 100 milliseconds where the long-term executes much less frequently.
- If it is stable, then the average rate of processes creation must be equal to the average departure rate of processes leaving the system.
- Most processes are either I/O bound or CPU bound.
- If the long-term processes spends most of its time doing I/O rather than doing computations, then the long-term scheduler controls the **degree of multiprogramming**, the long-term scheduler controls the **average rate of processes creation**.
- A CPU bound process, in contrast, generates I/O requests infrequently using more of its time doing computations. It's important that the long-term scheduler selects a good process mix of I/O bound and CPU bound processes.
- The short-term scheduler will have little to do if all processes are I/O bound, the ready queue will almost always be empty. If all processes are CPU bound, the I/O queue will almost always be empty, devices will not be utilized and the system will be unbalanced.
- The **middle-term scheduler** is advantageous to remove processes from memory and thus it can remove the degree of multaprogramming.
- The process is introduced into memory, and its execution is continued where it left off called swapping.

- A **context switch** is the process of saving the state of processor of a thread, so that it can be restored, and execution resumed from the same point later. This allows multiple processes to share a single CPU, and is an essential feature of a multitasking Operating System.
- The precise meaning of the phrase "context switch" varies significantly in usage. In a multitasking context, it refers to the process of storing the system state for one task, so that task can be paused, and another task resumed. A context switch can also occur as the result of an interrupt, such as when a task needs to access Disk storage, freeing up CPU time for other tasks. Some operating systems also require a context switch to move between User mode and Kernel mode tasks. The process of context switching can have a negative impact on system performance, although the size of this effect depends on the nature of the switch being performed.
- The kernel saves the context of the old process in its PCB and loads the saved context of the new process scheduled.

#### Operation on Processes

The process in various systems execute concurrently, and it can be created and deleted dynamically.

#### Process Creation

- Through a create-process system call a process may create several new processes where the creating process is called a parent process, and the new processes are called the children of that process.
- Each new process creates resources to accomplish its task such as CPU time, memory, files, I/O devices.
- The parent partitions its resources among its children.
- When a process creates a new process, two possibilities exist in terms of operations:
  - a) The parent continues to execute concurrently with its children.
  - b) The parent waits until some or all of its children have terminated.
- There are also two possibilities of the address space of the new process:
  - a) The child process is a duplicate of the parent process.
  - b) The child process has a new program loaded into it.

#### Process Termination

- When a final statement is executed a process terminates by exit() System call where all the resources are de-allocated.
- A parent terminates the execution of its children for various reasons:
  - a) The child goes beyond its usage of the resources that it has been allocated.
  - b) The task which was assigned to the child is no longer required.
  - c) The operating system does not allow a child to continue if its parent terminates.

#### Context Switch

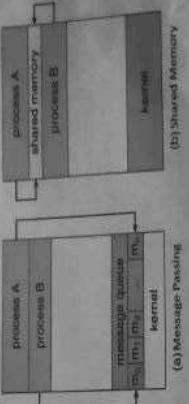
In Cascading termination, if a parent terminates normally or abnormally, then all its children must also be terminated.

#### Inter process Communication

- Process executing simultaneously in the operating system are independent processes or cooperating processes.
- A process is independent if it is not affected by the other processes executing in the system. Here the process that does not share data.
- Cooperating processes requires an inter process Communications (IPC).
- Two models of Inter process Communications are:

1. **Shared memory model :** Here, a region of memory that is shared by cooperating processes is established.

2. **Message Passing model :** Here, communication takes place by means of messages exchanged between the cooperating processes.



(a) Message Passing

#### Shared memory System

- Inter process communication using shared memory requires communicating processes to establish a region of shared memory. For example, producer consumer problem.
- A producer process generates information which is consumed by a consumer process. The solution to this problem is shared memory.
- To allow producer and consumer processes to run concurrently, we must have a buffer of items that can be filled by the producer and emptied by the consumer.
- The buffer resides in a region of memory that is shared by the producer and consumer processes. A producer produces one item while the consumer consumes another item.

Two types of buffers are used:

1. **Unbounded buffer :** No practical limit on the size of the buffer.
2. **Bounded buffer :** Assumes a fixed buffer size.

#### Message Passing System

- Mechanism to allow processes to communicate and synchronize their actions without sharing the same address space which is useful in a distributed environment.

#### Naming

- Processes that want to communicate use either direct or indirect communication.
- Each process that wants to communicate must explicitly name the recipient or sender of the communication in direct communication.

- A message passing facility provides at least two operations which are send and receive. Messages sent by a process can be either fixed or variable size.

#### Queuing System (S.Y.B.Sc., C.S.I./Wms-4)

- 1. **Blocking send :** Here, the sending process is blocked until the message is received by the receiving or by the mailbox.
- 2. **Non-blocking sends :** Here, the sending process sends the message and resumes the operation.
- 3. **Blocking receive :** Here, the receiver blocks until a message is available.
- 4. **Non-blocking receives :** Here, the receiver retrieves a valid message or a null.

#### Buffering

- Communicating processes reside in a temporary queue which are implemented in three ways:

1. **Zero capacity :** The link cannot have any messages waiting in it as the queue has a maximum length of zero.
2. **Bounded Capacity :** The queue has finite length n, where n message can resides in it. If the queue is not full when a new message is sent, the message is placed in the queue and the sender can continue execution without waiting. The sender blocks until space is available in the queue if the link is full.
3. **Unbounded capacity :** The queue's length is infinite where any number of the messages can wait in it, the sender never blocks.

#### QUESTIONS

1. Write a note on process
2. Explain the process states in detail
3. Write a note on process control block
4. Explain the term scheduling queues

## Chapter - 4

# Threads

Operating System (S.Y.B.Sc. C.S./SEm-4)

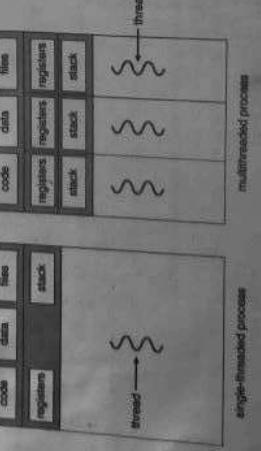
Threads

A thread is a stream of execution throughout the process code having its program counter which keeps track of basis of instruction to execute next, system registers bind its current working variables. Threads are also termed as lightweight processes. A thread uses parallelism which provides a way to improve application performance.

#### Major Types of Threads

Let us take an example where a web browser may have one thread to display images or text while another thread receives data from the network. Another example can be a word processor that may have a thread for displaying the UI or graphics while a word processor that may have a thread for responding to keystrokes received from the user and another thread to perform spelling and grammar checking in the background. In some cases, a single application may be required to perform several similar tasks.

A thread is a basic unit of CPU utilization which has a thread ID, a Program counter, a register set, and a stack.



#### Advantages/Benefits Of Threads

The advantages of multithreaded programming can be categorized into four major headings:

- Responsiveness :** Multithreading is an interactive concept for an application which may allow a program to continue running even when a part of it is blocked or is carrying a lengthy operation, which increases responsiveness to the user.

#### Testing and Debugging

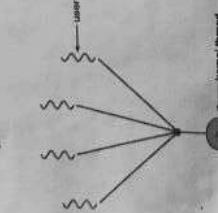
- Testing and debugging.** When a program is running in parallel on multiple cores, there are many different execution paths. Testing and debugging such concurrent programs is inherently more difficult than testing and debugging single-threaded applications.

#### Multithreading Models

All the threads must have a relationship between them [i.e., user threads and kernel threads]. Here is a list which tells the three common ways of establishing this relationship:

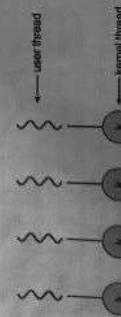
##### Many-to-One Model

In the one-to-one model maps every user thread to a kernel thread and provides more concurrency compare to many-to-one model.



##### One-to-One Model

In the one-to-one model maps every user thread to a kernel thread and provides more concurrency compare to many-to-one model.



##### Many-to-Many Model

In the many-to-many model, many user-level threads get mapped to a smaller or equal quantity of kernel threads. The number of kernel threads might be lesser to either a application or to a particular machine.

#### Resource sharing

Mostly threads share the memory and the resources of any process to which they fit in. The advantage of sharing code is that it allows any application to have multiple different threads of activity inside the same address space.

#### Economy :

In OS, allocation of memory and resources for process creation seems costly. Because threads can distribute resources of any process to which they belong; it became more economical to create and develop context-switch threads.

#### Utilization of multiprocessor architectures :

The advantages of multithreading can be greatly amplified in a multiprocessor architecture, where there exist threads which may run in parallel on diverse processors.

#### Multicore Programming

Multithreaded programming provides a mechanism for more efficient use of multiple cores and improved concurrency.

A recent trend in system design has been to place multiple computing cores on a single chip, where each core appears as a separate processor to the operating system.

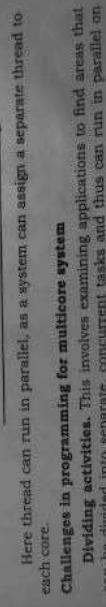
On a system with a single computing core, concurrency merely means that the execution of the threads will be interleaved over time



Concurrent execution on a single-core system.

#### Parallel Execution of a Multi-core system

|        | core 1 | T <sub>1</sub> | T <sub>3</sub> | T <sub>4</sub> | T <sub>5</sub> | T <sub>6</sub> | ... |
|--------|--------|----------------|----------------|----------------|----------------|----------------|-----|
| core 2 |        | T <sub>2</sub> | T <sub>4</sub> | T <sub>2</sub> | T <sub>4</sub> | T <sub>2</sub> | ... |



Here threads can run in parallel, as a system can assign a separate thread to each core.

#### Challenges in programming for multicore system

Dividing activities. This involves examining applications to find areas that can be divided into separate, concurrent tasks and thus can run in parallel on individual cores.

**Balance.** While identifying tasks that can run in parallel, programmers must also ensure that the tasks perform equal work of equal value. In some instances, a certain task may not contribute as much value to the overall process as other tasks; using a separate execution core to run that task may not be worth the cost.

**Data splitting.** Just as applications are divided into separate tasks, the data accessed and manipulated by the tasks must be divided to run on separate cores.

**Data dependency.** The data accessed by the tasks must be examined for dependencies between two or more tasks. In instances where one task depends

#### QUESTIONs

##### Objective Questions

###### 1. Multiple Choice Questions

1. Explain the concept of Thread in detail
2. What are the benefits of Multithreaded programming
3. Explain the concept of multicore programming
4. Explain the various types of multithreading models
5. What are the different challenges in multi core system?

###### 2. Objective Questions

1. This allows many user level threads to be mapped to many kernel threads
  - Process speed
  - Scheduling
  - Many-to-many model
  - None of these
2. Multi-processor system gives a
  - small system
  - loosely coupled system
  - tightly coupled system
  - both a and b
3. Multiprocessor system have advantage of
  - Expensive hardware
  - Increased Throughput
  - operating system
  - Scheduler
4. Multiprogramming of the computer system increases
  - Memory
  - Storage
  - CPU utilization
  - Cost
5. A common representation of process scheduling is a,
  - PCB
  - Scheduler
  - None of these
  - Quitting Diagram
6. Which of these is true about Operating System?
  - It acts as an interface between user and the computer hardware
  - It acts as an interface between user and the keyboard
  - It acts as an interface between the user and the keyboard hardware
  - It is a hardware.
7. Which of this is not a component of computer system?
  - Hardware
  - Operating system
  - Application program
  - Keyboard

#### 25

#### Operating System (S.Y.B.Sc. C.S./SEm-4)

#### Threads

on data from another, programmers must ensure that the execution of the tasks is synchronized to accommodate the data dependency.

**Testing and debugging.** When a program is running in parallel on multiple cores, there are many different execution paths. Testing and debugging such concurrent programs is inherently more difficult than testing and debugging single-threaded applications.

#### Multithreading Models

All the threads must have a relationship between them [i.e., user threads and kernel threads]. Here is a list which tells the three common ways of establishing this relationship:



#### Many-to-Many Model

In the many-to-many model, many user-level threads get mapped to a smaller or equal quantity of kernel threads. The number of kernel threads might be lesser to either a application or to a particular machine.





General structure of a process is shown in diagram above. Each process must request to get an entry in critical section. The section of code implementing this request is the entry section. If critical section is free (no other process is executing critical section), permission will be granted. Once the critical code is executed, process must exit the critical section so that other process can execute critical section.

**1. Mutual exclusion:** If process one process is executing in its critical section, then no other processes can enter critical sections.

**2. Progress:** Progress means that if a process is not using the critical section, then it should not stop any other process from accessing it. In other words, any process can enter a critical section if it is free.

**3. Bounded waiting:** Each process must have limited waiting time. No process should have to wait forever to enter the critical section.

#### 2.1.4 Peterson's Solution:

The Peterson's solution solves the critical section problem by satisfying all the three conditions listed above.

It is two process solution- P<sub>1</sub> and P<sub>2</sub>. When one process is in critical section, other process executes remainder section.

The two data items int turn and boolean flag are used to decide which process will enter critical section. Turn decides whose turn is it to enter a critical section. Flag has two values true and false. Flag decides whether a process is ready to execute critical code or not.

If turn==1, then Process P<sub>1</sub> is allowed to execute its critical section. If flag[i]==true, then P<sub>1</sub> is ready to enter the critical section. If turn==1 & flag[i]==true

For a process to be in critical section turn, (turn==1 & flag[i]==true).

True - E.g. if P<sub>1</sub> is in critical section, (turn==1 & flag[1]==true)

#### 1. Entry Section

- i) Process P<sub>1</sub> wants to enter the critical section. So set flag of P<sub>1</sub> as True.
- ii) After P<sub>1</sub>, P<sub>2</sub> has a next turn to enter the critical section. So set turn to 1.
- iii) Check for mutual exclusion. If P<sub>1</sub> is in critical section, Process P<sub>2</sub> cannot enter the critical section. If both turn and flag of P<sub>1</sub> are true, it means P<sub>1</sub> is in critical section. So process P<sub>2</sub> will wait. While leaving the critical section, process P<sub>1</sub> will set its flag as false and turns. So now P<sub>1</sub> will enter the critical section.

**Critical Section:** Now P<sub>1</sub> will execute critical section and modifies shared data/memory/file.

#### 2. Exit Section

- 1. Exit Section - P<sub>1</sub> will leave the critical section by setting flag false, so that P<sub>2</sub> can enter the critical section.
- 2. Data /Memory/File.

**Advantages:** - with the Peterson's Solution, Mutual exclusion is guaranteed, the progress requirement is satisfied and the bounded-waiting requirement is met.

**Drawback:** - It is only 2 process solution.

#### 2.1.5 Synchronization Hardware-

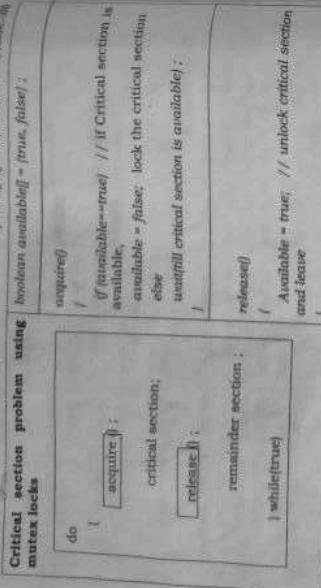
All these solutions are based on the concept of locking — that is, protecting critical regions through the use of locks. Some simple hardware instructions are available on many systems to solve the critical-section problem.

The critical-section problem can be solved in single processor systems if we prevent interrupts from occurring while a shared variable is being modified. In this way, we can ensure that the current set of instructions will be executed in order without any pre-emption. As interrupts are disabled, no other instructions can run, so no unexpected modifications could be made to the shared variable. This is often the approach taken by non-preemptive kernels.

But this solution is not as feasible in multiprocessor systems. Disabling interrupts on a multiprocessor can be time consuming, since the message is passed to all the processors. This message passing delays entry into each critical section, and system efficiency decreases.

#### 2.1.6 Mutex Locks-

A Mutex lock is a software tool to solve critical section problem. Mutual exclusion and ex. is exclusion. So Mutex stands for Mutual Exclusion. Mutex locks are used to protect critical section and thus to avoid race conditions.



Process must acquire lock before entering a critical section and process must release the lock before leaving. The acquire() function locks the critical region and release() function unlocks it. One Boolean variable available is used to check if critical section is free or not. If critical section is free, the lock is available, i.e. available=true. If critical section is 'busy' (locked), lock is unavailable i.e. available=false.

If the critical section is free, a call to acquire() succeeds, and the lock is then considered unavailable. A process that attempts to acquire an unavailable lock is blocked until the lock is released. Mutex locks are very simple to implement.

The main disadvantage of the implementation given here is that it requires busy waiting. While a process is in its critical section, any other process that tries to enter its critical section must loop continuously in the call to acquire(). In fact, this type of mutex lock is also called a spinlock because the process 'spins' while waiting for the lock to become available.

So we need more sophisticated ways for processes to synchronize their activities.

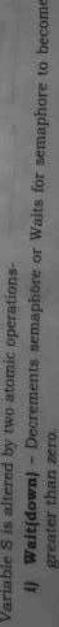
#### 2.1.7 Semaphores

A semaphore S is an integer variable which is used to achieve mutual exclusion for preventing race condition. Semaphore S is accessed only through two standard atomic operations: wait() and signal().

S can be +ve or -ve. If S=0, indicates shared resource is not available. Another process is using critical region. If S>0, indicates resource is available i.e. critical region is free.

Variable S is altered by two atomic operations-

- i) **Wait(down)** - Decrements semaphore or Waits for semaphore to become greater than zero.
- ii) **Signal(up)** - increments semaphore by 1.



A process which wants to access critical region, must follow following steps.

- i) Execute Wait.
- ii) If execution of wait is successful, access critical region. Decrement semaphore.
- iii) If execution of wait is unsuccessful, critical region is busy. Wait till process leaves the critical region and semaphore becomes greater than zero again.
- iv) Before leaving the critical region, increment semaphore by 1 by waking up other waiting processes so that another process can use it.

#### Types of Semaphores

##### 1. Binary Semaphores

Counting Semaphores is used to control access to the multiple resources available in the system. The variable S is assigned to number of resources available. Suppose S=5, i.e. 5 resources are available. Each process that requires a resource, executes wait operation. If resource is available, it will be allocated to the process and count of available resources is decremented by 1 (i.e. S=4). When process releases the source, it executes signal() operation and count of available resources will be incremented by 1 (i.e. S=5). When no resource is available (S = 0), process will block until count becomes greater than zero.

#### 2.1.8 Classic Problem of Synchronization:

All the synchronization problems can be solved by making use of semaphores. Here we are going to discuss three classic problems: Bounded Buffer Problem, Readers-Writers Problem and Dining Philosophers problem.

##### 1. Bounded Buffer Problem:

**Problem** - The bounded buffer problem [producer-consumer problem] is a classic example of concurrent access to a shared resource. Producers write data to the buffer and consumers read data from the buffer. Multiple producers and consumers share a single buffer.



- i) Producer tries to enter the data into full buffer.
- ii) Consumer tries to remove data from empty buffer.
- iii) Multiple producers try to write data in same buffer slots.
- iv) Multiple consumers try to read data from same buffer slots.

**Solution-**

All updates to the buffer state must be done in a critical section. So for the mutual exclusion **binary semaphore** is used.

If buffer has  $n$  slots. Producers must block if the buffer is full. Consumers must block if the buffer is empty. Two **counting semaphores** can be used for this. Suppose currently buffer is empty.

semaphore empty = n  
semaphore full = 0

```
void producer()
{
    while (TRUE)
    {
        void consumer()
        {
            while (TRUE)
            {
                wait (empty); // decrement no of
                empty slots; // enter critical
                region
                remove item;
                signal (mutex); // leave critical
                region
                signal (empty); // increment no of
                empty slots
            }
        }
    }
}
```

**2. Readers-Writers Problem-****Problem**

This deals with shared database. There is a shared database which multiple processes are accessing simultaneously for reading and writing. Multiple readers can read the database simultaneously. If two readers access the object at the same time, there is no problem. However, if two writers try to manipulate object at the same time, it will lead to inconsistent data. Also when one reader and one writer access the object at the same time, there may be problems.

To solve this situation, a writer should get exclusive access to the database. However, multiple readers can access the database at the same time. So mutual exclusion is required for writing-processes. Following semaphores are shared by readers and writers processes.

**Solution-**

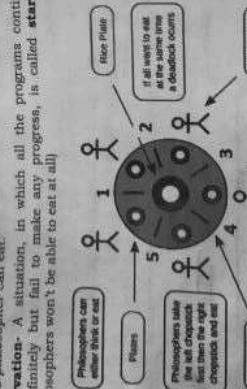
i.e. when a writer is accessing the database, no reader or writer may access it. However, multiple readers can access the database at the same time. So mutual exclusion is required for writing-processes. Following semaphores are shared by readers and writers processes.

**Deadlock**- All the philosophers took left chopstick simultaneously which as no philosopher can eat.

**Starvation**- A situation, in which all the programs continue to run indefinitely but fail to make any progress, is called **starvation**. All philosophers won't be able to eat at all.

Two problems can occur in this case-

- i) **Deadlock**- All the philosophers took left chopstick simultaneously which as no philosopher can eat.
- ii) **Starvation**- A situation, in which all the programs continue to run indefinitely but fail to make any progress, is called **starvation**. All philosophers can't eat at all.

**Figure 3: Dining Philosophers Problem****Solution-**

This problem is similar to the problem that arises in real computer programming when multiple programs need exclusive access to shared resources. Mutual exclusion is the basic idea of the problem. It can be achieved using semaphore and mutex variables.

Each philosopher can be in one of the 2 stages- Thinking and Eating

```
do
    THINK; // Thinking Stage
```

```
// pickup both left and right chopsticks if available and eat.
Else continue thinking stage
```

```
PICKUP(CHOPSTICK1), CHOPSTICK1 mod 5;
EAT,
PUTDOWN(CHOPSTICK1), CHOPSTICK1 mod 5;
```

```
semaphore write=1;
semaphore read = 1;
int reader_count = 0
```

**Writer Process-**

```
do {
    wait(write); // writer requests for critical section
    // Performs the write
    signal(write);
} while(true);
```

**Reader Process-**

```
do {
    wait(read); // Reader Wants to enter critical section
    reader_count++;
    if (reader_count == 1) // if there is at-least one reader in critical
    section,
        wait(write); // no writer can enter
    inside the signal(read); // other readers can enter while this reader is
    critical section
    // current reader performs reading here
    wait(read); // a reader wants to leave
    reader_count--;
    if (reader_count == 0) // if no reader is left in the critical section
    writer can enter
    signal(write);
} while(true);
```

When a reader wants to access the resource, first it increments the reader\_count value. Then, access the resource. After that, it decrements the reader\_count value. The semaphore write is used by the first reader (which enters in the Critical Section) and last reader (which leaves the Critical Section). This is because when the first readers enter in the Critical Section, the writer cannot enter in the Critical Section. Only new reader can access the critical section now.

Similarly, when the last reader leaves the Critical Section, it signals the write semaphore. This is because there are no readers that are using the critical section now. So, a writer can access the resource now.

One simple solution is to represent each chopstick with a semaphore. Each chopstick is numbered. A philosopher tries to grab a chopstick by executing a wait operation on that semaphore. She releases her chopsticks by executing the signal operation on the appropriate semaphores.

**2.1.9 Monitors**

A monitor is a collection of procedures, variables, and data structures that are all grouped together in a special kind of module or package. Procedures can call the procedures in a monitor, but they cannot directly access the monitor's internal data structures. Monitor is one of the ways to achieve Process synchronization. Monitor is supported by programming language to achieve mutual exclusion between processes.

**Structure of a Monitor**

1. It is the collection of condition variables and procedures combined together in a special kind of module or a package.
2. The processes running outside the monitor can't access the internal variable of monitor but they can call procedures of the monitor.
3. Only one process at a time can execute code inside monitors.

**Figure 4: Syntax and Structure of Monitor****Entry Queue**

A monitor guarantees mutual exclusion. Only one process can execute any monitor procedure at any time. If one process is in the monitor and all the same time second process invokes a monitor procedure, then second process blocks and placed in a waiting queue. So the monitor has to have a wait queue. If a process within a monitor blocks another one from the queue can enter.

**Queues associated with Condition Variables**

We also need a way for processes to block when they cannot proceed. In the producer-consumer problem, it is easy to write monitor procedures to wait-buffer-full and buffer-empty, but how should the producer blocks when it finds the

better full? E.g. In the producer-consumer problem, the producer must be woken when the buffer is full.

The solution lies in the introduction of condition variables, along with two operations on them, wait and signal. When a monitor procedure discovers that a buffer contains (e.g., the producer finds the buffer full), it does a wait on some condition variable, say, full. This causes the calling process to block. It also allows another process waiting in a queue to enter monitor now.

Let say we have 2 condition variables

// Declaring variable

**Wait operation**

x.wait(); Process performing wait operation on any condition variable is blocked. The suspended processes are placed in block queue of that condition variable. Note that each condition variable has its unique block queue.

**Signal operation**

x.signal(); When a process performs signal operation on condition variable, one or more processes waiting in a queue is given chance.

## 7.2 CPU SCHEDULING

### 7.2.1 Basic Concepts:

In multi-programming systems, several processes are kept in a memory at one time. When current process goes into the waiting state, operating system takes away CPU from current blocked process and gives CPU to another process stored in a memory. This pattern continues. Every time one process has to wait, another process can take over use of the CPU.

In multi-programmed operating systems, it is necessary to switch CPU rapidly from one process to another process to enhance speed and efficiency. CPU utilization is also improved as CPU doesn't sit idle. To do so, it is important to decide which process will run next, for how much time and how to switch from one process to another. Here, basic CPU scheduling concepts and scheduling algorithms are discussed.

#### CPU-I/O Burst Cycle:

Process execution consists of two cycles- CPU Execution and I/O Wait.

The Process execution continuously switches between CPU burst and I/O burst until the process terminates as shown in diagram.

CPU Burst- Amount of time process uses CPU.

I/O Burst- Amount of time required for I/O operations. So process is in waiting state [process is taken away from CPU] till I/O operations are carried out.



It is desirable to maximize CPU utilization and throughput and to minimize turnaround time, waiting time, and response time.

### 2.2.3 Scheduling Algorithms

#### 1. First Come First Served (FCFS).

This is the simplest scheduling algorithm used by operating system. Process which requests CPU first, is allocated the CPU first. Scheduler maintains the FIFO queue. The processes are arranged in First In First Out manner. When CPU is free, first process in a queue is scheduled next for execution.

**Example-**

Consider the following set of processes that arrive at time 0, with the length of the CPU burst given in milliseconds:

| Process        | Arrival Time | Burst Time |
|----------------|--------------|------------|
| P <sub>1</sub> | 0            | 9          |
| P <sub>2</sub> | 0            | 4          |
| P <sub>3</sub> | 0            | 5          |

If the processes arrive in the order P<sub>1</sub>, P<sub>2</sub>, P<sub>3</sub>, and are served in FCFS order, we get the result shown in the following Gantt chart, which is a bar chart that illustrates a particular schedule, including the start and finish times of each of the participating processes:

#### Gantt Chart-

|   | P <sub>1</sub> | P <sub>2</sub> | P <sub>3</sub> | PT                  |
|---|----------------|----------------|----------------|---------------------|
| 0 | 9              | 13             | 18             |                     |
|   |                |                |                | TAT = FT + AT       |
|   |                |                |                | Wait time = PT - BT |

Average Waiting time=  $(0+9+13)/3 = 7.33 \text{ ms}$

Average Turnaround Time=  $(9+13+18)/3 = 13.33 \text{ ms}$

**Note :-** FCFS scheduling algorithm is non-preemptive. Once the CPU is allocated to a process, that process keeps the CPU until it releases the CPU.

**Advantages-** Simple, easy to understand and implement.

#### Disadvantages-

- Average waiting time is more.
- High priority process will have to wait in queue.

#### 2. Shortest Job First (SJF)

This algorithm examines the burst time of each process in a queue. When CPU is available, the process with smallest burst time is scheduled for its execution.

#### Process Synchronization

#### CPU Schedules

When the process that CPU is currently executing goes in waiting state, operating system must select another process to execute. **CPU Schedules** or **short-term scheduler** is one that decides which process is to be executed next. The scheduler arranges the processes which are ready to execute in a queue depending upon algorithm. When current process goes into waiting state, scheduler schedules the next process in the queue for execution.

#### Preemptive and non-preemptive Scheduling

**Preemptive Scheduling:** Take the CPU away from current process and assign it to another process.

Suppose CPU is executing one process and another higher priority process is arrived. Then CPU is preempted from current process and higher priority process is executed first.

**Non-Preemptive Scheduling:** Once CPU is assigned to one process, it cannot be stopped until it completes the execution.

Suppose CPU is executing one process, and another higher priority process is arrived. In this case, CPU cannot be taken away from the current process until that process terminates.

#### Dispatcher-

A dispatcher is a special program which comes into play after the scheduler which takes that process to the desired state/queue. It is the module that gives control of the CPU to the process selected by the scheduler. This function involves the following:

- Switching to user mode.
- Jumping to the proper location in the user program to restart that program.
- The dispatcher should be as fast as possible, since it is invoked during every process switch. The time taken by the dispatcher to stop one process and start another is known as the **dispatch latency**.

Dispatch latency should be as low as possible.

#### 2.2.2 Scheduling Criteria

**CPU Utilization-** Keep the CPU busy all the time. Ideally CPU should not sit idle. In real system, it ranges from 40% (lightly loaded systems) to 90% (heavily loaded systems).

**Throughput-** Number of processes completed by CPU in unit time is known as Throughput. Throughput is a measure of CPU utilisation. Higher the throughput, higher is the CPU utilization, resulting in higher efficiency.

**Turn-around time (TAT)-** The time between submission of the process and the completion of the process. It is known as Turn-around time.

$$\text{TAT} = \text{Finish Time} - \text{Arrival Time}$$

**Burst Time-** Amount of time spent by the process in a ready queue is known as waiting time.

$$\text{Waiting time} = \text{TAT} - \text{Burst Time}$$

**Response time-** The amount of time from the submission of a request until the first response is produced. This measure, called response time, is the same as the time taken by the system to start responding.

#### Process Synchronization

When the process that CPU is currently executing goes in waiting state, operating system must select another process to execute. **CPU-burst** algorithm, because scheduling depends on the length of the time CPU burst of a process, rather than its total length. If the next CPU burst time of the next CPU burst of a process, rather than scheduling is used to break the tie.

This is non-preemptive SJF algorithm.

#### Example 1 - Arrival Times of each process is same.

Consider the following set of processes that arrive at time 0, with the length of the CPU burst given in microseconds:

| Process        | Arrival Time | Burst Time | Burst Time |
|----------------|--------------|------------|------------|
| P <sub>1</sub> | 0            | 0          | 9          |
| P <sub>2</sub> | 0            | 4          | 4          |
| P <sub>3</sub> | 0            | 5          | 5          |

Here, the sequence of execution will be P<sub>2</sub>  $\rightarrow$  P<sub>3</sub>  $\rightarrow$  P<sub>1</sub> as P<sub>2</sub> has minimum burst time as P<sub>1</sub>. We get the result shown in the following Gantt chart, which is a bar chart that illustrates a particular schedule, including the start and finish times of each of the participating processes.

#### Gantt Chart-

|   | P <sub>1</sub> | P <sub>2</sub> | P <sub>3</sub> | PT | TAT = FT + AT | Wait time = PT - BT |
|---|----------------|----------------|----------------|----|---------------|---------------------|
| 0 | 9              | 13             | 18             |    |               |                     |
|   |                |                |                |    |               |                     |
|   |                |                |                |    |               |                     |

Average Waiting time=  $(0+4+9)/3 = 4.33 \text{ ms}$

Average Turnaround Time=  $(4+9+18)/3 = 10.33 \text{ ms}$

**Advantages -** A compared to FCFS, average waiting time and average turnaround time is reduced in SJF algorithm.

#### Disadvantages-

- A process with high burst time may starve forever.
- High priority process with long CPU burst will have to wait in queue.
- Not a fair scheduling algorithm.

#### Example 2 : Arrival times of each process is different.

**Process**      **Arrival Time**      **Burst Time**

| Process        | Arrival Time | Burst Time |
|----------------|--------------|------------|
| P <sub>1</sub> | 0            | 0          |
| P <sub>2</sub> | 4            | 1          |
| P <sub>3</sub> | 5            | 2          |

**Gantt Chart**

- At instant 0, only one process  $P_1$  is arrived. So CPU starts executing  $P_1$ .
- At instant 1, another process  $P_2$  arrives with burst time less than  $P_1$ , as it is non-preemptive, once CPU starts executing one process, (CPU) cannot be taken away unless the process terminates. So CPU continues executing  $P_1$ .
- At instant 2, another process  $P_3$  arrives as shown in Gantt chart below.
- But as it is non-preemptive, CPU continues executing  $P_1$ .
- $P_1$  is completed at 3ms. At that point, CPU starts executing process with less burst time  $P_2$ . Once  $P_2$  is completed, CPU starts and finishes Process  $P_3$ .

| $P_1$   | 0       | 1       | 2       | 3 | $P_2$ | 3 | $P_3$ | 3 | 6 |
|---------|---------|---------|---------|---|-------|---|-------|---|---|
| $P_1=0$ | $P_1=8$ | $P_1=7$ | $P_1=0$ |   |       |   |       |   |   |
| $P_2=4$ |         | $P_2=4$ |         |   |       |   |       |   |   |
| $P_3=5$ |         |         | $P_3=5$ |   |       |   |       |   |   |
|         |         |         |         |   |       |   |       |   |   |

Average Waiting time =  $(0+8+11)/3 = 6.33 \text{ ms}$ Average Turnaround Time =  $(9+12+16)/3 = 12.33 \text{ ms}$ **Solve for FCFS and SJF.**

| Process | Arrival Time | Burst Time | Finish Time | Wait time = FT-BT-AT |
|---------|--------------|------------|-------------|----------------------|
| $P_1$   | 0            | 7          | 7           |                      |
| $P_2$   | 2            | 3          | 10          |                      |
| $P_3$   | 5            | 5          | 15          |                      |
| $P_4$   | 2            | 8          | 22          |                      |
| $P_5$   | 3            | 9          | 31          |                      |
| $P_6$   | 3            | 7          | 38          |                      |
| $P_7$   | 3            | 9          | 47          |                      |

Average Waiting time =  $(0+5+13+20+27)/6 = 12.17 \text{ units}$ Average Turnaround Time =  $(7+8+13+21+27+36)/6 = 18.67 \text{ units}$ 

| Process | Arrival Time | Burst Time | Finish Time | Wait time = FT-BT-AT |
|---------|--------------|------------|-------------|----------------------|
| $P_1$   | 0            | 10         | 10          |                      |
| $P_2$   | 1            | 15         | 25          |                      |
| $P_3$   | 2            | 30         | 55          |                      |
| $P_4$   | 3            | 39         | 94          |                      |
| $P_5$   | 3            | 39         | 133         |                      |

Average Waiting time =  $(0+8+11)/3 = 6.33 \text{ ms}$ Average Turnaround Time =  $(9+12+16)/3 = 12.33 \text{ ms}$ **Example 2 (MU Question Paper Apr 18)-****Solve for FCFS and SJF.**

| Process | Arrival Time | Burst Time | Finish Time | Wait time = FT-BT-AT |
|---------|--------------|------------|-------------|----------------------|
| $P_1$   | 0            | 9          | 9           |                      |
| $P_2$   | 1            | 13         | 22          |                      |
| $P_3$   | 2            | 8          | 30          |                      |
| $P_4$   | 2            | 11         | 41          |                      |
| $P_5$   | 3            | 7          | 48          |                      |
| $P_6$   | 3            | 9          | 57          |                      |

Average Waiting time =  $(0+5+13+20+27)/6 = 12.17 \text{ units}$ Average Turnaround Time =  $(7+8+13+21+27+36)/6 = 18.67 \text{ units}$ **3. Shortest Remaining Time First.**

Shortest remaining-time-first scheduling is also known as **Premptive SJF** algorithm. Job with less execution time is executed first. When new job comes Example

| Process | Arrival Time (ms) | Burst Time (ms) | Finish Time (ms) |
|---------|-------------------|-----------------|------------------|
| $P_1$   | 0                 | 9               | 9                |
| $P_2$   | 1                 | 4               | 13               |
| $P_3$   | 2                 | 5               | 18               |

**Gantt Chart-**

- At instant 0, only one process  $P_1$  is arrived. So CPU starts executing  $P_1$ .
- At time instant 1, another process  $P_2$  arrives with burst time 4ms. Till time 1,  $P_1$  is already executed for 1ms time and 8ms of process  $P_1$ 's remaining. So burst time of  $P_2$  is less than remaining time of  $P_1$ . As it is preemptive, CPU suspends  $P_1$  and starts executing process  $P_2$ .
- At instant 2, another process  $P_3$  arrives with burst time 5ms. Now at instant 2, remaining time of  $P_1$  is 5ms, remaining time of  $P_2$  is 3ms and burst time of  $P_3$  is 5ms. CPU continues executing  $P_2$  as  $P_2$ 's remaining time is shortest amongst all.
- $P_2$  is completed at 5ms. At that point CPU starts executing process with less burst time  $P_3$ . Once  $P_3$  is completed, CPU starts and finishes Process  $P_1$ .

| Process | Arrival Time | Burst Time | Finish Time | Wait time = FT-BT-AT |
|---------|--------------|------------|-------------|----------------------|
| $P_1$   | 0            | 2          | 2           |                      |
| $P_2$   | 1            | 4          | 5           |                      |
| $P_3$   | 2            | 5          | 10          |                      |

Average Waiting time =  $(0+4+5)/3 = 3.33 \text{ ms}$ Average Turnaround Time =  $(1+5+8)/3 = 4.33 \text{ ms}$ 

| Process | Arrival Time | Burst Time | Finish Time | Wait time = FT-BT-AT |
|---------|--------------|------------|-------------|----------------------|
| $P_1$   | 0            | 9          | 9           |                      |
| $P_2$   | 1            | 8          | 17          |                      |
| $P_3$   | 2            | 6          | 23          |                      |

Average Waiting time =  $(0+8+12)/3 = 8 \text{ ms}$ Average Turnaround Time =  $(9+17+23)/3 = 14.33 \text{ ms}$ **4. Priority Scheduling**

- A priority is associated with each process, and the CPU is allocated to the process with the highest priority.

**5. Round Robin Scheduling (RR)**

- Round Robin algorithm is also known as **Premptive FCFS scheduling NB** works in following manner.
  - Processes are arranged in a FIFO queue. Each process is assigned a time interval, called **quantum** or **Time Slice**.
  - When CPU is free, the first process from the queue is scheduled.
  - Once the quantum assigned is finished, CPU places that process at the end of FIFO queue and start executing next process as shown in diagram below.

Operating System (S.Y.B.Sc.C.S.) Sem.-4

#### Example-

Let us consider the same example taken above in non-preemptive Scheduling.

| Process        | Arrival Time [ms] | Burst Time [ms] |
|----------------|-------------------|-----------------|
| P <sub>1</sub> | 0                 | 9               |
| P <sub>2</sub> | 1                 | 4               |
| P <sub>3</sub> | 2                 | 5               |

#### Gantt Chart-

- At instant 0, only one process P<sub>1</sub> is arrived. So CPU starts executing P<sub>1</sub>.
- At time instant 1, another process P<sub>2</sub> arrives with burst time 4ms. Till time 1, P<sub>1</sub> is already executed for 1ms time and 8ms of process P<sub>1</sub> is remaining. So burst time of P<sub>2</sub> is less than remaining time of P<sub>1</sub>. As it is preemptive, CPU suspends P<sub>1</sub> and starts executing process P<sub>2</sub>.
- At instant 2, another process P<sub>3</sub> arrives with burst time 5ms. Now at instant 2, remaining time of P<sub>1</sub> is 8ms, remaining time of P<sub>2</sub> is 3ms and burst time of P<sub>3</sub> is 5ms. CPU continues executing P<sub>2</sub> as P<sub>2</sub>'s remaining time is shortest amongst all.
- P<sub>2</sub> is completed at 3ms. At that point CPU starts executing process with less burst time P<sub>3</sub>. Once P<sub>3</sub> is completed, CPU starts and finishes Process P<sub>1</sub>.

| Process                             | Arrival Time AT | Burst Time BT | Finish Time FT | Wait time= FT-BT-AT | TAT = FT-AT |
|-------------------------------------|-----------------|---------------|----------------|---------------------|-------------|
| P <sub>1</sub> =9                   | 0               | 9             | 9              | 0                   | 9           |
| P <sub>1</sub> =8 P <sub>1</sub> =6 | 1               | 4             | 5              | 0                   | 5           |
| P <sub>2</sub> =4 P <sub>2</sub> =3 | 2               | 5             | 10             | 3                   | 8           |

Average Waiting time=  $(9+0+3)/3 = 4 \text{ ms}$

Average Turnaround Time=  $(18+4+8)/3 = 10 \text{ ms}$

#### Priority Scheduling

A priority is associated with each process, and the CPU is allocated to the process with the highest priority.

#### Let Time Slice (Quantum) = 2ms

| Process        | Arrival Time | Burst Time | Owner process  | Quantum |
|----------------|--------------|------------|----------------|---------|
| P <sub>1</sub> | 0            | 9          | P <sub>1</sub> | 2       |
| P <sub>2</sub> | 0            | 4          | P <sub>2</sub> | 2       |
| P <sub>3</sub> | 0            | 5          | P <sub>3</sub> | 2       |

#### Gantt Chart

- As per FIFO queue, sequence of execution is P<sub>1</sub>  $\rightarrow$  P<sub>2</sub>  $\rightarrow$  P<sub>3</sub>.
- CPU starts executing process P<sub>1</sub>. As quantum is 2ms, P<sub>1</sub> will be executed for 2ms. After that P<sub>1</sub> is placed at the end of the queue. CPU executes P<sub>2</sub> for 2ms and after that P<sub>3</sub> is placed back at the end of the queue.
- This cycle continues until all the processes are finished.

| Process        | Arrival Time AT | Burst Time BT | Finish Time FT | Wait time= FT-BT-AT | TAT = FT-AT |
|----------------|-----------------|---------------|----------------|---------------------|-------------|
| P <sub>1</sub> | 0               | 5             | 5              | 0                   | 5           |
| P <sub>2</sub> | 0               | 2             | 7              | 5                   | 7           |
| P <sub>3</sub> | 0               | 2             | 9              | 7                   | 9           |

Average Waiting time=  $(20+37+30+32+39)/5 = 31.6 \text{ units}$

Average Turnaround Time=  $(30+49+37+42+54)/5 = 42.4 \text{ units}$

#### Multilevel Queue Scheduling

- Multi-level queue scheduling algorithm is used when we can classify processes into different groups.
  - In this scheduling, ready queue is partitioned into several separate queues.
  - Processes are divided into different queues based on their process type, memory size and process priority.
- Consider an example of a multilevel queue scheduling algorithm with five queues, listed below in order of decreasing priority:

  - System processes
  - Interactive processes
  - Interactive editing processes
  - Batch processes
  - Student processes

- Each queue has its own scheduling algorithm.

#### Example 2 (MU Question Paper Nov 17)

Consider following processes with time slice 5 units. Find average waiting for each process.

- Priorities are generally indicated by some fixed range of numbers e.g. 1 to 7, such as lower number represents higher priority.
- Priority scheduling can be either preemptive or non-preemptive.
- When a process arrives at the ready queue, its priority is compared with the priority of the currently running process.
- A preemptive priority scheduling algorithm will preempt the CPU if the priority of the newly arrived process is higher than the priority of the currently running process.
- A nonpreemptive priority scheduling algorithm will simply put the new process at the head of the ready queue.

#### Example

| Process        | Arrival Time [ms] | Burst Time [ms] | Priority |
|----------------|-------------------|-----------------|----------|
| P <sub>1</sub> | 0                 | 9               | 2        |
| P <sub>2</sub> | 0                 | 4               | 4        |
| P <sub>3</sub> | 2                 | 5               | 5        |

#### Gantt Chart

As per priorities, sequence of execution will be P<sub>3</sub>  $\rightarrow$  P<sub>1</sub>  $\rightarrow$  P<sub>2</sub>  $\rightarrow$  P<sub>2</sub>

| Process        | Arrival Time AT | Burst Time BT | Finish Time FT | Wait time= FT-BT-AT | TAT = FT-AT |
|----------------|-----------------|---------------|----------------|---------------------|-------------|
| P <sub>3</sub> | 0               | 5             | 5              | 0                   | 5           |
| P <sub>1</sub> | 5               | 9             | 14             | 5                   | 14          |
| P <sub>2</sub> | 14              | 4             | 17             | 3                   | 17          |

Average Waiting time=  $(0+5+14+17)/4 = 14.25 \text{ ms}$

#### Round Robin Scheduling (RR)

Round Robin algorithm is also known as Preemptive FCFS scheduling. RR works in following manner.

- Processes are arranged in a FIFO queue. Each process is assigned a time interval, called quantum or Time Slice.
- When CPU is free, the first process from the queue is scheduled.
- Once the quantum assigned is finished, CPU places that process at the end of FIFO queue and start executing next process as shown in diagram below.

Operating System (S.Y.B.Sc., C.S.I./Sem.-III)

| Process   | A  | B  | C | D  | E  | F  | G  | H  | I  | J  | K  | L  | M  | N  | O  | P  | Q  | R  | S  | T  | U  | V  | W  | X  | Y  | Z  |
|-----------|----|----|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| CPU Burst | 10 | 12 | 7 | 10 | 15 | 12 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 |

#### Gantt Chart.

| Process | Arrival Time AT | Burst Time BT | Finish Time FT | Wait time= FT-BT-AT | TAT = FT-AT |
|---------|-----------------|---------------|----------------|---------------------|-------------|
| A       | 0               | 5             | 5              | 0                   | 5           |
| B       | 5               | 5             | 10             | 5                   | 10          |
| C       | 10              | 5             | 15             | 5                   | 15          |
| D       | 15              | 5             | 20             | 5                   | 20          |
| E       | 20              | 5             | 25             | 5                   | 25          |
| F       | 25              | 5             | 30             | 5                   | 30          |
| G       | 30              | 5             | 35             | 5                   | 35          |
| H       | 35              | 5             | 40             | 5                   | 40          |
| I       | 40              | 5             | 45             | 5                   | 45          |
| J       | 45              | 5             | 50             | 5                   | 50          |
| K       | 50              | 5             | 55             | 5                   | 55          |
| L       | 55              | 5             | 60             | 5                   | 60          |
| M       | 60              | 5             | 65             | 5                   | 65          |
| N       | 65              | 5             | 70             | 5                   | 70          |
| O       | 70              | 5             | 75             | 5                   | 75          |
| P       | 75              | 5             | 80             | 5                   | 80          |
| Q       | 80              | 5             | 85             | 5                   | 85          |
| R       | 85              | 5             | 90             | 5                   | 90          |
| S       | 90              | 5             | 95             | 5                   | 95          |
| T       | 95              | 5             | 100            | 5                   | 100         |
| U       | 100             | 5             | 105            | 5                   | 105         |
| V       | 105             | 5             | 110            | 5                   | 110         |
| W       | 110             | 5             | 115            | 5                   | 115         |
| X       | 115             | 5             | 120            | 5                   | 120         |
| Y       | 120             | 5             | 125            | 5                   | 125         |
| Z       | 125             | 5             | 130            | 5                   | 130         |

Average Waiting time=  $(20+37+30+32+39)/5 = 31.6 \text{ units}$

#### 6. Multilevel Queue Scheduling-

- Processes are divided into different queues based on their process type, memory size and process priority.
- Consider an example of a multilevel queue scheduling algorithm with five queues, listed below in order of decreasing priority:

  - System processes
  - Interactive processes
  - Interactive editing processes
  - Batch processes
  - Student processes

- Each queue has its own scheduling algorithm.

Operating System (S.Y.B.Sc., C.S.I./Sem.-III)

Consider following processes with time slice 5 units. Find average waiting for each process.

- This process continues in Round Robin manner.
  - Fair scheduling scheme.
  - No starvation. All the processes will get equal chance.
- Example 1**
- Let Time Slice (Quantum) = 2ms**

| Process        | Arrival Time | Burst Time |
|----------------|--------------|------------|
| P <sub>1</sub> | 0            | 9          |
| P <sub>2</sub> | 0            | 4          |
| P <sub>3</sub> | 0            | 5          |

Gantt Chart

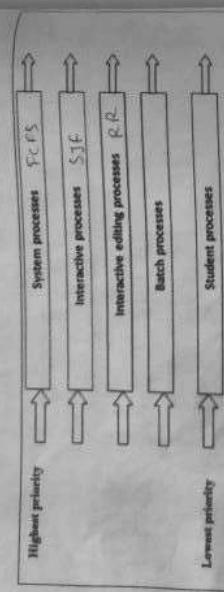
- As per FIFO queue, sequence of execution is P<sub>1</sub>  $\rightarrow$  P<sub>2</sub>  $\rightarrow$  P<sub>3</sub>.
- CPU starts executing process P<sub>1</sub>. As quantum is 2ms, P<sub>1</sub> will be executed for 2ms. After that P<sub>1</sub> is placed at the end of the queue. Then CPU executes P<sub>2</sub> for 2ms. After that P<sub>2</sub> is placed back at the end of the queue.
- This cycle continues until all the processes are finished.

| Process        | Arrival Time | Burst Time | Finish Time | Wait time= FT-BT-AT | TAT = FT-AT |
|----------------|--------------|------------|-------------|---------------------|-------------|
| P <sub>1</sub> | 0            | 9          | 10          | 12                  | 14          |
| P <sub>2</sub> | 0            | 4          | 10          | 14-9=9              | 18          |
| P <sub>3</sub> | 0            | 5          | 15          | 15-9=6              | 18          |

Average Waiting time =  $(9+6)/3 = 5$  msAverage Turnaround Time =  $(18+10+15)/3 = 14.33$  ms**Example 2 (MU Question Paper Nov 17)**

Consider following processes with time slice 5 units. Find average waiting for each process.

- Operating System / S.Y.B.Sc. C.S.I./Sem. 4**
- For example given below, queue1 (system process) uses FCFS (First Come First Serve), queue2 (interactive process) uses SJF (Shortest Job First) while queue3 uses RR (Round Robin) to schedule the processes and so on.



- 7. Multilevel Feedback Queue Scheduling**
- Each queue has absolute priority over lower priority queues. No process in the batch queue can run unless the queues for system processes, interactive processes, and interactive editing processes are all empty.
  - If an interactive editing process entered the ready queue while a batch process was running, the batch process would be preempted.
  - Here in multilevel feedback scheme, processes are classified based on CPU burst time.
  - Consider the following example with three queues numbered from 0 to 2 with queue 0 at highest priority and queue 2 at lowest priority.
  - Any process with burst time less than or equal to 8 milliseconds is given a highest priority i.e. is placed in queue 0.
  - Processes that need more than 8 but less than 24 milliseconds are also served quickly, although with lower priority than shorter processes.
  - This scheme generally keeps interactive processes in high priority queue.
  - To solve starvation problem, if process is in waiting state for too long in a lower priority queue, it is moved to higher priority queue.
  - A process entering the ready queue is put in queue 0. A process in queue 0 is executed for 8ms quantum. If it does not finish within this time, it is moved to the end of queue 1.
  - A process entering the ready queue is put in queue 0. A process in queue 0 is executed for 8ms quantum. If it does not finish within this time, it is moved to the end of queue 1.

**Advantages**

- Fair scheduling scheme.
- No starvation. All the processes will get equal chance.

**Example 1****Let Time Slice (Quantum) = 2ms**

| Process        | Arrival Time | Burst Time |
|----------------|--------------|------------|
| P <sub>1</sub> | 0            | 9          |
| P <sub>2</sub> | 0            | 4          |
| P <sub>3</sub> | 0            | 5          |

**Gantt Chart**

- As per FIFO queue, sequence of execution is P<sub>1</sub>  $\rightarrow$  P<sub>2</sub>  $\rightarrow$  P<sub>3</sub>.
- CPU starts executing process P<sub>1</sub>. As quantum is 2ms, P<sub>1</sub> will be executed for 2ms. After that P<sub>1</sub> is placed at the end of the queue. Then CPU executes P<sub>2</sub> for 2ms. After that P<sub>2</sub> is placed back at the end of the queue.
- This cycle continues until all the processes are finished.

**Let Time Slice (Quantum) = 2ms**Average Waiting time =  $(20+37+30+32+39)/5 = 31.6$  unitsAverage Turnaround Time =  $(30+49+37+42+54)/5 = 42.4$  units**6. Multilevel Queue Scheduling**

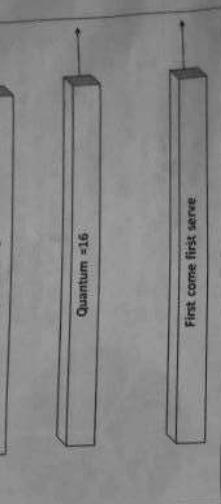
- Processes are divided into different queues based on their process type, memory size and process priority.
- Consider an example of a multilevel queue scheduling algorithm with five queues, listed below in order of decreasing priority:

  - System processes
  - Interactive processes
  - Interactive editing processes
  - Batch processes
  - Student processes

- Each queue has its own scheduling algorithm.

**process Synchronization**

- If queue 0 is empty, the process at the beginning of queue 1 is given a quantum of 16 milliseconds. If it does not complete, it is preempted and is put into queue 2.
- Processes in queue 2 are run on an FCFS basis but are run only when queues 0 and 1 are empty.

**2.2.4 Thread Scheduling**

- There are two types of threads in a system: Kernel Level Threads and User Level Threads. Some operating systems provide a combined user level thread and kernel level thread facility. In a combined thread approach, multiple threads within the same application can run in parallel on multiple processors and a blocking system call need not block the entire process. Multithreading models are three types.
- Many to Many Mapping**
  - Many to One Mapping**

- The many-to-many model multiplexes many number of user threads onto an equal number of kernel threads. In this model, developers can create as many user threads as required and the corresponding kernel threads can run in parallel on a multiprocessor machine. This model provides the best accuracy on concurrency and when a thread performs a blocking system call, the kernel can schedule another thread for execution.
- 2. Many to One Mapping**
- This model maps many user-level threads to one kernel-level thread. Thread management is done in user space by the thread library. When thread makes a blocking system call, the entire process will be blocked. Only one thread can access the kernel at a time, so multiple threads can run in parallel on the multiprocessors. If the user-level thread libraries are implemented in the operating system in such a way that the system does not support them, then the kernel threads use the many-to-one relationship mode.
- 3. One to One Mapping**
- There is one-to-one relationship of user-level thread to the kernel-level thread. This model provides more concurrency than the many-to-one model. It also allows another thread to run when a thread makes a blocking system call. It supports multiple threads to execute in parallel on multiprocessors.

### 2.3 DEADLOCK

#### 2.3.1 System Model - Resources:

Resource is a physical or virtual component available in a system to perform a requested operation.

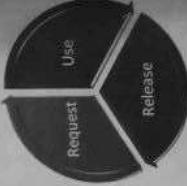
Physical resources- Every device connected to a system e.g. CPU, printer, scanner, etc.

Virtual Resources- Files, network connection, memories, etc.

A process must request a resource before using it, and must release the resource after using it. A process can request as many resources it requires to perform the desired task.

A process must acquire a resource in following sequence:

- Request-** Process places a request for resource, if resource is available, it will be granted to a process. Otherwise process will wait till the resource is available.
- Use-** Process performs desired task using the resource.
- Release-** Process must release the resource after using the resource.



#### Deadlock

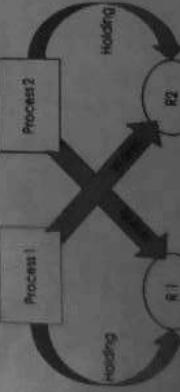
##### Introduction

In multiprogramming systems, multiple processes run at the same time and they share limited number of resources available in the system. So processes compete for finite number of resources. Process places a request for resource; if resource is available, it will be granted to a process. Otherwise process will wait till the resource is available.

##### Definition

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

The cause of deadlock: Each process needing what another process has. And the program cannot proceed further.



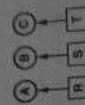
#### Example

- Two processes A and B, each want to scan a document and record it on a CD.

### 2.3.2 Deadlock Characterization

#### Necessary Conditions for occurrence of deadlock

- Mutual exclusion-** Resources are non-shareable. Resource should be used by a single process at a time.
- Hold-and-wait-** Processes currently holding resources can request new resources that are being held by other processes.



All these four conditions must hold good for a deadlock to occur in a system.

#### Resource Allocation Graph:

##### Deadlocks are represented precisely in terms of a directed graph called a system resource-allocation graph.

#### 3. No-preemption

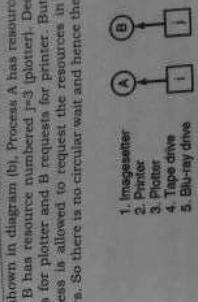
Resources previously granted cannot be forcibly taken away from a process. They must be released by the process holding them.

- Circular wait condition-** There must be a circular list of two or more processes, each of which is waiting for a resource held by the next member of the chain. Waiting processes-  $[P_0, P_1, P_2, \dots, P_n]$  are such that  $P_0$  is waiting for a resource that is held by  $P_1$ ,  $P_1$  is waiting for a resource that is held by  $P_2$ , and so on.  $P_n$  is waiting for resource that  $P_0$  has.

#### 4. Attacking Circular Wait:

To attack circular wait, assign a unique number to each resource type and shown in diagram 1(a) below. Each process requests the resources in increasing order of assigned numbers.

- A process can request first a printer and then an image scanner.
- As shown in diagram 1(b), Process A has resource numbered 1=2 (printer), and process B has resource numbered 1=3 (scanner). Deadlock can occur only when A requests for printer and B requests for scanner. But B cannot request for printer as process is allowed to request the resources in increasing order of assigned numbers. So there is no circular wait and hence there is no deadlock.



#### (a) Numerically ordered resources. (b) A resource graph.

#### 4.2.3.4 Deadlock Avoidance-

Deadlock avoidance mechanism requires that the system has some additional information about the resources- Each process must decide the maximum number of resources of each type that it may need before it begins its execution.

The deadlock avoidance algorithm 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.

#### Safe state

##### Definition

A state is said to be **safe** if there is some scheduling order in which every process before it begins execution. If all resources are available, process will start its execution. If one or more resources are busy, then process will go into waiting state.

Problem: many processes do not know how many resources they will need until they start running.  
Alternative solution is to allow process to request resources only when it doesn't have any resource. A process may request some resources and use them. Before it can request any additional resources, it must release all the resources that it is currently allocated.

#### Operating System (S.Y.B.Sc. C.S.I) Sem.-II

#### Process Synchronization

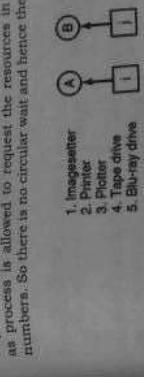
##### 3. Attacking no preemption:

- Resources assigned to the process can be taken away to prevent deadlock situation. If a process is holding some resources and requests another resource, that cannot be immediately allocated to it if that is the process currently holding all resources the process is currently holding are preempted. Some resources can be virtually held by the process.

##### 4. Attacking Circular Wait:

To attack circular wait, assign a unique number to each resource type and shown in diagram 1(a) below. Each process requests the resources in increasing order of assigned numbers.

- A process can request first a printer and then a tape drive, but it cannot request first a plotter and then an image scanner.
- As shown in diagram 1(b), Process A has resource numbered 1=2 (plotter), and process B has resource numbered 1=3 (image scanner). Deadlock can occur only when A requests for printer and B requests for image scanner. But B cannot request for image scanner as process is allowed to request the resources in increasing order of assigned numbers. So there is no circular wait and hence there is no deadlock.



#### (a) Numerically ordered resources. (b) A resource graph.

#### 4.2.3.4 Deadlock Avoidance-

Deadlock avoidance mechanism requires that the system has some additional information about the resources- Each process must decide the maximum number of resources of each type that it may need before it begins its execution.

The deadlock avoidance algorithm 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.

#### Safe state

##### Definition

A state is said to be **safe** if there is some scheduling order even if all of them suddenly request their maximum number of resources immediately. A safe state is not a deadlocked state. An unsafe state may lead to a deadlock.

#### Example-

Suppose total 10 resources are available and three processes  $P_1, P_2$  and  $P_3$  want to access resources. The processes have already acquired the few resources and also declared the maximum resources required for the execution as shown below-

#### Operating System (S.Y.B.Sc. C.S.I) Sem.-II

#### Process Synchronization

##### 3. Attacking Hold and Wait:

- In the above section we have seen necessary conditions for occurrence of deadlock. We can prevent the deadlock by assuring that at least one of the four conditions fails to hold good. So deadlock can be prevented by attacking the above mentioned conditions.

##### 1. Attacking Mutual Exclusion-

- If the resources are sharable, mutual exclusion is not required. In order to prevent the deadlock, make resources sharable. A process never needs to wait for a sharable resource. E.g. read only file can be accessed by many users at a time. But some resources cannot be sharable. E.g. Mutex locks- they cannot be shared by several processes at a time. So in these cases we cannot avoid mutual exclusion condition.

##### 2. Attacking Hold and Wait-

- To avoid hold and wait condition, allocate all the resources needed for a process before it begins execution. If all resources are available, process will start its execution. If one or more resources are busy, then process will go into waiting state.

Problem: many processes do not know how many resources they will need until they start running.

Alternative solution is to allow process to request resources only when it doesn't have any resource. A process may request some resources and use them. Before it can request any additional resources, it must release all the resources that it is currently allocated.

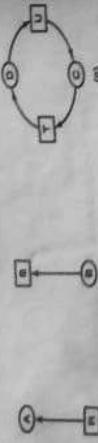
The graphs have two types of nodes:

1. Circles- Processes are shown as circles

2. Squares- Resources are shown as squares

A directed arc from a resource (square) to a process (circle) means that the resource is currently held by that process.

A directed arc from a process to a resource means that the process is currently waiting for that resource, as shown in diagram below-



- resource R1 assigned to process A

- process B is requesting/waiting for resource S

Process D has resource T and waiting for resource U (which process D has)

Process C has resource V and waiting for resource W (which process D has)

So diagram (b) represents deadlock.

If the graph contains no cycles, then no process in the system is deadlocked.

If the graph contains a cycle, then a deadlock exists in a system as shown in fig (c).

### 3.3 Deadlock Prevention-

In the above section, we have seen necessary conditions for occurrence of deadlock. We can prevent the deadlock by assuring that at least one of the four conditions fails to hold good. So deadlock can be prevented by attacking the above mentioned conditions.

#### 1. Attacking Mutual Exclusion-

If the resources are sharable, mutual exclusion is not required. In order to prevent the deadlock, make resources sharable. A process never needs to wait for a sharable resource. E.g. read only file can be accessed by many users at a time. But some resources cannot be sharable. E.g. Mutex locks, they cannot be shared by several processes at a time. So in these cases we cannot avoid mutual exclusion condition.

#### 2. Attacking Hold and Wait-

To avoid hold and wait condition, allocate all the resources needed for a process before it begins execution. If all resources are available, process will start its execution. If one or more resources are busy, then process will go into waiting state.

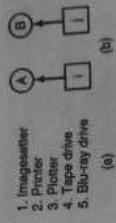
Problem: many processes do not know how many resources they will need until they start running.

Alternative solution is to allow process to request resources only when it doesn't have any resource. A process may request some resources and use them. Before it can request any additional resources, it must release all the resources that it is currently allocated.

#### 3. Attacking Circular Wait-

- To attack circular wait, assign a unique number to each resource (fig 3.3-1).
- shown in diagram (a), below. Each process requests the resources in increasing order of assigned numbers.
- A process can request first a printer and then a tape drive, but it cannot request first a plotter and then sin imagecutter.

As shown in diagram (b), Process A has resource numbered 1 (printer) and process B has resource numbered 2 (plotter). Deadlock can occur only when A requests for Plotter and B requests for printer. But B cannot request for printer as process is allowed to request the resources in increasing order of assigned numbers. So there is no circular wait and hence there is no deadlock.



(a) Numerically ordered resources.

(b) A resource graph.

#### 2.3.4 Deadlock Avoidance-

Deadlock avoidance mechanism requires that the system has some additional information about the resources-

Each process must declare the maximum number of resources of each type that it may need before it begins its execution. The deadlock avoidance algorithm 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.

#### Safe state

##### Definition

A state is said to be **safe** if there is some scheduling order, in which every process can run to completion even if all of them suddenly request their maximum number of resources immediately.

#### Example:-

Suppose total 10 resources are available and three processes  $P_1$ ,  $P_2$  and  $P_3$  want to access resources. The processes have already acquired the few resources and also declared the maximum resources required for the execution as shown below-

|       | Free: 0 | Free: 1 | Free: 2 | Free: 3 | Free: 4 | Free: 5 | Free: 6 | Free: 7 |
|-------|---------|---------|---------|---------|---------|---------|---------|---------|
|       | (a)     | (b)     | (c)     | (d)     | (e)     | (f)     | (g)     | (h)     |
| $P_1$ | 3       | 9       | 3       | 9       | 3       | 9       | 3       | 9       |
| $P_2$ | 2       | 4       | 4       | 0       | 0       | 0       | 0       | 0       |
| $P_3$ | 2       | 7       | 2       | 7       | 2       | 7       | 2       | 7       |

Operating System (S.Y.B.Sc.C.S.I./PDM-02)

As shown in fig (a),  $P_1$  requires maximum 9 resources out of which it already has 3.  $P_2$  has 2 and its maximum requirement is 4 and  $P_3$  has 2 and it needs 7 for its execution. So out of  $10 = 3+3+2 > 7$  resources, are already allocated, which mean only 3 resources are available.

Now  $P_1$  needs 6,  $P_2$  needs 2 and  $P_3$  needs 5 more resources to complete the execution. Only 3 resources are available, so  $P_2$  can immediately be allocated 2 resources as shown in fig (b).

Now  $P_2$  has all the resources it needs to complete its execution. As  $P_2$  terminates, it releases all the resources it had as shown in fig (c). Now available resources are 5.

As shown in fig (d) and fig (e),  $P_3$  is granted with 5 resources. Now  $P_3$  has all the resources required for its completion. As soon as  $P_3$  finishes the task, it releases all the resources it had.

Now  $P_1$ 's request can be fulfilled and all the three processes can finish the execution without any deadlock. So  $P_3 \rightarrow P_2 \rightarrow P_1$  is a safe sequence in which we can execute the processes without deadlock in the system.

#### Banker's Algorithm

It follows the safety algorithm and deadlock avoidance algorithm. It is modelled in the way a banker might deal with a group of customers to whom he has granted loan.

#### 1. Data structures needed to implement Banker's algorithm are -

a.  $n$  = Number of processes in system ( $P_1$ ,  $P_2$ , ...,  $P_n$ )

b.  $m$  = Number of Resource Classes.

c.  $E$  = Existing resource matrix.

e.g. Suppose there are four resource classes - Tape drives, plotters, scanners and bio-rays. There are 4 tape drives, 2 plotters, 3 scanners and 1 bio-ray available in the system.

d. Allocation

$$\text{Allocation} = \begin{bmatrix} P_1 & 0 & 0 & 1 \\ P_2 & 2 & 0 & 0 \\ P_3 & 0 & 1 & 0 \end{bmatrix}$$

$$\text{Allocation} = \begin{bmatrix} P_1 & 0 & 0 & 1 \\ P_2 & 2 & 0 & 0 \\ P_3 & 0 & 1 & 0 \end{bmatrix}$$

Need =  $\text{Need} = \text{Request}$

Need =  $\text{Need} = \text{Request}$

Process Signification

As shown in diagram Process  $P_1$  currently has no tape drive, no plotters, 1 scanner and 0 bio rays.  $P_2$  currently has 2 tape drives and 1 bio-ray.  $P_3$  has 1 plotter and 2 scanners.

e) Max =  $n \times m$  matrix which defines maximum number of resources each process needs to complete its execution.

f) Available = A vector of length  $m$  indicates number of available resources of each type after the allocation.

Out of 4 tape drives, 2 are already allocated to processes so now available tape drives are 2. Same is true for remaining resources. After allocation, 2 tape drives, 1 plotter, 0 scanners and 0 bio-rays are available.

g) Need =  $n \times m$  matrix indicating how many resources are required by each process in addition to complete its execution.

As per the diagram, Process  $P_1$  needs 2 more tape drives and 1 bio-ray to complete its execution.  $P_2$  needs 2 more tape drives and 1 scanner and  $P_3$  requires 2 tape drives and 1 plotter to complete its execution.

#### 2. Safety Algorithm-

This algorithm is designed to determine whether the system is in safe state or not.

a) Let  $\text{Finish}$  be vector of length  $n$  indicating whether a process is finished or not.

Initialize  $\text{Finish}[i] = \text{False}$  for  $i = 1, 2, \dots, n$  (Process i is not finished)

b) Find process  $i$  such that  $\text{Finish}[i] == \text{False}$

Verify if resources needed are less than availability  $\text{Need} \leq \text{Available}$

If yes, allocate needed resources to process. Update allocation matrix.

$\text{Allocation} = \text{Allocation} + \text{Need}$

If no such process exists, go to step d.

c) Process  $i$  finishes and releases all the allocated resources.

$\text{Available} = \text{Available} + \text{Allocation}[i]$

$\text{Finish}[i] = \text{True}$

Go to step b.

d) If  $\text{Finish}[i] == \text{true}$  for all  $i$ , then system is in safe state.

#### 3. Resource-request algorithm-

This algorithm is designed to verify if the request can be safely granted.

e) Let Request be the request matrix of process  $i$ .

If  $\text{Request} \leq \text{Need}$  go to step b.

Else error - process has exceeded maximum claimed resource.

b) If  $\text{Request} \leq \text{Available}$  go to step c.

Else  $\text{Process}$  wait since resources are not available.

c) Update the matrices-

$\text{Available} = \text{Available} - \text{Request}$

$\text{Allocation} = \text{Allocation} + \text{Request}$

$\text{Need} = \text{Need} - \text{Request}$

### Examples on Banker's Algorithm.

#### Example 1:

|                | Allocation |   |       |   | Max   | Available |
|----------------|------------|---|-------|---|-------|-----------|
|                | A          | B | C     | D | A B C | A B C     |
| P <sub>0</sub> | 0 1 0      |   | 7 5 3 |   | 3 3 2 |           |
| P <sub>1</sub> | 2 0 0      |   | 3 2 2 |   |       |           |
| P <sub>2</sub> | 3 0 2      |   | 9 0 2 |   |       |           |
| P <sub>3</sub> | 2 1 1      |   | 2 2 2 |   |       |           |
| P <sub>4</sub> | 0 0 2      |   | 4 3 3 |   |       |           |

To find-

1. Need Matrix

2. Safe state and sequence

**Ans :**

Need = Max - Allocation

|                | Allocation |       |       |   | Max   | Available | Need  |
|----------------|------------|-------|-------|---|-------|-----------|-------|
|                | A          | B     | C     | D | A B C | A B C     | A B C |
| P <sub>0</sub> | 0 1 0      | 7 5 3 | 3 3 2 |   | 7 4 3 |           |       |
| P <sub>1</sub> | 2 0 0      | 3 2 2 |       |   | 1 2 2 |           |       |
| P <sub>2</sub> | 3 0 2      | 9 0 2 |       |   | 6 0 0 |           |       |
| P <sub>3</sub> | 2 1 1      | 2 2 2 |       |   | 0 1 1 |           |       |
| P <sub>4</sub> | 0 0 2      | 4 3 3 |       |   | 4 3 1 |           |       |

Compare each processes' Need matrix with Available and find down a process such that Need[i] <= Available

1. For process P<sub>1</sub>, Need [P<sub>1</sub>] <= Available. So P<sub>1</sub> will be granted all the resources required.

Available= Available- Need=(3 3 2) - (1 2 2) = (2 1 0)

Allocation (P<sub>1</sub>) = Max= (3 2 2)

P<sub>1</sub> completes its execution and releases all the resources.

Available= Allocation (P<sub>1</sub>) + Available = (3 2 2) + (2 1 0) = (5 3 2)

Available= Available- Need=(5 3 2) - (0 1 1) = (5 2 1)

Allocation (P<sub>2</sub>) = Max= (2 2 2)

P<sub>2</sub> completes its execution and releases all the resources

Available= Allocation (P<sub>2</sub>) + Available = (2 2 2) + (5 2 1) = (7 4 3)

3. For process P<sub>4</sub>, Need (P<sub>4</sub>) <= Available. So P<sub>4</sub> will be granted all the resources required.

Available= Available- Need=(7 4 3) - (4 3 1) = (3 1 2)

Allocation (P<sub>4</sub>) = Max= (4 3 3)

Compute need array. Check whether system is in safe state or not. Also compute safe sequence.

#### Ans :

Need = Max - Allocation

|                | Allocation |         |         |         | Max     | Available | Need    |
|----------------|------------|---------|---------|---------|---------|-----------|---------|
|                | A          | B       | C       | D       | A B C D | A B C D   | A B C D |
| P <sub>1</sub> | 1 2 2 1    | 3 2 2   | 3 1 1 2 | 2 1 0 1 |         |           |         |
| P <sub>2</sub> | 1 0 3 3    | 1 2 3 4 | 0 2 0 1 |         |         |           |         |
| P <sub>3</sub> | 1 2 1 0    | 1 3 5 0 | 0 1 4 0 |         |         |           |         |

1. Start with P<sub>1</sub>. Compare Need with Available resources.

Here Need <= Available. So P<sub>1</sub> will be granted all the resources required.

Available= Available- Need=(3 1 1 2) - (2 1 0 1) = (1 0 1 1)

Allocation (P<sub>1</sub>) = Max= (3 2 2)

P<sub>1</sub> completes its execution and releases all the resources.

Available= Allocation (P<sub>1</sub>) + Available = (3 2 2) + (1 0 1 1) = (4 3 3 3)

2. Look for process P<sub>2</sub>. Compare Need with Available resources.

Here Need <= Available. So P<sub>2</sub> will be granted with all the resources required.

Available= Available- Need=(4 3 3 3) - (0 2 0 1) = (4 3 3 2)

Allocation (P<sub>2</sub>) = Max= (1 2 3 4)

P<sub>2</sub> completes its execution and releases all the resources.

Available= Allocation (P<sub>1</sub>) + Available = (1 2 3 4) + (4 1 3 2) = (5 3 6 6)

3. Look for process P<sub>3</sub>. Compare Need with Available resources.

Here Need <= Available. So P<sub>3</sub> will get all the resources required.

Available= Available- Need=(5 3 6 6) - (0 1 4 0) = (5 2 2 6)

Allocation (P<sub>3</sub>) = Max= (1 3 5 0)

P<sub>3</sub> completes its execution and releases all the resources.

Available= Allocation (P<sub>2</sub>) + Available = (1 3 5 0) + (5 2 2 6) = (6 5 7 6)

So all the processes are executed without any deadlock. We can conclude that System is in safe state.

**Safe sequence** = P<sub>1</sub> → P<sub>2</sub> → P<sub>3</sub>

#### 2.3.5 Deadlock Detection

If a system fail to prevent deadlock or deadlock avoidance algorithm fails to work, then Deadlock is going to occur in a system. So it is necessary to study deadlock detection algorithm. We are going to study deadlock detection algorithm for two different situations- (1) Deadlock detection with single resource of each type (2) Deadlock Detection with multiple resources of each type.

##### 1. Deadlock detection with single resource of each type

Consider a system with one resource of each type.

E.g. System with 1 printer, 1 scanner, 1 plotter and 1 CD recorder.

##### Steps to detect deadlock

1. Construct a resource graph.

2. If this graph contains one or more cycles, a deadlock exists.

### Process Synchronization

P<sub>0</sub> completes its execution and releases all the resources.

4. for process P<sub>0</sub>, Need (P<sub>0</sub>) <= Available = (4 3 3) + (3 1 2) = (7 4 5).

Available= Available- Need=(7 4 5) - (7 4 3) = (0 0 2)

Allocation (P<sub>0</sub>) = Max= (7 5 3)

P<sub>0</sub> completes its execution and releases all the resources.

Available= Available- Need=(7 5 3) + (0 0 2) = (10 5 7)

5. for process P<sub>2</sub>, Need (P<sub>2</sub>) <= Available. So P<sub>0</sub> will be granted all the resources required.

Available= Available- Need=(7 5 3) - (6 5 0) = (1 5 5)

Allocation (P<sub>2</sub>) = Max= (9 0 2)

P<sub>2</sub> completes its execution and releases all the resources.

Available= Available- Need=(9 0 2) + (1 5 5) = (10 5 7)

So all the processes are executed without any deadlock. We can conclude that System is in safe state.

**Safe sequence** = P<sub>1</sub> → P<sub>3</sub> → P<sub>4</sub> → P<sub>0</sub>

### Process Synchronization

If no cycle exists, the system is run deadlocked.

#### Example 2 (MU Nov 18)

Assume-

1. Total Resources in a System- A B C D

2. Available System Resources are- A B C D

3. Currently Allocated Resources-

|                | A | B | C | D |
|----------------|---|---|---|---|
| P <sub>1</sub> | 1 | 2 | 2 | 1 |
| P <sub>2</sub> | 1 | 0 | 3 | 3 |
| P <sub>3</sub> | 1 | 2 | 1 | 0 |

4. Maximum resources required by each process

|                | A | B | C | D |
|----------------|---|---|---|---|
| P <sub>1</sub> | 3 | 3 | 2 | 2 |
| P <sub>2</sub> | 1 | 2 | 3 | 4 |
| P <sub>3</sub> | 1 | 3 | 5 | 0 |

Conclusion

In above graph, there exists a cycle as shown above.

Processes A, B and C are deadlocked.

#### 2. Deadlock Detection with multiple resources of each type-

It is a matrix-based algorithm for detecting deadlock.

Let total P processes want to run in resource classes are available.

E is the existing resource vector with E<sub>i</sub> resources of class m.

E = {E<sub>1</sub>, E<sub>2</sub>, E<sub>3</sub>, ..., E<sub>m</sub>}

E.g. Consider system with 3 processes and 4 resource classes with 4 tape drives, 2 platters, 3 scanners and 1 Blu-ray.

P = 3 processes (P<sub>1</sub>, P<sub>2</sub>, P<sub>3</sub>)

m=4 resource classes

Let C be current allocation matrix. C<sub>j,i</sub> is the number of instances of resource j that are held by process i.

Let A be the available resource vector, with A<sub>i,j</sub> unallocated resources of class i, upto Am unallocated resources of class m.

Let R be the request matrix. R<sub>j,i</sub> is the number of instances of resource j that P<sub>i</sub> wants.

Process i wants.

A = {A<sub>1</sub>, A<sub>2</sub>, A<sub>3</sub>, ..., A<sub>m</sub>}

Let R be the request matrix. R<sub>j,i</sub> is the number of instances of resource j that P<sub>i</sub> wants.

R = {R<sub>1</sub>, R<sub>2</sub>, R<sub>3</sub>, ..., R<sub>m</sub>}

- The deadlock detection algorithm can now be given as follows.
- Look for process P<sub>i</sub> for which the i<sup>th</sup> row of R is less than or equal to A.
  - If such a process is found, add the i<sup>th</sup> row of C to A, mark the process.
  - and go back to step 1.
  - If no such process exists, the algorithm terminates.
- Consider request matrix 1, request of P<sub>3</sub> will be fulfilled. So no deadlock in system. Consider request matrix 2, request of neither of the processes will be fulfilled. So all processes are stuck and there is a deadlock in system.

### 2.3.6 Recovery from Deadlock

Once the deadlock detection is carried out by Deadlock Detection algorithms, next would be recover a system from deadlock. The recovery can be done through following measures-

#### 1. Process Termination

- i) Abort one process at a time:**
- Abort one process at a time, run deadlock detection algorithm to check whether deadlock is solved or not.
  - If not continuous step (a) until deadlock cycle is eliminated.

#### ii) Abort all deadlocked processes-

Abstoring all deadlocked processes can eliminate deadlock immediately, but there are consequences. The aborted processes may need to run again as result of the computation are discarded.

Selecting a process to abort is a difficult task. It depends on priority of the processes, its type, how long the process is running, how much computation the process is done, the types and number of resources process has, etc. That process should be aborted whose termination will incur minimum cost.

**iii) Starvation-**

To eliminate the deadlock, take away some resources from processes and give these pre-empted resources to other processes. In case of preemption, following factors need to be considered.

- iv) Selecting a victim-** Which resource to be taken away from which process? The process which has acquired the resources but currently doing some other tasks is chosen in this case. Temporarily take away a resource from its current owner and give it to another process.
- v) Rollback-** Once we select a process to be preempted, it cannot continue with its normal execution as some resources are taken away from the process. Hence the solution is to rollback that process to some safe state and restarts it again from that state.

- vi) Starvation-** If the same process is selected as a victim to pre-empt again and again, it will lead to starvation. As a result, the started process won't be able to finish the desired task. So we need to count number of rollbacks for each process. If rollbacks for a particular process exceed the threshold value, that process cannot be pre-empted.

#### QUESTIONS

1. Write a note on Semaphores

2. Define Deadlock. What are the characteristics of Deadlock?

3. Explain scheduling with its criteria

4. Explain FCFS scheduling algorithm with example.

5. Explain SJF scheduling algorithm with example.

6. Explain Priority scheduling with example.

7. Explain RR scheduling algorithm with example

8. What are the different methods of handling Deadlock?

9. What are different Dead lock prevention techniques?

10. What are different Deadlock Avoidance techniques?

11. Explain Resource allocation graph.

12. How to recover from the deadlock?

13. Explain Pre-emption and non-pre-emption techniques.

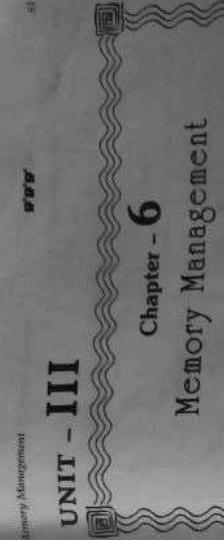
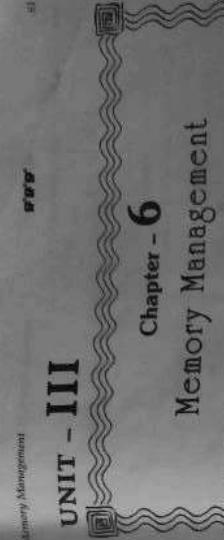
14. Write a short note on Thread scheduling.

#### Objective Questions

##### 1. Multiple Choice Questions

1. The software layer providing the virtualisation is called \_\_\_\_\_
  - Application Virtual Memory
  - Virtual Machine Monitor
  - System Virtual Monitor
  - Process Virtual Machine
2. You can close a file by using \_\_\_\_\_
  - Closed System Call
  - Create System Call
  - VPS stands for \_\_\_\_\_
  - VPS stands for \_\_\_\_\_
3. VPS stands for \_\_\_\_\_
  - Virtual Private Service
  - Virtual Private Server
  - Virtual Private Scan
  - Virtual Private Service
4. Unix \_\_\_\_\_ call is used for creating child process.
  - NEW
  - EXIT
  - CREATE
  - FORK
5. A Virtual machine is less efficient than a real machine when it access the hardware indirectly.
  - TRUE
  - False
  - Cannot be determined
  - None
6. Memory are normally classified according to their
  - speed
  - cost
  - Index
  - RAM
7. Static program of computer system are stored in
  - RAM
  - ROM
  - Hard Disk
  - CD
8. Which of the following condition is required for deadlock to be possible?
  - mutual exclusion
  - a process may hold allocated resources while awaiting assignment of other resources
  - no resource can be forcibly removed from a process holding it
  - all of the mentioned

#### 2. Operating System (S.Y.B.Sc., C.S.E., I.T.) (Sem. III)



6. 9. A system is in the safe state if:
  - the system can allocate resources so each process in some order and still avoid a deadlock.
  - there exist a safe sequence
  - all of the mentioned
  - none of the mentioned
10. Which one of the following is the deadlock avoidance algorithm?
  - banker's algorithm
  - round-robin algorithm
  - elevator algorithm
  - karn's algorithm

[Ans.: (1 - b), (2 - a), (3 - c), (4 - d), (5 - a), (6 - b), (7 - b), (8 - d), (9 - a), (10 - a)]

#### B. True or False

- Critical section is a code of segment that access shared variable and has to be executed as an atomic action on behalf of one process at a time.
  - Manual exclusion is, when one process executing in its critical section at the same time any other process can execute in its critical section.
  - A monitor is a collection of data structure, variables and procedures and it grouped together.
  - CPU scheduling is to make the system efficient, fast and fair.
  - PCFC meaning First come First serve.
  - I/O Burst Cycle consists of read and write operations.
  - Pre-emption process will not halt its execution because of some high priority process.
  - Waiting Time of process is less in Non-pre-emptive scheduling.
  - SJF stand for Shortest Job First.
  - Round Robin Scheduling Algorithm is designed for Time sharing System.
- [Ans.: True : 1, 3, 4, 5, 6, 8, 10   False : 2, 7, 9]

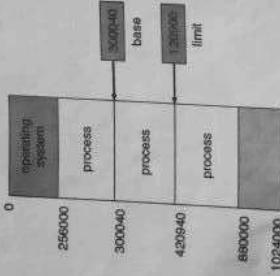


- MEMORY MANAGEMENT**
- Memory management is the functionality of an operating system, which handles or manages primary memory and moves processes back and forth between main memory and disk during execution. Memory management keeps track of each memory location either it is allocated to some process or it is free. It checks how much memory is to be allocated to processes. It decides which process will get memory at what time. It tracks whenever some memory gets freed or unallocated and correspondingly it updates the status.
- Responsibilities of memory management are:**
- To provide a detailed description of various ways of organizing memory hardware
  - To discuss various memory-management techniques, including paging and segmentation.

- To provide a detailed description of the Intel Pentium, which supports both page segmentation and segmentation with paging. Program must be brought from disk into memory and placed within a process for it to be run.
- Main memory can take many cycles.
  - Register access in one CPU clock (or level)
  - Main memory can take many cycles.
  - Cache sits between main memory and CPU registers.
  - Protection of memory required to ensure correct operation

#### MAIN MEMORY

- Memory consists of a large array of words or bytes each having its own address. The CPU fetches instructions from the memory according to the value of address. The program counter to improve the utilization of CPU, the computer uses several processes in the memory. To utilize the memory many memory management schemes are proposed. Selection of a memory management scheme depends on many factors such as hardware of the system.



#### ADDRESS BINDING

A user program goes through several steps such as compiling, loading, and linking. Addresses may be represented in different ways during these steps. Addresses in the source program are generally symbolic. A compiler will bind these addresses to relocatable addresses. The loader will in turn bind them to absolute addresses. Each binding is a mapping from one address space to another.

The binding of instructions and data can be done in the following ways.

1. Compile time - If it is known at compile time where the process will reside in memory, then absolute code can be generated. If address changes at compile time, the program is recompiled.
2. Load time - If it is not known at compile time where the process will reside in the memory, than the compiler must generate relocatable code.
3. Execution time - Binding delayed until run time if the process can be moved during its execution from one memory segment to another. Need hardware support for address maps. [e.g. base and limit registers].

#### LOGICAL V/S PHYSICAL ADDRESS SPACE

- The address generated by the CPU is a **logical address**, whereas the address actually seen by the memory hardware is a **physical address**.
- Addresses bound at compile time or load time have identical logical and physical addresses.
- Addresses created at execution time, however, have different logical and physical addresses.
- In this case the logical address is also known as a **virtual address**, and the two terms are used interchangeably by our text.
- The set of all logical addresses used by a program comprises the **logical address space**, and the set of all corresponding physical addresses comprises the **physical address space**.
- The run time mapping of logical to physical addresses is handled by the **memory-management unit, MMU**.
- The MMU can take on many forms. One of the simplest is a modification of the base-register scheme described earlier.
- The base register is now termed a **relocation register**, whose value is added to every memory request at the hardware level.
- Note that user programs never see physical addresses. User programs work entirely in logical address space, and any memory references or manipulations are done using purely logical addresses. Only when the address gets sent to the physical memory chips is the physical memory address generated.

#### Address Binding

• The address generated by the CPU is a logical address, whereas the address actually seen by the memory hardware is a physical address.

• Addresses bound at compile time or load time have identical logical and physical addresses.

• Addresses created at execution time, however, have different logical and physical addresses.

• In this case the logical address is also known as a **virtual address**, and the two terms are used interchangeably by our text.

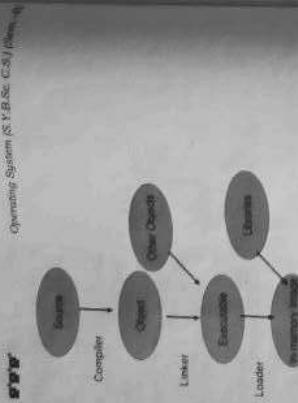
• The set of all logical addresses used by a program comprises the **logical address space**, and the set of all corresponding physical addresses comprises the **physical address space**.

• The run time mapping of logical to physical addresses is handled by the **memory-management unit, MMU**.

• The MMU can take on many forms. One of the simplest is a modification of the base-register scheme described earlier.

• The base register is now termed a **relocation register**, whose value is added to every memory request at the hardware level.

• Note that user programs never see physical addresses. User programs work entirely in logical address space, and any memory references or manipulations are done using purely logical addresses. Only when the address gets sent to the physical memory chips is the physical memory address generated.



#### Dynamic Loading

Routine is not loaded until it is called.

Useful when large amounts of code are needed to handle in frequently occurring cases.  
There is no special support from the OS is required implemented through program design.

#### Dynamic Linking

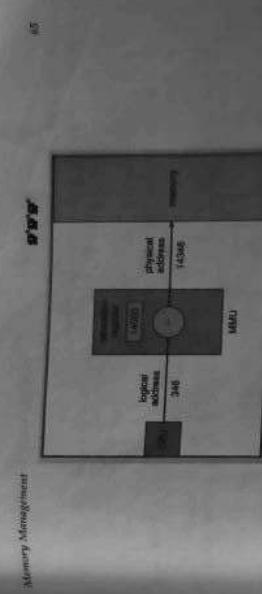
Linking postponed until execution time.  
\* Small piece of code, stub, used to locate the appropriate memory-resident library routine.  
\* Stub replaces itself with the address of the routine, and executes the routine.

\* Operating system needed to check if routine is in processes' memory address.  
\* Dynamic linking is particularly useful for libraries.

#### Memory management unit:

A memory management unit translates addresses between the CPU and physical memory. This translation process is often known as memory mapping because addresses are mapped from a logical space into a physical space.

Hardware device that maps virtual to physical addresses - 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.



#### e.g.

1. Here in the diagram the CPU has generated logical address for 346.
2. For which the MMU has generated relocation register for 140000.
3. In Memory physical address is located at (346+140000 = 14346).

#### Swapping

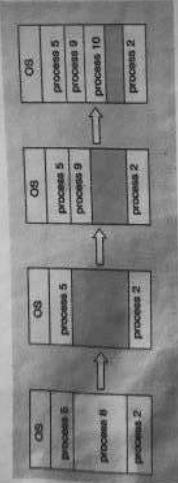
Swapping is a mechanism in which a process can be swapped temporarily out of main memory for move to secondary storage (disk) and make that memory available to other processes.

At some later time, the system swaps back the process from the secondary storage to main memory. Though performance is usually affected by swapping processes but it helps in running multiple and big processes in parallel and that's the reason swapping is also known as a technique for memory compaction. The total time taken by swapping process includes the time it takes to move the entire process to a secondary disk and then to copy the process back to memory, as well as the time the process takes competing to regain main memory.

Roll out, roll in - It is a swapping variant used for priority-based scheduling algorithms; lower-priority process is swapped out as higher priority process can be loaded and executed.

Major part of swap time is transfer time until transfer time is directly proportional to the amount of memory swapped.

- Contiguous Allocation**
  - Main memory usually divide into two partitions.
  - Resident operating system, usually held in low memory with interrupt vector.
  - User processes then held in high memory
  - Each (Single partition) allocation scheme used to protect user processes from each other and from changing operating system code and data.
  - Relocation register contains value of smallest physical address; limit register contains range of logical addresses – each logical address must be less than the limit register.
  - Multiple-partition allocation
    - In this type of allocation, main memory is divided into several fixed-sized partitions where each partition should contain only one process. When a partition is free, a process is selected from the input queue and is loaded into the free partition. When the process terminates, the partition becomes available for another process.



#### Dynamic Storage Allocation Problem

How to satisfy 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.

First-fit and best-fit better than worst-fit in terms of speed and storage utilization

#### Fragmentation

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

#### External fragmentation

- If exists when there is enough total memory space available to satisfy a request, but available memory space are not contiguous.
- Storage space is fragmented into large number of small holes.
- Both first fit and best fit suffers suffer from this.

#### Example:

- Consider a process is divided into 4 pages P0, P1, P2 and P3.
- Depending upon the availability, these pages may be stored in the main memory frames in a non-contiguous fashion as shown:
  - Process 1 is stored in frame F1 (P0, P1).
  - Process 2 is stored in frame F2 (P2, P3).
  - Process 3 is stored in frame F3 (P0).
  - Process 4 is stored in frame F4 (P1).

#### Translating Logical Address into Physical Address

- CPU always generates a logical address.
- A physical address is needed to access the main memory.

Following steps are followed to translate logical address into physical address-

#### Step-01:

CPU generates a logical address consisting of two parts-

- Page Number
- Page Offset

#### Step-02:

Page Number specifies the specific page of the process from which CPU wants to read the data.

Page Offset specifies the specific word on the page that CPU wants to read.

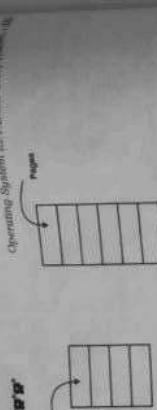
#### Step-03:

For the page number generated by the CPU,

- Page Table provides the corresponding frame number (base address of the frame) where that page is stored in the main memory.

#### Step-04:

The frame number combined with the page offset forms the required physical address.



- Paging is a fixed size partitioning scheme.
- In paging, secondary memory and main memory are divided into equal fixed size partitions.
- The partitions of secondary memory are called as pages.
- The partitions of main memory are called as frames.
- The partitions of main memory are called as frames.

#### Advantages:

The advantages of paging are-

- It allows to store parts of a single process in a non-contiguous fashion.
- It solves the problem of external fragmentation.
- The disadvantages of paging are-
  - It suffers from internal fragmentation.
  - There is an overhead of maintaining a page table for each process.

- Non-Contiguous Memory Allocation
- Non-Contiguous Memory Allocation is a memory allocation technique.
- It allows to store parts of a single process in a non-contiguous fashion.
- Thus, different parts of the same process can be stored at different places in the main memory.

#### Segmentation

- The goal is to shuffle the memory content, so as to place all free memory together in one large block.
- It is not always possible due to
  - If relocation is static and done at assembly or load time.
  - If relocation is dynamic and is done at execution time.

#### It is possible

- Only if relocation is dynamic and is done at execution time.
- Permit the logical address space to the processes to be non-contiguous.

#### Non-Contiguous Memory Allocation

- Non-contiguous memory allocation is a memory allocation technique.
- It allows to store parts of a single process in a non-contiguous fashion.
- Thus, different parts of the same process can be stored at different places in the main memory.

#### Techniques

- There are two popular techniques used for non-contiguous memory allocation.

#### Segmentation

- Segmentation is a memory partitioning scheme.
- Segments are the logical units of a program.
- Segments are mapped into pages.
- Pages are mapped into frames.

#### Disadvantages:

- It increases the complexity of the system.
- It increases the memory management overhead.
- It increases the memory access time.

- The time taken to fetch the instruction increases since now two memory accesses are required.

#### Page Table

Page Table is a data structure used by the virtual memory system to store the mapping between logical addresses and physical addresses.  
Logical addresses are generated by the CPU for the pages of the processes therefore they are generally used by the processes.  
Physical addresses are the actual frame address of the memory. They are generally used by the hardware or more specifically by RAM subsystems.

The image given below considers,

Physical Address Space =  $\log 2 M = m$  bits

Logical Address Space =  $M$  words

Page Size =  $P$  words

Page offset =  $\log 2 P = p$  bits

Physical Address =  $\log 2 M = m$  bits

Logical Address Space =  $M$  words

Page Size =  $P$  words

Page offset =  $\log 2 P = p$  bits



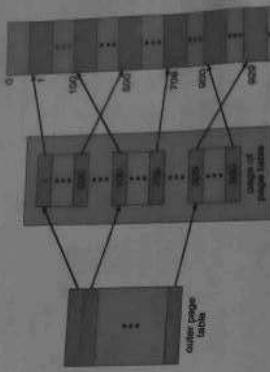
The CPU always accesses the processes through their logical addresses. However, the main memory recognizes physical address only.

In this situation, a unit named as Memory Management Unit comes into the picture. It converts the page number of the logical address to the frame number of the physical address. The offset remains same in both the addresses.

To perform this task Memory Management unit needs a special kind of mapping which is done by page table. The page table stores all the Frame numbers corresponding to the page numbers of the page table.

In other words, the page-table maps the page number to its actual location (frame number) in the memory.

In the image given below shows, how the required word of the frame is accessed with the help of offset.



A logical address from a 32-bit machine with 1K page size is divided into -  $m$  page number consisting of 12 bits - a page offset consisting of 10 bits

- Since the page table is paged, the page number is further divided into - a 12-bit page number - a 10-bit page offset
- Thus, a logical address is as follows:

page number . page offset

$p_1 \quad p_2 \quad p_3 \quad p_4$

$12 \quad 10 \quad 10$

- where  $p_1$  is an index into the outer page table, and  $p_2$  is the displacement within the page of the inner page table

#### Hashed Page Table

Common in address spaces  $> 32$  bits

- The virtual page number is hashed into a page table
  - This page table contains a chain of elements hashing to the same location
  - Each element contains
    - (1) the virtual page number
    - (2) the value of the mapped page frame
    - (3) a pointer to the next element
  - Virtual page numbers are compared in this chain searching for having small RAM

#### Advantages and Disadvantages of Paging

- This reduces external fragmentation but still suffer from internal fragmentation
- This is simple to implement and assumed as an efficient method
- Due to equal size of the pages and frames, swapping becomes very easy.

#### Segmentation

- Till now, we were using Paging as our main memory management technique. Paging is closer to Operating system, rather than the User. It divides all the functions which needs to be loaded in the same page.
- Operating system doesn't care about the User's view of the process. It may divide the same function into different pages and those pages may or may not be loaded at the same time into the memory. It decreases the efficiency of the system.
- It is better to have segmentation which divides the process into the segments. Each segment contains same type of functions such as main function can be included in one segment and the library functions can be included in the other segment.

In Operating Systems Segmentation is a memory management technique in which, the memory is divided into the variable size parts. Each part is known as Segment. The details about each segment are stored in a table called as segment table.

Segment table contains mainly two information about segment:

- Base : It is the base address of the segment
- Limit : It is the length of the segment

#### Translation of Logical address into Physical address by segment table

CPU generates a logical address which contains two parts:

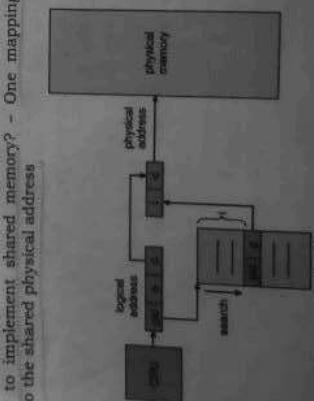
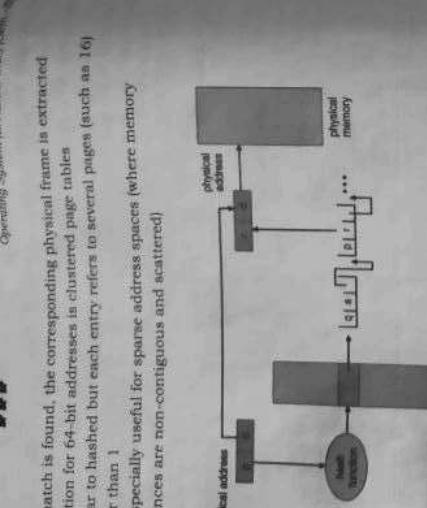
- Segment Number
- Offset

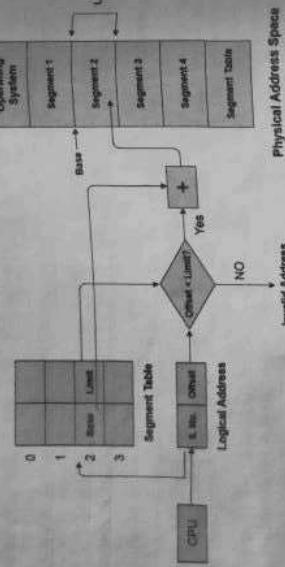
- The Segment number is mapped to the segment table. The limit of the respective segment is compared with the offset. If the offset is less than the limit then the address is valid otherwise it throws an error as the address is invalid.
- In the case of valid address, the base address of the segment is added to the offset to get the physical address of actual word in the main memory.

#### Inverted Page Table :

- Rather than each process having a page table and keeping track of all possible logical pages, - track all physical pages
- One entry for each real page of memory
- Entry consists of - the virtual address of the page stored in that real memory location, - information about the process that owns that page
- Decreases memory needed to store each page table - but increases time needed to search the table when a page reference occurs
- Use hash table to limit the search to one/few page-table entries - TLB can accelerate access

- But how to implement shared memory? - One mapping of a virtual address to the shared physical address



**Advantages of Segmentation**

1. No internal fragmentation
  2. Average Segment Size is larger than the actual page size.
  3. Less overhead.
  4. It is easier to relocate segments than entire address space.
  5. The segment table is of lesser size as compare to the page table in paging.
- Disadvantages**
1. It can have external fragmentation.
  2. It is difficult to allocate contiguous memory to variable sized partition.
  3. Costly memory management algorithms.
- Virtual Memory**
- In practice, most real processes do not need all their pages, or at least not all at once, for several reasons:
    1. Error handling code is not needed unless that specific error occurs, some of which are quite rare.
    2. Arrays are often over-sized for worst-case scenarios, and only a small fraction of the arrays are used in practice.
  - 3. Certain features of certain programs are rarely used, such as the routine to balance the federal budget. ↗
  - The ability to load only the portions of processes that were needed (and only when they were needed) has several benefits:
    - o Programs could be written for a much larger address space (virtual memory space) than physically exists on the computer.
    - o Because each process is only using a fraction of their total address space, there is more memory left for other programs, improving CPU utilization and system throughput.
    - o Less I/O is needed for swapping processes in and out of RAM, speeding things up.

**Benefits of having Virtual Memory**

1. Large programs can be written, as virtual space available is huge compared to physical memory.
2. Less I/O required, leads to faster and easy swapping of processes.
3. More physical memory available, as programs are stored on virtual memory, so they occupy very less space on actual physical memory.
4. System libraries can be shared by mapping them into the virtual address space of more than one process.
5. Processes can also share virtual memory by mapping the same block of memory to more than one process.
6. Process pages can be swapped during a fork() system call, eliminating the need to copy all of the pages of the original (parent) process.

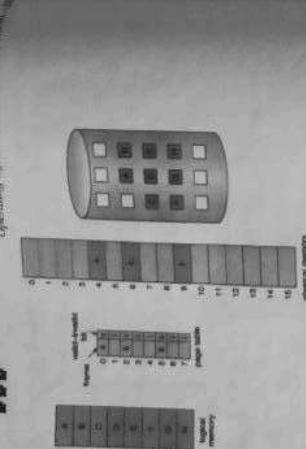
**Virtual memory is implemented using the following techniques:**

- Demand paging
- Demand Segmentation

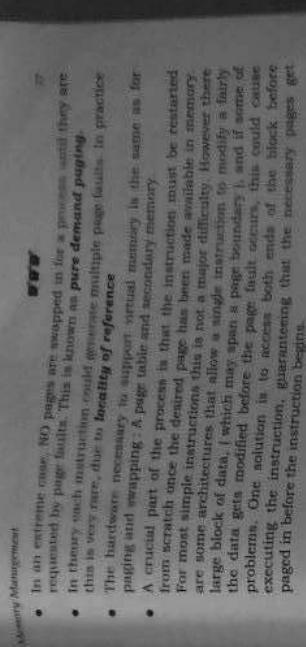
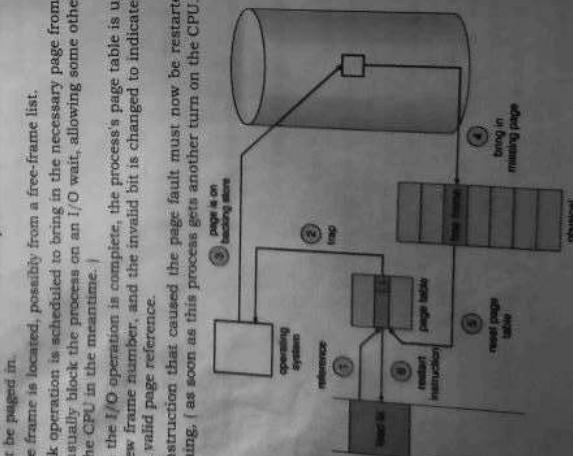
**Demand Paging :**

- The basic idea behind **demand paging** is that when a process is swapped in, its pages are not swapped in all at once. Rather they are swapped in only when the process needs them (on demand). This is termed a **lazy swapper**, although a **pager** is a more accurate term. Pages that are not loaded into memory are marked as **invalid** in the page table, using the least significant bit of the page table entry may either be blank or invalid bit. (The rest of the page table entry may contain information about where to find the swapped-out page on the hard drive.)
- If the process only ever accesses pages that are loaded in memory (memory resident pages), then the process runs exactly as if all the pages were loaded in to memory.

e) Operating System (S.Y.B.Sc.-C.S.I) (Sem.-III)



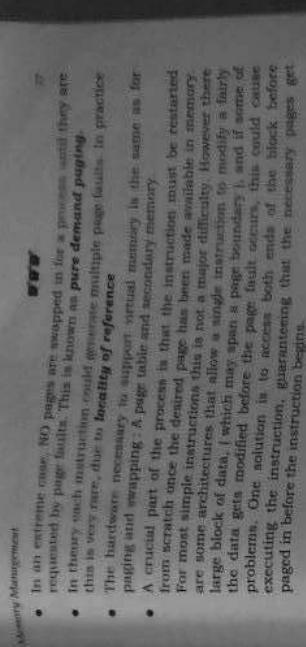
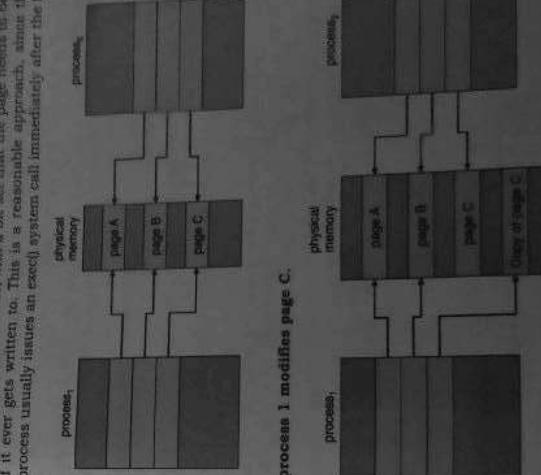
- On the other hand, if a page is needed that was not originally loaded up, then a **page fault trap** is generated, which must be handled in a series of steps:
1. The memory address requested is first checked, to make sure it was a valid memory request.
  2. If the reference was invalid, the process is terminated. Otherwise, the page must be paged in.
  3. A free frame is located, possibly from a free-frame list.
  4. A disk operation is scheduled to bring in the necessary page from disk. (This will usually block the process on an I/O wait, allowing some other process to use the CPU in the meantime.)
  5. When the I/O operation is complete, the process's page table is updated with the new frame number, and the invalid bit is changed to indicate that this is now a valid page reference.
  6. The instruction that caused the page fault must now be restarted from the beginning, (as soon as this process gets another turn on the CPU.)



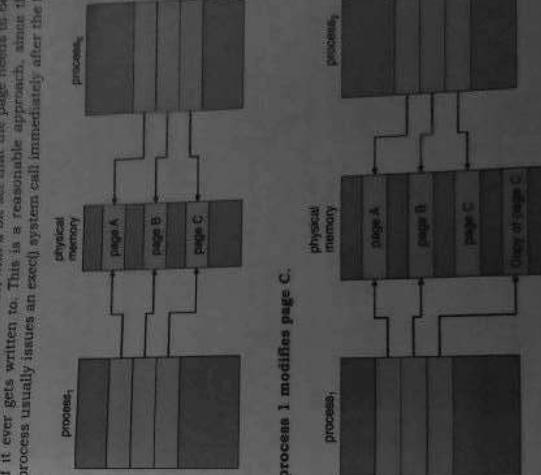
- In an extreme case, NO pages are swapped in for a process until they are requested by page faults. This is known as **multiple demand paging**.
- This is very rare, due to **locality of reference**.
  - The hardware necessary to support virtual memory is the same as for paging and swapping : A page table and secondary memory.
  - A crucial part of the process is that the instruction must be restarted from scratch once the desired page has been made available in memory.
  - For most simple instructions this is not a major difficulty. However there are some architectures that allow a single instruction to modify a fairly large block of data, i.e. which span a page boundary. If, and if some of the data gets modified before the page fault occurs, this could cause problems. One solution is to access both ends of the block before executing the instruction, guaranteeing that the necessary pages get pagged in before the instruction begins.

**Copy-on-Write**

- The idea behind a **copy-on-write fork**, is that the pages for a parent process do not have to be copied for the child until one or the other of the processes changes the page. They can be simply shared between the two processes in the meantime, with a bit set that the page needs to be copied if it ever gets written to. This is a reasonable approach, since the child process usually issues an exec system call immediately after the fork.



- The idea behind a **copy-on-write fork**, is that the pages for a parent process do not have to be copied for the child until one or the other of the processes changes the page. They can be simply shared between the two processes in the meantime, with a bit set that the page needs to be copied if it ever gets written to. This is a reasonable approach, since the child process usually issues an exec system call immediately after the fork.





| reference bit | 0 | 1 | 2 | 3 | 0 | 4 | 2 | 3 | 0 | 3 | 2 | 1 | 2 | 0 | 1 | 7 | 0 |
|---------------|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| page frames   | 0 | 1 | 2 | 3 | 0 | 4 | 2 | 3 | 0 | 3 | 2 | 1 | 2 | 0 | 1 | 7 | 0 |
| 0             | 1 | 2 | 3 | 0 | 4 | 2 | 3 | 0 | 3 | 2 | 1 | 2 | 0 | 1 | 7 | 0 |   |
| 0             | 1 | 2 | 3 | 0 | 4 | 2 | 3 | 0 | 3 | 2 | 1 | 2 | 0 | 1 | 7 | 0 |   |
| 0             | 1 | 2 | 3 | 0 | 4 | 2 | 3 | 0 | 3 | 2 | 1 | 2 | 0 | 1 | 7 | 0 |   |
| 0             | 1 | 2 | 3 | 0 | 4 | 2 | 3 | 0 | 3 | 2 | 1 | 2 | 0 | 1 | 7 | 0 |   |
| 0             | 1 | 2 | 3 | 0 | 4 | 2 | 3 | 0 | 3 | 2 | 1 | 2 | 0 | 1 | 7 | 0 |   |
| 0             | 1 | 2 | 3 | 0 | 4 | 2 | 3 | 0 | 3 | 2 | 1 | 2 | 0 | 1 | 7 | 0 |   |

- LRU is considered a good replacement policy, and is often used. The problem is how exactly to implement it. There are two simple approaches commonly used:

1. **Counters:** Every memory access increments a counter, and the current value of this counter is stored in the page-table entry for that page. Then finding removing objects from the middle of the stack, a doubly linked list is the recommended data structure.

Note that both implementations of LRU require hardware support, either for incrementing the counter or for managing the stack, as these operations must be performed for **every** memory access.

- Neither LRU or OPT exhibit Belady's anomaly. Both belong to a class of page-replacement algorithms called **stack algorithms**, which can never exhibit Belady's anomaly. A stack algorithm is one in which the pages kept in memory for a frame set of size  $N$  will always be a subset of the pages kept for a frame size of  $N + 1$ . In the case of LRU (and particularly the stack implementation thereof), the top  $N$  pages of the stack will be the same for all frame set sizes of  $N$  or anything larger.

#### LRU Approximation Page Replacement

- Unfortunately full implementation of LRU requires hardware support, and few systems provide the full hardware support necessary.
- However, many systems offer some degree of HW support, enough to approximate LRU fairly well (in the absence of ANY hardware support, FIFO might be the best available choice.)
- In particular, many systems provide a **reference bit** for every entry in a page-table, which is set anytime that page is accessed. Initially all bits are set to zero, and they can also all be cleared at any time. One bit of precision is enough to distinguish pages that have been accessed since the last clear from those that have not, but does not provide any finer grain of detail.

#### Additional Reference-Bit Algorithm

- Finer grain is possible by storing the most recent 8 reference bits for each page in an 8-bit byte in the page table entry, which is interpreted as an unsigned int.
- At periodic intervals (clock interrupts), the OS takes over, and right-shifts each of the reference bytes by one bit.

#### Enhanced Second-Chance Algorithm

- The enhanced second chance algorithm looks at the reference bit and the modify bit (dirty bit) as an ordered pair, and classifies pages into one of four classes:
  1. (0, 0) - Neither recently used, nor modified.
  2. (0, 1) - Not recently used, but modified.
  3. (1, 0) - Recently used, but clean.
  4. (1, 1) - Recently used and modified.
- This algorithm searches the page table in a circular fashion (in as many as four passes), looking for the first page it can find in the lowest numbers category, i.e., it first makes a pass looking for a (0, 0), and then if it can't find one, it makes another pass looking for a (0, 1), etc.
- The main difference between this algorithm and the previous one is the preference for replacing clean pages if possible.

#### Counting-Based Page Replacement

- There are several algorithms based on counting the number of references that have been made to a given page, such as:
  - **Least Frequently Used, LFU:** Replace the page with the lowest reference count. A problem can occur if a page is used frequently initially and then not used any more, as the reference count remains high. A solution is to not use any more, as the reference count periodically, yielding a time-decaying average reference count.
  - **Most Frequently Used, MFU:** Replace the page with the highest reference count. The logic behind this idea is that pages that have already been referenced a lot have been in the system a long time, and we are probably done with them, whereas pages referenced only a few times have only recently been loaded, and we still need them.
- In general, counting-based algorithms are not commonly used, as their implementation is expensive and they do not approximate OPT well.

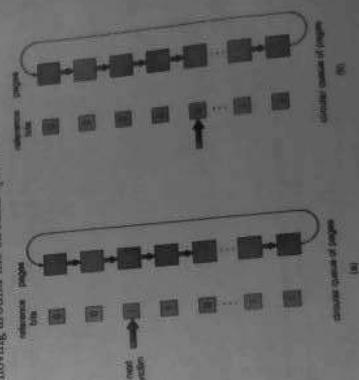
#### Page Buffering Algorithms

- There are a number of page-buffering algorithms that can be used in conjunction with the afore-mentioned algorithms, to improve overall performance and sometimes make up for inherent weaknesses in the hardware and/or the underlying page-replacement algorithms:
  - Maintain a certain minimum number of free frames at all times. When a page-fault occurs, go ahead and allocate one of the free frames from the free list first, to get the requesting process up and running again as quickly as possible, and then select a victim page to write to disk and free up a frame as a second step.

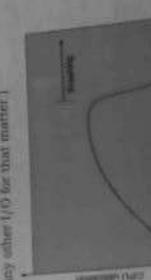
Keep a list of modified pages, and when the I/O system is otherwise idle, write it to disk, and then clear the modify bit. thereby increasing the chance of finding a "clean" page for the next potential victim.

- Keep a pool of free frames. Since the data in the page is not actually cleared out when it was made free, it can be made an active page again without having to copy the page to disk. This is useful when an algorithm loads in any new data from disk. (In fact, if a fault is needed again soon,

- The **second chance algorithm** is essentially a FIFO, except the reference bit is used to give pages a second chance at staying in the page-table.
- When a page must be replaced, the page-table is scanned in a FIFO (circular queue) manner.
- If a page is found with its reference bit not set, then that page is scanned as the next victim.
- If, however, the next page in the FIFO **does** have its reference bit set, then it is given a second chance.
- The reference bit is cleared, and the FIFO search continues.
- If some other page is found that did not have its reference bit set, then that page will be selected as the victim, and this page (the one being given the second chance) will be allowed to stay in the page-table.
- If, however, there are no other pages that do not have their reference bit set, then this page will be selected as the victim when the FIFO search circles back around to this page on the second pass.
- If all reference bits in the table are set, then second chance algorithm to replacement.
- This algorithm is also known as the **clock algorithm**, from the hints of the clock moving around the circular queue.



- The reference bit is cleared, and the FIFO search continues.
- Some applications (most notably database programs) understand their data accessing and caching needs better than the general-purpose OS, and should therefore be given reign to do their own memory management.
- Sometimes such programs are given a **ram disk partition** to work with, containing raw data blocks and no file system structure. It is then up to the application to use this disk partition as extended memory or for whatever other reasons it sees fit.
- Thrashing:**
- If a process cannot obtain its minimum required number of frames, then it must be swapped out, freeing up frames for other processes. This is an intermediate level of CPU scheduling.
- But what about a process that can keep its minimum, but cannot keep all of the frames that it currently owns on a regular basis? In this case it is forced to page out pages that it will need again in the very near future, leading to large numbers of page faults.
- A process that is spending more time paging than executing is said to be thrashing.
- Cause of Thrashing:**
- Early process scheduling schemes would control the level of multiprogramming, allowing based on CPU utilization, adding in more processes when CPU utilization was low.
- The problem is that when memory filled up and processes started spending lots of time waiting for their pages to page in, then CPU utilization would drop, causing the scheduler to add in even more processes and exacerbating the problem. Eventually the system would essentially grind to a halt.
- Local page replacement policies can prevent one thrashing process from taking pages away from other processes, but it will tend to clog up the I/O queue, thereby slowing down any other process that needs to do even a little bit of paging (in any other I/O for that matter!)



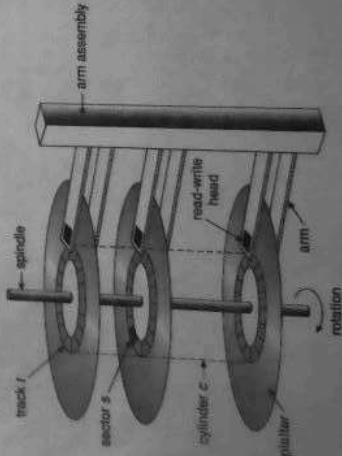
- To prevent thrashing we must provide protection with as many frames as they really need – right now – but how do we know what that is?
- The **locality model** states that processes typically access memory references in a **given locality**, making lots of references to the same general area of memory before moving periodically to a new locality as are illustrated in the current diagram. If we could just keep all memory frames as are provided in the current diagram, then page faults would occur, probably on switches from one locality to another (I.E. switch page between pages and another is called.)

## Mass Storage Structure

### Overview of Mass-Storage Structure

#### Magnetic Disks

- Traditional magnetic disks have the following basic structure:
  - One or more platters in the form of disks covered with magnetic media.
  - Hard disk platters are made of rigid metal, while floppy disks are made of more flexible plastic.
  - Each platter has two working surfaces. Older hard disk drives would sometimes not use the very top or bottom surface of a stack of platters, as these surfaces were more susceptible to potential damage.
  - Each working surface is divided into several concentric rings called tracks. The collection of all tracks that are the same distance from the edge of the platter, i.e., all tracks immediately above one another in the following diagram, is called a cylinder.
  - Each track is further divided into sectors, traditionally containing 512 bytes of data each, although some modern disks occasionally use larger sector sizes. (Sectors also include a header and a trailer, including checksum information among other things. Larger sector sizes reduce the fraction of the disk consumed by headers and trailers, but increase internal fragmentation and the amount of disk that must be marked bad in the case of errors.)
  - The data on a hard drive is read by read-write heads. The standard configuration among other things: Larger sector sizes reduce the seek time of the disk simultaneously from one cylinder to another. (Other configurations, according independent read-write heads, may speed up disk access, but involve serious technical difficulties.)
  - The storage capacity of a traditional disk drive is equal to the number of heads (i.e., the number of working surfaces) times the number of tracks per surface, times the number of sectors per track, times the number of bytes per sector. A physical block of data is specified by providing the head-sector-cylinder number at which it is located.

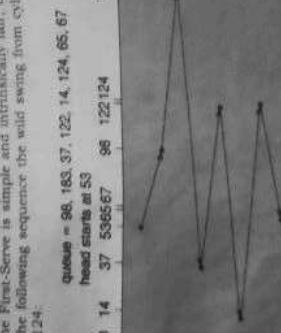


## Disk Scheduling

- Disk transfer speeds** are limited primarily by **seek times** and **rotational latency**. When multiple requests are to be processed there is also **starvation**, delay in serving for either requests to be processed.
- Bandwidth** is measured by the amount of data transferred divided by the total amount of time from the first request being made to the last transfer being completed. (For a series of disk requests)
- Both bandwidth and access time can be improved by processing requests in a good order.
- Disk requests include the disk address, memory address, number of sectors to transfer, and whether the request is for reading or writing.

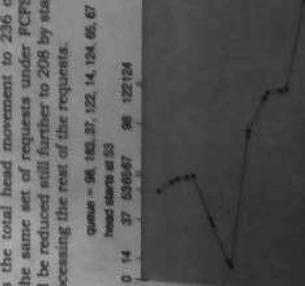
### FCFS Scheduling

First-Come First-Serve is simple and intuitively fair, but not very efficient. Consider the following sequence: the head swings from cylinder 122 to 14 and then back to 124:



### SSTF Scheduling

**Shortest Seek Time First** scheduling is more efficient, but may lead to starvation if a constant stream of requests arrives for the same general area.



## Memory Management

### Moving Head Disk Mechanism

- In operation, the disk rotates at high speed, such as 7200 rpm, 1/20 revolutions per second. (The rate at which data can be transferred from the disk to the computer.)
- The positioning time, aka, the **seek time** or **random access time** is the time required to move the heads from one cylinder to another, and the time to settle down after the move. This is typically the slowest step in the process and the predominant bottleneck to overall transfer rates.
- The **rotational latency** is the duration of time required for the desired sector to rotate around and come under the read/write head. This can range anywhere from zero to one full revolution, and on average will require one-half revolution. This is another physical step and is usually the second slowest step behind seek time. (For a disk rotating at 7200 rpm, the average rotational latency would be 1/2 revolution, 1/20 seconds, or just over 4 milliseconds, a long time by computer standards.)

- The **transfer rate**, which is the time required to move one byte electronically from the disk to the computer. (Some authors may define the term transfer rate to refer to overall transfer rate, including seek time and rotational latency as well as the electronic data transfer rate.)
- Floppy disks are normally removable. Hard drives can also be removable, and some are even **hot swappable**, meaning they can be removed while the computer is running, and a new hard drive inserted at their place.
- Disk drives are connected to the computer via a cable known as the I/O Bus. Some of the common interface formats include Enhanced Integrated Drive Electronics, EIDE, Advanced Technology Attachment, ATA, Serial ATA, SATA, Universal Serial Bus, USB, Fiber Channel, FC, and Small Computer Systems Interface, SCSI.

- The host controller is at the computer end of the I/O bus, and the disk controller is built into the disk itself. The CPU issues commands to the host controller via I/O ports. Data is transferred between the magnetic surface and onboard cache by the disk controller, and then the data is transferred and on to the host controller and the motherboard memory at electronic speeds.

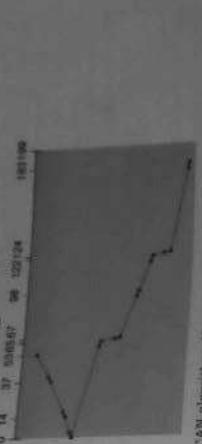
### Magnetic Tapes

- Magnetic tapes were once used for common secondary storage before tape drives of hard disk drives, but today are used primarily for backup tape building.
- Accessing a particular spot on a magnetic tape can be slow, but once reading or writing commences, access speeds are comparable to disk drives.
- Capacities of tape drives can range from 20 to 200 GB, and compression varies from that, cache to the host controller and the motherboard memory at electronic speeds.

## Memory Management

### SCAN Scheduling

- The SCAN algorithm, i.e., the shorter distances between birth and death of the disk to the origin, minimize moves back and forth from tail building.
- Accessing a particular spot on a magnetic tape can be slow, but once reading or writing commences, access speeds are comparable to disk drives.
- Capacities of tape drives can range from 20 to 200 GB, and compression varies from that, cache to the host controller and the motherboard memory at electronic speeds.

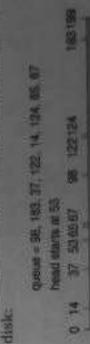


- Under the SCAN algorithm, if a request arrives just ahead of the moving head, then it will be processed right away, but if it arrives just after the head has passed, then it will have to wait for the head to pass going the other way on the return trip. This leads to a wide variation in access times which can be improved upon.

- Consider, for example, when the head reaches the high end of the disk requests with high cylinder numbers just missed the passing head, which means they are all recent requests, whereas requests with low numbers have been waiting for a much longer time. Making this return scan from high to low then ends up accessing recent requests first and making older requests wait much longer.

### C-SCAN Scheduling

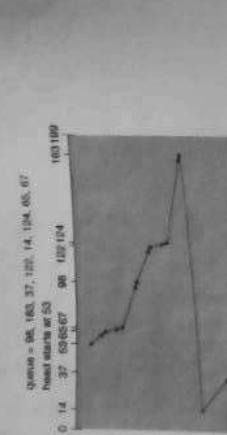
- The Circular SCAN algorithm improves upon SCAN by treating all requests in a circular queue fashion - Once the head reaches the end of the disk, it returns to the other end without processing any requests, and then starts again from the beginning of the disk.



### LOOK Scheduling

- LOOK** scheduling improves upon SCAN by having about half the queue of pending requests, and not moving the heads with requests towards the end of the queue.

### LOOK Scheduling



#### Disk Management

- Before a disk can be used, it must be **low-level formatted**, which means laying down all the headers and trailers marking the beginning and end of each sector, included in the header and trailer are the linear sector numbers, and **error-correcting codes**, **ECC**, which allow damaged sectors to not only be detected, but, in many cases for the damaged data to be recovered (depending on the extent of the damage) Sector sizes are traditionally 512 bytes, but may be larger, particularly in larger drives.
- ECC calculation is performed with every disk read or write, and if damage is detected but the data is recoverable, then a **soft error** has occurred. Soft errors are generally handled by the on-board disk controller, and never seen by the OS.
- Once the disk is low-level formatted, the next step is to partition the drive into one or more separate partitions. This step must be completed even if the disk is to be used as a single large partition, so that the partition table can be written to the beginning of the disk.
- After partitioning, then the filesystems must be **logically formatted**, which involves laying down the master directory information [FAT table], initializing free lists, and creating at least the root directory of the filesystem. Disk partitions which are to be used as raw devices are not logically formatted. This saves the overhead and disk space of the filesystem structure but requires that the application program manage its own disk storage requirements.]

#### Boot Block

- Computer ROM contains a **bootstrap** program (OS independent) with just enough code to find the first sector on the first hard drive on the first controller, load that sector into memory, and transfer control over to it. (The ROM bootstrap program may look in floppy and/or CD drives before accessing the hard drive, and is smart enough to recognize whether it has found valid boot code or not.)
- The first sector on the hard drive is known as the **Master Boot Record**, **MBR**, and contains a very small amount of code in addition to the **partition table**. The partition table documents how the disk is partitioned into logical disks and indicates specifically which partition is the **active** or **boot** partition.

#### Swap Space

- Some systems support multiple swap spaces on separate disks in order to speed up the virtual memory system.
- Swap-Space Location**
- The first sector on the hard drive can be physically located in one of two locations:
  - As a large file which is part of the regular filesystem. This is easier to implement, but inefficient. Not only must the swap space be accessed through the directory system, the file is also subject to fragmentation issues. Catching the block location helps in finding the physical blocks, but that is not complete fix.
  - As a raw partition, possibly on a separate or little-used disk. This allows the OS more control over swap space management, which is usually faster and more efficient. Fragmentation of swap space is generally not a big issue, as the space is re-initialized every time the system is rebooted. The downside of keeping swap space on a raw partition is that it can only be grown by repartitioning the hard drive.

- In the mapping system shown below for Linux systems, a map of swap space is kept in memory, where each entry corresponds to a 4K block in the swap space. Zeros indicate free slots and non-zeros refer to how many processes have a mapping to that particular block ( $> 1$  for shared pages only.)
- |          |        |      |
|----------|--------|------|
| swap map | page → | disk |
| 0        | ↓      | ↓    |
| 1        | ↓      | ↓    |
| 0        | ↓      | ↓    |
| 3        | ↓      | ↓    |
| 0        | ↓      | ↓    |
| 1        | ↓      | ↓    |
- Historically OSes swapped out entire processes as needed. Modern systems swap out only individual pages, and only as needed. For example process code blocks and other blocks that have not been changed since they were originally loaded are normally just freed from the virtual memory system rather than copying them to swap space, because it is faster to go find them again in the filesystem and read them back from there than to write them out to swap space and then read them back.)

#### Swap Space Management : An Example

- Historically OSes swapped out entire processes as needed. Modern systems swap out only individual pages, and only as needed. For example process code blocks and other blocks that have not been changed since they were originally loaded are normally just freed from the virtual memory system rather than copying them to swap space, because it is faster to go find them again in the filesystem and read them back from there than to write them out to swap space and then read them back.)
- In the mapping system shown below for Linux systems, a map of swap space is kept in memory, where each entry corresponds to a 4K block in the swap space. Zeros indicate free slots and non-zeros refer to how many processes have a mapping to that particular block ( $> 1$  for shared pages only.)

#### RAID Structure

- The general idea behind RAID is to employ a group of hard drives together with some form of duplication, either to increase reliability or to speed up operations, (or sometimes both.)
- RAID** originally stood for **Redundant Array of Inexpensive Disks**, and was designed to use a bunch of cheap small disks in place of one or two larger, more expensive ones. Today RAID systems employ large possibly expensive disks as their components, switching the definition to **Independent Disks**.
- The more disks a system has, the greater the likelihood that one of them will go bad at any given time. Hence increasing disks on a system actually **decreases the Mean Time to Failure**, **MTTF** of the system.
- If, however, the same data was copied onto multiple disks, then the data would not be lost unless **both** copies of the data were damaged.

- In a **duo boot** (or longer multi-boot), systems, this user may be given a choice of which operating system to boot, with a default option to be chosen in the event of no response within some time frame.
- Once the kernel is found by the boot programs, it is loaded into memory and the boot process is transferred over to the OS. The kernel will normally continue launching important system services and finally associate one or more logins, **maintenance** or **safe mode**, in which very few system services are started. These modes are designed for system administrators to repair problems or otherwise maintain the system.

#### Bad Blocks

- wear out over time. For these reasons all disks are shipped with a few bad blocks, and additional blocks are added to the disk after a certain time. If many blocks go bad then the entire disk will need to be replaced, but a few here and there can be handled through other means:
  - In the old days, bad blocks had to be checked for manually. Formattting and then erasing certain disk analysis tools would identify bad blocks and attempt to read the data off them on the last time the newly requested track. Then the bad blocks would be mapped out and taken out of future requests. Sometimes the data could be recovered, and sometimes it was lost forever.
  - Modern disk controllers make much better use of the error-correcting codes, so that bad blocks can be detected earlier, and the data actually recovered.
  - Note that re-mapping of sectors from their normal linear progression can throw off the disk scheduling optimizations of the OS, especially if the replacement sectors are physically far away from the sector it is replacing. For this reason most disks normally keep a few spare sectors on each cylinder, as well as at least one spare cylinder. Whenever possible a bad sector will be mapped to another sector on the same cylinder, or at least a cylinder as close as possible. **Sector skipping** may also be performed, in which all sectors between the bad sector and the replacement sector are moved down by one, so that the linear progression of sector numbers can be maintained.
  - If the data on a bad block cannot be recovered, then a **hard error** has occurred, which requires replacing the (file) from backups, or rebuilding them from scratch.

#### Swap-Space Management

- Modern systems typically swap out pages as needed, rather than swapping out entire processes. Hence the **swapping** acronym is part of the virtual memory management system.
  - Managing swap space is obviously an important task for modern OSes.
  - Swap-Space Use**
    - The amount of swap space needed by an OS will vary greatly according to how it is used. Swap systems require an amount equal to total RAM, minus virtual memory needed by RAM, and swap systems use little or none virtual memory (S.V.R.S.) (See III)

#### Memory Management

- simultaneously, which is a **MUCH** slower primitive (less for a single disk going bad. More specifically, the second disk would have to go bad before the first disk was restarted, which brings the Mean Time To Repair into play. For example if two disks were involved, such with a MTTF of 100,000 hours and a MTTR of 10 hours, then the **Mean Time to Data Loss** would be 3000 - 10/6 hours, or 57,000 years).
- This is the basic idea behind disk **mirroring**, in which a system contains identical data on two or more disks.
- Note that a power failure during a write operation could cause both disks to corrupt data, if both disks were writing simultaneously at the same time, so that they will both become corrupted at that time. In the same way, if two hosts become corrupted at the same time in the RAM as a write cache, and alternate solution involves having multiple hosts which is not lost in the event of a power failure and which is protected by error-correcting codes.

#### RAID Levels

- Mirroring provides reliability, but is expensive. Striped, unstriped performance, but does not improve reliability. Accordingly, there are a number of different schemes that combine the principles of striping and striping in different ways, in order to balance reliability versus performance versus cost. These are described by different RAID levels, as opposed to full mirroring. The numbers following, 'C' indicates a copy and 'P' indicates parity, i.e., checksum bits 1
- Raid Level 0 : This level includes striping only, with no mirroring.
- Raid Level 1 : This level stores mirrored data, no striping.
- Raid Level 2 : This level stores error-correcting codes on additional disks, allowing for any damaged disk to be reconstructed from the lost data. As a result a single parity bit is all that is needed to recover the lost data from an array of disks. Level 3 also includes striping, which requires only three extra disks to protect 4 disks worth of data, as opposed to full mirroring. The number of disks required is a function of the error-correcting algorithm, and the facets by which the particular Raid (bits) share identified.)

- Raid Level 3 : This level is like level 2, except that it adds adding of the fact that each disk is still doing its own error detection, so that when an error occurs, there is no question about which disk in the array has the bad disk. As a result a single parity bit is all that is needed to recover the lost data from an array of disks. Level 4 is like level 3, employing block-level striping instead of bit-level striping. The benefits are that multiple blocks can be read independently, and changes to a block only require writing one block (data and parity) rather than involving all disks. Note that these disks can be added seamlessly to the system provided they are initialized to all zeros so that does not affect the parity results.
- Raid Level 5 : This level is like level 4, except the parity blocks are distributed over all disks, thereby more evenly balancing the load on the system. For every given block on the disk(s), one of the disks will hold the parity information for that block and the other N-1 disks will hold the data. Since the same data is stored on the disks, the data will be lost if all the disks with the same data that block and the other N-1 disks will hold the data. Since the same data is stored on the disks, the data will be lost if all the disks with the same data that block and the other N-1 disks will hold the data.

cannot hold both data and parity for the same block, as both would be lost in the event of a disk crash.

Raid Level 6 - This level extends raid level 5 by storing multiple bits of error recovery codes, i.e. such as the Reed-Solomon codes [1] for each bit position of data, rather than a single parity bit. In the example shown below 2 bits of ECC are stored for every 4 bits of data, allowing data recovery in the face of up to two simultaneous disk failures. Note that this still involves only 50% increase in storage needs, as opposed to 100% for simple mirroring which could only tolerate a single disk failure.



(e) RAID 2: mirrored stripe with connecting codes.



(f) RAID 3: bit-interleaved parity.



(g) RAID 4: block interleaved parity.



(h) RAID 5: block interleaved distributed parity.

**FILE SYSTEM INTERFACE****File**

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

**File Attributes**

- Different filers keep track of different file attributes, including:
  - o Name - Some systems give special significance to names, and particularly extensions like .txt, .rtf, etc., and some do not. Some experiments may be of significance to the OS (e.g. .l), and others only to certain applications (e.g. .1).
  - o Identifier
  - o Type - Text, executable, other binary, etc.
  - o Location - on the hard drive
  - o Size
  - o Protection
  - o Time & Date
  - o User ID

**Stable-Storage Implementation**

- The concept of stable storage involves a storage medium in which data is never lost, even in the face of equipment failure in the middle of a write operation.
- To implement this requires two (or more) copies of the data, with separate failure modes.

An attempted disk write results in one of three possible outcomes:

1. The data is successfully and completely written.
2. The data is partially written, but not completely. The last block written may be garbled.
3. No writing takes place at all.

Whenever an equipment failure occurs during a write, the system must detect it, and return the system back to a consistent state. To do this requires two physical blocks for every logical block, and the following procedure:

**PPW**

**File pointer** - records the current position in the file, for the next read or write access.

**File-open count** - How many times has the current file been opened simultaneously by different processes? and not yet closed? When this counter reaches zero the file can be removed from the table.

Some systems provide support for file locking.

A shared lock is for reading only.

A exclusive lock is for writing as well as reading.  
An advisory lock is informational only, and not enforced. (A 'Keep Out' sign, which may be ignored.)

A mandatory lock is enforced. (A truly locked door.)

UNIX uses advisory locks, and Windows uses mandatory locks.

**FILE Types**

- Windows (and some other systems) use special file extensions to indicate the type of each file:

| file type      | usual extension                 | function   |
|----------------|---------------------------------|--|
| executable     | .exe, .COM, .BIN<br>or none     | read-to-run machine-language program                                     |
| object         | .obj, .C, .java, .pert,<br>.asm | compiled, machine language, not linked                                   |
| source code    | C, C++, Java, pert,<br>.asm     | source code in various languages   |
| batch          | .bat, .sh                       | commands to the command interpreter                                      |
| startup        | .ini, .hml, .tes                | textual data, documents  |
| word processor | .rtf, .rtf,<br>.docx            | various word-processor formats   |
| library        | .lib, .a, .so, .db              | libraries of routines for programmers                                    |
| print or view  | .gif, .pdf, .jpg                | ASCII or binary file in a format for printing or viewing                 |
| archive        | .tar, .zip, .tar.gz             | related files grouped into one file, sometimes compressed, for archiving |
| multimedia     | .mp3, .m4v, .mov, .avi          | binary file containing audio or AVI information                          |

**FILE Structure**

- Some filers contain an internal structure, which may or may not be known to the OS
- For the OS to support file formats, increases the size and complexity of the OS

**File Operations**

The file ADT supports many common operations:

- Creating a file
- Writing a file
- Reading a file
- Repositioning within a file
- Deleting a file

**Truncating a file**

Most filers require that files be opened before access, and closed after all access is complete. Normally the programmer opens the file automatically at file access. (Windows has some rare situations open the file, automatically at file access, when opening a document without currently open files is selected in the file menu, initializing the windows procedure)

**Memory Management**

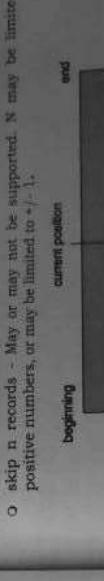
UNITX treats all files as sequences of bytes, with no further consideration of internal structure. (With the exception of executable binary programs, which it must know how to load and find the first executable binary program). Machine filers have two *forks* - a *read fork* - a *write fork*, and a *data fork*. The *read fork* contains information relating to the I/O, such as length and contains the code or data as appropriate.

Internal File Structure

- Disk files are accessed in units of physical blocks, typically 512 bytes or some power-of-two multiple thereof. (larger physical disks use larger block integers)
- Internally files are organized in units of logical units, which may be as small as a single byte, or may be a larger size corresponding to some data record or space.) that occurs.
- As a rule, half a physical block is wasted for each file, and the larger the block sizes the more space is lost to internal fragmentation.

**Access Methods****Sequential Access**

- A sequential access file emulates magnetic tape operation, and generally supports a few operations:
  - o read next - read a record and advance the tape to the next position.
  - o write next - write a record and advance the tape to the next position.
  - o rewind
  - o skip n records - May or may not be supported. N may be limited to +/- 1.

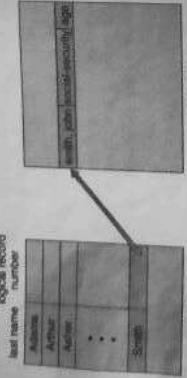
**Direct Access**

- Jump to any record and read that record. Operations supported include:
  - o read n - read record number n. (Note an argument is now required.)
  - o write n - write record number n. (Note an argument is now required.)
  - o jump to record n - could be 0 or the end of file.
  - o jump to record n - could be 0 or the end of file.
  - o Query current record - used to return back to this record later
  - o Sequential access can be easily emulated using direct access. The inverse is complicated and inefficient.

| Implementation of Direct Access |  |
|---------------------------------|--|
| read                            | $CP = 0$<br>read $CP$ ,<br>$SP = CP + 1$ |
| read more                       | $CP = SP$                                |
| write, write                    | $CP = CP + 1$                            |

#### Other Access Methods

- An indexed access scheme can be easily built on top of a direct access system. Very large files may require a multi-tiered indexing scheme, i.e. indexes of indexes.



#### DIRECTORY STRUCTURE

##### Storage Structure

- A disk can be used in its entirety for a file system.
- Alternatively, a physical disk can be broken up into multiple partitions, slices, or minidisks, each of which becomes a virtual disk and can have its own filesystem.
- Or, multiple physical disks can be combined into one volume, i.e. a larger virtual disk, with its own filesystem spanning the physical disks.



#### Single Level Directory

The simplest method is to have one big list of all the files on the disk. The entire system will contain only one directory which is supposed to mention all the files present in the file system. The directory contains one entry per each file present on the file system.

#### Tree-Structured Directories

An obvious extension to the two-tiered directory structure, and the one with which we are all most familiar. Each user / process has the concept of a current directory from which all (relative) I-searches take place. Files may be accessed using either absolute pathnames (relative to the root of the tree) or relative pathnames (relative to the current directory.) Directories are stored the same as any other file in the system, except there is a bit that identifies them as directories, and they have some special structure that the OS understands.

One question for consideration is whether to allow the removal of directories that are not empty - Windows requires that directories be emptied first, and UNIX provides an option for deleting entire sub-trees.

#### Operating System (S.Y.B.Sc. C.S./Sem. 4)

##### Memory Management



#### Acyclic-Graph Directories

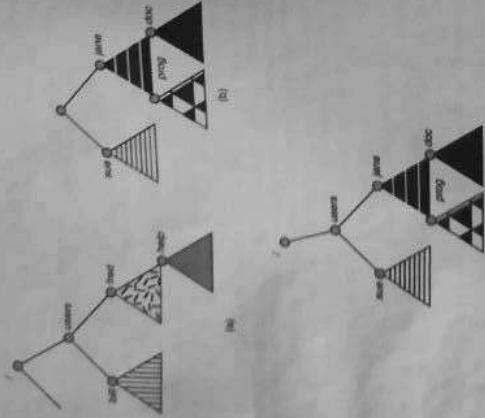
- When the same file is accessed in more than one place in the directory structure (e.g. because they are being shared by more than one user / process), it can be useful to provide an acyclic-graph structure. (Note the directed arcs from parent to child.)
- UNIX provides two types of links for implementing the acyclic-graph structure. (See 'man ln' for more details.)
- A hard link (usually just called a link) involves multiple directory entries that both refer to the same file. Hard links are only valid for ordinary files in the same filesystem.

##### The Structured Directory System

##### File-System Mounting

- Hard links require a reference count, containing information about where to find the linked file. Symbolic links may be used to link directories and/or files in other filesystems, as well as ordinary files in the current filesystem.
- For symbolic links there is some question as to what to do with the symbolic links when the original file is moved or deleted.
- One option is to find all the symbolic links and adjust them also.
- Another is to leave the symbolic links dangling and discover that they are no longer valid the next time they are used.
- What if the original file is removed, and replaced with another file having the same name before the symbolic link is next used?

Filesystems may be mounted read-only, or have other restrictions imposed.



The traditional Windows OS runs an extended two-tier directory structure, where the first tier of the structure separates volumes by drive letters, and a tree structure is implemented below that level.

- Macintosh runs a similar system, where each new volume that is found is automatically mounted and added to the desktop when it is found.
- More recent Windows systems allow filesystems to be mounted to any directory in the filesystem, much like UNIX.

#### File Sharing

- On a multi-user system, more information needs to be stored for each file:

- The owner (user) who owns the file, and who can control its access.
- The group of other user IDs that may have some special access to the file.
- What access rights are afforded to the owner (User), the Group, and to the rest of the world (the universe, a.k.a. Others.)
- Some systems have more complicated access control, allowing or denying specific accesses to specifically named users or groups.

The following low-level operations are often controlled:

- Read - View the contents of the file.
- Write - Change the contents of the file.
- Execute - Load the file onto the CPU and follow the instructions contained therein.
- Append - Add to the end of an existing file.
- Delete - Remove a file from the system.
- List - View the name and other attributes of files on the system.

#### Protection

- Files must be kept safe for reliability against accidental damage, loss protection, against deliberate malicious access. (The former is usually managed with backup copies. This section discusses the latter.)

- One simple protection scheme is to remove all access to a file. However this makes the file unusable, so some sort of controlled access must be arranged.

#### Types of Access

- The following low-level operations are often controlled:
- Read - View the contents of the file.
- Write - Change the contents of the file.
- Execute - Load the file onto the CPU and follow the instructions contained therein.
- Append - Add to the end of an existing file.
- Delete - Remove a file from the system.
- List - View the name and other attributes of files on the system.

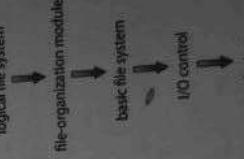
#### Higher-level operations, such as copy, can generally be performed through combinations of the above.

#### Access Control

- One approach is to have complicated **Access Control Lists, ACLs**, which specify exactly what access is allowed or denied for specific users or groups.
- The AFS uses this system for distributed access.
- Control is very finely adjustable, but may be complicated, particularly when the specific users involved are unknown. (AFS allows some wild cards, so for example all users on a certain remote system may be trusted, or a given username may be trusted when accessing from any remote system.)

- UNIX uses a set of 9 access control bits, in three groups of three. These correspond to R, W, and X permissions for each of the Owner, Group, and Directories. The RWX bits control the following privileges for ordinary files and directories:

- Hard disks have two important properties that make them suitable for secondary storage of files in file systems: (1) Blocks of data can be rewritten in place, and (2) they are direct access, allowing any block of data to be accessed with only (relatively) minor movements of the disk heads and rotational latency.
- Disks are usually accessed in physical blocks, rather than a byte at a time.
- Block sizes may range from 512 bytes to 4K or larger.
- File systems organize storage on disk drives, and can be viewed as a layered design:



- The advent of the Internet introduces issues for accessing files stored on remote computers.
- The original method used was FTP, allowing individual files to be transferred across systems as needed. FTP can be either account and password mounted onto a local directory or **anonymous**, not requiring any user name or password access.
- Various forms of **distributed file systems** allow multiple systems to be connected across a network, using normal file structures, and increased sharing of resources across the network.
- The WWW has made it very easy again to access files on remote systems without maintaining their filesystems, generally using (anonymous) FTP as the underlying file transport mechanism.

#### The Client-Server Model

- When one computer system remotely mounts a filesystem that is physically located on another system, the system which physically stores the files acts as a **server**, and the system which mounts them is the **client**.
- User IDs and group IDs must be consistent across both systems for the system to work properly.
- The same computer can be both a client and a server. (E.g. cross-linked file systems.)

- There are several security concerns involved in this model:
- Spoofing (a computer pretending to be a different computer) as a potential security risk.
- Servers may restrict remote access to read-only.
- Servers restrict which filesystems may be remotely mounted. Generally, the information within those subsystems is limited, relatively public, and protected by frequent backups.

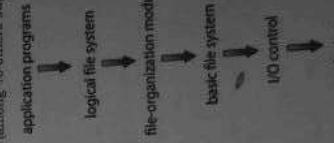
#### Distributed Information Systems

- The **Domain Name System, DNS**, provides for a unique naming system across all the internet.
- Domain names are maintained by the **Network Information System, NIS**. NIS has several security issues. NIS+ is more secure version but has not yet gained the same widespread acceptance as NIS.
- Microsoft's **Common Internet File System, CIFS**, establishes a **network login** for each user on a networked system, with shared file access. Older Windows systems used **domains**, and newer systems (XP, 2000), use active **directories**. Usernames must match across the network for this system to be valid.
- A newer approach is the **Lightweight Directory Access Protocol, LDAP**, which provides a **secure single sign-on** for all users to access all resources on a network. This is a secure system which is gaining in popularity, and which has the maintenance advantage of combining authentication information in one central location.

- At the lowest layer are the physical devices, consisting of the magnetic media, motion, media controls, and the electronics connected to them and controlling them. Modern disk put more and more of the electronics directly on the disk drive itself, leaving relatively little work for the disk controller card to perform.
- **I/O Control** consists of **device drivers**, special software programs (written in assembly) which communicate with the devices by reading and writing special codes directly to and from memory addresses corresponding to the controller cards + registers. Each controller card (device) on a system has a different set of addresses (registers, aka **ports**) that it listens to, and a unique set of command codes and results codes that it sends out.

- The **logical file system** level works directly with the device drivers in terms of retrieving and arranging raw blocks of data, without any consideration for what is in each block. Depending on the system, blocks may be referenced with a single block number, (e.g. block # 2345641, or with head-sector-cylinder combinations).
- The **file organization module** knows about files and their logical blocks, and how they map to physical blocks on the disk. In addition to translating from logical to physical blocks, the file organization module also maintains the list of free blocks, and allocates free blocks to files as needed.
- The **basic file system** level manages the directory structure and the mapping of data itself. This level manages the directory structure and the mapping of data as well as block number information for finding the data on the disk.
- The layered approach to file systems means that much of the code can be used uniformly for a wide variety of different file systems, and only certain layers need to be filesystem specific. Common file systems in use include the UNIX file system, NFS, the Berkeley Box File System, FFS, Windows FAT, FAT32, NTFS, CD-ROM systems ISO 9660, and for Linux the extended file systems ext2 and ext3 (among 40 others supported.)

#### Application Programs



- When a local disk file is unavailable, the result is generally known, immediate, and is generally non-recoverable. The only reasonable response is for the response to fail.
- However, when a remote file is unavailable, there are many possible reasons, and whether it is unrecoverable is not readily apparent. Hence most robust access systems allow for blocking or delayed response, in the hopes that the remote system (or the network) will come back up eventually.
- Files must be kept safe for reliability against accidental damage, loss protection, against deliberate malicious access. (The former is usually managed with backup copies. This section discusses the latter.)
- One simple protection scheme is to remove all access to a file. However this makes the file unusable, so some sort of controlled access must be arranged.
- The following low-level operations are often controlled:
- Read - View the contents of the file.
- Write - Change the contents of the file.
- Execute - Load the file onto the CPU and follow the instructions contained therein.
- Append - Add to the end of an existing file.
- Delete - Remove a file from the system.
- List - View the name and other attributes of files on the system.

#### Higher-level operations, such as copy, can generally be performed through combinations of the above.

#### Access Control

- One approach is to have complicated **Access Control Lists, ACLs**, which specify exactly what access is allowed or denied for specific users or groups.
- The AFS uses this system for distributed access.
- Control is very finely adjustable, but may be complicated, particularly when the specific users involved are unknown. (AFS allows some wild cards, so for example all users on a certain remote system may be trusted, or a given username may be trusted when accessing from any remote system.)
- UNIX uses a set of 9 access control bits, in three groups of three. These correspond to R, W, and X permissions for each of the Owner, Group, and Directories. The RWX bits control the following privileges for ordinary files and directories:

**File-System Implementation**

- File systems store several important data structures on the disk:
  - A **base-control-block**, (per volume) i.e., the **boot block** in UNIX or the **partition boot sector** in Windows, which contains information about how the system off this disk. This will generally be the first sector of the volume if there is a bootable system loaded on that volume, or the block will be left vacant otherwise.
  - A **volume control block**, (per volume) i.e., the **master file table** in UNIX or the **super-block** in Windows, which contains information such as the partition table, number of blocks on each filesystem, and pointers to free blocks and free FCB blocks.
  - The **File Control Block**, **FCB**, (per file) containing details about ownership, size, permissions, dates, etc. UNIX stores this information in inodes, and NTFS in the master file table as a relational database structure.

**A typical file-control block**

- There are also several key data structures stored in memory:
  - An in-memory mount table.
  - An in-memory directory cache of recently accessed directory information.
  - A system-wide open file table, containing a copy of the FCB for every currently open file in the system, as well as some other related information.
  - A per-process open file table, containing a pointer to the system open file table, as well as some other information. (For example the current file position pointer may be either here or in the system file table, depending on the implementation and whether the file is being shared or not.)
  - When a new file is created, a new FCB is allocated and filled out with important information regarding the new file. The appropriate directory is modified with the new file name and PCB information.
  - When a file is accessed during a program, the open() system call reads in the FCB information from disk, and stores it in the system-wide open file table. An entry is added to the per-process open file table referencing the system-wide table, and an index into the per-process table is returned by the open() system call. UNIX refers to this index as a **file descriptor**, and Windows refers to it as a **file handle**.

- If another process already has a file open, then a new process cannot write to the same file, unless it is explicitly given a copy of the existing entry in the per-process table or a pointer to the system-wide table is modified to point to the system-wide table entry in question. If this happens, memory access for this file is written out to disk if necessary.
- Physical disks are commonly divided into smaller units called **partitions**. They can also be combined into larger units, but this is done currently mostly for RAID installations.
- Partitions can either be used as raw devices (with no file system implemented with FCBs) and initial directory structures to build a filesystem like ext2/ext3 are generally used for swap spaces, and files, and can be used for various programs such as databases that choose to manage their own disk storage system. Partitions containing filesystems can generally only be accessed using the file system structure by ordinary users, but can often be accessed as a raw device (say by root).

- The boot block is accessed as part of a raw partition, by the boot program prior to any operating system being loaded. Modern boot programs understand multiple OSes and filesystem formats, and can give the user a choice of several available systems to boot.
- The root partition contains the OS kernel and at least the key portions of the OS needed to complete the boot process. At boot time the root partition is mounted, and control is transferred from the boot program to the kernel found there. Older systems required that the root partition be completely within the first 1024 cylinders of the disk, because that was as far as the boot program could reach. Once the kernel had control, then it could access partitions beyond the 1024 cylinder boundary.]

- Continuing with the boot process, additional filesystems get mounted, adding their information into the appropriate mount table structure. As a part of the mounting process, the file systems may be checked for errors, or inconsistencies, either because they have been cleaned, or just as a safety measure. The last time they were used, or just for generic purposes. Filesystems may be mounted either automatically or manually. In UNIX, a mount point is indicated by setting a flag in the memory copy of the structure, so all future references to that inode get re-directed to the root directory of the mounted filesystem.

**Virtual File Systems**

- Virtual File Systems**, VFS, provide a common interface to multiple different filesystem types. In addition, it provides for a unique identifier (inode) for files across the entire space, including across all filesystems of different types. (UNIX inodes are unique only across a single filesystem, and consequently do not carry across networked file systems.)

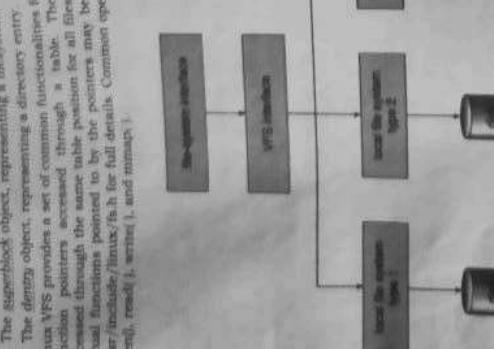
The VFS in Linux is based upon four key object types:

- The **inode** object, representing an individual file
- The **file object**, representing an open file

Versus System (S.Y.B.Sc. I-5) (Sem. II)

- The **staterlock object**, representing a filesystem.
- The **directory object**, representing a directory entry.
- Linux VFS provides a set of common functions for each filesystem, using function pointers accessed through a table. The same functionality is accessed through the same table position for all filesystem types, though the actual functions pointed to by the pointers may be filesystem specific. See [here](#) for full details. Common operations provided include [open\(\)](#), [read\(\)](#), [write\(\)](#), and [mmap\(\)](#).

Operating System (S.Y.B.Sc. C.S./IT) Sem-I

**Directory Implementation**

- Directories need to be fast to search, insert, and delete, with a minimum of wasted disk space.

**Linear List**

- A linear list is the simplest and easiest directory structure to set up, but it does have some drawbacks.
  - Finding a file (or searching one does not already exist upon creation) requires a linear search.
  - Deletions can be done by moving all entries, flinging an entry as deleted, or by moving the last entry into the newly vacated position.
  - Starting the list makes searches faster, at the expense of more complex insertions and deletions.
  - A linked list makes insertions and deletions into a sorted list easier, with overhead for the links.

**Hash Table**

- A hash table can also be used to speed up searches.
  - Hash tables are generally implemented in **addition to** linear or other structures.

**Allocation Methods**

- There are three major methods of storing files in disk, contiguous, linked, and indexed.

**Contiguous Allocation**

- Contiguous Allocation requires that all blocks of a file be kept together contiguously.

Performance is very fast, because reading successive blocks of the same file, the next adjacent cylinder, storage allocation involves the same sectors allocated earlier (i.e., fragmentation problem, etc.). The disadvantage is that file fragmentation required for moving the disk blocks from spot-to-spot may now justify the benefits of keeping files contiguously when possible.

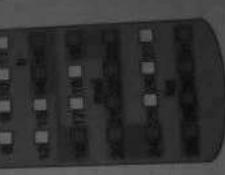
(Even file systems that do not try to obtain more disk contiguous can benefit from certain utilities that compact the disk and move all files contiguous to the process.)

Problems can arise when files grow, or if the exact size of a file is unknown at creation time:

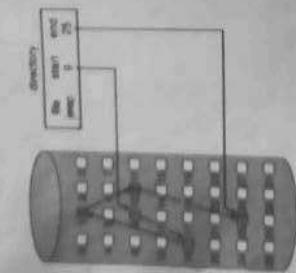
- Over-estimation of the file's final size increases external fragmentation and wastes disk space.
- Under-estimation may require that a file be moved or a process aborted if the file grows beyond its originally allocated space.
- If a file grows slowly over a long time period and then total free space must be allocated initially, then a lot of space becomes unusable before the file fills the space.

A variation is to allocate file space in large contiguous chunks, called **extents**. When a file outgrows its original extent, then no additional ones are allocated. For example an extent may be the size of a cylinder track or even cylinder aligned on an appropriate track or cylinder boundary. The high-performance file system Veritas uses extents to optimize performance.

| File   | start | length |
|--------|-------|--------|
| Count  | 0     | 2      |
| File 1 | 14    | 3      |
| File 2 | 10    | 6      |
| File 3 | 20    | 4      |
| File 4 | 8     | 2      |



- Disk files can be stored as linked lists, with the exception of the storage units contained by each block. If a block may be 500 bytes instead of 512, I linked allocation would result in memory blocks that are addressed by a 32-bit address, which is why some systems have moved to 64-bit file pointers.
- Another important aspect of disk management is keeping track of where allocation free space.
- **Bit Vector**
- One simple approach is to use a bit vector to which each bit represents a disk block, set to 1 if free or 0 if allocated.
- Fast algorithms exist for quickly finding contiguous blocks of a given size.
- The down side is that a 4GB disk requires over 3000 bits to store the bitmap. (For example...)



#### Indirect Allocation

- Indirect Allocation combines all of the inodes for accessing each file into a common block (for this file), as opposed to spreading them all over the disk or storing them in a FAT table.

Some disk space is wasted (relative to linked lists or FAT tables) because an entire index block must be allocated for each file, regardless of how many data blocks the file contains. This leads to questions of how big the index block should be, and how it should be implemented. There are several approaches:

**Linked Scheme** - An index block is one disk block, which can be read and written in a single disk operation. The first index block contains some header information, the first N block addresses, and it increments a pointer to additional index blocks.

**Multi-Level Index** - The first index block contains a set of pointers to secondary index blocks, which in turn contain pointers to the actual data blocks.

**Combined Scheme** - This is the scheme used in UNIX modes, in which the first 12 or so data blocks pointers are stored directly in the inode, and then singly, doubly, and triply indexed pointers provide access to more data blocks as needed.

The advantage of this scheme is that for small files, which may be 1, the data blocks are readily accessible (up to 4K with 4K block sizes), files up to about

#### NFS

- The resulting data structures could be VERY inefficient if not implemented carefully. For example, freeing up a 1 GB file on a 1 TB file system could involve updating thousands of blocks of tree list bit maps if the file was spread across the disk.
- NFS uses a combination of techniques, starting with dividing the disk up into hundreds of 1 megablock of a manageable size, each having their own space map.
- Free blocks are managed using the counting technique, but rather than write the information to a table, it is recorded in a log-structured transaction record. Adjacent free blocks are also coalesced into a larger single free block.
- An in-memory space map is constructed using a balanced tree data structure, constructed from the log data.
- The combination of the in-memory tree and the on-disk log provide for very fast and efficient management of these very large files and free blocks.

#### NFS

- NFS stands for Network File System, a file system developed by Sun Microsystems, Inc. It is a client server system that allows users to access files across a network and treat them as if they resided in a local file directory. For example, if you were using a computer linked to a second computer via NFS, you could access files on the second computer as if they resided in a directory on the first computer. This is accomplished through the processes of exporting (the process by which an NFS server provides remote clients with access to its files) and mounting (the process by which file systems are made available to the operating system and the user).
- The NFS protocol is designed to be independent of the computer, operating system, network architecture, and transport protocol. This means that systems using the NFS service may be manufactured by different vendors, use different operating systems, and be connected to networks with different architectures. These differences are transparent to the NFS application, and thus, the user.

#### The Mount Protocol

- The NFS mount protocol is similar to the local mount protocol, establishing a connection between a specific local directory (the mount point) and a specific device from a remote system.
- Each server maintains an export list of the local filesystems (directory subtrees) which are exportable, who they are exportable to, and what restrictions apply (e.g. read-only access.)
- The server also maintains a list of currently connected clients, so that they can be notified in the event of the server going down and for other reasons.
- Autounmount and automount are supported.

#### The NFS Protocol

- Implemented as a set of remote-procedure calls (RPCs):
- Searching for a file in a directory
- Reading a set of directory entries
- Manipulating links and directories
- Accessing file attributes
- Reading and writing files

- 1.4K (cylinder 4K blocks) are allocatable with only a single location. Which is good for cache 1, and higher disk after with more sectors containing multiple cylinders, which is why sector relocations & latency issues often addressed by a 32-bit free-space management.

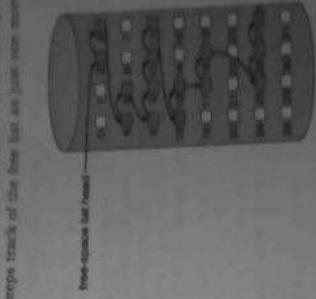
- Another important aspect of disk management is keeping track of where allocation free space.

#### Linked List

- One simple approach is to use a list structure to which each bit represents a disk block, set to 1 if free or 0 if allocated.

- Fast algorithms exist for quickly finding contiguous blocks of a given size.

- The down side is that a 4GB disk requires over 3000 bits to store the bitmap. (For example...)



#### Counting

- When there are multiple contiguous blocks of free space then the length, and length of the successive address of the group and the number of contiguous free blocks. As long as the average length of the contiguous groups of free blocks is greater than two this offers a savings in space requirement for the free list.

#### Space Maps

- Sun's ZFS file system was designed for HUGE memory and uses a ZFS directioan, and even file systems directioan, and even file systems

#### Free-Space List on Disk

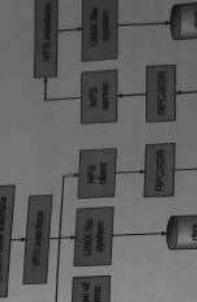
- A variation on linked list free lists is to use blocks of blocks of blocks of free blocks. If a block holds up to N addresses, then the free block on this block list contains up to N-1 addresses of free blocks and a pointer to the next block of free addresses.

#### Grouping

- Traversing the list until a finding a consecutive block of a given size is not easy, but fortunately are not frequently needed operations. Generally the system just adds and removes single blocks from the beginning of the free list.

#### FAT Table

- The FAT table keeps track of the free list as block numbers (block free list on the table).



#### Path Name Translation

- These procedures can be involved only after a file handle for the remoted mounted directory has been established. Servers do not maintain information about their clients from one access to another.

#### Three Major Layers of NFS Architecture

- UNIX file system interface based on the open, read, write, and close calls, and file descriptors. Virtual File System (VFS) layer - distinguishes local files from remote ones, and local files are further distinguished according to their file system types. A The NFS activates file system specific operations to handle local requests according to their file system types. A Call the NFS protocol procedures for remote requests v NFS service layer - bottom layer of the architecture. A implements the NFS protocol.

#### Path Name Translation

- /usr/local/dir1/file.txt into separate directory entries, or components: (1) user, (2) local, and (3) dir. Path-name translation is done by breaking the path into component names and performing a separate NFS lookup call for every pair of component name and directory entry. Once a full path is crossed, every component lookup causes a separate RPC to the server.

- NFS Remote Operations v Nearly one-to-one correspondence between regular UNIX file calls and the NFS protocol RPCs except opening and closing files. NFS adheres to the remote-service paradigm, but employs buffering and caching techniques for the sake of performance. v File-blocks cache - when a file is opened, the kernel checks with the remote server whether to fetch or validate the cached attributes. Cached file blocks are used only if the corresponding cached attributes are up to date v File-attribute cache - the attribute cache is updated whenever new attributes arrive from the server v Clients do not free delayed-write blocks until the server confirms that the data have been written to disk.

#### QUESTIONS

1. List and explain different file attributes.
2. What different are the file types? Explain.
3. List and explain the various file operations?

4. Write a short note on access methods.
5. Explain the concepts of file system mounting.
6. Explain file sharing.
7. Write a short note on different types of Directory.
8. Write a short note on Magnetic disks
9. Explain the concept of Disk Formatting in detail.
10. Write a short note on Proc-Space Management.
11. Explain Swapping.
12. Define Paging. Explain Page Fault.
13. Explain the structure of Page Table.
14. Write a short note of Threading.
15. Explain Paging.

#### Objective Questions

##### I. Multiple Choice Questions

1. Which of the following is the fastest means of memory access for CPU?
  - a) Registers
  - b) Cache
  - c) Main memory
  - d) Virtual Memory
2. CISC stands for \_\_\_\_\_
  - a) Complex Information Sensed CPU
  - b) Complex Instruction Set Computer
  - c) Complex Intelligence Sensed CPU
  - d) Complex Instruction Set CPU
3. Access in which records are accessed from and inserted into file, is classified as
  - a) direct access
  - b) sequential access
  - c) random access
  - d) duplicate access
4. To create a file
  - a) allocate the space in file system
  - b) make an entry for new file in directory
  - c) allocate the space in file system & make an entry for new file in directory
  - d) none of the mentioned
5. Mapping of file is managed by
  - a) file metadata
  - b) page table
  - c) virtual memory
  - d) file system
6. A file control block contains the information about
  - a) file ownership
  - b) file permissions
  - c) location of file contents
  - d) all of the mentioned

Ques.

Ans.

Ques.

Ans.