Introduction and Operating System Services:

Introduction to Operating System: Basics of Operating Systems: Definition, Operations – Dual-Mode and Multi-Mode, Services, System Calls – Types. Operating System Structure: Layered Structure, Microkernel's, Modules, Hybrid Systems – Mac OS X, iOS, Android

1. INTRODUCTION

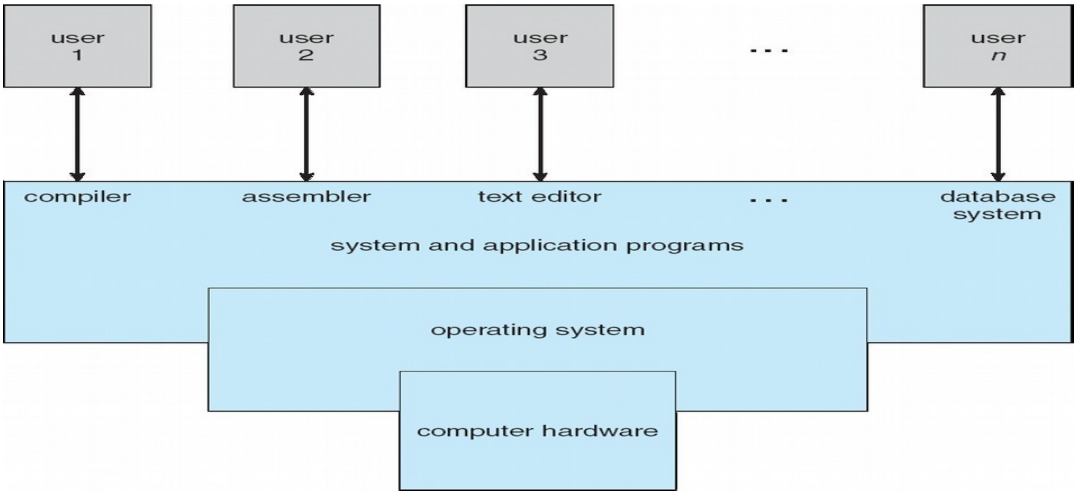
- An operating system **acts as an intermediary between the user of a computer and the computer hardware.**
- The purpose of an operating system is to provide an environment in which a user can execute programs in a **convenient and efficient** manner.
- An operating system is software that **manages the computer hardware**. The hardware must provide appropriate mechanisms to ensure the correct operation of the computer system and to prevent user programs from interfering with the proper operation of the system.
- The operating system is a **control program** which acts as a **resource manager** and provides a **convenient,secure,and**

efficient environment for the execution of user programs.

- The operating system is a program that runs all the time- right from **boot up till shut down**.

1.1 What Operating Systems Do:Basic

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



- **The hardware**—the central processing unit (CPU), the memory, and the input/output (I/O) devices—provides the basic computing **resources** for the system.
- The **application programs**—such as word processors, spreadsheets,compilers, and Web browsers—define the ways in which these resources are used to solve users' computing problems.

- **The operating system** controls the hardware and coordinates its use among the various application programs for the various users.
- The operating system provides the means for proper use of these resources in the operation of the computer system.

1.2 Challenges in the Operating System Design

- An operating system is **large and complex**, it must be created piece by piece. Each of these pieces should be a **well-delineated portion of the system**, with **carefully defined inputs, outputs, and functions.**

- Since the operating system should provide an efficient and secure environment for the execution of user programs,it must **protect** itself as well as other programs.

- **The Operating system design varies considerably with respect to various environments.**For example,Mainframe operating systems are designed primarily to optimize utilization of hardware whereas Personal computer (PC) operating systems is mainly designed for convenience with support for complex games, business applications, and everything in between. Operating systems for mobile computers is designed to provide an environment in which a user can easily interface with the computer to execute programs.
- Hence while designing the operating system the goals have to

be listed with respect to the user and system point of view.

- There are two views to be considered while designing an operating system:**User View and the System view.**

USER VIEW

- The user's view of the computer varies according to the interface being used.
- In the case of **Personal Computers**,the operating system is designed mostly for **ease of use**, with some attention paid to performance and none paid to resource utilization ie how various hardware and software resources are shared.Such systems are optimized for the single-user experience rather than the requirements of multiple users.
- In case of **mainframe computers**,wherein a user sits at a terminal connected to a mainframe or a minicomputer,the operating system is designed **to maximize resource utilization ie** to assure that all available CPU time, memory, and I/O are used **efficiently** and that no individual user takes more than her **fair** share.

- In the case of Workstations,wherein users have dedicated resources at their disposal, but they also share resources such as networking and servers,including file, compute, and print servers. The operating system designed is to compromise between **individual usability and resource utilization.**

- In the case of mobile computing wherein we discuss many varieties of mobile computers, such as smartphones and tablets,Which are basically standalone units for individual users,but connected to networks through cellular or other wireless technologies,the users providing the inputs via a touch screen,the operating system is mainly designed for user **convenience,efficient resource management and also power management stratergies.**

- In case of embedded systems,which has very little User interface,except for some numeric keypads or indicator lights and start/stop button,the operating systems are designed primarily **to run without user intervention with very critical time constraints.**

SYSTEM VIEW

- From the computer’s point of view, the operating system is the program most intimately involved with the hardware. In this context, we can view an operating system as a **resource allocator**.
- 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.

- The OS should decide **how to allocate the resources** to specific programs and users so that it can operate the computer system efficiently and fairly and **resolve various conflicting requests.**
- Since the Operating system should be a control program,it should **prevent errors and improper use of the computer** . It is especially concerned with the operation and control of I/O devices.

1.3 IMPORTANCE OF OPERATING SYSTEM

- The operating system is the **one program running at all times** on the computer—usually called **the kernel**. (Along with the kernel, there are two other types of programs: **system programs,** which are associated with the operating system but are not necessarily part of the kernel, and **application programs**, which include all programs not associated with the operation of the system.)
- The Operating system has so much growing importance due to the fact that **computing is now pervasive in nature** ie With the widespread use of Personal Computers and Hand held devices,Computing is not restricted to rather one place.
- Mobile operating systems often include not only a core kernel but also **middleware**—a set of software frameworks that provide additional services to application developers. For example, each of the two most prominent mobile operating systems—**Apple's i OS and Google's Android —features a core**

kernel along with middleware that supports databases, multimedia, and graphics.

1.4 COMPUTER SYSTEM OPERATION

- A modern general-purpose computer system consists of one or more CPU s and a number of device controllers connected through a common bus that provides access to shared memory.
- Each device controller is in charge of a specific type of device (for example, disk drives, audio devices, or video displays).
- The CPU and the device controllers can execute in parallel, competing for memory cycles. To ensure orderly access to the shared memory, a memory controller synchronizes access to the memory.
- For a computer to start running—for instance, when it is powered up or rebooted —it needs to have an initial program to run. This initial program,or **bootstrap program**, tends to be simple. Typically, it is stored within the computer hardware in **read-only memory (ROM) or electrically erasable programmable read-only memory (EEPROM)**, known by the general term **firmware.**

1.5 FUNCTIONS OF BOOTSTRAP PROGRAM

- It **initializes all aspects of the system**, from CPU registers to device controllers to memory contents.
- The bootstrap program must know **how to load the operating system and how to start executing that system** . To accomplish this goal, the bootstrap program must **locate the operating-system kernel and load it into memory.**
- Once the kernel is loaded and executing, it can start providing services to the system and its users. **Some services are provided outside of the kernel, by system programs that are loaded into memory at boot time to become system processes, or system daemons that run the entire time the kernel is running.**
- Once this phase is complete, the system is fully booted, and the system **waits for some event** to occur.
- The occurrence of an event is usually signaled by an **interrupt from either the hardware or the software.**
- **Hardware** may trigger an interrupt at any time by sending a **signal to the CPU** , usually by way of the system bus.
- **Software** may trigger an interrupt by executing a special operation called a **system call** (also called a monitor call).

1.6 HANDLING INTERRUPTS

- When the CPU is interrupted, it stops what it is doing and immediately transfers execution to a fixed location.

- The fixed location usually contains the **starting address** where the **service routine for the interrupt is located.**
- The interrupt service routine executes; on completion, the CPU resumes the interrupted computation.
- The interrupt must transfer control to the appropriate interrupt service routine.
- The straightforward method for handling this transfer would be to invoke a generic routine to examine the interrupt information. The routine, in turn, would call the i**nterrupt-specific handler**.
- However, interrupts must be handled **quickly.**
- Generally implemented by maintaining a **table of pointers to interrupt routines** to improve speed. **(INTERRUPT VECTOR TABLE)**
- The interrupt routine is called indirectly through the table, with no intermediate routine needed.
- Generally, the table of pointers is stored in **low memory** (the first hundred or so locations). These locations hold the addresses of the interrupt service routines for the various devices. On encountering any interrupt,it performs a look up in the table regarding which ISR to be invoked.
- More recent architectures **store the return address on the system stack**.
- If the interrupt routine needs to modify the processor state—for instance, by modifying register values—it must explicitly save the current state and then restore that state before returning. After the interrupt is serviced, the saved return

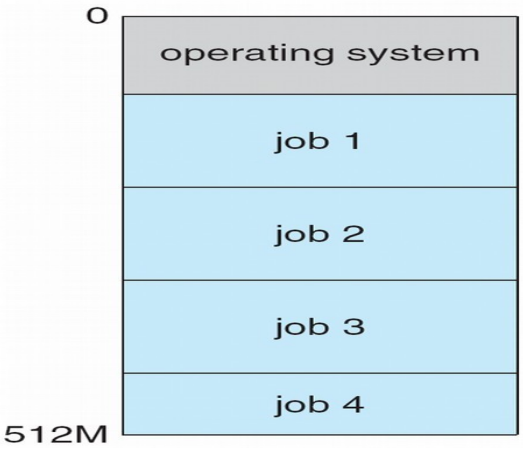
address is loaded into the program counter, and the interrupted computation resumes as though the interrupt had not occurred.

2. OPERATING SYSTEM STRUCTURE

- Operating systems vary greatly in their makeup, since they are organized along many different lines.
- One of the most important aspects of operating systems is the ability to multiprogram. Since a single program cannot, in general, keep either the CPU or the I/O devices busy at all times,modern operating systems are designed to do Multiprogramming to increase CPU Utilization.
- Single users frequently have multiple programs running.
- **Multiprogramming** is the ability of the system to organize jobs in the main memory which are ready to be executed so that the CPU always has one to execute.
- The idea is as follows:The jobs are intially kept in hard disk(job pool).
- From the job pool,the operating system picks up those jobs which are ready to be executed into the main memory.
- The operating system picks up job from the main memory(CPU scheduling) and allocates CPU so that it begins to execute .
- Eventually, the job may have to wait for some task, such as an I/O operation, to complete,or wait for an event etc,the operating system simply switches to, and executes, another job. When that job needs to wait, the CPU switches to another job, and so on.

- In a non-multiprogrammed system, the CPU would sit idle. But here,the CPU is given some task to be executed so that it is never idle.

FIG:MEMORY LAYOUT FOR MULTIPROGRAMMING



ADVANTAGES OF MULTIPROGRAMMING

- Multiprogrammed systems provide an environment in which the various system resources (for example, CPU , memory, and peripheral devices) are **utilized effectively,** but they do not provide for user interaction with the computer system.

MULTITASKING

- Time sharing (or multitasking) is a logical extension of multiprogramming. In time-sharing systems, the CPU executes multiple jobs by switching among them, but the switches occur so frequently that the users can interact with each program while it is running.

- Time sharing **requires an interactive computer system**, which provides direct communication between the user and the system.
- The user gives instructions to the operating system or to a program directly, **using a input device** such as a keyboard, mouse, touch pad, or touch screen, and waits for immediate results on an output device. Accordingly, the **response time should be short—typically less than one second.**
- A time-shared operating system **allows many users to share the computer simultaneously.** Since each action or command in a time-shared system tends to be short, only a little CPU time is needed for each user. As the system switches rapidly from one user to the next, each user is given the impression that the entire computer system is dedicated to his use, even though it is being shared among many users.
- A time-shared operating system **uses CPU scheduling and multiprogramming** to provide each user with a small portion of a time-shared computer.
- Time sharing and multiprogramming require that several jobs be kept simultaneously in memory. If several jobs are ready to be brought into memory,job scheduling need to be performed and CPU scheduling need to be done among them for the allocation of CPU.

MULTIPROCESSOR SYSTEMS

Multiprocessor systems are those systems which has more than one CPU.

ADVANTAGES

- Increased Throughput due to parallel execution
- Increased Reliability
- Economy of Sale

3. OPERATING SYSTEM OPERATIONS

- Modern operating systems are interrupt driven. Ie The Operating system will be waiting for any event to happen.
- Events are almost always signaled by the occurrence of an interrupt or a trap.
- A **trap** (or an exception) is a software-generated interrupt caused either by an error (for example, division by zero or invalid memory access) or by a specific request from a user program that an operating-system service be performed.
- The interrupt-driven nature of an operating system defines that system's general structure. Ie For each type of interrupt, separate segments of code in the operating system determine what action should be taken.
- An interrupt service routine**(ISR)** is provided to deal with the interrupt.
- When many user programs and Operating system code need to share the hardware and software resources of the computer system, we need to ensure that an error in a user program could cause problems only for the one program running.

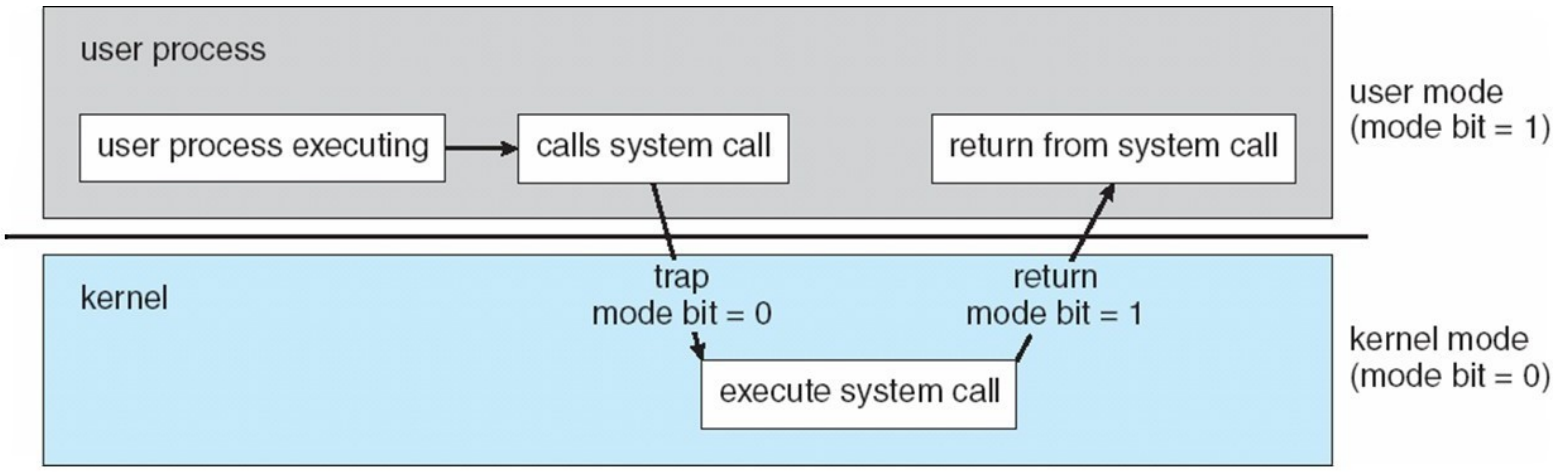
- With sharing enabled,the Operating System should not only protect itself from corruption due to malicious programs,but also protect other processes from getting corrupted.
- For example, if a process gets stuck in an infinite loop, this loop could prevent the correct operation of many other processes.
- A properly designed operating system must ensure that an incorrect (or malicious) program cannot cause other programs to execute incorrectly.

3.1 IMPLEMENTATION OF SECURE EXECUTION ENVIRONMENT

Dual-Mode and Multimode Operation

- In a system,the two different programs in execution are 1)User Process 2)System Process
- In order to ensure the proper execution of the operating system, we must be able to distinguish between the execution of operating-system code and user defined code.
- The set of instructions which should be executed only by the kernel are called as **Privileged Instructions.**Thus there is a need to differentiate between the user and system processes.
- The approach taken by most computer systems is to provide hardware support that allows us to differentiate among various modes of execution.
- At the very least, we need two separate modes of operation: **user mode** and **kernel mode** (also called supervisor mode, system mode, or privileged mode).

- A bit, called the mode bit, is added to the hardware of the computer to indicate the current mode:if modebit=0 then it implies kernel mode and if modebit=1,it implies user mode.
- With the mode bit, we can distinguish between a task that is executed on behalf of the operating system and one that is executed on behalf of the user. When the computer system is executing on behalf of a user application, the system is in user mode.
- when a user application requests a service from the operating system (via a system call), the system must transition from **user to kernel mode** to fulfill the request.
- At system boot time, the **hardware starts in kernel mode**. The operating system is then loaded and starts user applications in user mode. Whenever a trap or interrupt occurs, the hardware switches from user mode to kernel mode (that is, changes the state of the mode bit to 0). Thus, whenever the operating system gains control of the computer, it is in kernel mode. The system always switches to user mode (by setting the mode bit to 1) before passing control to a user program.



- The dual mode of operation provides us with the means for protecting the operating system from errant users—and errant users from one another.
- The hardware allows privileged instructions to be executed only in kernel mode. **If an attempt is made to execute a privileged instruction in user mode, the hardware does not execute the instruction but rather treats it as illegal and traps it to the operating system.**
- The instruction to switch to kernel mode is an example of a privileged instruction. Some other examples include I/O control, timer management, and interrupt management.
- The concept of modes can be extended beyond two modes. **(Multimode)**
- In this case the CPU uses more than one bit to set and test the mode. Eg:**CPU s that support virtualization** frequently have a separate mode to indicate when the virtual machine manager (VMM)—and the virtualization management software—is in control of the system. In this mode, the VMM has more privileges than user processes but fewer than the kernel. It needs that level of privilege so it can create and manage virtual machines, changing the CPU state to do so. Sometimes, too, different modes are used by various kernel.

- Alternatively instead of having different mode for virtualization,The CPU designer can use other methods to differentiate operational privileges. Eg:The Intel 64 family of CPUs supports four privilege levels.

4. Operating-System Operations

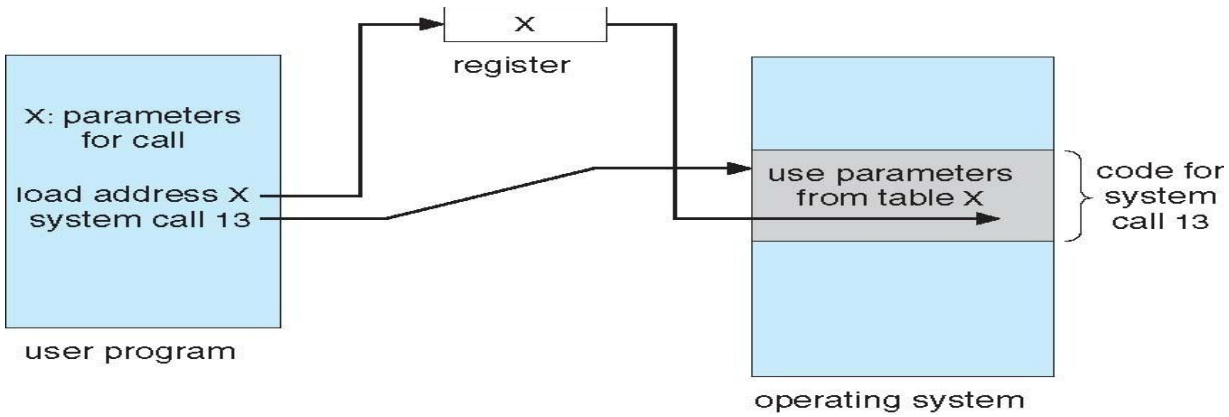
The life cycle of instruction execution in a computer system.

- **Initial control** resides in the operating system, where instructions are executed in **kernel mode.**
- When **control is given to a user application**, the mode is set to **user mode**. Eventually, control is switched back to the operating system via an **interrupt, a trap, or a system call.**

SYSTEM CALLS

- **System calls** provide the means for a user program to ask the operating system to perform tasks reserved for the operating system on the user program's behalf.
- A system call is **invoked in a variety of ways**, **depending on the functionality provided by the underlying processor.** In all forms, it is the method used by a process to request action by the operating system. A system call usually **takes the form of a trap to a specific location in the interrupt vector.**
- This trap can be executed by a generic trap instruction, although some systems (such as MIPS) have a **specific syscall** instruction to invoke a system call.

- When a **system call is executed,** it is typically treated by the hardware as a **software interrupt.** Control passes through the **interrupt vector to a service routine** in the operating system, and the **mode bit is set to kernel mode**. The system-call service routine is a part of the operating system.
- The kernel examines the interrupting instruction to **determine what system call has occurred; a parameter indicates what type of service the user program is requesting**.
- Additional information needed for the request may be passed through any of the following methods
  - **Registers:Simple and fast,but cannot be used if there are large number of parameters.**
  - **Stack:Parameters can be stored and retrieved from stack using push and pop operation.**
  - **Block or table method** :In this method,the parameters are stored in memory (with pointers to the memory locations passed in registers). The kernel verifies that the parameters are correct and legal, executes the request, and returns control to the instruction following the system call.



WHY DO WE REQUIRE ATLEAST DUAL MODE OPERATION

- The lack of a hardware-supported dual mode can cause serious shortcomings in an operating system.
- For instance, **MS-DOS** was written for the Intel 8088 architecture, which has **no mode bit** and therefore **no dual mode**.
- A user program running awry can wipe out the operating system by writing over it with data; and multiple programs are able to write to a device at the same time, with potentially disastrous results.
- Modern versions of the Intel CPU do provide dual-mode operation.
- Accordingly, most contemporary operating systems—such as Microsoft Windows 7, as well as Unix and Linux take advantage of this dual-mode feature and **provide greater protection for the operating system.**
- Once hardware protection is in place, **it detects errors that violate modes.**

STEPS TAKEN BY OPERATING SYSTEM ON DETECTION OF ERRORS THAT VIOLATE MODES

CASE: If a user program fails in some way—such as by making an attempt either to execute an illegal instruction or to access memory that is not in the user’s address space

- **STEP 1:**Hardware traps to the operating system.
- **STEP 2**:The trap transfers control through the interrupt vector to the operating system, just as an interrupt does.
- **STEP 3:**The OS identifies the type of interrupt,executes the corresponding ISR(Interrupt Service Routines) and displays suitable error messages and terminates the program .
- **STEP 4:** An appropriate error message is given, and the memory of the program may be dumped. The memory dump is usually written to a file so that the user or programmer can examine it and perhaps correct it and restart the program.

IMPORTANCE OF TIMER

- We must ensure that the operating system maintains control over the CPU .
- We cannot allow a user program to get stuck in an infinite loop or to fail to call system services and never return control to the operating system.
- To ensure that the control will be returned to kernel mode ,we require the use of timers.
- A **timer can be set to interrupt the computer after a specified period.** The period may be fixed (for example, 1/60 second) or variable (for example, from 1 millisecond to 1 second).
- A **variable timer is generally implemented by a fixed-rate clock and a counter.**

IMPLEMENTATION

- The operating system **sets the counter**. Every time the clock ticks, the **counter is decremented**. When the **counter reaches 0, an interrupt occurs**.
- Before turning over control to the user, the operating system ensures that the timer is set to interrupt. **If the timer interrupts, control transfers automatically to the operating system,** which may treat the interrupt as a fatal error or may give the program more time.

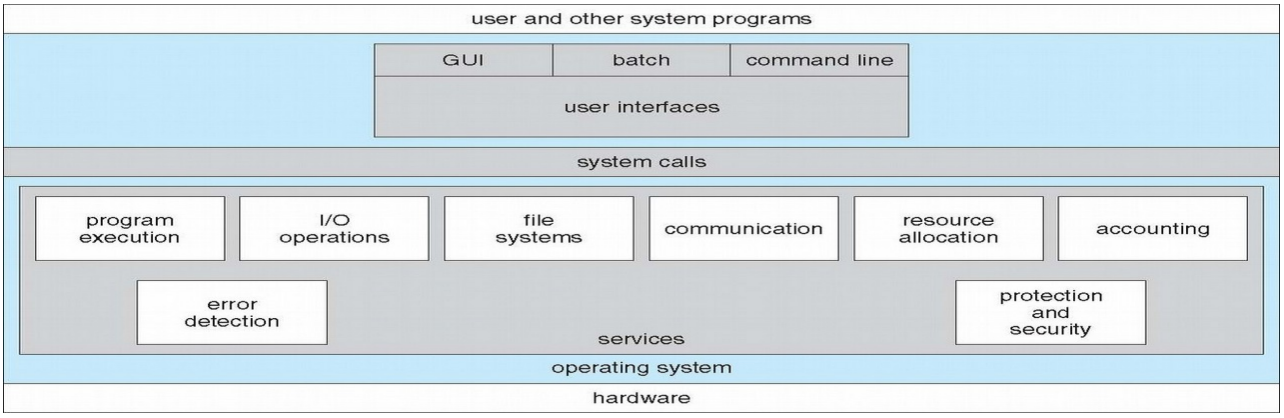
Instructions that modify the content of the timer are privileged instructions

- Timer can be used to prevent a user program from running too long.
- A simple technique is to **initialize a counter with the amount of time that a program is allowed to run.**
- A program with a 7-minute time limit, for example,would have its counter initialized to 420. Every second, the timer interrupts,and the counter is decremented by 1. As long as the counter is positive, control is returned to the user program.
- When the counter becomes negative, the operating system terminates the program for exceeding the assigned time limit.

Operating-System Services

- **An operating system provides an environment for the execution of programs.It provides certain services to programs and to the users of those programs.**

- The services provided by the OS can be broadly classified from two view points:**USER AND SYSTEM**



USER VIEW POINT SERVICES

1.User interface. Almost all operating systems have a user interface **(UI)**. This interface can take several forms like CLI,GUI,Batch interface etc.

- **command-line interface (CLI)**, which uses text commands and a method for entering them (say, a keyboard for typing in commands in a specific format with specifications).
- **Batch interface,** in which commands and directives to control those commands are entered into files, and those files are executed.
- **Graphical user interface (GUI)** is a window system with a pointing device to direct I/O , choose from menus, and make selections and a keyboard to enter text.

2.Program execution. The system must be able to **load** a program into memory and to **run** that program. The program must be able to **end its execution**, either **normally or abnormally** (indicating error).

3.I/O operations. A running program may require I/O , which may involve a file or an I/O device. For specific devices, special functions may be desired (such as recording to a CD or DVD drive or blanking a display screen). **For efficiency and protection, users usually cannot control I/O devices directly. Therefore, the operating system must provide a means to do I/O .**

4. **File-system manipulation.** The programs need to **read and write files and directories**. They also need **to create and delete** them by name, **search** for a given file, and list file information. Finally, some operating systems include **permissions management to allow or deny access to files or directories based on file ownership**. Many operating systems provide a variety of file systems, sometimes to allow personal choice and sometimes to provide specific features or performance characteristics.

5. **Communications:** For the working of the system,One process needs to exchange information with another process. Such communication may occur between **processes that are executing on the same computer or between processes that are executing on different computer systems tied together by a computer network.**

- Communications may be implemented via **shared memory,** in which two or more processes read and write to a shared section of memory, or **message passing**, in which packets of information in predefined formats are moved between processes by the operating system.

6. **Error detection:**The operating system needs to be detecting and correcting errors constantly. **Errors may occur in the CPU** and **memory hardware** (such as a memory error or a power failure), in **I/O devices** (such as a parity error on disk, a connection failure on a network, or lack of paper in the printer), and in the **user**

program (such as an arithmetic overflow, an attempt to access an illegal memory location, or a too-great use of CPU time).

- For each type of error, the operating system should take the appropriate action.
- To ensure **correct and consistent** computing the OS does one of the following

-halt the system

-terminate an error-causing process or **return an error code to a process for the process to detect and possibly correct.**

SYSTEM VIEWPOINT SERVICES

- These services exists not for helping the user but rather for ensuring the efficient operation of the system itself.

1.Resource allocation: For the multiple jobs running at the same time, limited resources must be allocated in most efficient manner.

- The operating system manages many different types of resources.
- Resources such as CPU cycles, main memory, and file storage may have **special allocation code,** whereas others (such as I/O devices) may have much more **general request and release code**.
- For instance, in determining **how best to use the CPU** , operating systems have **CPU -scheduling routines** that take into

account the **speed of the CPU** , the **jobs** that must be executed, the **number of registers** available, and other factors.

- For the efficient use of **I/O devices** also routines exist to allocate printers, USB storage drives, and other peripheral devices.

2.Accounting: For **efficient allocation** of resources,the operating system must **keep track of which users use how much and what kinds of computer resources**. This record keeping may be used for accounting (so that **users can be billed**) or simply for **accumulating usage statistics**. Usage statistics may be a **valuable tool for researchers who wish to reconfigure the system to improve computing services.**

3.Protection and security. The owners of information stored in a multiuser or networked computer system should be given the control of decision to use of that information.

- When several separate processes execute concurrently, **one process should not interfere with the others or with the operating system itself.**
- **Protection** involves ensuring that all **access to system resources is controlled. Protection is all about what to implement.**
- **Security is all about how to implement**
- Such security starts with requiring each user to **authenticate** himself or herself to the system, usually by means of a **password,** to gain access to system resources. It extends to d**efending**

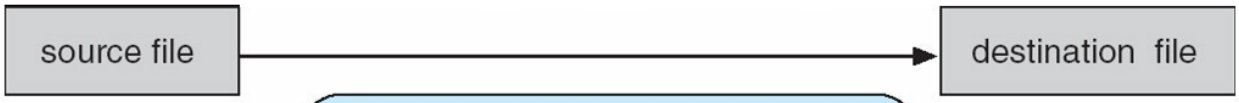
external I/O devices, including network adapters, from invalid access attempts and to recording all such connections for detection of break-ins.

SYSTEM CALLS

- System Calls **provide an interface to the services made available by an operating system**. These calls are generally available as **routines written in C and C++**, although certain **low-level tasks** (for example, tasks where hardware must be accessed directly) may have to be written using **assembly-language instructions.**
- **EG:To write a simple program to read data from one file and copy them to another file.**
- **STEP 1:GET NAMES OF FILES-**These names can be specified in many ways, depending on the operating-system design. One approach is for the program to **ask the user for the names**. In an **interactive system, this approach will require a sequence of system calls, first to write a prompting message on the screen and then to read from the keyboard the characters that define the two files**. On **mouse-based and icon-based systems**, a menu of **file names is usually displayed in a window.**
- The user can then use the mouse to select the source name, and a window can be opened for the destination name to be specified. This sequence requires many I/O system calls.

- **STEP 2:open the input file and create the output file** . Each of these operations requires another system call. Possible error conditions for each operation can require additionalsystem calls.
    - **ERROR CONDITIONS:** No input file ,NO access permission for the input file,Output file already exists,
    - In these cases, the program should print a message on the console (another sequence of system calls) and then terminate abnormally (another sequence of system call).

- **STEP3:**When both files are set up, we enter a l**oop that reads from the input file (a system call) and writes to the output file (another system call).**
- **STEP 4:**Each read and write must **return status information regarding various possible error conditions.**
    - **POSSIBLE ERRORS:** the end of the file ,a hardware failure in the read (such as a parity error).The write



ut device

Example System Call Sequence

```
Acquire input file name
  Write prompt to screen
  Accept input
Acquire output file name
  Write prompt to screen
  Accept input
Open the input file
  if file doesn't exist, abort
Create output file
  if file exists, abort
Loop
  Read from input file
  Write to output file
Until read fails
Close output file
Write completion message to screen
Terminate normally
```

- The application developers design programs according to an **application programming interface (API)**. The API specifies a **set of functions** that are available to an application programmer, including the parameters that are passed to each function and the return values the programmer can expect.
- **Three of the most common API** s available to application programmers are
  - the **Windows API** for Windows systems
  - the **POSIX API** for POSIX -based systems(which include virtually all versions of UNIX , Linux, and Mac OS X
  - the **Java API-**API for programs that run on the Java virtual machine.
  - A programmer **accesses an API via a library of code** provided by the operating system.
  - In the case of UNIX and Linux for programs written in the C language, the library is called **libc**. N
- Behind the scenes, **the functions that make up an API typically invoke the actual system calls on behalf of the application programmer.**
- For example, the Windows function **CreateProcess()** (which unsurprisingly is used to create a new process) **actually invokes the NT CreateProcess() system call** in the Windows kernel.

- Why would an application programmer prefer programming according to an API rather than invoking actual system calls?

- One benefit concerns **program portability.** An application programmer designing a program using an API can expect her program to compile and run on any system that supports the same API (although, in reality, architectural differences

often make this more difficult than it may appear).

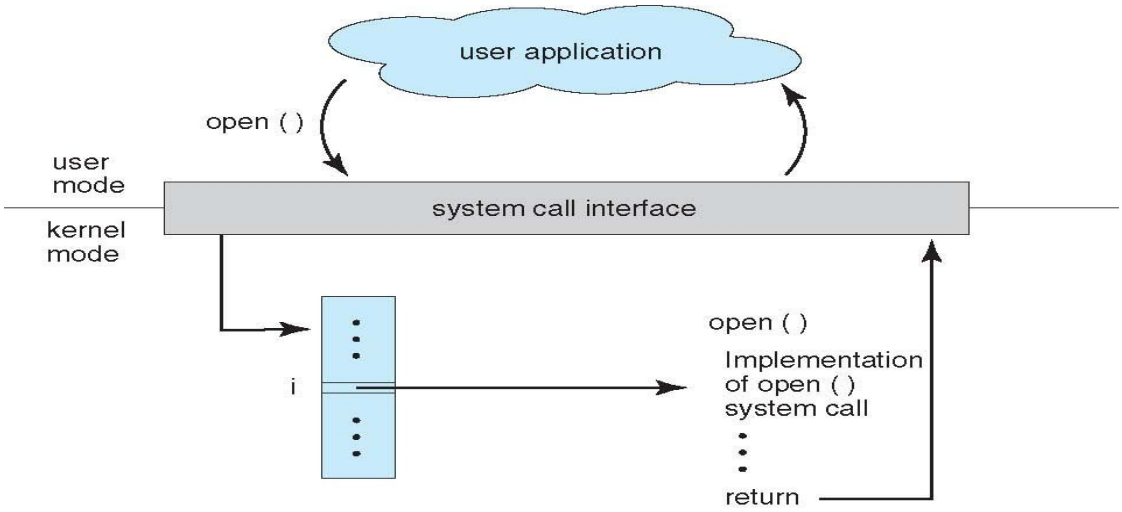
- Actual **system calls can often be more detailed and difficult to work** with than the API available to an application programmer. Nevertheless, there often exists a strong correlation between a function in the API and its associated system call within the kernel. In fact, many of the POSIX and Windows API s are similar to the native system calls provided by the UNIX , Linux, and Windows operating systems.
- **ABSTRACTION:**The caller need know nothing about how the system call is implemented or what it does during execution. Rather, the caller need only obey the API and understand what the operating system will do as a result of the execution of that system call. Thus, most of the details of the operating-system interface are hidden from the programmer by the API and are managed by the run-time support library.
- For most programming languages, the run-time support system (a set of functions built into libraries included with a compiler) provides a **system- call interface** that **serves as the link to system calls made available by the operating system.**

- The **system-call interface intercepts function calls in the API and invokes the necessary system calls within the operating system.**

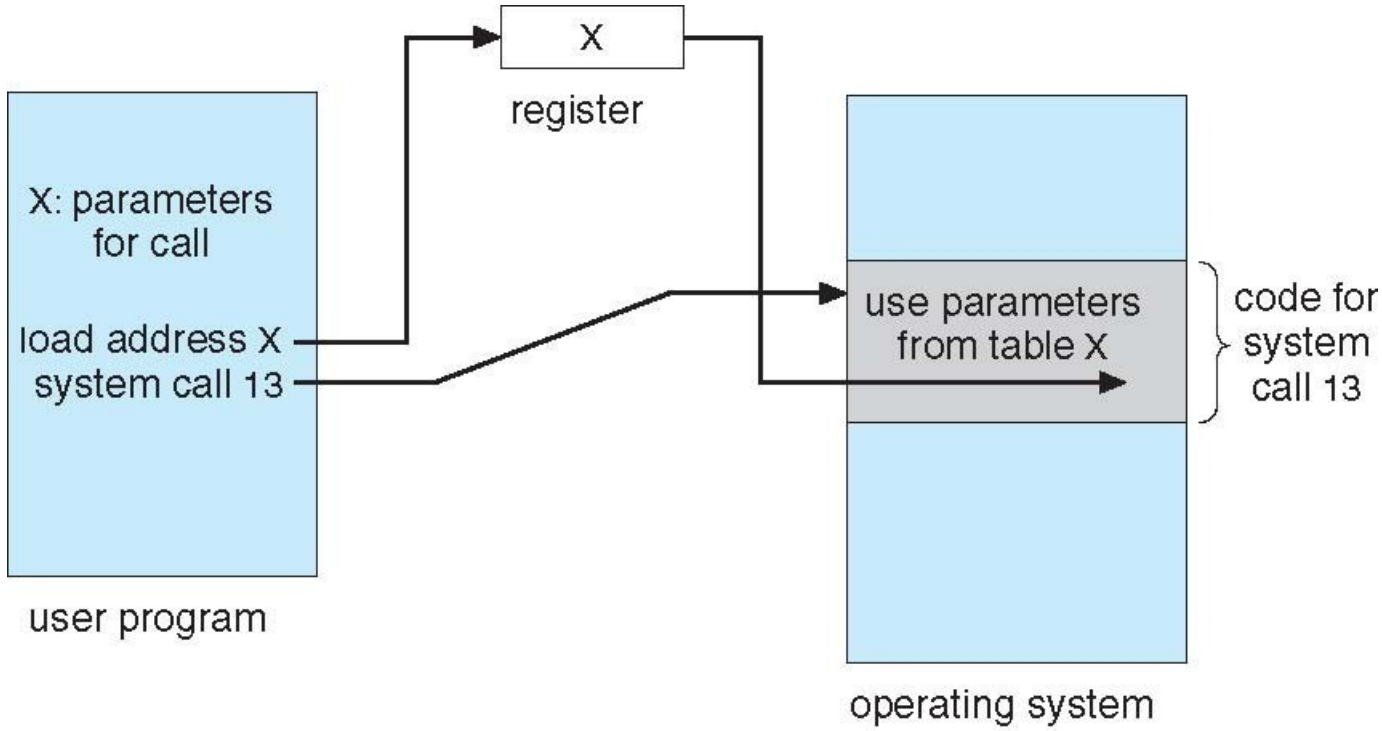
- **IMPLEMENTATION:**
- A **number is associated with each system call**, and the system-call interface maintains a **table indexed according to these numbers**.
- The **system call interface then invokes the intended system call in the operating-system kernel and returns the status of the system call and any return values.**

- operating system handles a user application invoking the open() system call.

SYSTEM CALL -API-OS RELATIONSHIP



- System calls occur in different ways, depending on the computer in use.
- Often, more information is required than simply the identity of the desired system call.ie **parameters** need t be passed in system calls like the input file or device to use as the source, as well as the address and length of the memory buffer into which the input should be read.
- **Three general methods** are used to pass parameters to the operating system.
  - **Registers:**The **simplest** approach is to pass the parameters in but not suitable if the parameters to be passed exceeds the no of registers.
  - **Block, or table,** in memory, and the address of the block is passed as a parameter in a register. This is the approach taken by Linux and Solaris.



- **Stack:** Parameters also can be placed, or pushed, onto the stack by the program and popped off the stack by the operating system.

- Some operating systems prefer the block or stack method because those approaches do not limit the number or length of parameters being passed.

TYPES OF SYSTEM CALLS

- System calls can be grouped roughly into **six** major categories:

● process control
● file manipulation
● device manipulation
● information maintenance,

● communications
● protection.

Process Control

When a process is in execution various scenarios happen.

- For a program to be executed it has to be first **loaded** into the main memory.This is done via **load()** system call.
- A running program **needs to be able to halt its execution either normally (end()) or abnormally (abort())**.
- Under either normal or abnormal circumstances, the operating system must transfer control to the invoking **command interpreter.**
  - In an interactive system, the command interpreter simply continues with the **next command** assuming user will issue appropriate command.
  - In a **GUI system,** a **pop-up window** might alert the user to the error and ask for guidance.
  - In a batch system, the command interpreter usually terminates the entire job and continues with the next job.
- A process or job executing one program may want to **load()** and **execute()** another program. Eg:keeping track of mouse clicks to execute the necessary commands.
- If control returns to the existing program when the new program terminates, we must **save the memory image of the existing program;** thus, we have effectively created a mechanism for one program to call another program. If both

programs continue concurrently, we have created a new job or process to be multiprogrammed. Often, there is a system call specifically for this purpose **(create process() or submit job()).**

- The process has to be controlled while in the system,like the the ability to determine and reset the attributes of a job or process, including the job's priority, its maximum allowable execution time, and so on.For this we can use the **get process attributes() and set process attributes() system calls**.
- To terminate a job or process that we created we use **(terminate process()) system call** if it is incorrect or is no longer needed.
- Having created new jobs or processes, we may need to wait for them to finish their execution. We may want to wait for a certain amount of time to pass **(wait time())**. or wait for a specific event to occur **(wait event())**. The processes should signal when that event has occurred **(signal event()).**
- Quite often, two or more processes may **share** data. To ensure the integrity of the data being shared, operating systems often provide **system calls allowing a process to lock shared data.** Then, no other process can access the data until the lock is released. Typically, such system calls include acquire **lock() and release lock()** .

SUMMARY OF SYSTEM CALLS UNDER PROCESS CONTROL

● end, abort

● load, execute
● create process, terminate process
● get process attributes, set process attributes
● wait for time
● wait event, signal event
● allocate and free memory

File Management

- There are various system calls which comes under file management.
- Ability to create or delete files via **create() and delete()** files.
- Either system call requires the **name of the file and perhaps some of the file's attributes**.
- Once the file is created, we need to **open()** it and to use it. We may also **read() , write() , or reposition()** (rewind or skip to the end of the file, for example).
- Finally, we need to **close()** the file, indicating that we are no longer using it.
- We may need these same sets of operations for directories if we have a directory structure for organizing files in the file system.
- **Attributes** are those properties that define a file.Eg:**name, file type, protection codes, accounting information,** and so onwith respect to attributes,the operating system should be able to **get file attributes()or** In addition, for either

files orsometimes set some of the file attributes**(setfileattributes())**

- Some operating systems provide many more calls, such as calls for **file move() and copy()** .

SUMMARY OF SYSTEM CALLS UNDER FILE MANAGEMENT

● File management
● create file, delete file
● open, close
● read, write, reposition
● get file attributes, set file attributes

Device Management

- A process may need several resources to execute —main memory, disk drives, access to files, and so on.
- If the resources are **available**, they can be **granted**, and control can be returned to the user process.
- Otherwise, the process will have to **wait** until sufficient resources are available.
- The devices controlled by the operating system can be **physical devices** (for example, disk drives), or abstract **virtual** devices (for example, files).
- A system with multiple users may require us to first **request()** a device, to ensure exclusive use of it. After we are finished with the device, we **release()** it.These functions are similar to the open() and close() system calls for files.

- Once proper synchronization is not present,it can result in a situation called **deadlock**.
- Once the device has been requested (and allocated to us), we can **read() ,write()** , and (possibly) **reposition()** the device, just as we can with files.
- Since,the files and device system calls are very similar many Operating systems like Linux,Uniix etc use a file-device system call where I/O devices are identified by special file names, directory placement, or file attributes.
- The user interface can also make files and devices appear to be similar, even though the underlying system calls are dissimilar.

SUMMARY OF DEVICE MANAGEMENT SYSTEM CALLS

● request device, release device
● read, write, reposition
● get device attributes, set device attributes
● logically attach or detach devices

Information Maintenance

- For the purpose of **transferring information between the user program and the operating system** there are many system calls.For example, most systems have a system call to return the **current time()** and **date()** .
- Other system calls may return information about the system, such as the **number of current users,** the **version number of**

the operating system, the **amount of free memory or disk space**, and so on.

- Another set of system calls is helpful in debugging a program.**EG:** dump() system call which is useful for debugging. A program **trace lists** each system call as it is executed.
- Microprocessors provide a CPU mode known as **single step, in which a trap is executed by the CPU after every instruction.** The trap is usually caught by a debugger.
- Many operating systems provide a **time profile of a program** to indicate the amount of time that the program executes at a particular location or set of locations.
- A time profile requires either a **tracing facility or regular timer interrupts**.
- In addition, the operating system keeps i**nformation about all its processes,** and system calls are used to access this information.
- Generally, calls are also used to reset the process information **(get process attributes() and set process attributes())**.

SUMMARY OF SYSTEM CALLS UNDER INFORMATION MAINTANANCE

● get time or date, set time or date
● get system data, set system data
● get process, file, or device attributes
● set process, file, or device attributes

communication

- There are **two common models** of interprocess communication: the **message passing model** and the **shared-memory model**.
- In the **message-passing model**, the communicating processes **exchange messages** with one another to transfer information. Messages can be exchanged between the processes either directly or indirectly through a common **mailbox**.
- Before communication can take place, a connection must be **opened**. The name of the other communicator must be known.
- Each computer in a network has a **host name** by which it is commonly known. A host also has a **network identifier**, such as an **IP address**. Similarly, each process has a **process name**, and this name is translated into an identifier by which the operating system can refer to the process. The **get hostid()** and **get processid()** system calls do this translation. The identifiers are then passed to the general purpose **open() and close()** calls provided by the file system or to specific **open connection() and close connection()** system calls.
- The recipient process usually must give its permission for communication to take place with an **accept connection()** call.
- The deamon program(system programs which will be running in the background (execute a **wait for connection()** call and are awakened when a connection is made.
- The client(source) and the server(daemon) then exchange messages by using **read message()** and **write message()** system calls. The **close connection()** call terminates the communication.
- In the shared-memory model, processes use shared memory

create() and shared memory **attach()** system calls to create and gain access to regions of memory owned by other processes.

- Normally, the operating system tries to prevent one process from accessing another process's memory,but in a Shared memory model it requires that two or more processes will be sharing the same shared region in memory.
- The operating system should ensure that they are not writing to the same location simultaneously.

COMPARISON OF SHARED MODEL WITH MESSAGE PASSING

Shared Memory model	**Message Passing**
Can be used for large volume of data	Useful for small amount of data
A little difficulty in implementation	Easier to implement
Fast ,as transfer happens within memory system	Slow
Requires kernel intervention	Does not require kernel intervention

SUMMARY OF SYSTEM CALLS UNDER COMMUNICATION

● create, delete communication connection
● send, receive messages
● transfer status information
● attach or detach remote devices

PROTECTION

- Protection provides a mechanism for controlling access to the resources provided by a computer system.
- Typically, system calls providing protection include **set permission() and get permission()** , which manipulate the permission settings of resources such as files and disks. The **allow user() and deny user()** system calls specify whether particular users can—or cannot—be allowed access to certain resources.

SUMMARY ABOUT SYSTEM CALLS UNDER PROTECTION

● Set permission()
● Get permission()
● Deny permission()
● Allow user()
● Deny user()

Operating-System Structure

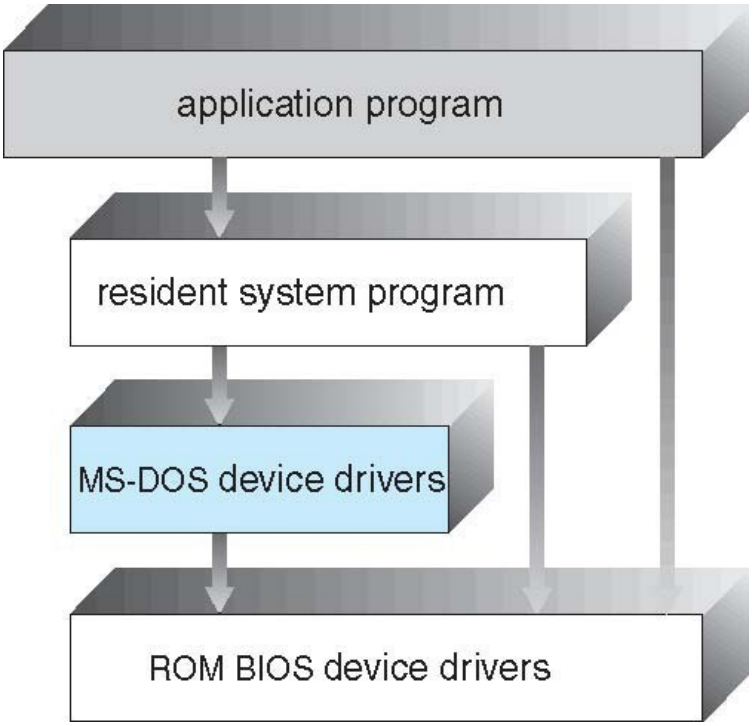
- A system as large and complex as a modern operating system must be engineered carefully if it is to function properly and be modified easily.
- In **monolithic systems**,every functionality is written in the **single module** and hence it is very **difficult to debug or to do modifications.**
- The common approach is to **partition the task into small components, or modules.**
- Each of these modules should be a well-defined portion of the system, with carefully defined inputs, outputs, and functions.

Simple Structure

- MS-DOS is an example of a system which do not have well-defined structures.It was originally designed and implemented by a few people who had no idea that it would become so popular. It was written to provide the most **functionality in the least space,** so it was not carefully divided into modules.
- **In MS-DOS , the interfaces and levels of functionality are not well separated.**
- For instance, **application programs are able to access the basic I/O routines** to write directly to

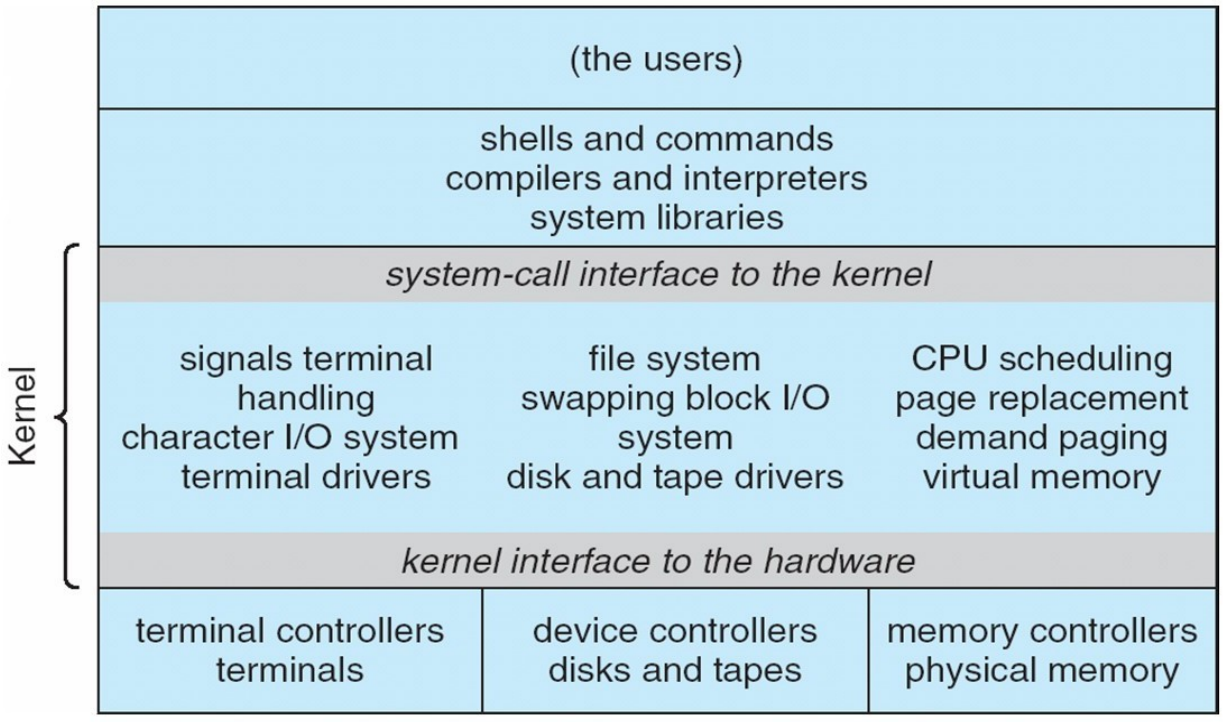
the display and disk drives. Such freedom leaves MS-DOS **vulnerable to errant (or malicious) programs**, causing entire system crashes when user programs fail.

- MS-DOS was also **limited by the hardware,No** dual mode or hardware protection.



- The original UNIX operating system also had a limited structure -**but maintained two separable parts: the kernel and the system programs.**

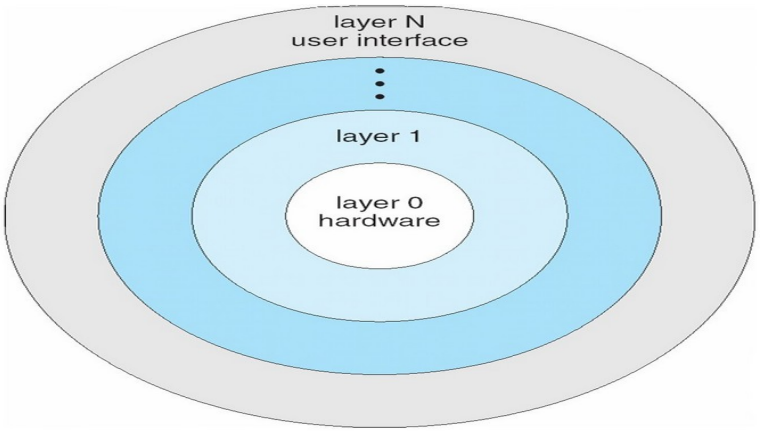
- Everything below the system-call interface and above the physical hardware is the kernel. The kernel provides the file system, CPU scheduling, memory management, and other operating-system functions through system calls. Taken in sum, that is an enormous amount of functionality to be combined into one level. This monolithic structure was **difficult to implement and maintain.** It had a distinct performance advantage, **however: there is very little overhead in the system call interface or in communication within the kernel.**



Traditional Unix Structure

Layered Approach

- With proper hardware support, operating systems can be broken into pieces that are smaller and more appropriate .
- The operating system can then **retain much greater control** over the computer and over the applications that make use of that computer.
- **Implementers have more freedom in changing the inner workings of the system and in creating modular operating systems by providing data abstraction or information hiding.**
- In a layered system , the operating system is broken into a **number of layers (levels)**.
- The **bottom layer (layer 0) is the hardware**;
- **The highest (layer N) is the user interface.**



- A typical operating-system layer—say, **layer M -consists of data structures and a set of routines that can be invoked by higher-level layers.**
- Layer M, in turn, can invoke operations on lower-level layers.

ADVANTAGES :

- **simplicity of construction.** The layers are selected so that each uses functions (operations) and services of only lower-level layers.
- **Simplifies debugging and system verification.** Each layer can be debugged without affecting the other layers. Thus, the design and implementation of the system are simplified.
- Each layer is implemented only with **operations provided by lower-level layers**.
- A layer **does not need to know how these operations are implemented;** it needs to know only what these operations do. Hence, **each layer hides the existence of certain data structures, operations, and hardware from higher-level layers.**

CHALLENGES/DRAWBACKS

- Appropriately defining the various layers are difficult.
- **Careful planning** is necessary for the design Eg:For example, the **device driver for the backing store**

(disk space used by virtual-memory algorithms) must be at **a lower level than the memory-management** routines, because memory management requires the ability to use the backing store.

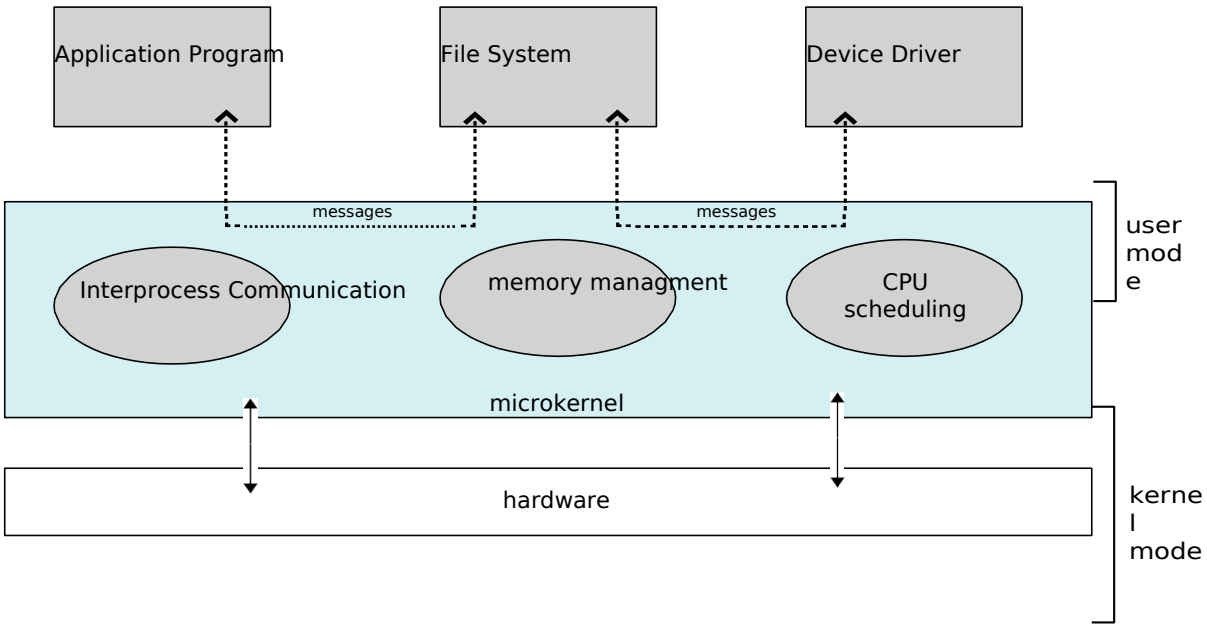
- tend to be **less efficient** than other types.For instance, when a user program executes an I/O operation, it executes a system call that is trapped to the I/O layer, which calls the memory-management layer, which in turn calls the CPU -scheduling layer, which is then passed to the hardware. At each layer, the parameters may be modified, data may need to be passed, and so on. **Each layer adds overhead to the system call. The net result is a system call that takes longer than does one on a nonlayered system.**
- The **backing-store driver** would normally be **above the CPU scheduler**, because the driver may need to wait for I/O and the CPU can be rescheduled during this time. **However, on a large system, the CPU scheduler may have more information about all the active processes than can fit in memory. Therefore, this information may need to be swapped in and out of memory, requiring the backing-store driver routine to be below the CPU scheduler.**

SOLUTION ADOPTED IN MODERN OS

Fewer layers with more functionality are being designed, providing most of the advantages of modularized code while avoiding the problems of layer definition and interaction.

Microkernels

- Researchers at Carnegie Mellon University developed an operating system called **Mach** that **modularized the kernel using the microkernel approach**.
- In this approach,the operating system is structured **by removing all nonessential components from the kernel and implementing them as system and user-level programs. The result is a smaller kernel.**
- Microkernels **provide minimal process and memory management, in addition to a communication facility.**



- The main function of the microkernel **is to provide communication between the client program and the various services that are also running in user space.**
- Communication is provided through **message passing.**

ADVANTAGES

- **Extensions of the operating system is easier** . All new services are added to user space and consequently do not require modification of the kernel.
- When the kernel have to be modified, the changes tend to be fewer, because the microkernel is a **smaller kernel and it is easier to compile the kernel.**
- The resulting operating system is **easier to port from one hardware design to another**.
- The microkernel also provides **more security** and **reliability**, since most services are running as user rather than kernel processes. **If a service fails, the rest of the operating system remains untouched.**
- Examples of operating systems which uses microkernel approach :
    - **Tru64 UNIX** (formerly Digital UNIX) provides a

UNIX interface to the user, but it is implemented with a Mach kernel.

- The **Mac OS X kernel (also known as Darwin)** is also partly based on the Mach microkernel.
- **QNX , a real-time operating system for embedded systems.**

DISADVANTAGES

- **The performance of microkernels can suffer due to increased system-function overhead.**
  - **CASE:**Consider the history of Windows NT . The first release had a layered microkernel organization. This version's performance was low compared with that of Windows 95.

Modular Approach

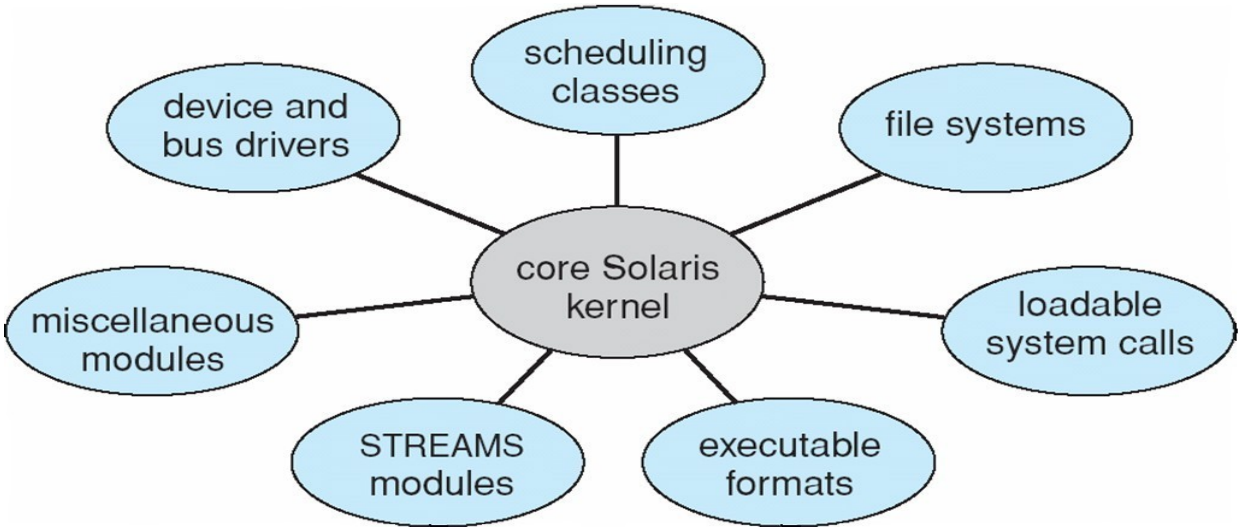
- This is the best current methodology adopted in various operating-system design. This type of design is common in modern implementations of UNIX , such as Solaris, Linux, and Mac OS X , as well as Windows.
- It involves designing kernel modules as **loadable kernel modules**. Here, the kernel has a set of core components and links in additional services

via modules, either at boot time or during run time.

- The **idea of the design is for the kernel to provide core services** while **other services are implemented dynamically**, as the kernel is running.

ADVANTAGES

- Addition of new features to kernel is very easy
- The overall result resembles a layered system in that each kernel section has defined, **protected** interfaces; but it is **more flexible than a layered system,** because any module can call any other module.
- The approach is also similar to the microkernel approach in that **the primary module has only core functions and knowledge of how to load and communicate with other modules.**
- It is **more efficient,** because modules do not need to invoke message passing in order to communicate.
- The **Solaris operating system structure,is organized around a core kernel** with **seven types of loadable kernel modules:**



1. Scheduling classes
2. File systems
3. Loadable system calls
4. Executable formats
5. STREAMS modules
6. Miscellaneous
7. Device and bus drivers

- **Linux also uses loadable kernel modules,** primarily for supporting device drivers and file systems.

Hybrid Systems

- Modern Operating systems combine different structures, resulting in **hybrid** systems that address **performance, security, and usability issues**.

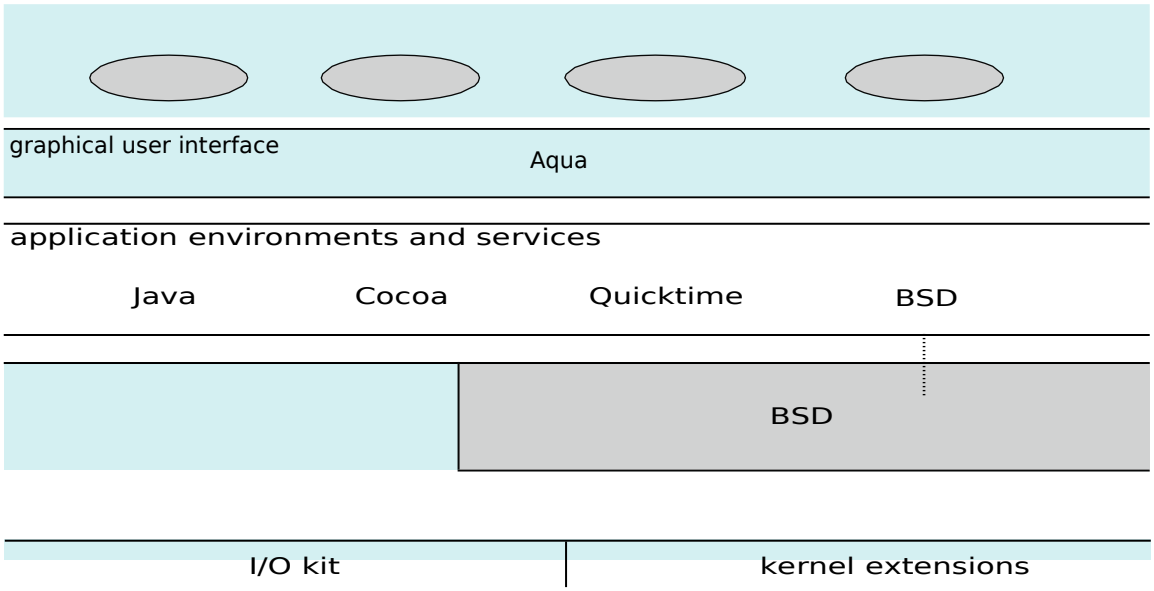
- For example, both Linux and Solaris are monolithic, because having the operating system in a single address space provides very efficient performance. However, they are also modular, so that new functionality can be dynamically added to the kernel.

- **Windows** is largely **monolithic** as well (again primarily for performance reasons), but it retains some behavior typical of **microkernel** systems, including providing support for separate subsystems (known as operating-system personalities) that run as user-mode processes. Windows systems also provide support for **dynamically loadable kernel modules**.

- **Three powerful hybrid systems:** the **Apple Mac OS X** operating system and the two most prominent mobile operating systems—**i OS and Android.**

Mac OS X

- The **Apple Mac OS X operating system uses a hybrid structure.**


graphical user interface
Aqua
application environments and services
Java
Cocoa
Quicktime
BSD
BSD
I/O kit
kernel extensions

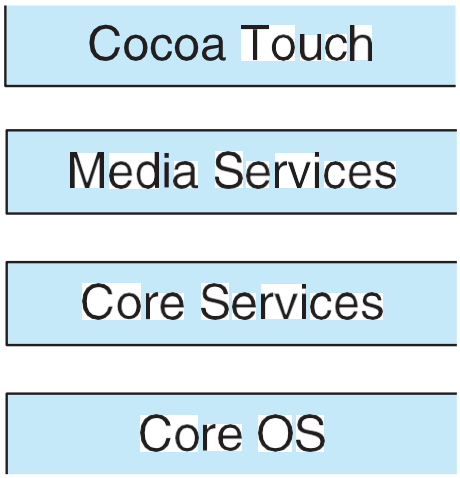
- It is a **layered system**.
- The **top layers** include the **Aqua user interface** and a **set of application environments and services**.
- The **Cocoa environment** specifies an **API for the Objective-C programming language**, which is used for writing Mac OS X applications.
- Below these layers is the **kernel environment, which consists primarily of the Mach microkernel and the BSD UNIX kernel.**
- **Mach** provides **memory management**;support for **remote procedure calls (RPC s)** and **interprocess communication**(IPC) facilities, including m**essage passing; and thread scheduling.**
- The **BSD component** provides a **BSD command-line interface, support for networking and file systems, and an implementation of POSIX API s, including Pthreads.**
- In addition to Mach and BSD , the kernel environment provides an **I/O kit** for development of device drivers and **dynamically loadable modules** (which Mac OS X refers to as kernel extensions).

SUMMARY OF MAC Osx

Aqua	**GUI**
Cocoa	**API For Objective C Programming Language**
Mach	**Memory management,RPC,IPC,message passing,thread scheduling**
BSD	**CLI,Support for network and file systems,POSIX APIs,**
I/O Kit	**Device Drivers**

iOS

- iOS is a **mobile operating system designed by Apple** to run its smartphone, the iPhone, as well as its tablet computer, the iPad.
- iOS is **structured on the Mac OS X operating system, with added functionality pertinent to mobile devices,**
- It does not directly run Mac OS X applications. The structure of i OS is as shown below



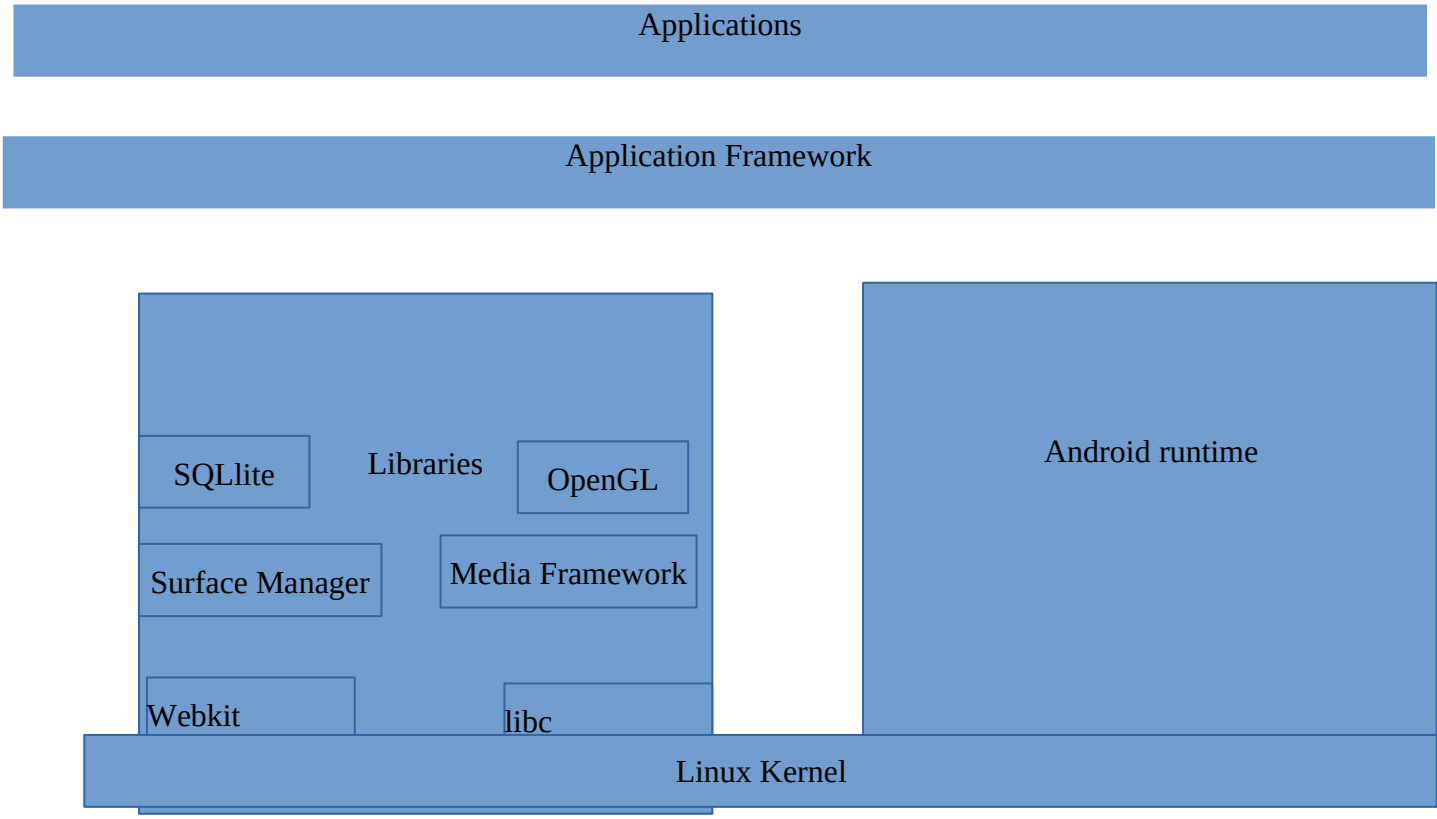
- **Cocoa Touch** is an **API for Objective-C** that provides several frameworks for developing applications that run on i OS devices.
- The fundamental difference between Cocoa, mentioned earlier, and **Cocoa Touch** is that the latter **provides support for hardware features unique to mobile devices, such as touch screens.**
- The **media services layer** provides services for **graphics, audio, and video.**

Android

- The **Android operating system** was designed by the **Open Handset Alliance** (**led primarily by Google**)

and was developed for Android smartphones and tablet computers.

- Android **runs on a variety of mobile platforms** and is **open-sourced**, partly explaining its rapid rise in popularity.
- Android is similar to iOS in that it is a **layered stack of software** that provides a **rich set of frameworks for developing mobile applications** .



- In the **bottom of this software stack is the Linux kernel,** although it has been modified by Google and is currently outside the normal distribution of Linux releases.

NOTE:

- iOS is designed to run on Apple mobile devices and is close-sourced, Android runs on a variety of mobile platforms and is open-sourced,

END OF UNIT -1

MODULE 2

PROCESS

What it is ???

•Program in execution state

How it executes …

- Using resources CPU, Memory, I/O devices

When Resources are allocated…

-Process creation time, Execution time

- System has many processes
- How process executes

- Sequentially or Parallely

- “””Thread””””” concept

- How many Thread a process can have??
- OS responsible for ‘””P ‘’R management….
- Process scheduling..
- Communication…

PROCESS…

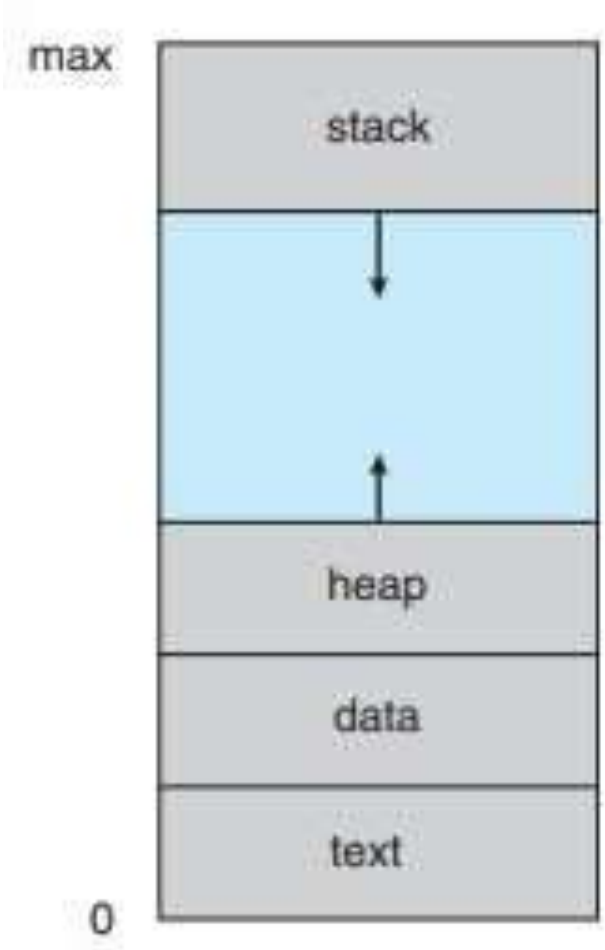
- How many Thread a process can have??

- Single threaded , Multithreaded

-OS responsible for ‘’’’P ‘’R management….

-Process scheduling..

-Communication…



PROCESS IN MEMORY..

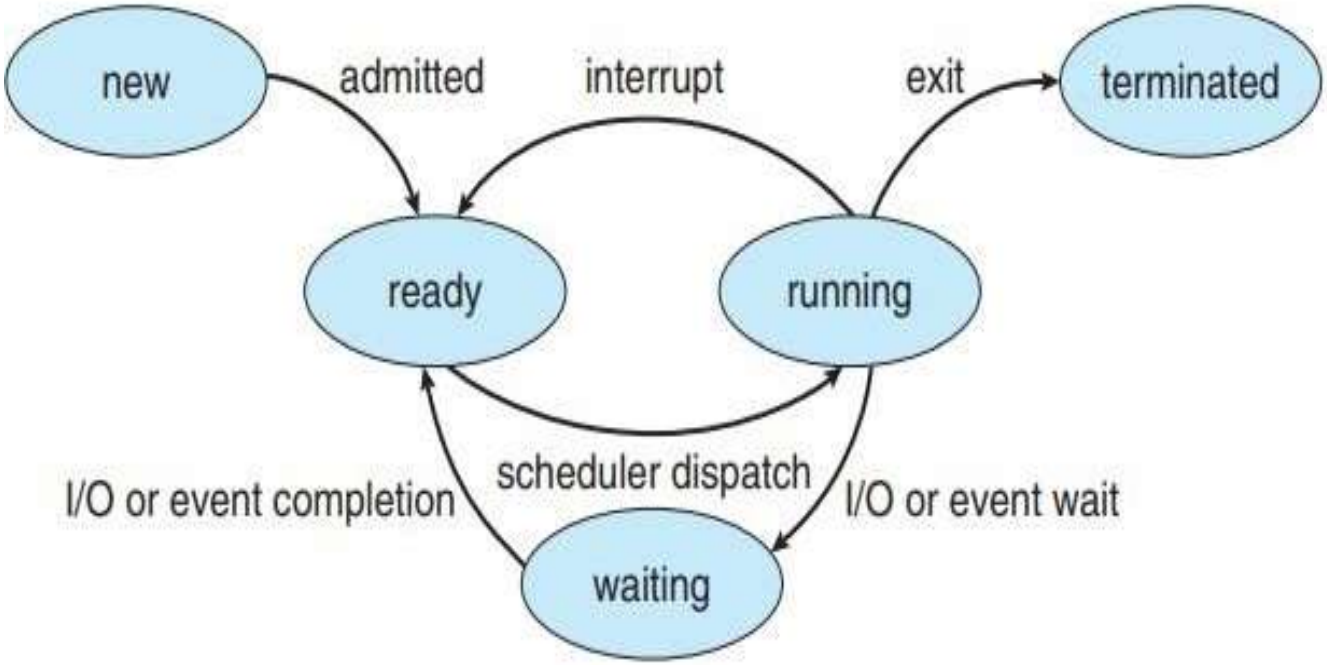
- **Process State**

As a process executes, it changes state. The state of a process is defined in part by the current activity of that process.

A process may be in one of the following states:

- NEW
- RUNNING
- WAITING
- READY
- TERMINATED

- **Only one P is running in Processor/CPU**



Process states

Process Control Block (PCB)

- Each process is represented in the operating system by a process control block (PCB)
- TCB
- It contains many pieces of information associated with a specific process

1. Process state.

The state may be new, ready, running, waiting, halted, and so on.

2. Program counter.

3. CPU registers.

4. CPU-scheduling information.

-Process priority

5. Memory-management information.

- page table

- segment table

- register values

6. Accounting information.

-amount of CPU and real time used,
-time limits, account numbers
- job or process numbers

7. I/O status information.

-list of I/O devices allocated to the process
-list of open files

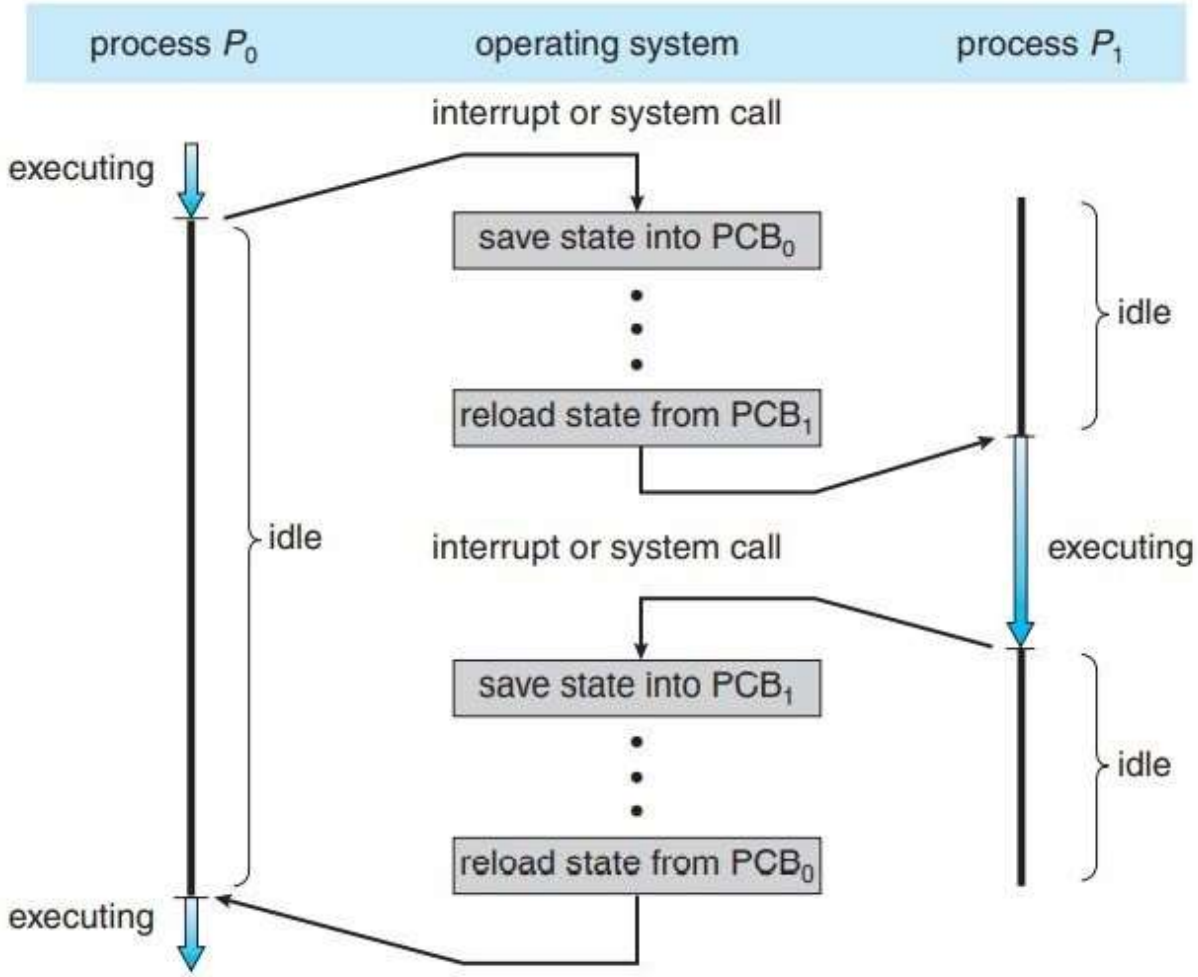


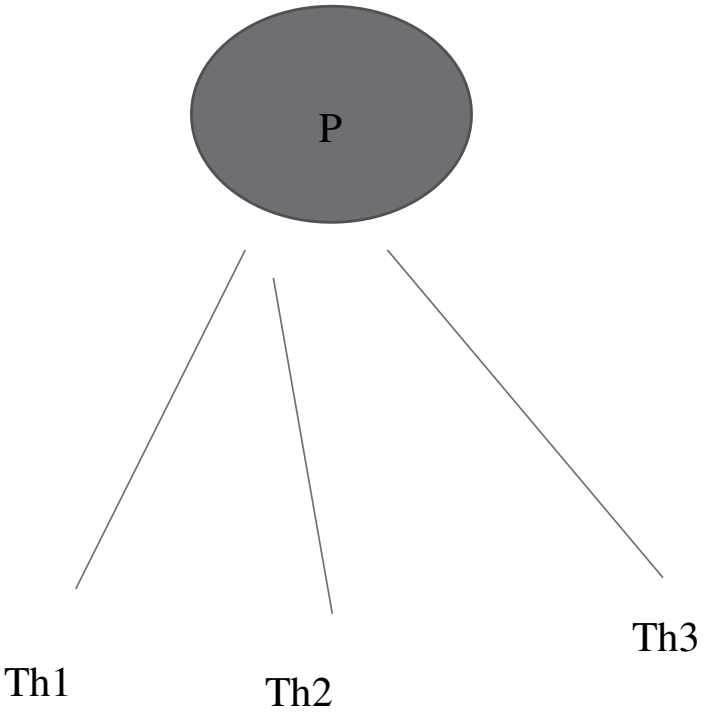
Figure 3.4 Diagram showing CPU switch from process to process.

Thread

-Single thread of execution

Eg: Process running MS word application

- **Allow only one Task at a time**



PCB..

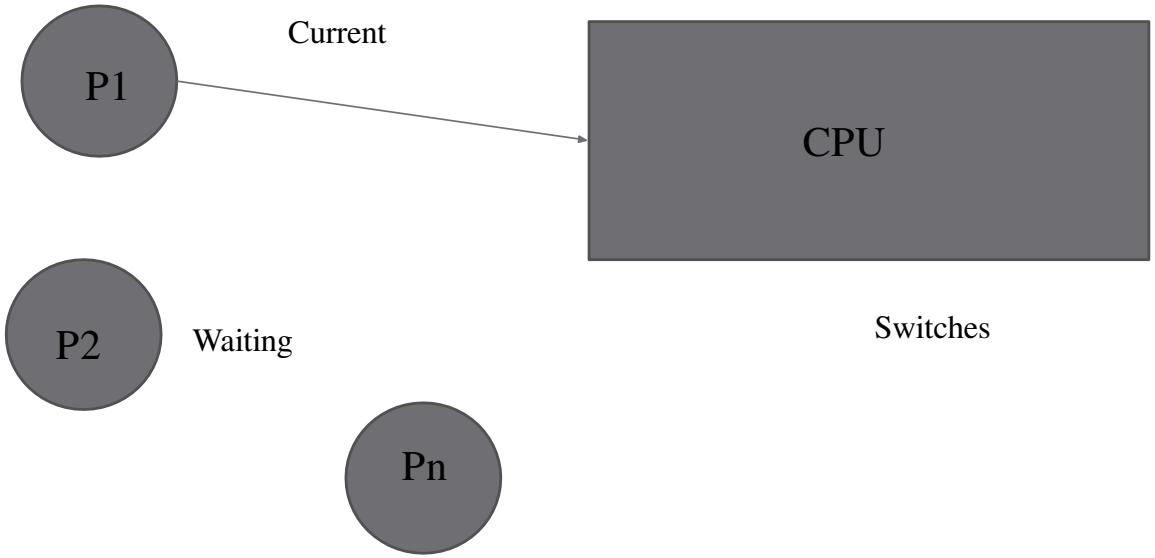
PROCESS SCHEDULING

- Multiprogramming

- to maximize CPU utilization.

-Time sharing is to switch the CPU among processes

- Process Scheduler



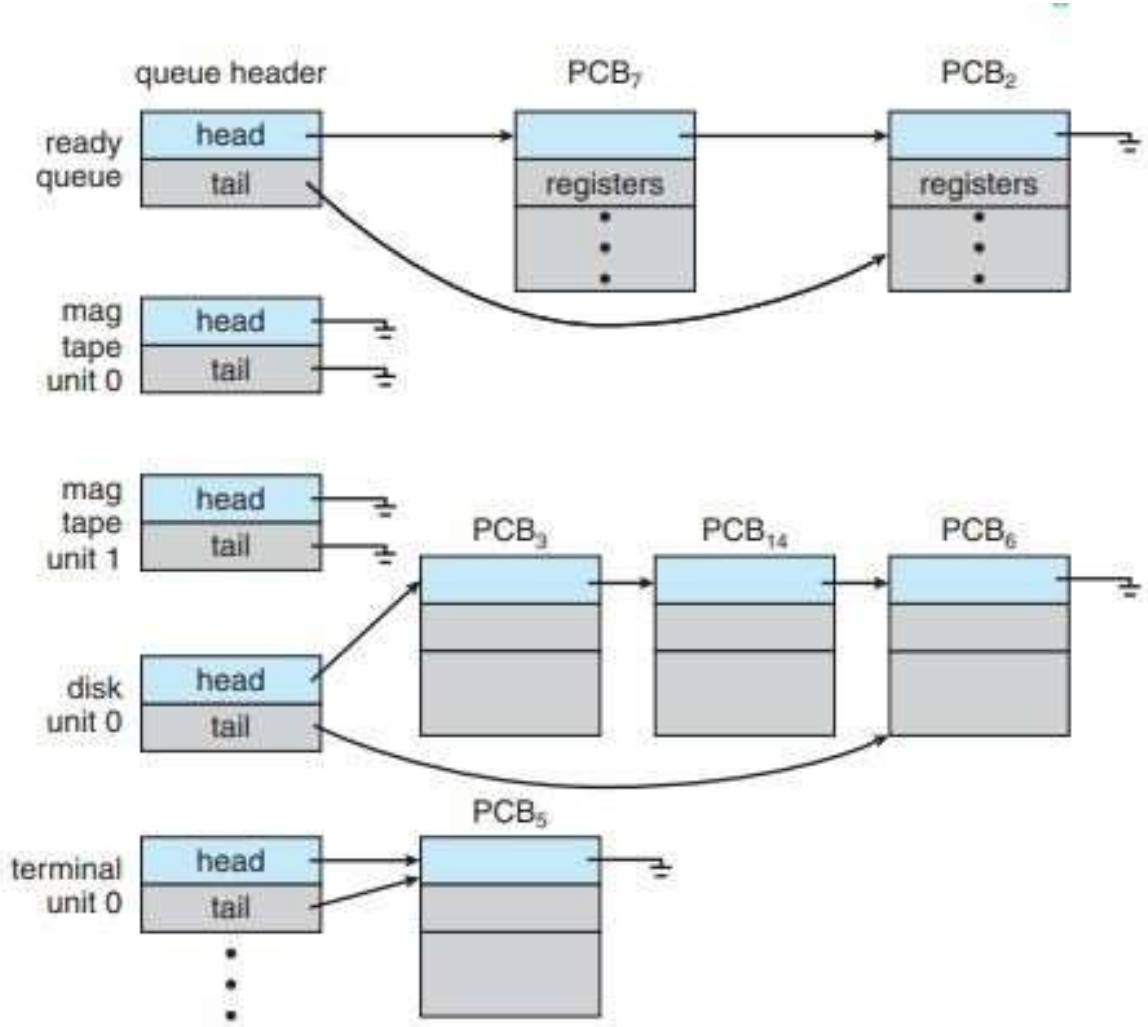


Figure 3.5 The ready queue and various I/O device queues.

Scheduling queues:

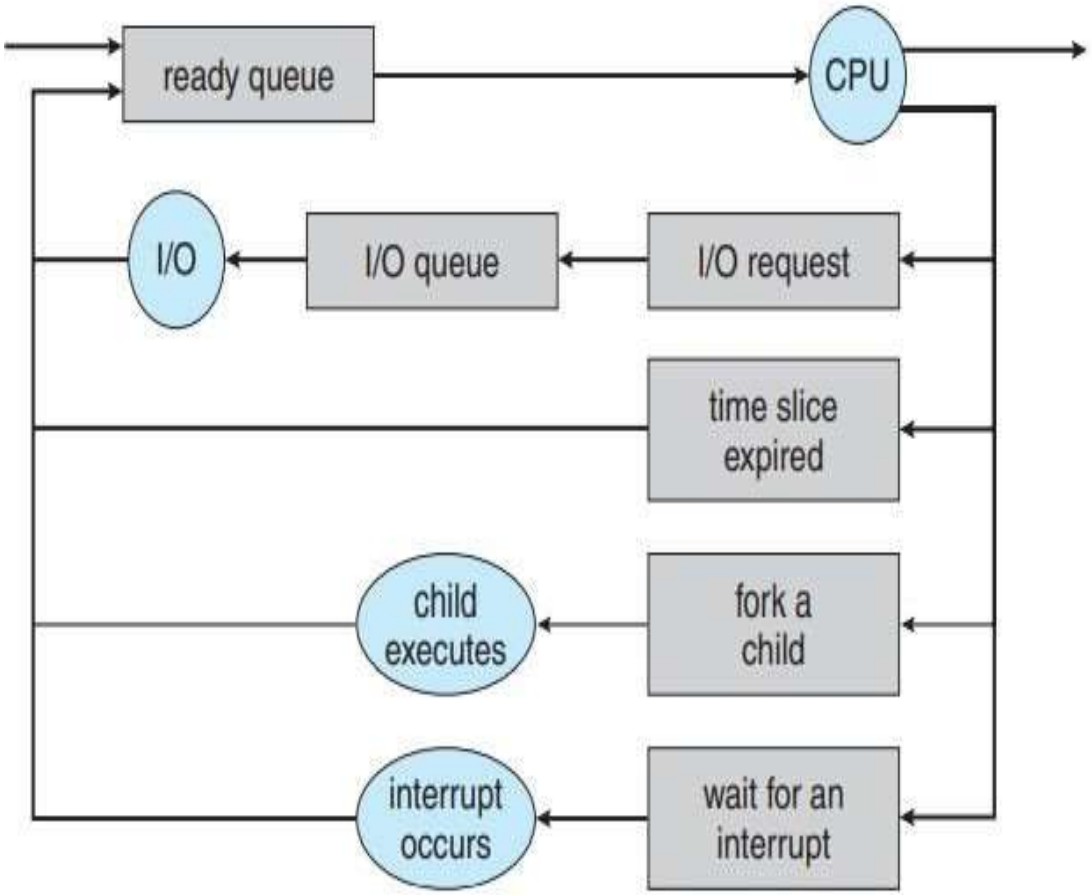
Job queue, Ready queue

PCB pointers

I/O request

Device queue

3 events occurs



QUEUE diagram for Process Scheduling

- A common representation of process scheduling is a queueing diagram
- Each rectangular box represents a queue. 
- Two types of queues are present: the ready queue and a set of device queues.
- The circles represent the resources that serve the queues, 
- arrows indicate the flow of processes in the system.
- new process is initially put in the ready queue. It waits there until it is selected for execution, or dispatched.
- Once the process is allocated the CPU and is executing, one of several events could occur:
- .

•1. The process could issue an I/O request and then be placed in an I/O queue.

• 2.The process could create a new child process and wait for the child's termination.

•3. The process could be removed forcibly from the CPU, as a result of an interrupt, and be put back in the ready queue.

In the first two cases, the process eventually switches from the waiting state to the ready state and is then put back in the ready queue.

-**A process continues this cycle until it terminates, at which time it is removed from all queues and has its PCB and resources deallocated**

Schedulers

-Short term Scheduler (CPU scheduler)

selects from among the processes that are ready to execute and allocates the CPU to one of them.

More frequent

-Long tern scheduler (Job scheduler)

selects processes from this pool and loads them into memory for execution.

Less frequent

Control Multiprogramming

- I/O bound process
- CPU bound process

medium-term scheduler (SWAPPING)

The key idea behind a medium-term scheduler is that sometimes it can be advantageous to remove a process from memory (and from active contention for the CPU) and thus reduce the degree of multiprogramming.

Later, the process can be reintroduced into memory, and its execution can be continued where it left off.

This scheme is called swapping.

The process is swapped out, and is later swapped in, by the medium-term scheduler.

Swapping may be necessary to improve the process mix or because a change in memory requirements has overcommitted available memory, requiring memory to be freed up.

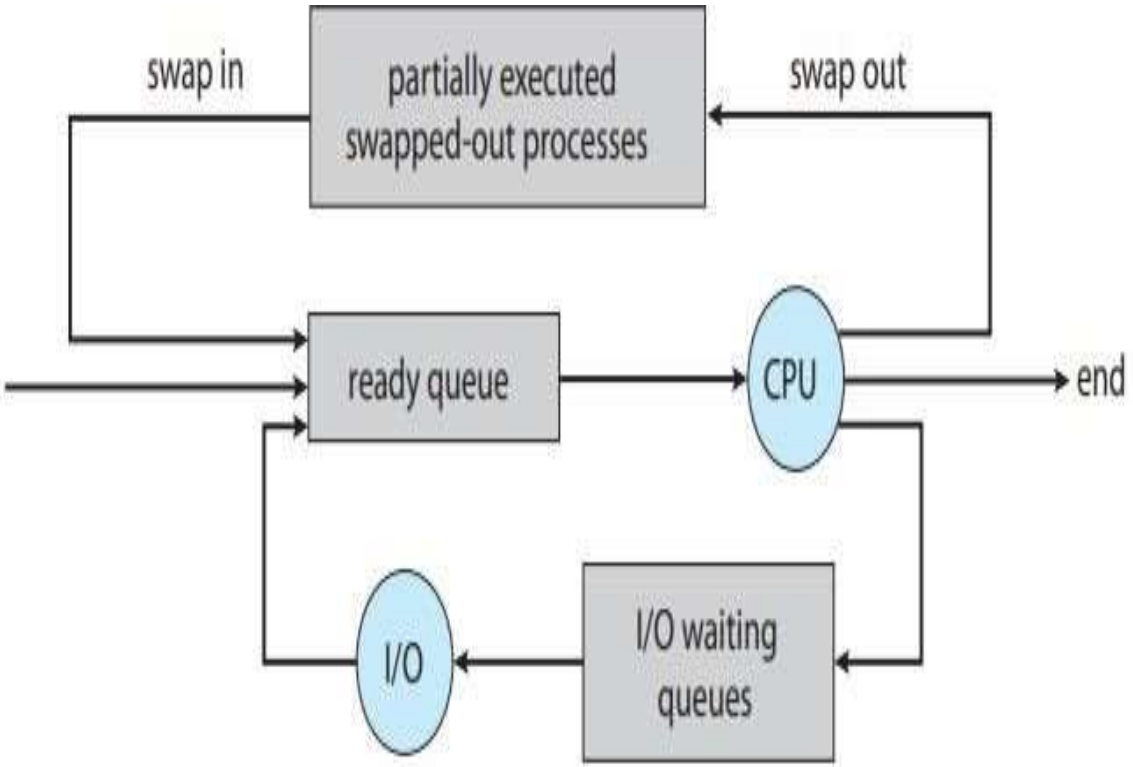


Figure 3.7 Addition of medium-term scheduling to the queueing diagram.

CONTEXT SWITCH

- Switching the CPU
- state save
- the kernel saves the context of the old process
- rescheduled to run.

- Context-switch time is pure overhead
- Switching speed varies from machine to machine, depending on the memory speed, the number of registers
- A typical speed is a few milliseconds.

Operation on **Process**:

- Creation
- Deletion
- execution
- Kill ----Normal------Abnormal
- Pid
- Init : The init process (which always has a pid of 1) serves as the root parent process for all user processes.

“we can obtain a listing of processes by using the ps command. For example, the command

ps –el

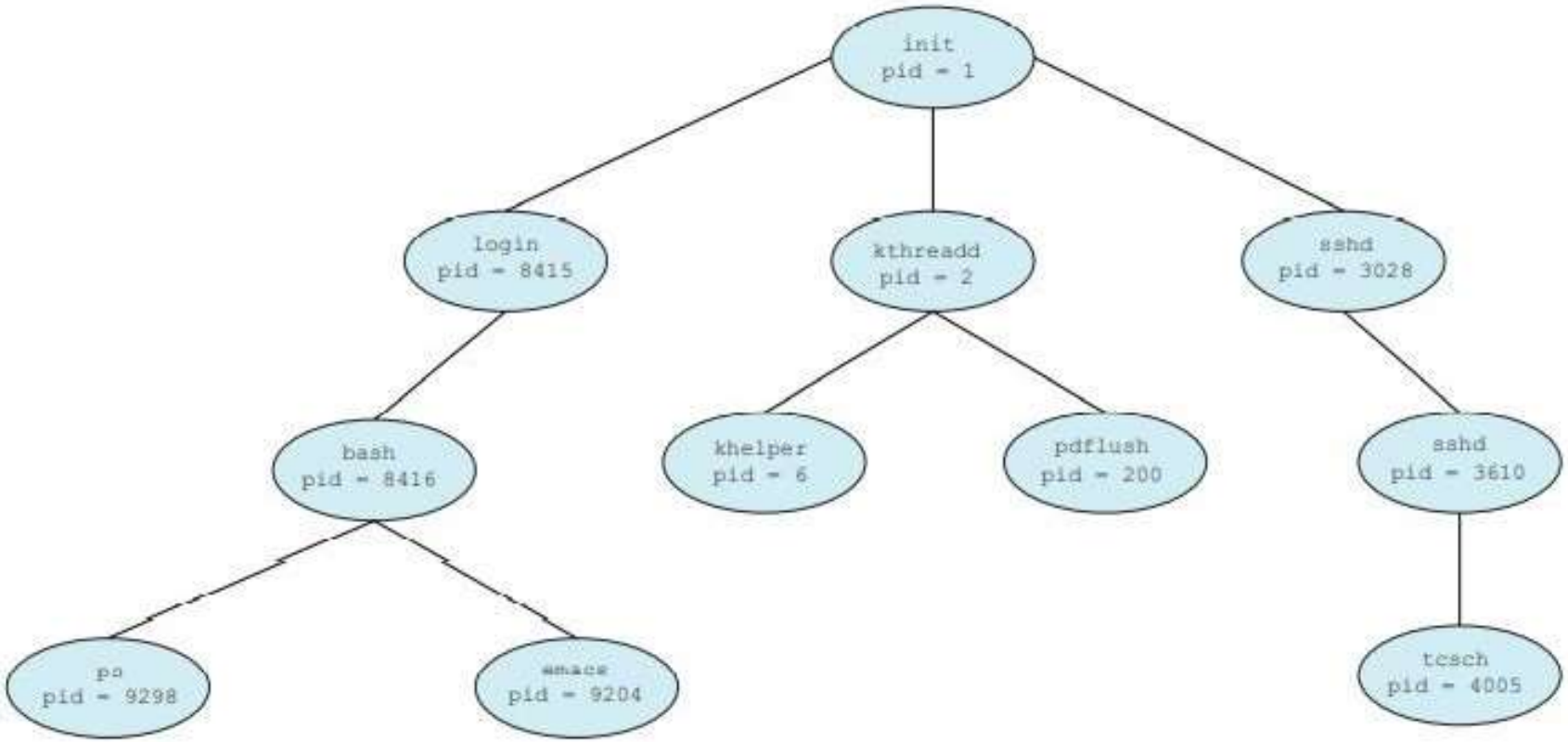


Figure 3.8 A tree of processes on a typical Linux system.

When a process creates a new process, two possibilities for execution exist:

1. **Pp parallel with Cp**

2. Pp waits for Cp

There are also two address-space possibilities for the new process:

1. **Cp is copy of Pp**

2. Cp has program loaded into it

~~Directory~~---~~folder1~~ ---- ~~open~~ ~~close~~ ~~append~~

folder2 --- rename exit

Fork() exec() wait()

CreateProcess(); // WINDOWS

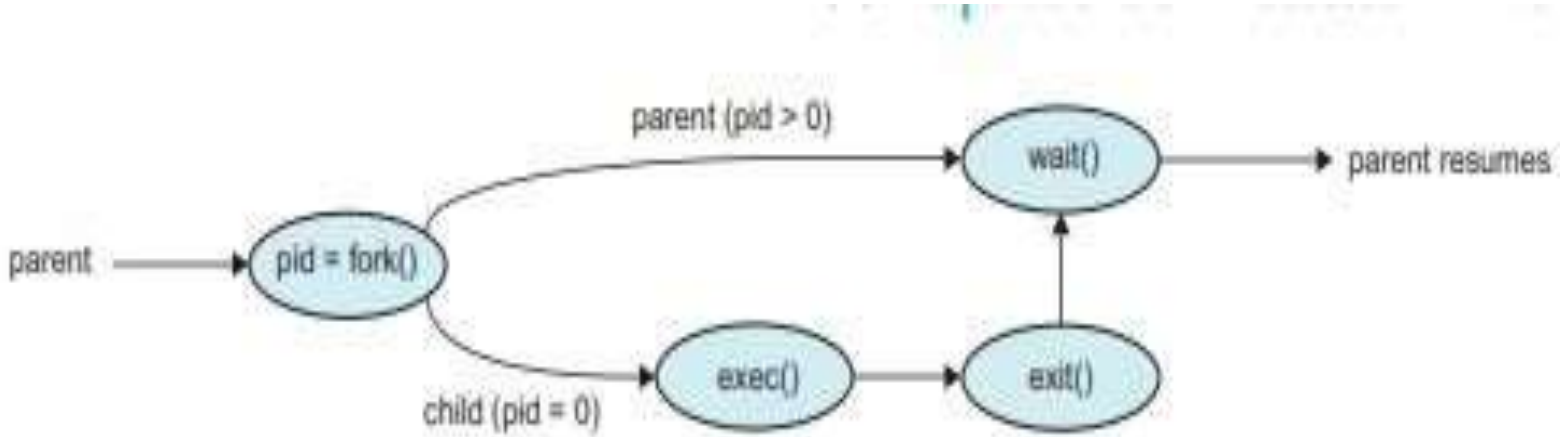


Figure 3.10 Process creation using the `fork()` system call.

Process termination

A parent may terminate the execution of one of its children for a variety of reasons, such as these:

•The child has exceeded its usage of some of the resources that it has been allocated.

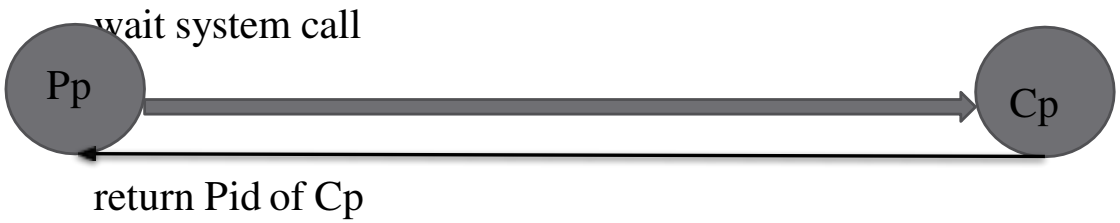
• task assigned to the child is no longer required.

• Once Pp exits Cp will terminates

-"Cascading Termination"

TerminateProcess() in Windows

exit(): system call by process to OS to terminate


wait system call
Pp
Cp
return Pid of Cp

- The wait() system call is passed a parameter
- This system call also returns the process identifier of the terminated child so that the parent can tell which of its children has terminated:
- status of Cp can be obtained

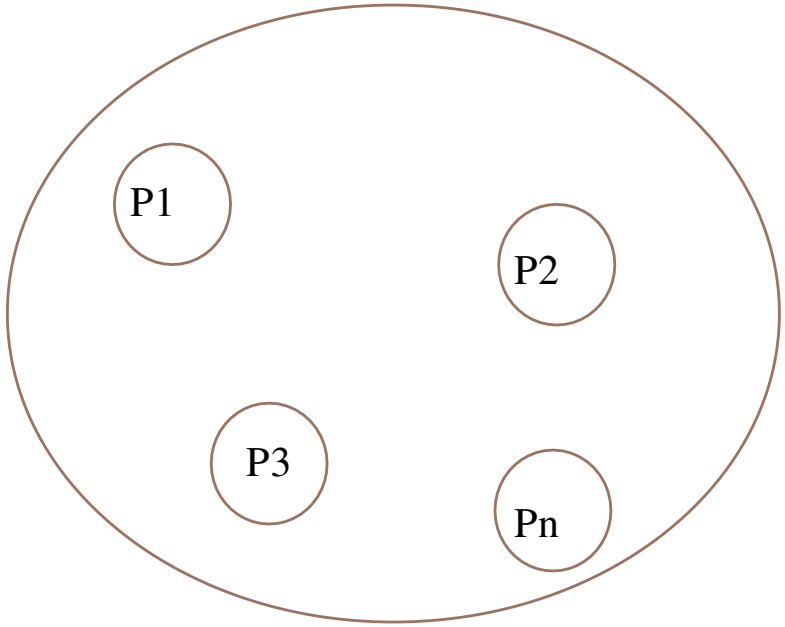
init() in Linux

pid is 1
create various user P
Eg: web server, print server , ssh server

child process of init is

- kthread // kernel thread
- sshd // secure shell

INTER PROCESS COMMUNICATION (IPC)



- OS supports all Process in parallel execution
- Independent Process
- Cooperative Process

Why IPC ????

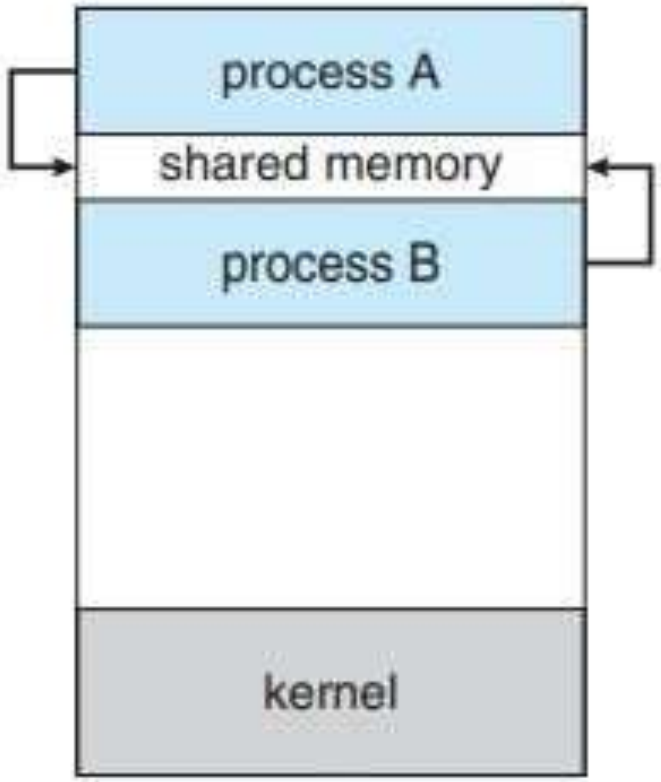
-Information Sharing

-Computation Speed up

-Modularity

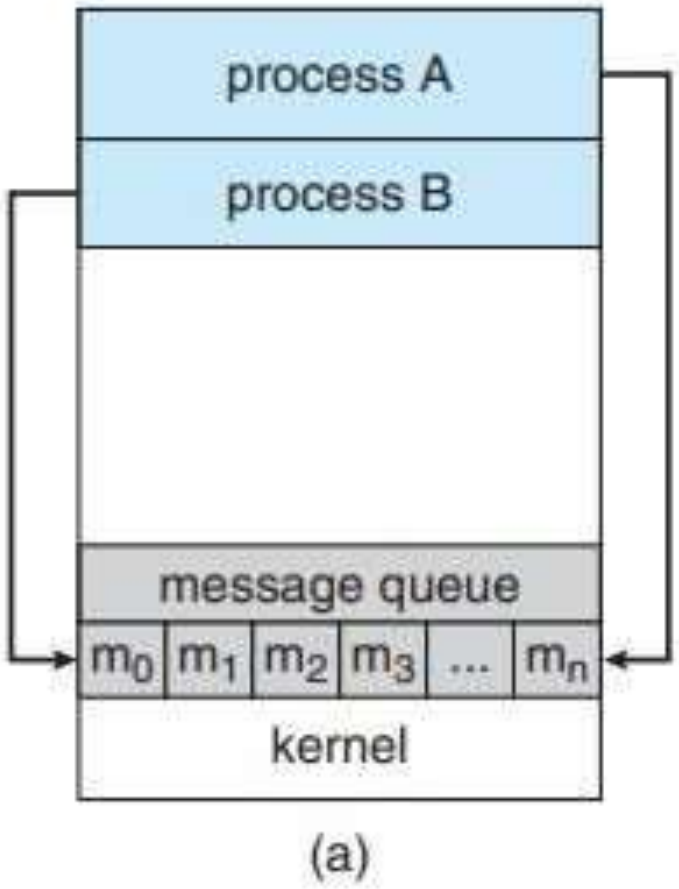
- Convenience

SHARED MEMORY MODEL

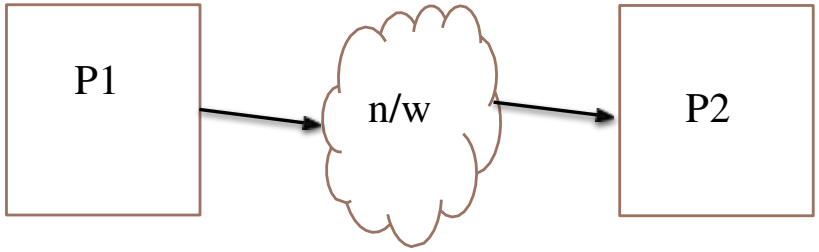


(b)

Message pass model



(a)



S1

S2

send(msg)

receive(msg)

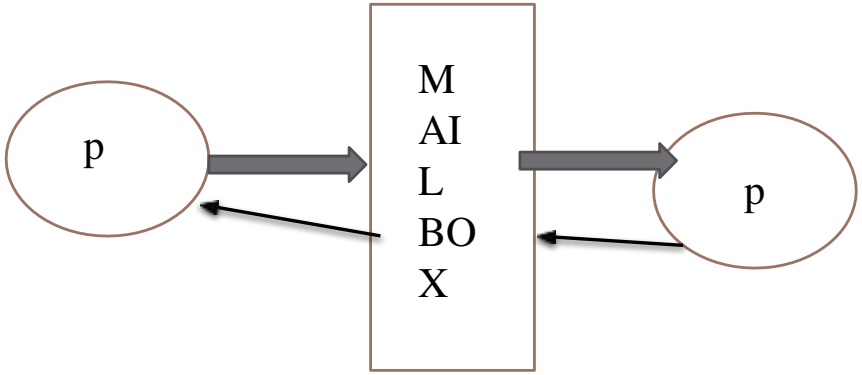
- Direct or indirect communication
- Synchronous or asynchronous communication
- Automatic or explicit buffering

  - `send(P, message)`—Send a `message` to process `P`.
  - `receive(Q, message)`—Receive a `message` from process `Q`.

  - `send(P, message)`—Send a `message` to process `P`.
  - `receive(id, message)`—Receive a `message` from any process.

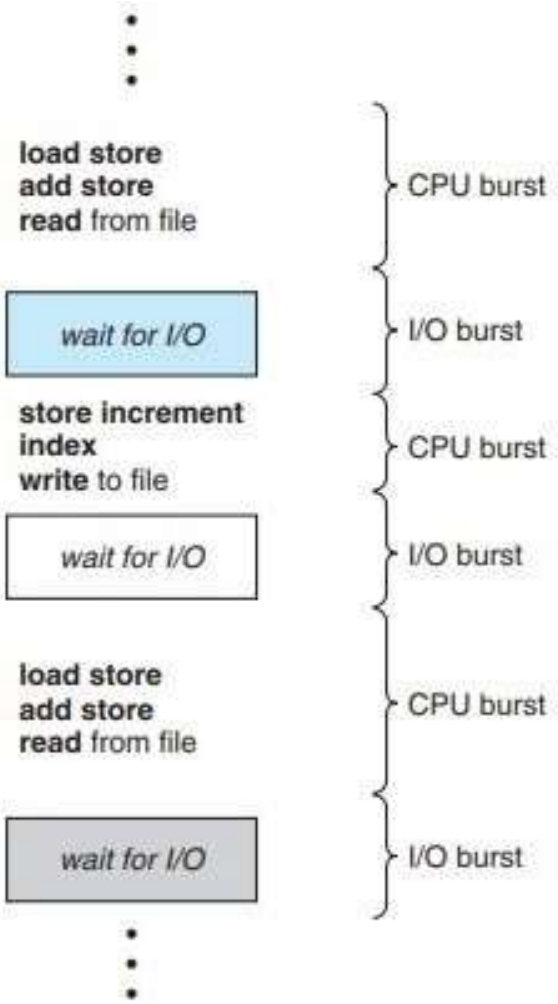
indirect mode

- use mail box
- it is data structure

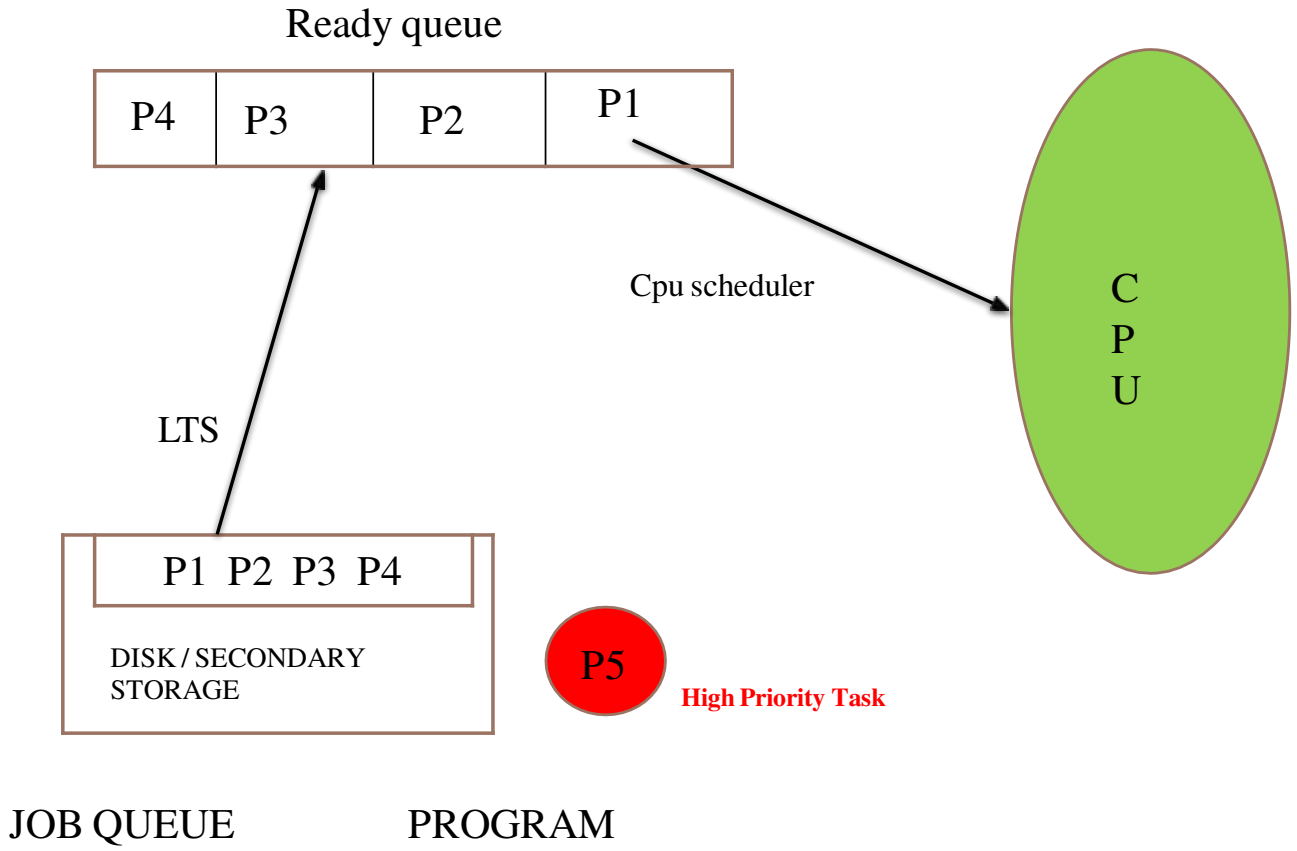


- BUFFERS
- ZERO CAPACITY
- BOUNDED CAPACITY
- UN BOUNDED CAPACITY

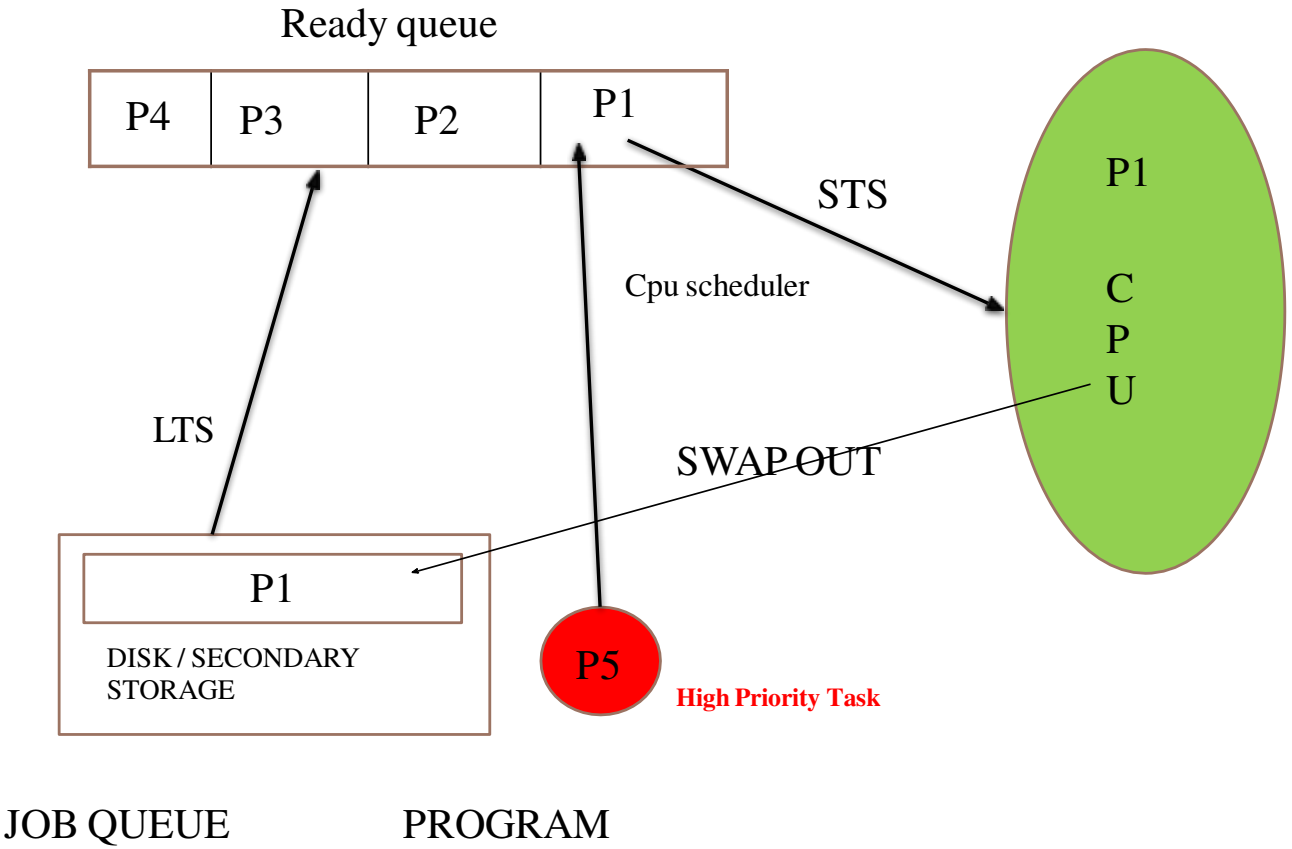
CPU SCHEDULING::



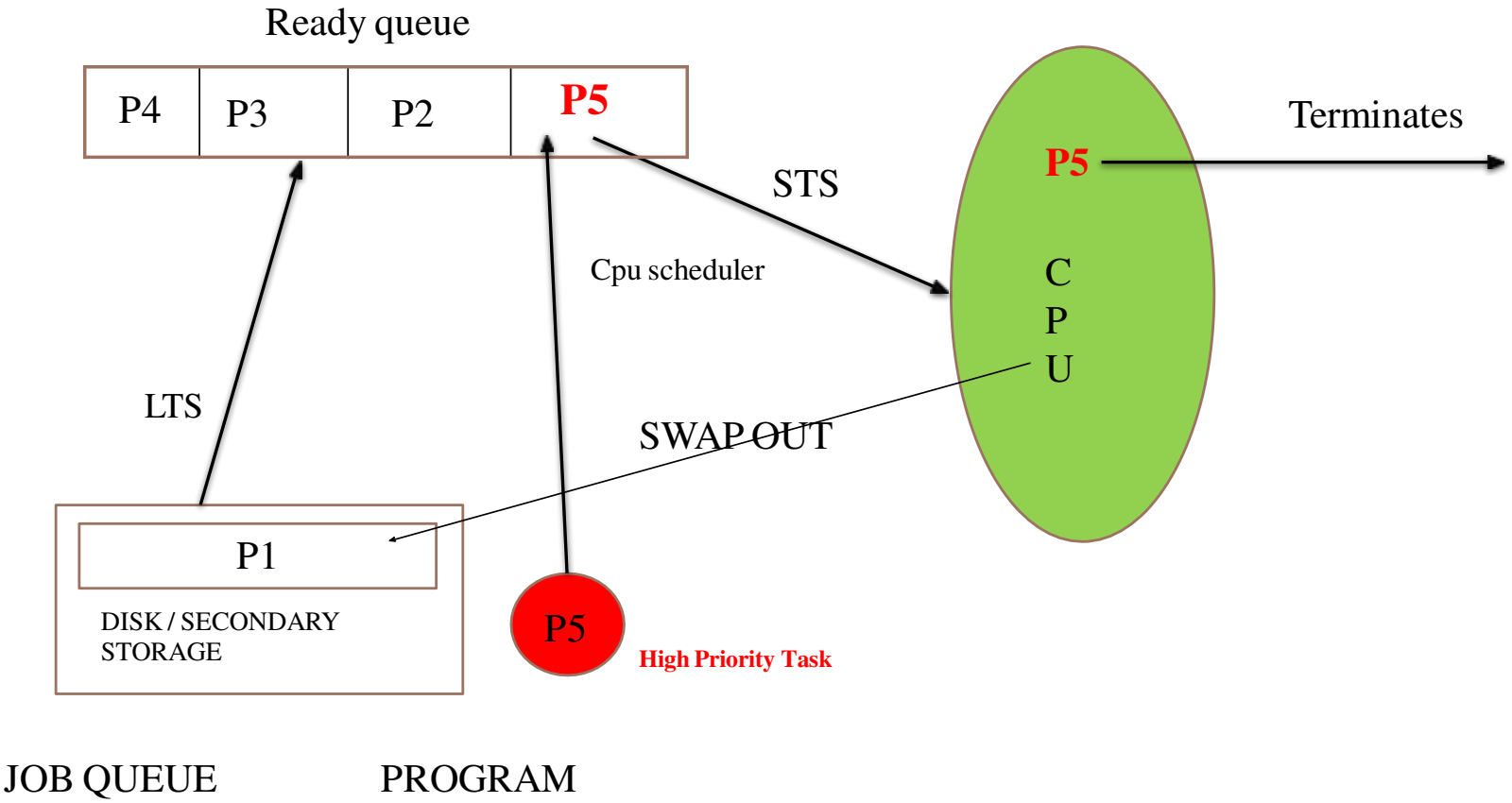
Cpu scheduler



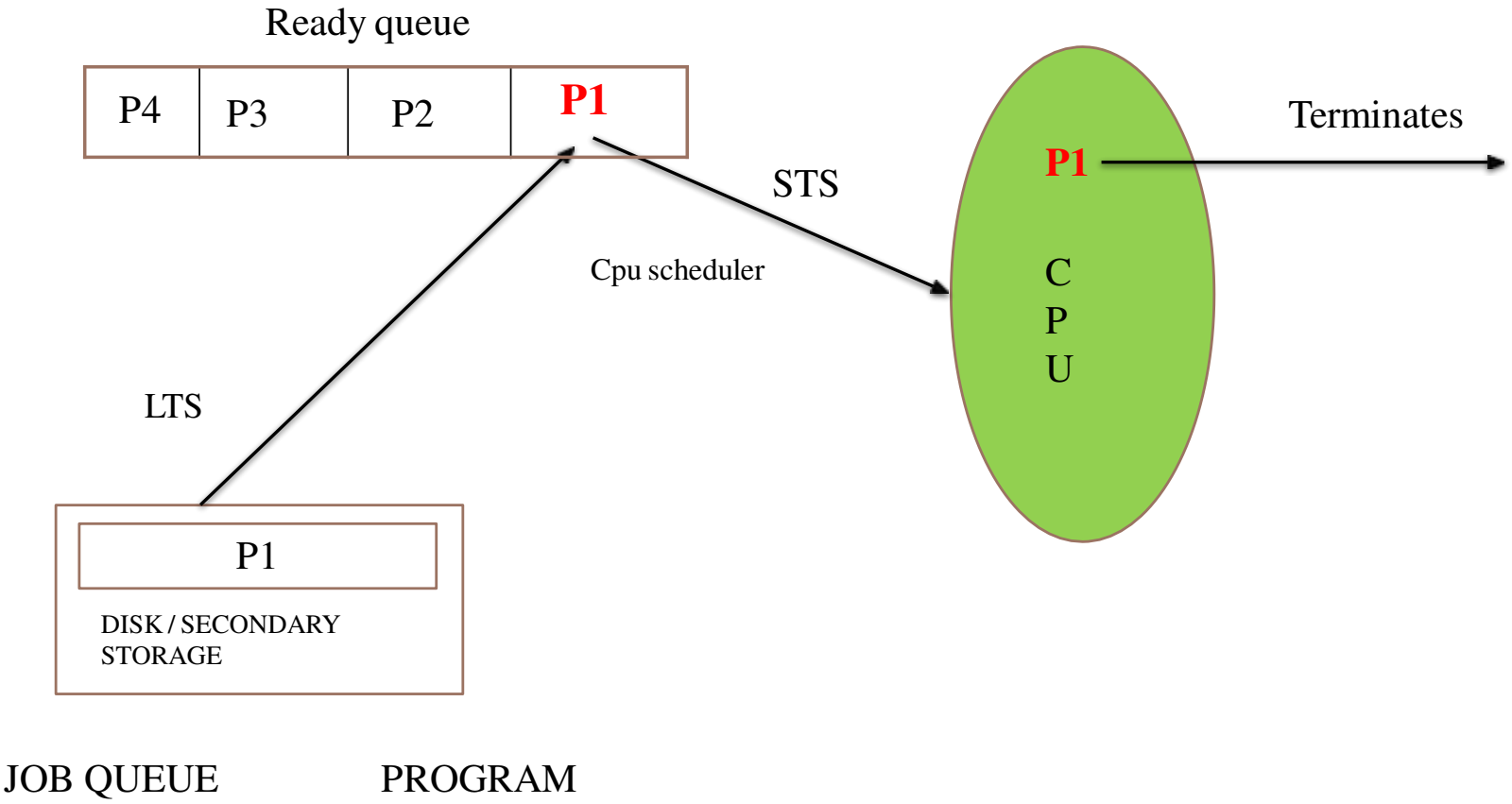
Cpu scheduler



Cpu scheduler



Cpu scheduler

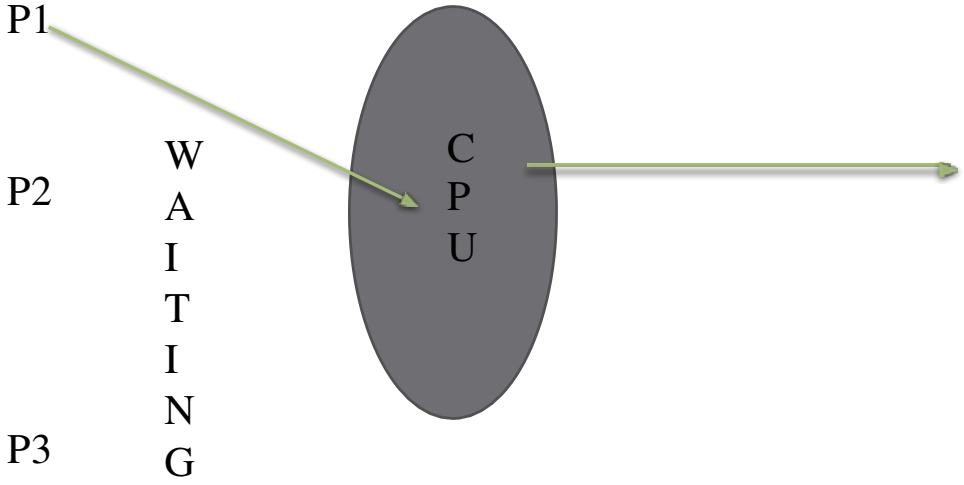


SCHEDULING CRITERIA

1. CPU UTILIZATION
2. THROUGHPUT
3. TURN AROUND TIME
4. WAITING TIME
5. RESPONSE TIME

PREEMPTIVE & NON – PREEMPTIVE SCHEDULING

NON-PREEMPTIVE



ONCE ALLOCATE CPU
CANNOT TAKE BACK

ALL REMAINING PROCESS
HAS TO WAIT

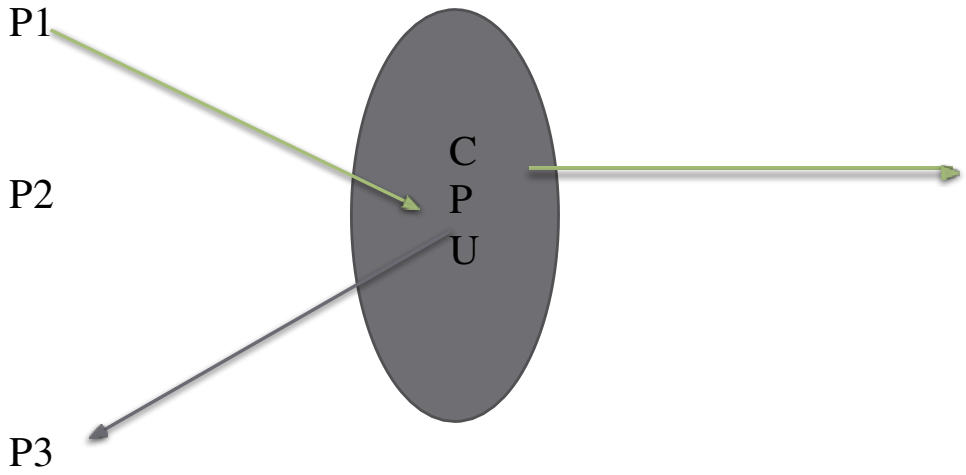
ONCE COMPLETE ….

THEN CPU ALLOCATED

PROCESS VOLUNTARILY RELESE CPU
AFTER EXECUTION….

EG: FCFS

PREEMPTIVE SCHEDULING



CPU CONTROL CAN BE SWITCH ANYTIME

PROCESS NEED NOT WAIT LONG TIME

PROCESS FORCIBLY RELESE CPU WHILE EXECUTING….

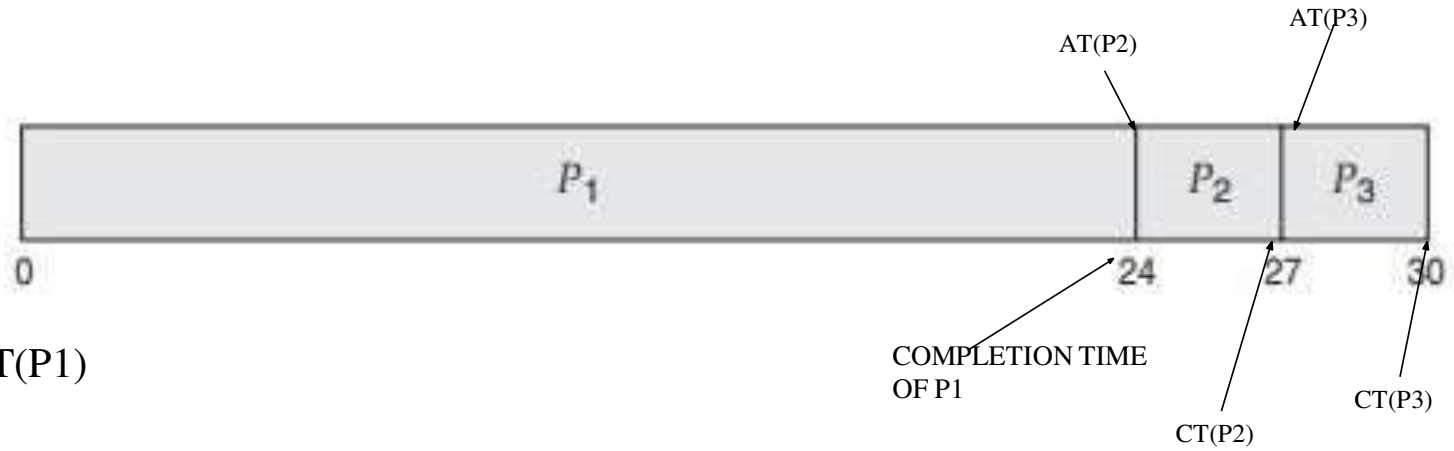
EG:: SJF
ROUND ROBIN
PRIOTIRY

- **FCFS TECHNIQUE:::**

Process	Burst Time
P_1	24
P_2	3
P_3	3

process arriving in the order P1, P2, P3….

GANTT CHART….



AT(P1)

WT(P1) ==0 msec

WT(p2) ==24

WT(p3)==27

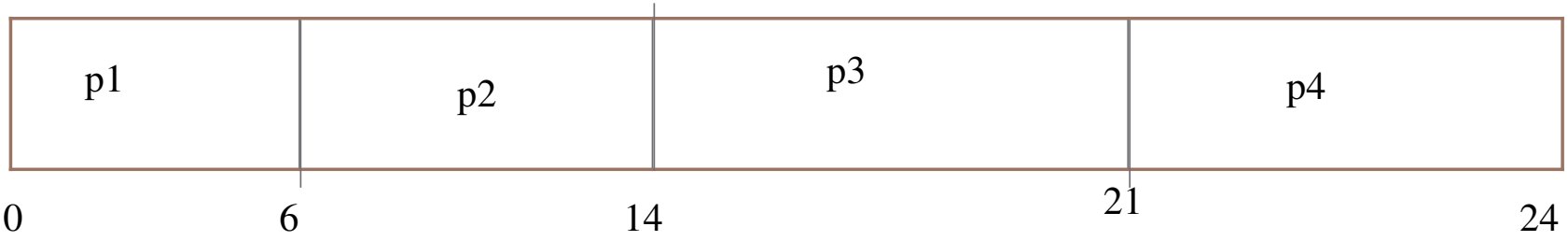
AWT= (0+24+27)/3== 17msec

EXAMPLE-2

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

wt(p1) =0
wt(p2)=6
wt(p3)=14
wt(p4)=21

AWT=10.25 msecc



- **SHORTEST JOB FIRST SCHEDULING (SJF)**

Process with Minimum CPU time / Burst Time is allocated to CPU

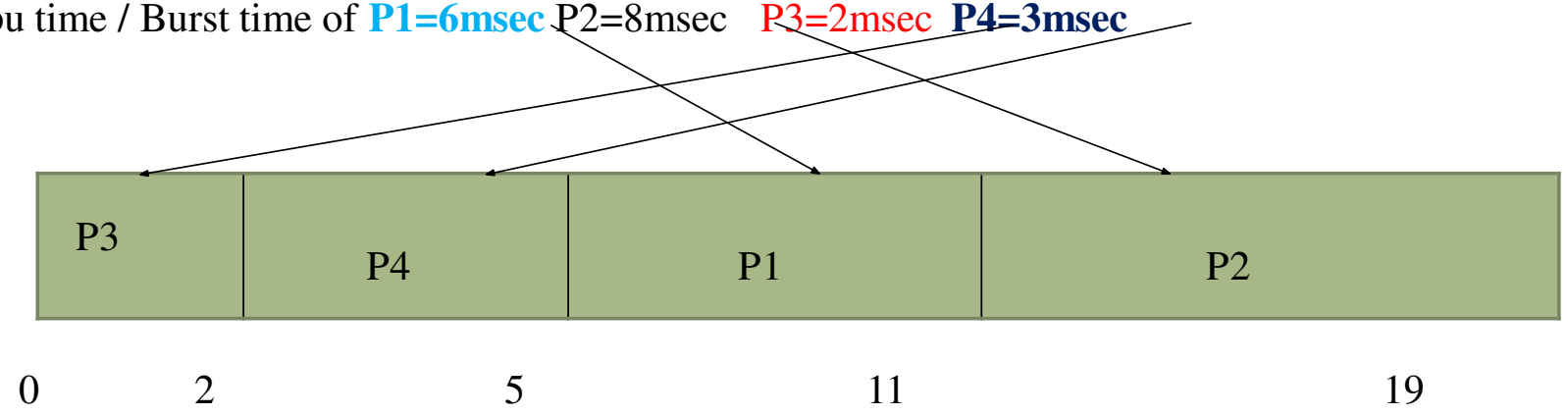
Process with more CPU time is scheduled Last

More efficient technique

It gives minimum waiting time for processes

Eg: Consider a process having 4 process P1 P2 P3 P4

Cpu time / Burst time of **P1=6msec** P2=8msec P3=2msec **P4=3msec**



Above diagram is called as GANTT CHART

- Waiting time of P3 =WT(P3)=0msec
- P4 =WT(P4)=2msec
- P1 =WT(P1)=5msec
- P2 =WT(P2)=11msec

- Average waiting time of all process in system
- **AWT=(sum of all waiting time of all processes)/Number of Processes**

== (p1+p2+p3+p4)/4

== (5+11+0+2) / 4

== 18 /4

==**4.5 msec**

SJF(shortest job firs) Example -2

Process	Burst time(msec)
P1	6
P2	8
P3	2
P4	1

- -

_____total BT= 17 msec _____

Gantt Chart ::

0 1 3 9 17

WT(P4)=0 WT(P3)=1 WT(P1)=3 WT(P2)=9

AWT== WT(P4+P3+P1+P2)/4

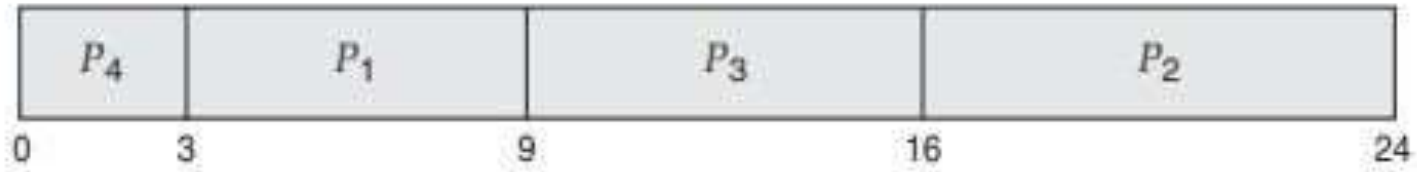
== 0+1+3+9 / 4

== 3.25 msecc

SJF SCHEDULING

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

Using SJF scheduling, we would schedule these processes according to the following Gantt chart:



- Shortest Remaining time First Schedule Algorithm (SRTF)

Process	Arrival Time	Burst Time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

- Process P1 is started at time 0, since it is the only process in the queue.
- Process P2 arrives at time 1.
- The remaining time for process P1 (7 milliseconds) is larger than the time required by process P2 (4 milliseconds)
- process P1 is preempted, and process P2 is scheduled.

- The average waiting time for this example is

$[(10 - 1) + (1 - 1) + (17 - 2) + (5 - 3)]/4 =$

$= 26/4 = 6.5$ milliseconds.

PRIORITY SCHEDULING

Priority….???

Preemptive or Non pre emptive

Indefinite Blocking time/ STARVATION

“”Aging “”

Low Number ---- High

High Number------ Low

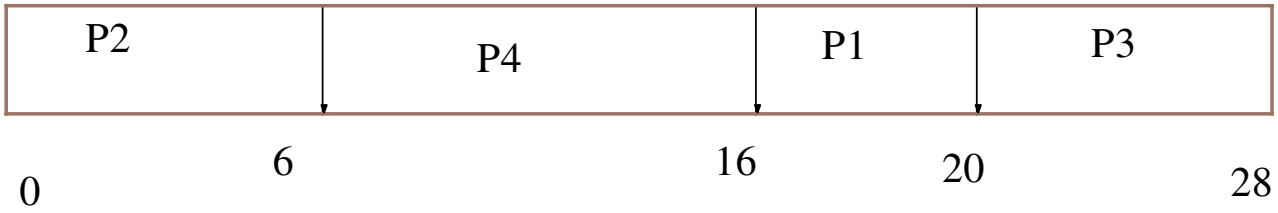
PRIORITY SCHEDULING

Process	Burst time(msec)	PRIORITY
P1	4	2
P2	6	0
P3	8	3
P4	10	1

- -

_____total BT= 28 msec _____

Gantt Chart ::



WT OF P2=0, P4=6 P1=16 P3=20

AWT== WT(P4+P3+P1+P2)/4
== 0+6+16+20/ 4
== 10.5 msec

- ROUND ROBIN SCHEDULING (RRS)
- For time sharing systems
- Pre-emption of process
- Time slice/ time quantum
- 10-100 msec

Process	Burst Time
P_1	24
P_2	3
P_3	3

Time quantum= 4msec

SYNCHRONIZATION

- * Co-operating process is the one that can affect or be affected by other processes executing in the system
- They can directly share a logical address space (code and data) or be allowed to share data only through files or messages.
- But concurrent access to shared data may result in data inconsistency.
- Consider the producer consumer problem which shares a common data buffer. producer process adds item to the buffer whereas consumer process removes item from buffer. The code for the producer and consumer are as shown below

producer

```
while (true)
{
  while (counter == BUFFERSIZE);
      /* do nothing -> buffer
   is full */
  buffer[in] = nextproduced;
  in = (in+1) % BUFFERSIZE
  counter++;
}
```

consumer

```
while (true)
{ while (counter == 0);
   /* do nothing --
      empty buffer */
  nextconsumed = buffer[out];
  out = (out+1) % BUFFERSIZE
  counter--;
}
```

Assume at any time t, value of counter as 5

counter = 5

when producer process executes at time t_1, it performs counter++; making counter value as = 6

At time t_2, when the consumer process executes, it performs the operation, counter--; making value of counter = 5.

If consumer executes entially, value of counter = 4 and then if producer executes counter value = 5. one would expect the answer to be 5 under these circumstances. But in reality, when

both the process execute concurrently, the value differs when
Consider counter ++ operation. It is a non atomic operation which is generally broken down and executed as follows.

reg1 = counter

reg1 = reg1 + 1

counter = reg1, reg1 = one of local CPU registers.

Similarily, counter -- can be broken down as follows

reg2 = counter

reg2 = reg2 - 1

counter = reg2.

where reg2 also is a local CPU register.

NOTE: Though reg1 and reg2 are same physical registers, during context switches the contents of registers are saved and restored by interrupt handler

The concurrent execution is done by executing low level instructions in an interleaved manner. consider the following execution

Time	Process	operation	instruction	content of register
T0	Producer	execute	reg1 = counter	reg1 = 5
T1	producer	execute	reg1 = reg1 + 1	reg1 = 6
				context switch
T2	consumer	execute	reg2 = counter	reg2 = 5
T3	consumer	execute	reg2 = reg2 - 1	reg2 = 4
				context switch
T4	producer	execute	counter = reg1	counter = 6
T5	consumer	execute	counter = reg2	counter = 4

We arrived at counter = 4 ⇒ which is incorrect where in real counter = 5.

When several processes access and manipulate data concurrently and the outcome of the execution depends on the order in which access takes place is called race condition.

To guard against race-condition, we require synchronization where we allow only one process to manipulate counter at a time.

With multicore systems, where multiple threads execute concurrently, in different processing cores, synchronization is very important

THE CRITICAL SECTION PROBLEM

consider a system containing n processes $[P_0, P_1 \ldots P_{n-1}]$.
Each process has a segment of code called critical section, in which processes may be changing common variables, updating a table, writing a file and so on.

The critical section problem is to design a protocol that processes can use to co-operate.
Each process must request permission to enter the critical section. This section of code complementing this request is the entry section.
The critical section is followed by exit section and remainder section.

General structure of a process

```
do
{
    ENTRY SECTION
        critical section
    EXIT SECTION
        remainder section
} while (TRUE);
```

Any solution to the critical section must satisfy the following requirements

1. Mutual Exclusion

If process Pi is executing in the critical section, then no other process is allowed to enter the critical section.

2. Progress

At any instant of time, if the critical section does not have any process executing, then any process that are not executing in remainder section, can decide which process will enter the critical section and the decision should be taken in finite time.

3. Bounded waiting (No starvation)

Any process after giving the request to enter the critical section should be allowed within a **bound** / limit after the request is made.

Many Kernel mode processes will be executing concurrently & various Kernel data structures are prone to race condition.

Eg) Consider kernel data structure that maintains a list of all open files. This data structure gets modified when a new file is opened or closed.

If two processes were to open the same file simultaneously it could result in race condition

Other Kernel data structures which are prone to possible race conditions are

* structures for maintaining memory allocation.
* for maintaining process.
* interrupt handling Table.

Two general approaches for handling critical section in OS are

(1) preemptive Kernels
(2) Non-preemptive Kernels.

• A preemptive kernel allows a process to be preempted while it is running in Kernel mode.

• A non-preemptive kernel does not allow a process running in kernel mode to be preempted. ie it runs until it blocks or voluntarily yield control of the CPU.

A non-preemptive kernel is free from race conditions on kernel data structures, as only one process is active in the kernel at a time.

Pre-emptive Kernels are those which are most suitable for SMP architectures, since in these environments, it is possible for two kernel mode processes to run simultaneously on different processors.

A preemptive kernel is more responsive, since there is less risk that kernel mode process will run for arbitrarily long period before relinquishing the processor to waiting processes.

Petersons solution - also known as Two process solution.

- It is a software based solution to the critical section problem
- Petersons solution is restricted to two processes that alternate execution between their critical sections and remainder sections.
- Before we arrive in the petersons solution, consider the following algorithm

ATTEMPT - 1

```
turn = i   // indicates turn is for i process

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

In this code mutual exclusion is guaranteed, but not progress.

If j process wants to enter the critical section and turn = i j has to wait till condition becomes turn = j.

ATTEMPT 2.

```
do
{
  flag[i] = true;
  while(flag[j]);
  Critical section
  flag[i] = false.
  remainder section
} while(1);
```

flag[0] = true if P_1 wants to enter
flag[2] = true if P_2 wants to enter

ensures progress
But in a pre emptive kernel,
consider 2 process.

P_1	P_2
flag[1] = True;	
	context switch
	Flag[2] = True
while(flag[2]); // wait	
	while(flag[1]); // wait

Both the process will have to wait infinitely i.e it does not guarantee bounded waiting

ATTEMPT 3

```
do
{ flag[i] = true;
  turn = j;
  while ( flag[j] && turn == j);
    CS
    flag[i] = false;
    remainder section.
} while (1);
```

This algorithm satisfies all conditions and is known as Petersons solution
It satisfies
* Mutual Exclusion
* progress
* Bounded Wait.

Analysis

P_1	P_2
flag[1] = true	
	flag[2] = true
Turn = 2	
	Turn = 1
while (flag[2] && turn == 2); // waits	enters CS
	flag[i] = F
Condition broken i can now enter CS.	

SYNCHRONIZATION HARDWARE

- The software based solutions such as petersons solution are not guaranteed to work on modern computer architectures.
  We can efficiently solve the critical section problem in modern computer architectures using hardware based solutions.
- Any solution to critical section problem requires a simple tool - a lock.
- Race conditions are prevented by ensuring the critical section regions be protected by locks
- A process must acquire a lock before entering the critical section and it should release the lock on exit.

Eg)

Process 1

```
while (1)
{ while (lock != 0); // wait
  lock = 1; // lock
  CS
  lock = 0; // unlock
}
```

lock = 0

Process 2

```
while (1)
{ while (lock != 0); // wait
  lock = 1; // lock
  CS
  lock = 0; // unlock
}
```

Yes This scheme ensures mutual exclusion?

Analyze:-

lock = 0;

P1: while (lock != 0);

context switch

P2: while (lock != 0)
P2: lock = 1;

P1: lock = 1;

... Both process in the critical section!! - Hence no mutual exclusion.

The above procedure failed because while (lock != 0) and lock = 1 is not atomic. (cannot be broken down).

- Hence we rely on hardware support.
- In a uniprocessor environment, the same can be achieved by disabling interrupts ensuring that the instruction is executed completely.
- But disabling interrupts is not a feasible solution in multiprocessor environment, as it can be time consuming

The message has to be passed to all processors - but can cause delays. The system efficiency ~~deg~~ decreases due to the delays in the critical section. Also the working of clock will be affected if we use hardware based interrupts.

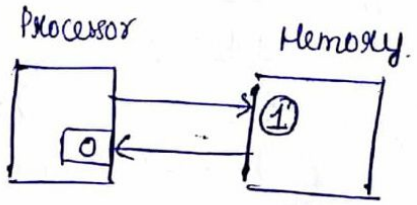
- Many modern computer systems provide special hardware instructions that allow us either to test and modify the content of a word or to swap the contents of words atomically
- Test and set instruction c) and swap c) instructions help us in atomic execution of instructions.

Test and set instruction

The definition of Test and set instruction is as follows

```
boolean TestAndSet (boolean *target)
{boolean rv = *target.
 *target = TRUE
 return rv;
}.
```

returns previous value of the memory location



target = 0
rv = 0
target = 1
return rv.

Even if 2 CPUs execute test and set at the same time, the hardware ensures that one test and set does both its steps before other one starts.

usage of test and set instruction in Locking

```
do
{ while (TestAndSet(&Lock))
      ; // do nothing.

  // critical section

  lock = false;

  // Remainder section
} while (TRUE);
```

Assume initial lock = 0

```
P1: while (TestAndSet &Lock)
        // Lock = 1
    CS
    Lock = F;
```

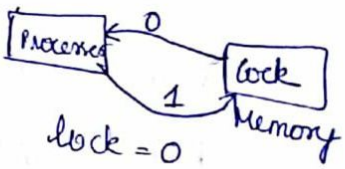
P2 executes while (TestAndSet &lock)
// since lock = 1, P2 waits for this condition to become false and then enters

Swap instruction

swap() instruction operates on two words in an atomic manner.

The definition of swap() instruction

```
void swap (boolean *a, boolean *b)
{
    boolean temp = *a;
    *a           = *b;
    *b           = temp;
}
```



lock = 0

temp = 0
*a = 1
*b = 0;

Like in the case of TestAndset instruction(), even if two CPU's execute swap instruction, one will execute atomically followed by the other.

- In Intel, this is done using xchg instruction.
- If the system supports swap() instruction, the mutual exclusion is obtained as follows.

```
do
{ key = TRUE;
  while (key == TRUE)
    swap (&LOCK, &key);
    // critical section
  lock = false
    // remainder section
} while (TRUE);
```

Assume initial value
lock = 0 / False & key = true

P1: → while (key == true)
lock = True
key = false

P2: key = true
while (key == True)
lock = T
key = T

Remember
lock = global variable
key = Local variable.

Bounded waiting mutual exclusion with TestAndset()

```
boolean waiting[n];
boolean lock;
```

→ common data structures initialized to false.

```
do
{ waiting[i] = TRUE;
  key        = TRUE;
  while (waiting[i] && key);
    key = TestAndSet(&LOCK);
  waiting[i] = FALSE;
    // CS
  j = (i+1)%n;
  while (j != i) && (!waiting[j])
    j = (j+1)%n;
  if (j == i)
    lock = False;
  else
    waiting[j] = FALSE;
  // remainder section
} while(TRUE);
```

Guarantee for mutual exclu
A process can enter CS only if
either waiting[i] = F or key =

P_1	P_2
waiting[i] = TRUE	waiting[2] = Tru
Key = TRUE	Key = True
key = TestAndset(Lock)	
Lock = 0	→ Lock = 1
Key = 0	Key = True
Waiting[i] = F	// It cannot enter CS
j = 2%10 = ②	
Waiting[2] = False	

progress requirement is also met because when a process leaves the critical section it sets lock = false or waiting[j] = false.

Bounded waiting is also met when process leaves CS, it scans entire array in cyclic order ie any process waiting to enter CS can do so within (n-1) terms.

Implementing TestAndset instruction in a multiprocessor architecture is a difficult task.

SEMAPHORES

The hardware based solutions to the critical section problem are difficult and complicated for application programmers to use.

- semaphore is a tool we use to overcome the implementation difficulty.

- Semaphore is an integer variable that apart from initialization, can be accessed only through two standard atomic operations : wait() and signal().

wait() and signal are atomic operations:

wait() operation → was originally termed as Proberen (to test)
signal() operation → originally called as Verhogen (to increment)

Definition of wait()

```
wait(S)
{ while(S<=0)
      ; // no operation
  S--;
}
```

Definition of signal()

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

All the modifications to the integer variable has to be done atomically. Also mutual exclusion has to be ensured.

Usage:

Semaphores → binary semaphores (takes only value range between 0 or 1).
Semaphores → counting semaphore

Binary semaphores are also known as mutex locks (which are locks that ensure mutual exclusion).

Counting semaphores can be used to control access to a given resource consisting of a finite number of instances. In this case semaphore is initialized to the number of resources available. when they want to use a resource they execute wait() operation on the semaphore decrementing the value of semaphore. After use of resource on return, it performs the operation signal() incrementing the count.

when count = 0, all resources are currently in use. In that case the process will have to wait / ~~not~~ block until count becomes greater than 0.

Eg) A server can support a maximum of 5 requests. S=5

P1 → request → wait(S) S=4
P2 → req → wait(S) S=3
P3 → req → wait(S) S=2
P4 → req → wait(S) S=1
P5 → req → wait(S) S=0
P6 → req → wait(S) S=-1 (magnitude indicates no. of process blocked)

when P_1 finishes usage it performs signal (S) $\Rightarrow S = -1+1 = 0$ and P_6 process will be chosen to executed.

Show how semaphore can be used to solve the synchronization problem between two concurrent process P_1 and P_2. P_1 with statement S_1 to be executed and P_2 with statement S_2 to be executed. Condition : S_2 should be executed only after S_1 is completed.

S_1;
signal (synch); — P_1

---

wait (sync)
S_2; — P_2

Implementation

- main disadvantage of semaphore is <u>busy waiting</u>
- When one process enters the CS, any other process that enters the critical section must loop continuously in the entry code.
- In a single CPU sys busy waiting wastes the CPU cycles when some other process could have used productively. This type of semaphore is called <u>spinlock</u>, because the process spins on the processor while another process is in the critical section.
- To overcome busy waiting, we will modify the wait and signal operation as follows. The semaphore definition is as follows.

```
typedef struct
{
    int value;
    struct process *list; // A facility to add the process in the queue while performing wait()
} semaphore;
```

ence a wait operation adds the process to the queue, whereas signal() operation will remove the process from the queue.

```
wait(semaphore *s)
{ s->value --;
  if (s->value < 0)
  { add this process to s->list
    block();
  }
}
```

```
signal(semaphore *s)
{ s->value ++;
  if (s->value <= 0)
  { remove process from s->list
    wakeup();
  }
}
```

If a process on executing wait() operation, if it is not able to enter the critical section, the particular process will be added to the queue list and it is blocked.

When the process finishes execution in the critical section, it signals the semaphore (increments value) and the process waiting in the list will be loaded in ready queue & the wait() of signal() is executed. So only one process enters the CS and all others will be blocked. This problem is known as Thundering Herd problem. A solution to this problem is to wakeup a specific process. Hence signal defintion is modified as

```
wait(semaphore *s)
{ s->value --;
  if (s->value < 0)
  { // add process to s->list
    block();
  }
}
```

```
signal(semaphore *s)
{ s->value ++;
  if (s->value <= 0)
  { remove process from s->list
    wakeup(P);
  }
}
```

wakeup(P) invokes only one process P. Hence only that process will be loaded in the ready queue and made to execute wait() operation.

The list of waiting processes can be easily implemented by a link field in each process control block (PCB).

The operation of semaphore should be executed atomically. This is a critical section problem. In a single CPU system, we can ensure this by disabling all interrupts. But in a multiprocessor systems, we should use some alternate strategies.

Deadlock & Starvation

The implementation of semaphore in a queue might result in a situation called deadlock where 2 or more processes are waiting for each other to complete, consider the system consisting of 2 processes P0 and P1, each accessing S and Q semaphore initialized to 1

P0	P1
wait (S)	Wait (Q)
Wait (Q).	wait (S).
:	
signal(S)	signal(Q)
signal(Q)	signal(S)

each process successfully got hold of S and Q respectively. since each one is waiting for other both is blocked indefinitely Such condition is a deadlock condition.

Priority Inversion

one of the scheduling challenge is when a high priority process wants to modify kernel data, and a low priority process is already modifying the kernel data. The situation becomes worse when a medium process with Priority M comes into picture.

The entry of the High priority process is now decided by a medium priority process if L < M < H and medium priority process gave the request first and L is in critical section. This problem is called priority ~~Inheritance~~. Inversion problem.

d solution to this is priority Inheritance protocol where we temperorily make the priority of the process highest while it is executing in critical section and once it is done, its priority will be changed to orginal priority. Now H will run and not H.

CLASSIC PROBLEMS OF SYNCHRONIZATION

3 classic large scale problems of synchronization under consideration

(1) Bounded Buffer Problem

(2) The Readers writers problem

(3) the Dining philosophers problem.

Bounded Buffer problem

- There is a buffer of size N, each space capable of holding only one item
- Two processes producer and consumer access the buffer in such a way that producer produces item into the buffer, whereas consumer consumes the item in a mutually exclusive manner.
- To ensure synchronization, we use semaphores.
- Mutex semaphore provides mutual exclusion at buffer pool
- Full and empty semaphores are used to count the occupied and empty buffer spaces

- Initialization values

mutex = 1

full = 0

empty = n.

Consumer

```
do
{ wait (full);  // If count == 0
  wait (mutex);
  // remove an item
     from buffer
  CS
  signal (mutex)
  signal (empty);
```

Producer

```
do
{ wait (empty), // If(count == N)
  wait (mutex);
  // CS
  add item
  signal (mutex),
  Signal (full);
} while (TRUE).
```

The Readers - writers problem.

- Suppose that a database has to be shared among several concurrent processes.
- some of these processes want to read the database (Reader process)
- some of these wants to update (writer processes).

- We can allow N readers to read simultaneously.
- We cannot allow a writer and a reader or two writers to access the database concurrently.

ie Writers should be given exclusive access to the database while writing and this synchronization is called reader writers problem.

There are various variations to the reader writer problem based on priorities

- First Readers writers problem gives priority to reader processes. ie NO reader should be kept waiting unless a writer has already gained access to critical section
- The second readers writers problem prioritizes writer processes ie once a writer is ready, writer is allowed to perform the write as soon as possible. ie if a writer is waiting to access the object, no new readers may start reading
- Both the solution may cause starvation - in the first case writers may starve whereas readers might starve in second scenario.

Solution / Implementation of first Readers writers problem

The reader processes share the following datastructure

```
semaphore mutex, wrt
int readcount
```

mutex variable is to ensure mutual exclusion when variable readcount is updated.
wrt is common to both writer & reader process.
Readcount keeps track of the readers in the critical section

Initialization values

mutex = 1
wrt = 1
readcount = 0

Structure of writer process

```
do
{ wait(wrt);
   // writing is performed
  signal(wrt);
} while(TRUE);
```

Structure of reader process

```
do
{ wait(mutex);
  readcount ++
  if(readcount == 1)
  wait(wrt);
  signal(mutex);
  // reading performed
wait(mutex);
  readcount --;
if(readcount == 0)
  signal(wrt);
  signal(mutex);
} while(TRUE)
```

NOTE:- If a writer is in the critical section and n readers are waiting, one reader is queued on wrt, and (n-1) readers are queued on mutex
On signal(wrt), the scheduler decides whether to wake up just one writer process in (wrt) or all the reader processes.

Locking solution

Acquire the read write lock by specifying the mode
If process wishes to read the shared data, then it requests the read write lock in the read mode.
If any process wants to write, then it should request for the read-write lock in write mode.

Reader writer locks are most useful in the following situations.

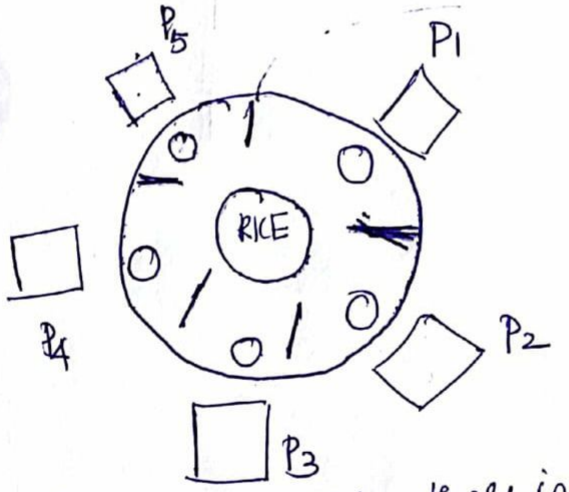
- In applications where it is easy to identify the reader and writer processes
- In applications where there are more number of readers than writers.

↳ Because the synchronization primitives causes performance overheads which can be compensated by allowing an concurrent readers)

THE DINING PHILOSOPHERS PROBLEM

There are 5 philosophers who spends their lifes — thinking and eating.

- These 5 philosophers are seated on a circular table on 5 chairs as shown below



- In the centre of the table, there is a bowl of rice and the table has five single chopsticks laid in the manner shown in the diagram.
- when a philosopher gets hungry, he grabs the chopsticks closest to him ie of left and right neighbours
- At a time the philosopher can pick up only one chopstick at a time.

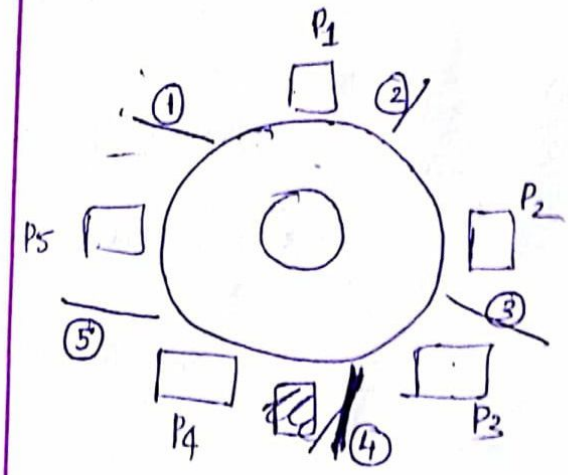
Once eating is done, the philosopher can put down both chopsticks and starting thinking again.

- Once a philosopher is eating No preemption is done on those chopsticks.

Aim:- To make sure that no philosophers are starved.

Sol:- Represent each chopstick with a semaphore
Each philosopher grabs a chopstick by executing wait() operation on that semaphore and release by executing signal operation.

```
Shared data structure
        Semaphore chopstick[5];    // all initialized to 1
Consider the structure of philosopher i
do
{ wait (chopstick[i]);
  wait (chopstick[(i+1) % 5]);
  // eat
  signal (chopstick[i]);
  signal (chopstick[(i+1) % 5]);
  // think.
} while (TRUE)
```



ie pick right first & then left

The above solution can result in deadlock. when the 5 philosophers become hungry at same time & grabs their right chopsticks.

All elements on the chopsticks will be 0 and each one when they try to grab their left chopstick, the wills be delayed forever

possible solutions:-

- Allow almost 4 philosophers to sit simultaneously
- Allow philosophers to pick up the chopsticks with both are available

- Use an asymmetric solution.
  ie odd philosophers pick up her left chopstick followed by right and even philosophers pick up their right chopstick first followed by left.

MONITORS

- semaphores provide convenient & effective mechanism for proper synchronization But using them incorrectly will result in timing errors, which are very difficult to detect, since these errors happen only if some particular execution sequences that takes place which do not always occur.

- All the processes share a semaphore variable mutex, which is initialized to 1. Each process must execute wait(mutex) to enter the critical section. After leaving the critical section, the process must execute signal(mutex).

If the sequence is not followed in the correct order then the following scenarios can happen

case 1: signal(mutex)
  //cs
  wait(mutex)
// Two or more process will be in critical section simultaneously.
- violation of mutual exclusion requirement

case 2: wait(mutex) // Deadlock will occur.
  //cs
  wait(mutex)

Even if a single process is not behaving, the entire processes can get affected.

To deal with such errors, researchers have developed high level language constructs.
one such fundamental high level synchronization construct is monitor type.

Usage

An abstract data type (ADT) encapsulates private data with public methods to operate on that data.

A monitor is an ADT which presents a set of programmer defined operations that are provided mutual exclusion within the monitor. It also contains the declaration of variables whose values define the state of an instance of that type, along with the bodies of procedures or functions that operate on those variables.

NOTE:- The shared data can only be accessed by procedures Local data are accessed only via Local procedures.

```
monitor monitor_name
{
// shared variable declaration
    Procedure P1 ()
    {
      }
    Procedure P2()
    {
    }
  :
  Procedure Pn()
      {
      }
      Initialization code.()
        {
        }
```

The monitor ensures that only one process at a time is active in the monitor.

The synchronization schemes are provided by conditional variables.

A programmer can define one or more variables of type condition.

condition x, y;

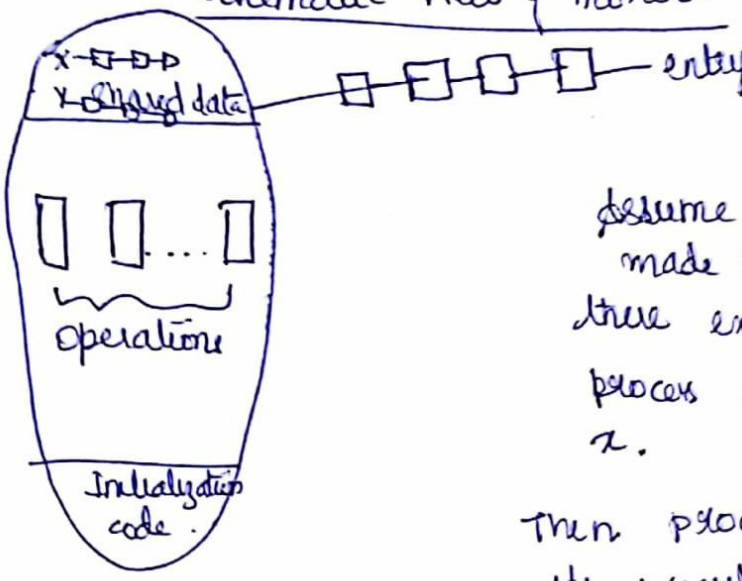
The only operation that can be invoked on a condition variable are wait() and signal

Eg). x.wait() and x.signal

x.wait() ⟶ process invoking this operation is suspended until another process invokes x.signal()

x.signal() ⟶ resumes exactly one suspended process.

Schematic view of monitor



Assume x.signal operation was made by process P and there exists another suspended process Q associated with condition x.

Then process Q should resume its execution signalling P, otherwise both will be in CS.

Two possibilities which exists are

1. signal and wait().

P either waits until Q leaves the monitor or waits for another condition

2. signal and continue

Q either waits until P leaves the monitor or waits for another condition.

Signal & continue is more reasonable as P was already in monitor.

UNIT - 4 DEADLOCKS

For a process to execute it requires resources. In a multiprogramming environment several processes compete for the limited amount of resources.

A process should request for the resource. If it is available at the time of request they are allocated to the process. If the resource is not available that moment of time, the process enters the wait state

Deadlock is a situation where a waiting process is never able to change state, because resources requested by it are held by other waiting processes.

Analogy:- when two trains approach each other on crossing, both shall come to a stop, and neither shall start up again until other has gone

System Model:-

There are various kinds of resources in the system like CPU cycles, files, I/O devices (Printers, DVD drives) etc If a system has two CPUs, then the resource type CPU has two instances.

Thus by instance, we refer to the no of resource types of similar class.

If a process requests for an instance of a resource type, the allocation of any instance of same-type will satisfy the request.

eg: 2 printers on the same floor.

If the two printers are on different floors, then they can be considered as different instances of different resource types

eg. Printer on 1st floor Printer on 3rd floor.

A process, for its execution it requires resources and it should utilize the resource in the following sequence

(a) Request :- The process should first request the resource. If the request cannot be granted immediately it should be put in waiting queue until it can acquire the resource.

(b) Use : The process can operate on the resource (eg: if the resource is a printer, the process can print on the printer.

(c) Release : The process should release the resource.

Request() and Release() are system calls.
open(), close(), allocate(), deallocate() etc are memory system calls.
A system table records whether each resource is free or allocated. It also maintains information like to which process it is allocated.
A set of processes is in a deadlocked state if every process in that set is waiting for some other resource held by another process in the set.

The resources can be physical or Logical resource

eg: Physical resource :- printers, tape drives, memory space, CPU cycles etc.

Logical Resources :- Files, semaphores, Monitors etc.

Eg of deadlock :- consider a system with 3 CD RW drives. Suppose one process requires 2 drives and if each one is already allocated with one drive each, each process is waiting for other processes or events of Release CD drive Hence all the 3 processes will be in deadlocked state

Deadlocks can occur wherein multiple resource types are also involved. Suppose Pi is holding the DVD and process Pj is holding the printer. If Pi requests for printer and Pj for the DVD drive deadlock occurs.

Multithreaded programs are good candidates for deadlocks, as multiple threads can compete for resources. Hence it is the responsibility of the application developer to ensure proper synchronization with respect to access of resources.

DEADLOCK CHARACTERIZATION:-

In a deadlock, processes never finish executing and all the resources are tied up such that no new process can be executed.

NECESSARY CONDITIONS.

A deadlock situation can arise if the following four conditions hold simultaneously in the system.

1. Mutual exclusion:- A condition where only one process can access the resource at a time. For this the resource should be non-sharable. Eg) Printer

2. Hold and wait:- A condition where process holds at least one resource and waits for other resources currently held by other processes

3. No preemption:- Resources cannot be preempted ie a resource is released only a process finishes of execution voluntarily

4. Circular wait:- A set $\{P_0, P_1, P_2 \ldots P_n\}$ of waiting processes must exist such that $P_0 \rightarrow P_1 \rightarrow P_2 \rightarrow P_3 \ldots P_{n-1} \rightarrow P_n \rightarrow P_0$. Such a condition is called circular wait

RESOURCE ALLOCATION GRAPH

Deadlocks can be described more precisely using RAG (Resource allocation graph). The Graph consists of a set of vertices and a set of Edges.

Standard Representations

○ → Process

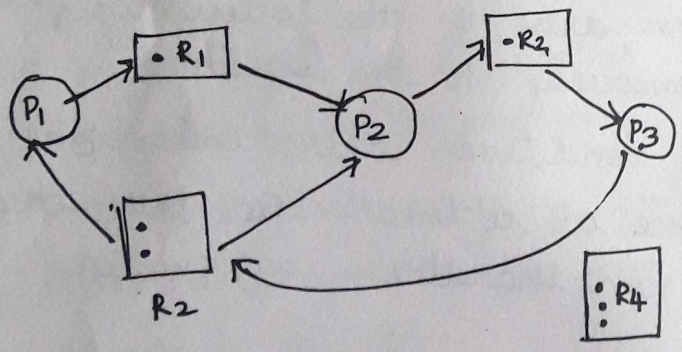
▪ → single Instance of Resource

⁘ → Multiple (4) instances of resource

$P_i \rightarrow R_j$ → Request Edge

$R_j \rightarrow P_i$ → Assignment Edge.

when a process gets the resource without any wait, immediately the request edge is converted to assignment Edge.



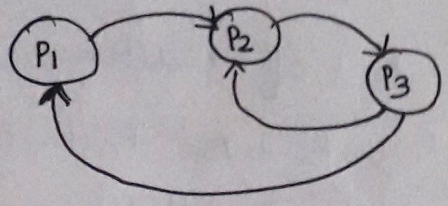
Consider the RAG given above.

Process P_1 is requesting for Resource R_1 held by P_2

Process P_2 is requesting for resource R_2 held by P_3

Process P_3 is requesting for resource R_2 held by P_1 and P_2

The corresponding wait for graph for the above RAG is



ie $P_1 \rightarrow P_2 \rightarrow P_3 \rightarrow P_2$

$P_1 \rightarrow P_2 \rightarrow P_3 \rightarrow P_1$

Deadlock prevention strategy uses a set of methods for ensuring atleast one of the four conditions cannot hold. Deadlock avoidance algorithm requires that the OS be given additional information concerning which resources will be requested by which all processes during its lifetime. With this additional knowledge, it can decide for each request, whether or not a proces should wait.

Deadlock detection strategy does not use any prevention or avoidance strategy. Rather it will allow normal request by processes but checks whether new request has caused the system to be in deadlock or not. and an algorithm to recover from deadlock.

If a system neither employs the above strategies, a deadlock will occur and system performance might degrade and has to be then restarted manually. Even though this approach seems inefficient, it does not have to bear the overhead of prevention or avoidance algorithms.

DEADLOCK PREVENTION:-

For a deadlock to occur, the four necessary conditions which should hold simultaneously are

(1) Mutual Exclusion

(2) Hold & wait

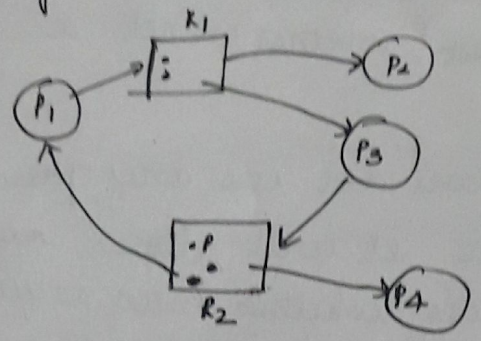
(3) No preemption

(4) Circular wait.

A deadlock prevention algorithm works towards breaking any one of the four necessary conditions

The presence of cycle in a RAG may cause a deadlock to occur. If there are only one instance of resource type, then the presence of cycle surely indicate a deadlock whereas only multiple instance resources are present, the presence of a cycle may or may not cause a deadlock.

Eg)



$P_1 \rightarrow P_2 \rightarrow P_3 \rightarrow P_1$

But since P_4 or P_2 is not waiting for any other resource, they can finish off execution, and give of to the requesting processes.

summary

In an RAG, the cycle indicates a sure deadlock if only one instance of each resource is present

A cycle in a system with multiple instances of resource may or may not cause deadlock

In a RAG, if there is no cycle then there is no deadlock.

METHODS FOR HANDLING DEADLOCKS

A deadlock problem is handled in any one of the following ways

- Use a protocol to prevent or avoid deadlocks ensuring that system will never enter a deadlocked state
- Allow the system to enter deadlock state, detect it and then recover
- Ignore the problem all together as if deadlock will never occur in the system

(a) Mutual Exclusion

Mutual Exclusion condition is a must for non-sharable resources. Eg) A printer cannot print two documents simultaneously.

Eg2) A read write file cannot give access to two processes requesting for conflicting operations - read & write.

We cannot prevent deadlocks by not allowing non-sharable resources. But we should ensure proper synchronization of these resources

(b) Hold and Wait:

To ensure that hold and wait condition never occurs in the system, a process should never hold any other resources when it is requesting some resource. There are two protocols widely accepted

(i) Each process should request for all the resource it needs before it ~~for~~ executes.

(ii) Allow a process to request for some resource only if it has none.

Illustration of difference between the two protocols

P1 → copy data from DVD drive to a file on disk, sort the file and print the file

~~P2~~ 1st protocol says only if DVD drive, file and printer is available then process will execute.

Disadvantage:- printer even though required towards the end is held throughout execution

second protocol allows a process to request for additional resources only if it has none.

Initially P1 given DVD drive and file // performs copy to file

Then P1 is given file and printer // to perform printing

After it copies onto file it should release it and then request again

Disadvantages

- Low resource utilization
- Starvation may occur (A process needs several resources to and might have to wait indefinitely).

(c) No preemption :-

To break this condition, we use a protocol that if a process request for any resource which cannot be granted immediately all the resources from the requesting process are preempted & given to other processes.

An alternative strategy is to check whether the resource requested by process is available and if not available, check if any waiting process holds it if so all the resources from the process in waiting Queue will be preempted and given to the requesting resource

(d) circular wait :-

To prevent this condition, we ensure that the resources must be requested based on some logical ordering only. For every resource in the system, we assign a value eg: F(tapedrive) = 1, F(disk drive) = 5, F(printer) = 12 etc.

A process can request for any resource in increasing enumeration: ie it cannot request for a disk drive holding Printer since F(printer) ie 12 > F(diskdrive) ie 5. such an ordering will break this circular wait condition.

DEADLOCK AVOIDANCE

This approach of handling deadlocks, requires some additional information about how resources are to be requested ie if P and Q process are asking for two resources R_1 and R_2, We need to know in advance which order each of the process requests for the resource.

- With the knowledge about, the requesting pattern, the system can allocated resources in such a way that the deadlock situation can be avoided.

- Various algorithms / models require various informations. The simplest and most useful model requires that each of the ~~informed~~ process requires declaration of the maximum number of resource of each type required for its execution.

- A deadlock avoidance algorithm dynamically examines the resource allocation state to ensure that circular wait condition will never happen.

- The resource allocation state is defined by the no: of resources allocated and the maximum resource types required for the execution of program.

Safe state:-

A state is safe if the system can allocate resources ~~to~~ ~~not~~ each process in some order such that deadlock is avoided.

A safe state occurs only if there is a safe sequence.

A ~~safe~~ sequence of processes $<P_0\ P_1, P_2 \ldots P_n>$ is defined as the safe sequence for the current allocation

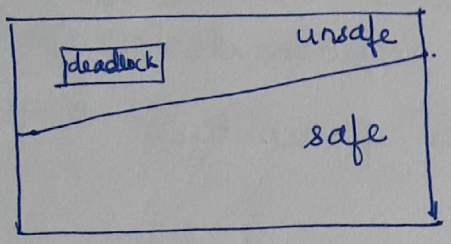
state if for process P_i, the resource request of P_i can still be satisfied by the current available resources + the resources held by all P_j, with $j < i$.

If the request cannot be immediately granted, the process P_i need to wait till all P_j finishes off execution.

A system is said to be in unsafe state if there is no safe sequence.

A deadlocked state is an unsafe state. But not all unsafe states are not deadlocks.

ie an unsafe state might result in a deadlock.



The operating system can go on allocating resources as long as the system remains safe. It should avoid all unsafe or deadlocked states.

Eg) A system has 12 magnetic tape drives. There are three processes P_0, P_1 and P_2 with the following requirements

	Max	Current Needs (Allocated)	Need
P_0	10	5	5
P_1	4	2	2
P_2	9	2	7
		9	

Available = 12 − 9 = 3//

At time to system is in safe state

$< P_1\ P_0\ P_2 >$ satisfies the safety condition.

P_1 can be given 2, and once it finishes execution it will leave off all the 4 resources to the pool.

Available = 3 + 4 = 7

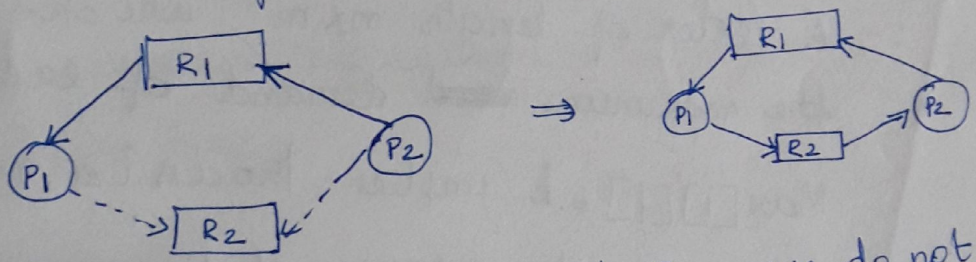
Now P_2 can be given 7 resources and after it

finishes off execution it returns all 7 back to pool.
Pool = 7.

Now P_0 need of 5 more tape drives can be satisfied and after execution it returns everything back to pool.

Resource Allocation Graph Algorithm

- This algorithm can be applied for single instance resource types.
- In addition to request edge and assignment edge, we introduce claim edge ie P_1 - - -> R_2 means P_1 wants R_2
- If the claim edge inclusion can result in a cycle, we do not convert it to request edge all we will convert claim edge to request edge



P_1 also wants R_2 and P_2 also wants R_2. We do not assign R_2 to P_2 and then allow P_1 to request for R_2 as it will result in deadlock

Bankers Algorithm

RAG algorithm is only suitable for resources of single instance type only.
For multiple instances of resource, we can use bankers algorithm.
The name was chosen because this algorithm uses an algorithm that bank uses that a bank will never allocate its available cash in such a

way that it can no longer satisfy the needs of all customers.

when a new process enters the system, it must declare the maximum no: of resources required for execution & this no: should never exceed the total resources in the system.

when any request comes, the algorithm must check whether this request if granted might put the system in deadlocked state or not.

several data structures are used for this algorithm. some of it are

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

If Available [j] = k implies k instances of resource j is available

Max :- A vector of length n x m, which defines the maximum ~~need~~ demand of each process

Max [i][j] = k implies process i requires maximum k instances of j.

Allocation: An n x m matrix, that defines the no: of resources of each type currently allocated to each process.

Allocation [i][j] = k ⇒ Process i is allocated with k instances of resource j.

Need : An n x m matrix, which indicates remaining need for each process

Need [i][j] = k ⇒ process i needs k instance of j to finish off execution

Needi - Maxi - Allocatedi

These data structures vary in size and value.

(a) Safety Algorithm

This algorithm helps us to find whether or not a system is in a safe state.

① Let Work and Finish be two vectors of length m and n respectively. Initialize

Work = Available and

Finish[i] = False for ∀ i from 0 to n-1

② Find an i such that:

Finish[i] == false and

Needi <= Work

if no such i exists go to step 4

③ Work = work + Allocationi

Finish[i] = true :

Go to step 2

④ If finish[i] == true for ∀ i → from 0 to n then system is in safe state else it is in unsafe state

(b) Resource Request Algorithm

This algorithm determines whether request can be granted safely or not.

Let Requesti be the request vector for process i.

If Requesti[j] == k, then process Pi requests for k instances of j at that time.

① If Request$_i$ ≤ Need$_i$ go to step 2
else raise error condition

② if request$_i$ ≤ Available$_i$ go to step3
else raise error.

③ Available = Available − Request$_i$
Allocated$_i$ = Allocated$_i$ + Request$_i$ // System pretends to allocate
Need$_i$ = Need$_i$ − Request$_i$

If resource allocation is safe then no deadlock else it can result in deadlock

Eg)

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

① Compute Need$_i$

	Need$_i$		
P_0	7	4	3
P_1	1	2	2
P_2	6	0	0
P_3	0	1	1
P_4	4	3	1

we claim that currently s/m is in safe state support a process requests for additional resource

we check whether Request i ≤ Available else it will check for whether it can result in safe / unsafe sequence.

when the system is in this state

Numerical :- Allocated

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

A system has a total of 10 instances of A, 5 instances of B and 7 instances of C.

The Available array is computed as

Available (A) = 10 − 7 = 3

Available (B) = 5 − 2 = 3

Available (C) = 7 − 5 = 2.

The Need matrix is computed as Max ij − Allocation ij

Need

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

We find a safe sequence as.

P_0 → Need 0 ≤ Avail ? → N

(743) ≤ (332) → N.

Not starting with P_0

P_1 Need 1 ≤ Available

(122) ≤ (332) Y

Allocated (P_1) = 200 + 122 Need i

322

Available − 332 − 122 = 210

After P_1 finishes of execution it returns 3 2 2 back to pool making Available =

$$\begin{array}{r} 2\ 1\ 0 + \\ 3\ 2\ 2 \\ \hline \end{array}$$

Available = 5 3 2

Is Need 2 ≤ Available = ?

(6,0,0) ≤ 532 N

So do not add to safe sequence $<P_1, >$

Is Need 3 ≤ Available?

(0 1 1) ≤ (5 3 2) Y

$$\text{Allocated}_3 = \begin{array}{r} 2\ 1\ 1 + \\ 0\ 1\ 1 \\ \hline 2\ 2\ 2 \end{array}$$

$$\text{Available} = \begin{array}{r} 5\ 3\ 2 - \\ 0\ 1\ 1 \\ \hline 5\ 2\ 1 \end{array}$$

After P_3 finishes of execution it can return (2 2 2) back to pool

$$\text{Available} = \begin{array}{r} 5\ 2\ 1 + \\ 2\ 2\ 2 \\ \hline 7\ 4\ 3 \end{array}$$

safe $<P_1, P_3$

Is $\text{Need}_4 \leq$ Available

(4,3,1) ≤ (7,4,3) → Y.

$$\text{Allocated}_4 = \begin{array}{r} 0\ 0\ 2 + \\ 4\ 3\ 1 \\ \hline 4\ 3\ 3 \end{array}$$

$$\text{Available} = \begin{array}{r} 7\ 4\ 3 - \\ 4\ 3\ 1 \\ \hline 3\ 1\ 2 \end{array}$$

After P_4 finishes off execution it returns (4,3,3) back to pool.

$$\text{Available} = \begin{array}{r} 3\ 1\ 2 + \\ 4\ 3\ 3 \\ \hline 7\ 4\ 5 \end{array}$$

safe $<P_1, P_3, P_4$

Is Need $P_0 \leq$ Available?

(7 4 3) ≤ (7 4 5)? → Y.

Allocated P_0 = 0 1 0 +
7 4 3
7 5 3

Available = 7 4 5 −
7 4 3
0 0 2

After P_0 finishes of execution it returns (7,5,3) back to pool

Available = 0 0 2 +
7 5 3
7 5 5

safe sequence = $<P_1, P_3, P_4, P_0>$

Is Need $P_2 \leq$ ~~Avail~~ Available

$(6,0,0) \leq (7,5,5) \rightarrow Y$.

Allocated P_2 - 3 0 2 +
6 0 0
9 0 2

Available = 7 5 5 −
6 0 0
1 5 5

After P_2 finishes of execution it returns (9,0,2) back to pool

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

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

All process have finished of execution returning all resources back after execution

Note: Final Available array should tally with total available

The system is then currently safe with one such safe sequence as $<P_1, P_3, P_4, P_0, P_2>$

can we grant an immediate request from process P_1 as (1 0 2)

check Request$_i \leq$ Need$_i$

$(1,0,2) \leq (1,2,2)$? $\rightarrow Y$.

check Request$_1 \leq$ ~~Needs~~ Available

$(1,0,2) \leq 3\ 3\ 2 \rightarrow Y$

3 3 2 −
1 0 2
2 3 0

Allocation can be granted immediately

Available = 2 3 0

Can the system be in safe state if a request (3,3,0) by P_4 comes? can it be granted immediately?

Request $P_4 \leq$ Need P_4?

$(3,3,0) \leq (4,3,1)$? Y

Request $P_4 \leq$ Available

$(3,3,0) \leq (2,3,0) \rightarrow$ N

Hence a request of P_4 cannot be granted immediately after P_1's request.

can we satisfy a request from P_0 immediately

Request $P_0 = (0,2,0)$

Request $P_0 \leq$ Need P_0

$(0,2,0) \leq (7,4,3)$ Y

Request $P_0 \leq$ Available

$(0,2,0) \leq (2,3,0) \rightarrow$ Y

can the allocation be done?

The allocation cannot be done immediately as starting with P_0 will leave the system in unsafe state

$<P_1, P_3, P_4, P_0, P_2>$ is the safe sequence.

Deadlock Detection

If the system does not employ either a deadlock prevention or a deadlock avoidance algorithm, then deadlock situation might occur. The system should then

(1) Provide an algorithm that examines the state of the system to determine whether a deadlock has occured

(2) Provide an algorithm to recover from the deadlock.

Single instance of Each Resource Type:-

From the RAG (Resource allocation graph algorithm), we can derive the wait For Graph (WFG).
We check whether there is a cycle or not.
An algorithm to detect cycle in the graph requires an order of n^2 operations.

Several instances of Resource type:-

When multiple instances of resources are there the RAG algorithm cannot be used. The deadlock detection algorithm employs several time varying data structures that are similar to those used in bankers algorithm

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

Allocation:- An nxm matrix that defines the no: of resources of each type currently allocated to each process.

Request: An nxm matrix indicates the current request of each process.
If Request[i][j] = k, Process Pi requesting k more instances of Resources Rj

Detection Algorithm

① Let work & Finish be two vectors of length m and n respectively.
Initialize work = Available

For $i = 0, 1, 2 \ldots n-1$, if Allocation $i \neq 0$, then Finish[i] = false else Finish[i] = true.

② Find an index i such that both
(a) Finish[i] == false
(b) Request i ≤ work
if no such i exists → go to step 4

③ work = work + Allocation
Finish[i] = true;
Go to step 2.

④ If Finish[i] == false for some i, $0 \leq i \leq n$, then the system is in deadlocked state.
if Finish[i] == false for some i, then process P_i is deadlocked state

requires $\boxed{m \times n^2 \text{ operations}}$ to detect whether s/m is in deadlocked state or not.

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

Available
A B C
0 0 0

Safe seq = $\langle P_0, P_2, P_3, P_1, P_4 \rangle$ results in finish[i] == true for all i

Assume P_2 makes an additional request 001 instead of 000 then s/m will be in deadlocked state. Even if we preempt the resource from P_0, deadlock will continue with P_1, P_2, P_3 and P_4.

Detection Algorithm usage:-

How frequently should we invoke the detection algorithm

Factors to be considered are

- How often a deadlock is likely to occur
- How many processes will be affected when a deadlock happens.

① if deadlocks occur frequently then deadlock detection algorithm should be invoked frequently. Resources allocated to deadlocked processes will be idle until the deadlock can be broken.

② Deadlocks can occur if some process request for resource that cannot be granted immediately. The request may be the final request that completes the chain of processes waiting for resources. should be invoke the algorithm very frequently ie after every request. Thus we can identify the deadlock immediately and the process that caused the deadlock. But this algorithm can cause a huge overhead

③ Invoke the deadlock detection algorithm at frequent intervals. eg: every hour once.

④ Invoke the deadlock detection algorithm based on a condition ie if the CPU utilization goes below 40%. etc.

In methods 3 & 4 we won't be able to identify the process which caused the deadlock

RECOVERY FROM DEADLOCK.

when a detection algorithm determines that deadlock exists, several alternatives are available.

Recovery options

(1) manual

(2) automatic.

To break deadlock there are 2 options

(1) abort one or more processes that causes circular wait

(2) To preempt resources from one or more processes waiting or in deadlock processes.

Process Termination.

Two methods are widely used

(*) Abort all deadlocked processes

Adv:- no complication

Disadv: huge overhead - There might be processes which have almost finished execution.

(*) Abort one or more processes at a time until the deadlock cycle is eliminated

Adv:- ↓ overhead as we are not aborting all the processes

Disadv:- overhead involved in choosing which process to be aborted

Factors to be considered while aborting

- Priority of process
- How long process has been given CPU
- How many & what type of resources a process has used
- How many more resources are required
- How many process need to be determinated
- whether process is interactive or batch

Never abort a process while it is in the middle of updation eg) updating a file.

Resource preemption

Three issues to be addressed if preemption is required to deal with deadlocks are

(I) Selecting a victim :-

Victim is the process which is chosen to be aborted. Factors such as no: of resources held by process, amt of CPU time used, priority etc are to be considered

Goal :- minimum cost while pre-emption.

(II) Roll Back :-

one resources are preempted from a process, we must roll back the state of the process to a safe state and restart from that state.

2 ways

(I) Total Rollback :- Restart from beginning

(II) partial Rollback :- Cost effective, to the state before deadlock occured.

But this method requires keeping additional information about the state of all running processes.

(III) Starvation :- If resources are preempted every time from the same process, starvation can occur. Hence victim selection should happen such that starvation should not occur

Sol :- Keep count of no: of times a process is chosen as victim.

ie Keep the Rollback count.

check if the victim is preempted as the minimum count specified - if so choose some other process as victim.

UNIT-5 MEMORY MANAGEMENT

MEMORY MANAGEMENT STRATERGIES - CHAPTER-8

- why do we require memory management?-

- To efficiently use the CPU, we require CPU scheduling
- For CPU scheduling to happen, efficiently, we require the programs which are ready for execution to be in the main memory. Because the main memory is the largest memory that CPU can access other than registers and cache.
- The size of the main memory is limited and hence to efficiently utilize the memory, we require memory management.
- The CPU fetches instruction from memory according to the value of the program counter.
- An instruction execution cycle fetches instruction from memory, then the instructions are decoded & may cause the operands fetched from memory. After the instruction is executed, the memory unit sees only a stream of memory addresses. There is no need for efficiently mapping the CPU generated address to the actual or physical location in main memory. Hence for the efficient utilization of main memory and for the better CPU utilization, we require main memory management.
- Most of the programs have to be brought into the main memory from the secondary storage devices. Hence to efficiteively bring in at less time, we require secondary storage management too

- In order to speed up the memory access, all the instructions which are frequently accessed are kept in the cache. Since the cache size is limited there is a need to efficiently manage the cache.

Thus by memory management, we require efficient main memory management, cache management, secondary storage management.

BASIC HARDWARE.

- The CPU can access only the registers and main memory which are built in the processor itself.
- For any instruction to be executed atleast it should be in the main memory.
- Registers that are built into CPU can accessed in one CPU cycle whereas, main memory, where a memory access requires transaction on memory bus, may take more CPU cycles. During that time, the processor might have to stall since it does not have data to continue execution.
- To reduce the time lag between the fast CPU and the slow memory, the remedy is to add fast memory (cache) between CPU and main memory.
- For ensuring protection of the OS from user processes and to ensure protection across the various processes, there are various implementation mechanisms.
- Each process has a seperate memory space. Thus there is a range of legal addresses that any process can access. This protection is ensured using two registers - Base register and the limit register.

The base register holds the smallest legal physical memory address and the limit register specifies the size of the range.

Eg: If base register holds - 300040 and limit reg - 120900, then the program can legally access all addresses from (300040 through 420939 (inclusive).

- For every CPU generated address, the value is compared with the base and limit values. Hardware protection using base & Limit registers is as shown below.

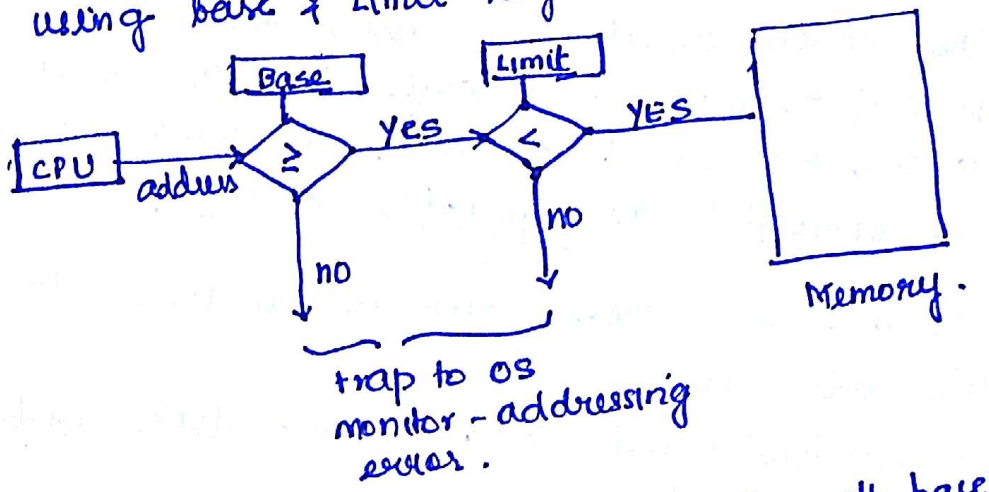


Fig:- Hardware address protection with base and limit regs

Any address is compared with base address - If less ↓ trying to access OS area → trap.

≥ base address - check with limit

If address < limit - grant access

else deny.

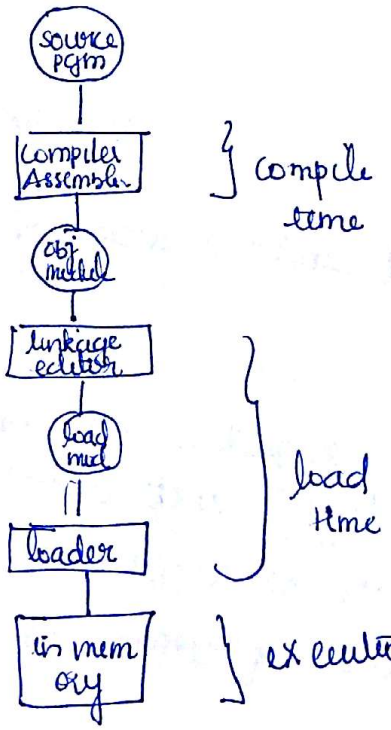
The base and limit registers are only accessed by the OS through privileged instructions. This scheme prevents any deliberate or accidental modification of code or data structures of either OS or other users.

ADDRESS BINDING:-

A program to be executed if residing in disk should be brought into main memory.
The processes on the disk that are waiting to be brought into memory form the input queue. For this the process from input queue should be selected and brought into memory. If any process finishes off execution its memory space in main memory is declared available.

Most of the modern OS, allows a user process to reside in any part of the physical memory. Thus the address space at computer starts at 0000, but user program might be represented as symbolic address (count).

A compiler will bind these symbolic addresses to relocatable addresses.
The linker and loader will load the relocatable addresses to absolute addresses.
Each binding is a mapping from one address space to another.



Multistep processing of user programs

The binding of instructions and data to memory address can be done at any step along the way

* compile time :- Here the absolute code is generated at compile time itself. If any changes to starting location happens, recompilation becomes necessary

* Load Time :- If the address binding does not happen at compile time, it is generated as relocatable address and final binding is delayed until load time.

* Execution Time :- If a process can be moved from ~~one~~ during its execution from one segment to another, then its final binding is delayed until run time / execution time

Logical Versus Physical Address space :-

An address generated by the CPU is known as the Logical address whereas address seen by the memory unit i.e one loaded into MAR (Memory Address Register) is known as physical address.

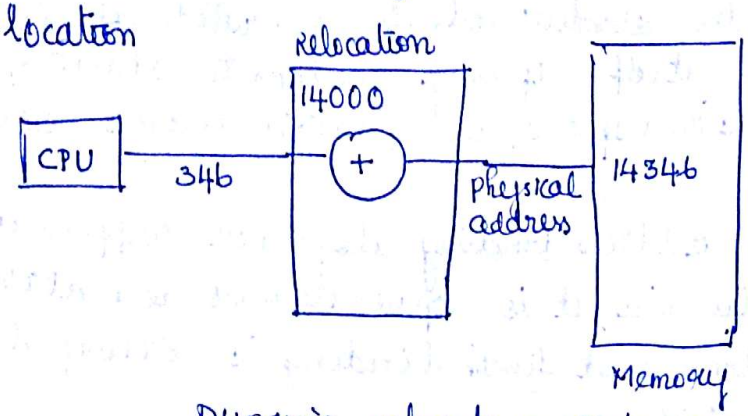
The compile time and load time binding generates identical logical & physical addresses whereas run time binding results in different logical and physical Addresses.

The Logical address is also known as virtual address and the set of all logical addresses forms the Logical address space for any program.

Memory Management unit is responsible for mapping logical addresses to physical addresses. It is a hardware unit. MMU uses various techniques to do logical to physical address mapping

PTO

(I) The simplest scheme is to use the base register (also known as relocation register). Every address generated is added to the value of the relocation register to get the actual location



Dynamic relocation Technique.

(II) Dynamic Loading

A program to be executed, should be loaded into main memory. For better memory utilization, we make use of a technique known as dynamic loading.

A routine is not loaded until it is called. All routines are kept in the disk in relocatable format, and only when a routine is called it is loaded into main memory. Thus the user program size is never limited to the size of main memory.

Adv:- * All infrequently called routines are not loaded into main memory (error routines etc).
* Less I/O required to load balance concurrency as the program loaded is very smaller.
* Higher concurrency support

NOTE: Dynamic Loading does not require any special support from the OS.

The programmer has to design their programs to take advantage of this method.

Dynamic Linking and shared Libraries

There are two types of Linking

Static Linking :- Here the library routines are treated like any other object module and are combined into a binary program image by the loader

Dynamic Linking :- Here linking is postponed till execution time. This is generally used with system library routines. This allows many programs to share the image of library routine rather than each one having its own.

A small piece of code called stub is included in the binary image of each library routine. While Executing, the stub replaces itself with the code of the library routine. The stub knows to locate the library routine in the main memory.

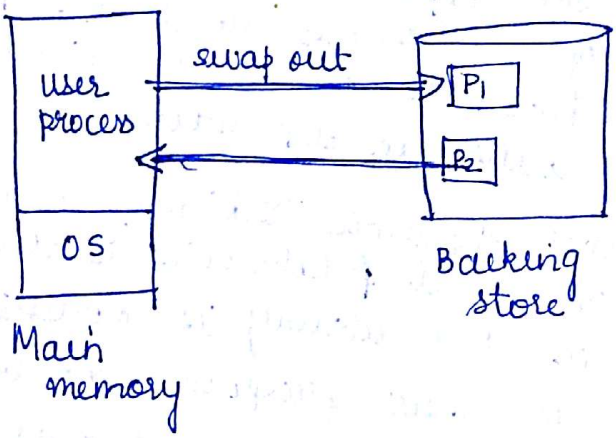
With dynamic linking, the concept of shared Libraries is supported i.e if a library is modified, all the new programs can be linked to modified one whereas the old programs can continue to refer to old version.

NOTE Dynamic Linking Requires help from the Operating systems. Because only the OS can check whether the required routine is in memory or not, ensuring memory protection

SWAPPING:

A process must be in main memory if it has to be executed But size of the main memory is limited and to increase the CPU utilization, we have to increase the degree of multiprogramming. (No of processes loaded in the main memory).
To make space for a process in the main memory, the OS might swap out some process out of the main memory and swap in the new process into the memory. This mechanism is known as swapping. Ideally a swapper is called when the CPU scheduler wants to reschedule the CPU.

In the case of priority scheduling, this mechanism is known as roll in - roll out where in a high priority process is swapped in and a lower priority process is swapped out.



Normally a process is swapped back into the same memory space as it occupied previously. If compile time or load time binding is done whereas a process can be swapped to a different location, if it is a run time binding.

The OS maintains a ready queue from which the CPU scheduler picks up processes ready to be executed and loads into main memory.

The CPU scheduler calls the dispatcher which checks if the process to be executed is already in main memory or not. If there are no free space in the main memory, the dispatcher swaps out a process from main memory and swaps in desired process.

The context switch time in such a swapping is very high. For efficient CPU utilization, we want the execution time of each process to be relatively longer than context switch time.

The major part of ~~com~~ swap time is transfer time. The total transfer time is directly proportional to the amount of memory swapped.

A 10MB process might be swapped quickly than a 256MB process.

We need to swap in only what is required and on execution, the memory should be released. For these, system calls are issued.

To swap a process, it must be completely idle. We should never swap a process which has pending I/O because it requires changing I/O buffers.

There should be a balance with respect to the swapping required. standard swapping which is used in most OS provides too little execution time and more swapping time. Most OS like UNIX disable swapping at up till a ~~process~~ particular threshold. only if there is insufficient memory for a process to be swapped in, the swapper is called

Microsoft windows allows partial swapping ie the swapped out process is not loaded back until a reference to it comes.

CONTIGUOUS MEMORY ALLOCATION

The main memory should accomodate both the OS and the various user processes.

To efficiently allocate the main memory, we have two possible methods :-

(I) contiguous allocation
(II) Non-contiguous allocation.

The main memory is divided into two partitions :-

(I) for the OS
(II) for user processes.

We can place the OS in the higher or lower memory. But since the interrupt vector is placed in the lower end of memory, the OS also we place at the lower end of the memory.

In contiguous memory allocation, we place the processes in single contiguous block.

Memory mapping and Protection :-

For memory mapping and protection, we use the relocation and limit registers.

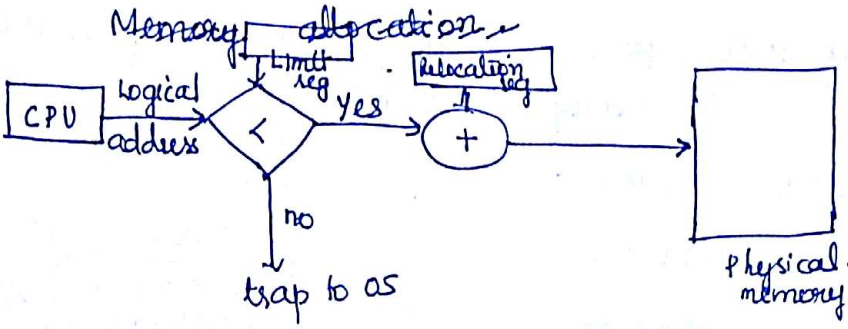
The relocation register contains value of the smallest physical address and limit register contains the range of logical addresses.

Eg: Relocation = 100040
Limit = 74600

whenever any logical address comes, it is checked with the limit register value. If the address generated is less than the limit, then this value is added with the value of relocation register to get the physical address. Else if address is greater than limit, then a trap to OS is generated.

when the CPU scheduler selects a process for execution, the dispatcher loads the relocation and limit registers with the correct values as part of the context switch. Thus we ensure protection of OS and other user processes

Transient OS code refers to the OS code which is brought into memory, only when it is executed or needed. eg: Device driver programs etc.



Hardware support for relocation & limit registers

MEMORY ALLOCATION:-

One of the simplest method for allocating memory is to divide memory into several fixed sized partitions. Each partition may contain exactly one process. Hence the degree of multiprogramming is bounded by the no: of partitions.

In Multiple partition method, whenever a process finishes execution, the memory partition becomes free.

In fixed partition scheme, the OS keeps a table indicating which parts of memory are free and which are occupied. The available free memory in any partition is known as a hole.

whenever a process arrives and needs memory, the biggest hole that can fit in a the process is allocated. when the process finishes off execution, the allocated memory is made available free.

At any time, we have various process waiting to get loaded into main memory in the input queue and some

free blocks. At any point of time, the process requirement is compared with the size of the available memory and if allocation to the process happen till no further process can be loaded into.

The available free blocks (holes) are scattered throughout. If the hole is large it is again divided to satisfy the process requirement and other one as a hole.

Eg: Consider a 1000K free memory block. The jobs are arriving in the following sequence

job1 : 250K arrives

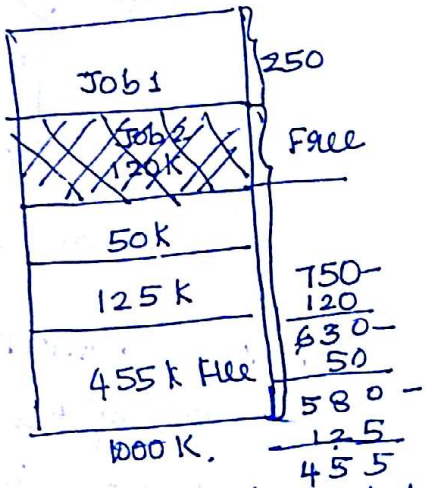
job2 : 120K arrives

job3 : 50K arrives

job4 : 125K arrives.

job2 finishes execution.

job5 : 50K arrives.



This procedure is a particular instance of Dynamic storage allocation problem, which concerns how to satisfy a request of size n from a list of free holes.

Job5: 50K can be allocated in 120K which became free or 455K free

3 algorithms are generally used for dynamic storage allocation problem

(i) First fit :- Allocates the first hole that is big enough to satisfy the request. We stop searching once we find a hole to satisfy the request

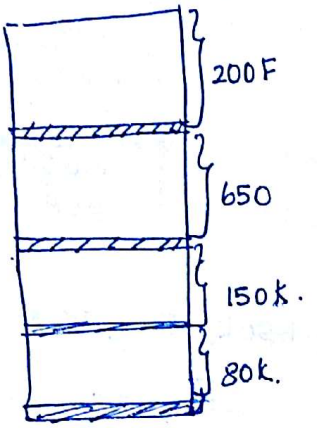
(ii) Best fit :- Allocates a hole which can rightly fit in a process (smallest hole that is big enough to hold a process). Requires searching the complete memory.

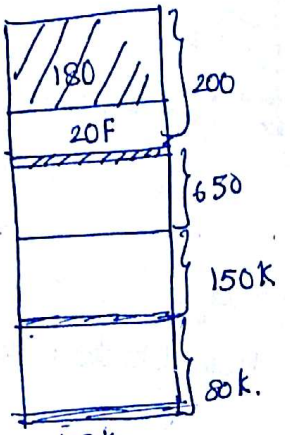
Worst fit :- Here we partition the largest hole. It also requires searching the entire memory.

Eg: consider the available memory blocks in the following order 200K, 650K, 150K, 80K (in order). consider the following requests arriving in order

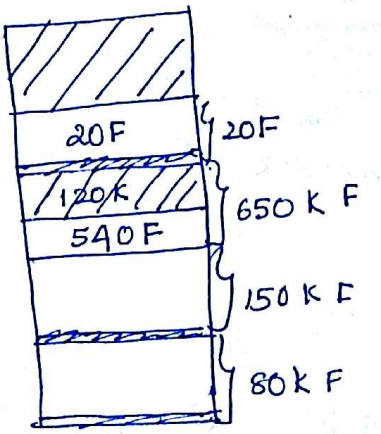
Job 1 : 180K
Job 2 : 120K
Job 3 : 50K
Job 4 : 500K.

First fit



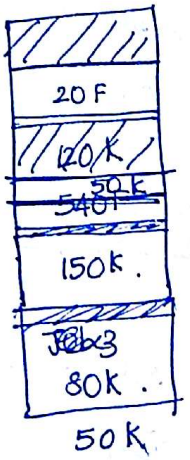


180K allocation

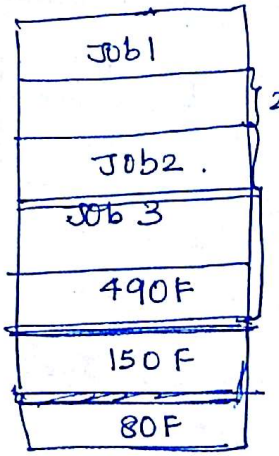


650 − 120 = 540

120K allocation.



540 − 50 = 490F



Job 4 allocation

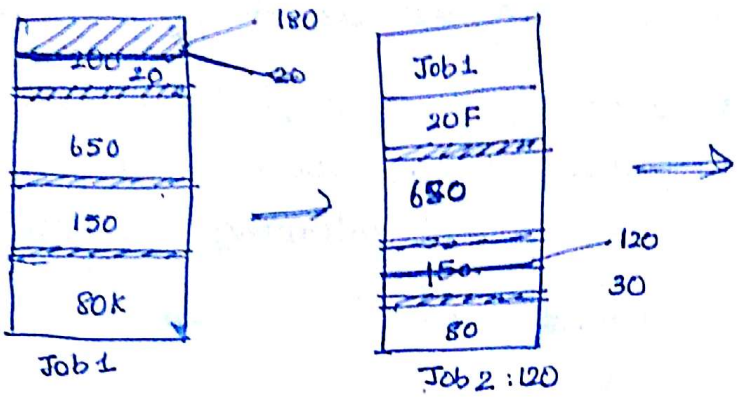
Choices: 20F X
490F X
150F X
80F X.

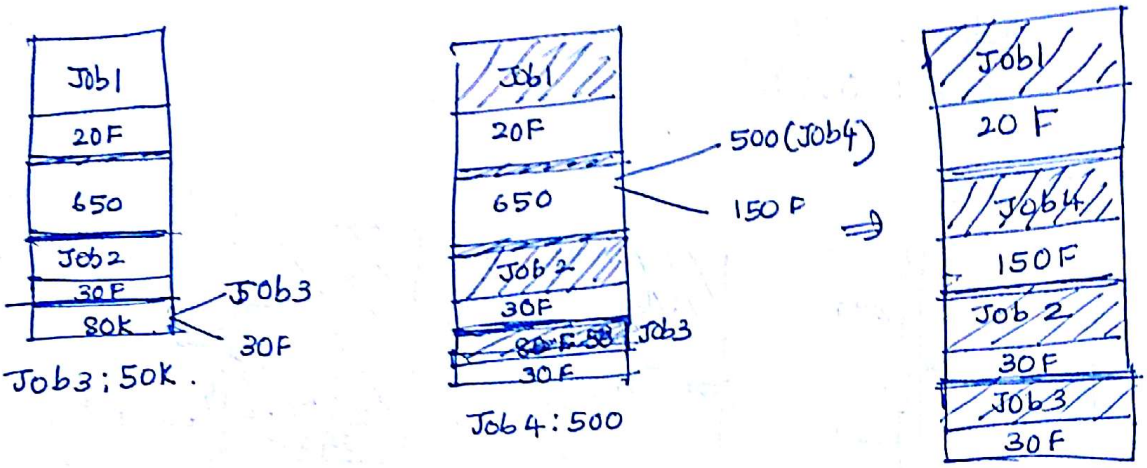
First fit cannot satisfy the request.

200, 650, 150, 80K

Best fit

Job 1 : 180K
Job 2 : 120K
Job 3 : 50K
Job 4 : 500K



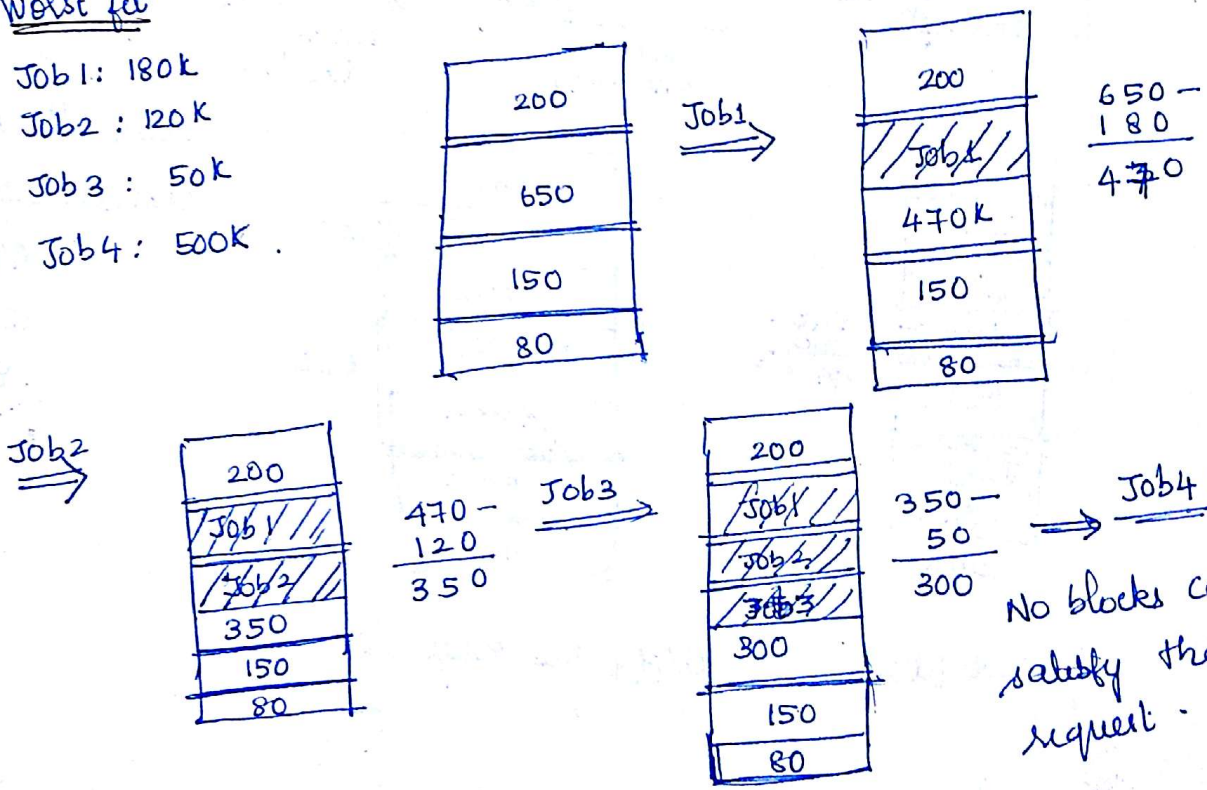


All job requests were satisfied.

Mem blocks : 200K, 650K, 150K, 80K.

Worst fit

Job 1 : 180K
Job 2 : 120K
Job 3 : 50K
Job 4 : 500K.



Fragmentation

As processes are loaded and removed from memory, the memory space is broken down into little space.

External fragmentation exists when there is enough total memory space to satisfy the request, but not individually as the memory spaces are not contiguous. Best fit and first fit can cause external fragmentation.

Factors depends upon the request arriving, the way we divide memory etc. Stastical analysis reveal that for every N blocks 0.5N blocks are lost in fragmentation (50% rule)

Internal fragmentation refers to the fragment of memory left when a partition

Eg: Req: 18642 bytes Available hole size = 18644

After Allocation the remaining 2 bytes within the block is there and the overhead to keep track of the memory is more than that of memory itself

Solutions to external fragmentation problem

(i) Compaction :- The goal is to shuffle all the memory contents to one end to combine all the holes to other end to form a bigger hole.

Challenges :- possible only in case of execution time binding (relocation is dynamic)
overhead of running compaction algorithm is expensive

(ii) Permit Logical address spaces to be non-contiguous. - ie allow process Logical addresses space scattered across main memory

There are various methods by which we can implement Non-contiguous allocation of memory. They are

(I) Paging
(II) segmentation
(III) segmentation with paging.

PAGING

A memory management scheme that allows the physical address space of a process to be non-contiguous. The advantage of paging is that it ~~allow~~ avoids the considerable problem of fitting chunks of memory of varying sizes on to backing store (which also suffers from fragmentation problem and with access much slower than the main memory.

Traditionally paging was implemented using hardware. Now recent designs have implemented paging by integrating the hardware with the OS.

BASIC METHOD :-

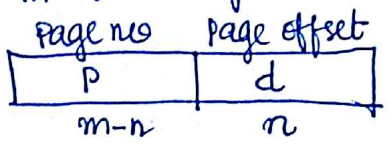
The main memory is divided into fixed size blocks called frames and the logical memory is divided into blocks of same size called pages

To execute a process, its pages are loaded into any available free frames in the memory.

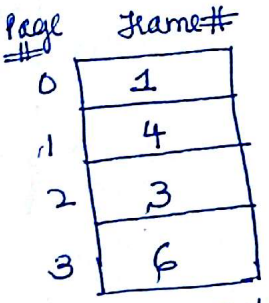
Every address generated by CPU consists of two parts :- pageno (P) and offset (d). The page no is used as an index to page table, whereas offset defines the location of desired byte on physical memory from the base address.

The page size typically varies across OS (512 bytes to 16MB) ~~etc~~. They are generally expressed as power of 2 (2^N).

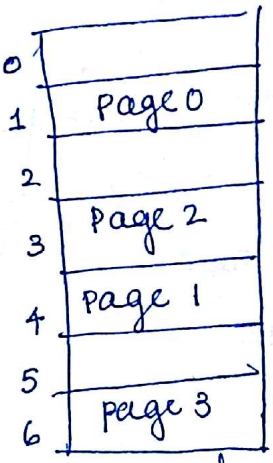
If the size of Logical address space is 2^m and page size is 2^n, then Lower bits indicate page offset & higher order bits $m-n$ designate the page number



Logical memory

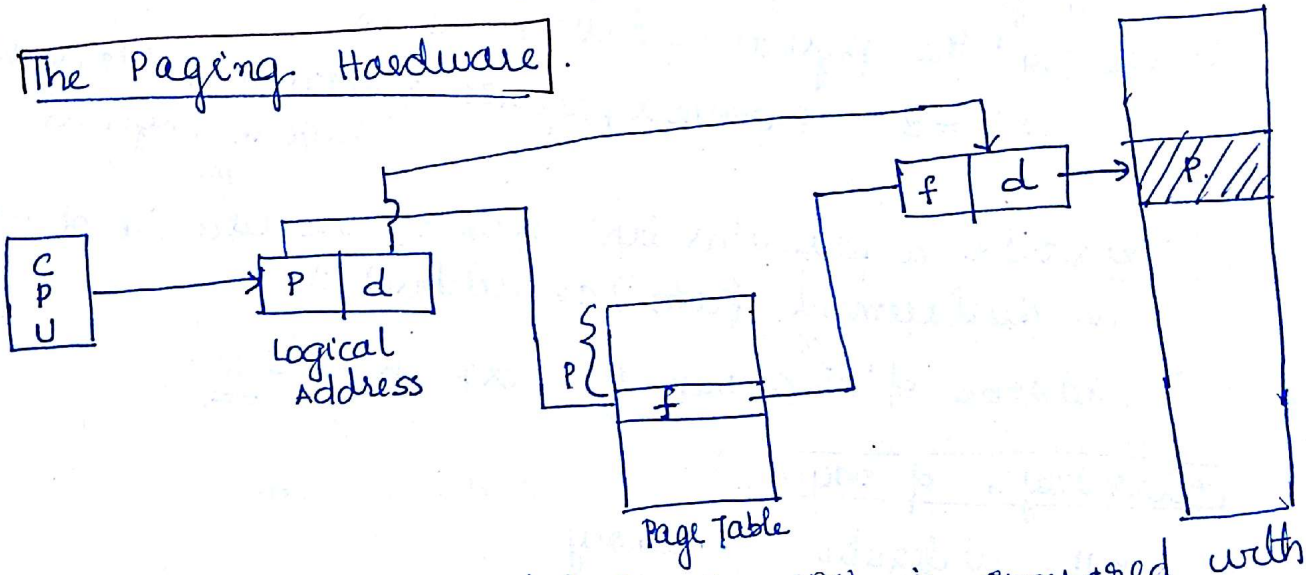


Page Table



Physical memory

The Paging Hardware.

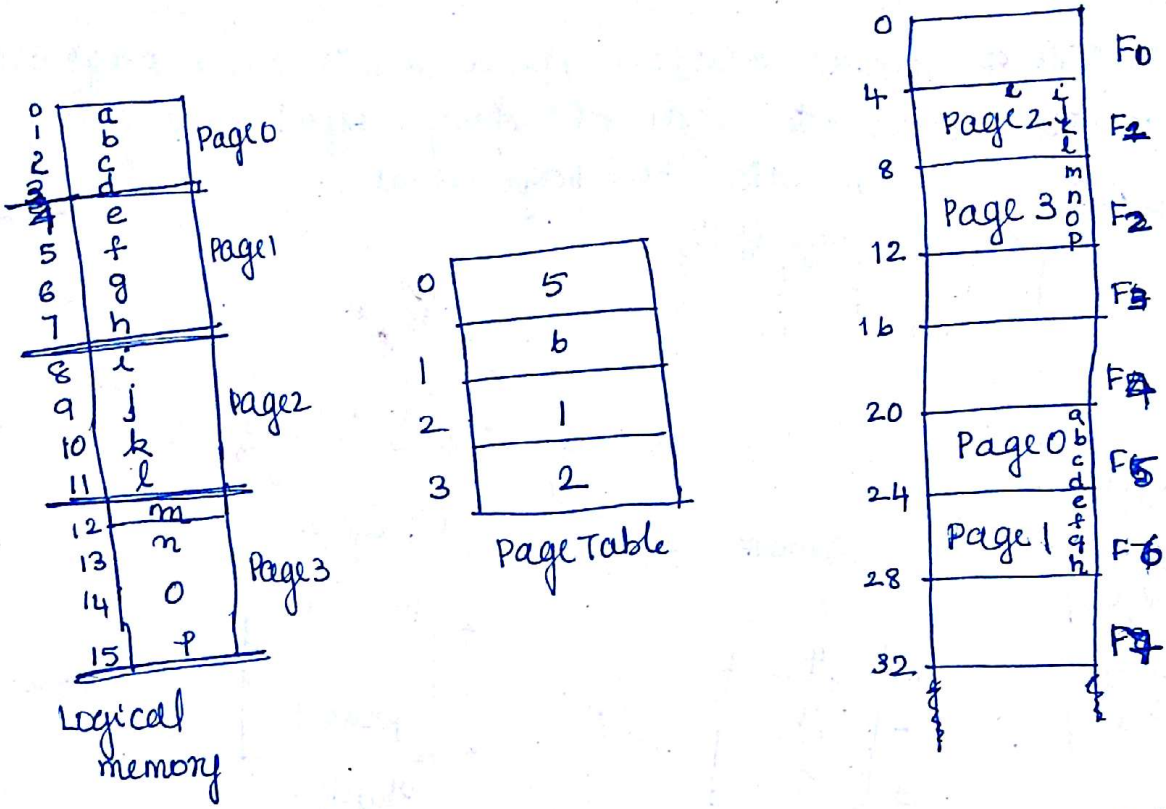


The page no generated by the CPU is compared with the page table to retrieve the frame number. Along with the frame number the offset is added to give the required address in physical memory

Example :-

Consider page size = 4 bytes

physical mem size = 32 bytes



Now page table entry indicates page3 is mapped to frame 2
So we find the physical address as follows.

$2 * 4 = 8$ Frame no x page size ⇒ Starting address of page in physical memory

To retrieve a particular byte say 0, we add the offset ie displacement from base address ie 2

address of '0' in memory $= 2 \times 4 = 8 + 2 = 10$

Advantages of paging:

* Better utilization of memory
* No external fragmentation

Disadvantages:-

* Internal fragmentation can be more

eg: process size = 72766 bytes
page size = 2048 bytes
Requirement in terms of pages:

```
        35
2048 | 72766
       6144
      -----
       11326
       10240
      -----
        1086
```

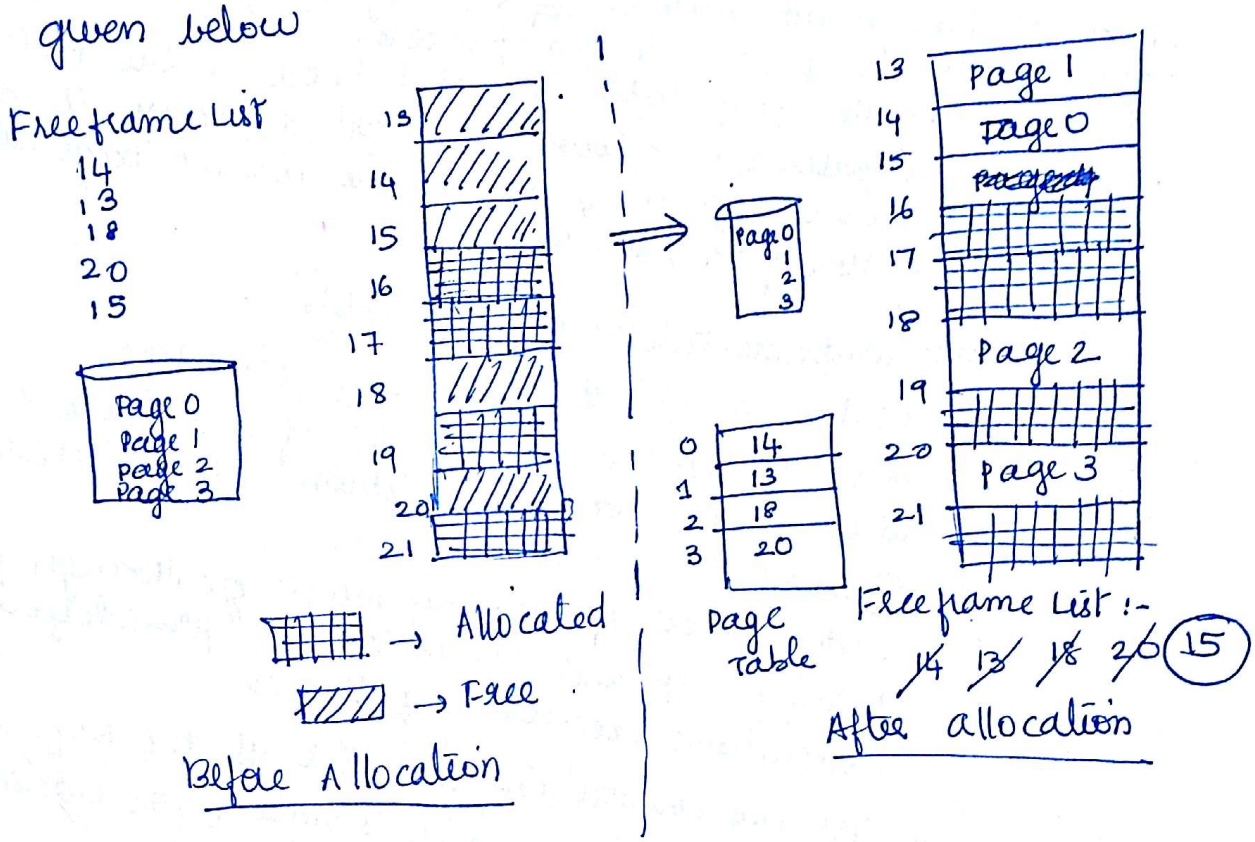
35 pages + 1086 bytes
So we require 36 blocks

2048 − 1086 = 962 bytes within 36th block is free

It is desirable to have small page sizes as we can reduce the internal fragmentation problem. But it can increase the overhead involved in each page table entry tracking and the size of page table increases.

Free frame allocation :-

If a process requires n pages → n frames must be allocated to it. For this the OS should keep track of available free frames and maintains in a data structure ie freeframe List. consider the example given below



The user view of program is continuous whereas in reality the user program is scattered throughout the physical memory. The mapping is done by the address translation hardware by the OS, and these complexities are hidden from the user

To know the details about which frames are allocated and which are free, the os maintains another data structure called frame table

The frame table has one entry for each physical frame indicating whether it is free or allocated to which page of which process

If the user makes an I/O system call, the os should map the same correctly on to the correct physical frame. For this it maintains a copy of instruction counter and register contents. Paging therefore increase the context switch time.

HARDWARE SUPPORT:

- Each OS has its own method of storing page tables.
- Most OS allocate page table per process.
- A pointer to the page table is maintained in the PCB
- when the dispatcher is allowed to start a proces, it must reload the user registers & define the correct page table from values stored in user page table.

Hardware implementation of page table

(i) can be done as set of dedicated registers.
- These registers are made up of high speed logic to make the paging address translation efficient.
- Every access of memory must go through the paging map and all these are privileged instructions executed by the OS.
- This implementation is good if the size of the page table is very small. (256 entries)
- For machines which permit 1 million entries for page table, this mechanism cannot be used.

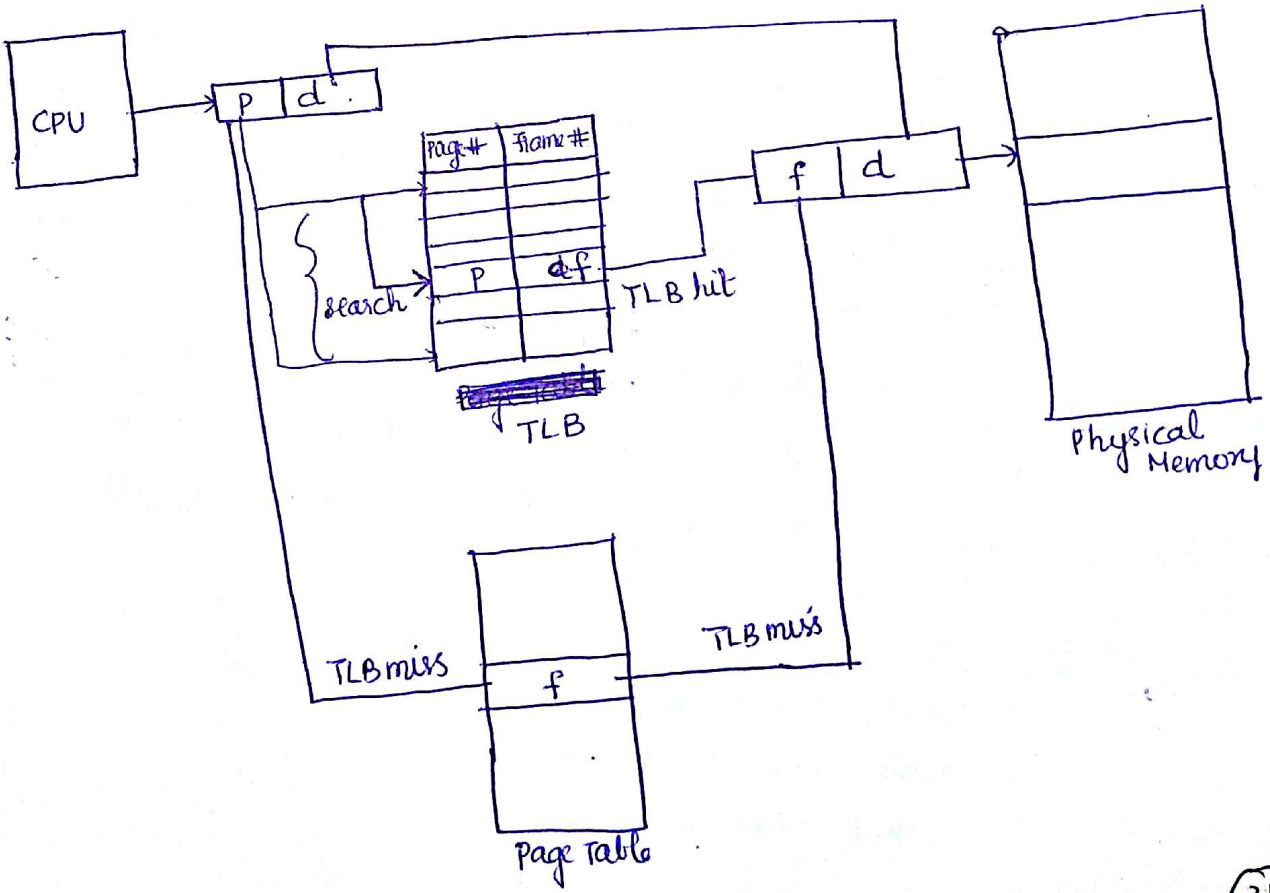
(ii) Page ~~table~~ Table kept in main memory and accessed using PTBR (page Table base register) which points to the paging Table.

changing page table required only changing the value of PTBR (page table Base register) substantially reducing context switch time

Problem with this approach is the time taken to access a user memory location. If we want to access user location i, we must access page table in main memory, retrieve the frame no where i is stored and then access the desired location. Thus the main memory is accessed twice (slowed down by factor of 2)

Sol:- use of TLB (Translation LOOK ASide Buffer)

- TLB – associative high speed memory
- consists of 2 parts a key (tag) and value.
- when a page is referred, it is compared with the keys and on match, the value in the field is retrieved.
- Here the search is fast, but hardware is expensive and hence size of TLB is very small (64 – 1024 entries).



Working of TLB in Paging

- The TLB contains only a few of the page Table entries
- when a logical address is generated by the CPU, first the TLB is searched to find the match of page no. If present, (TLB hit) then we retrieve the frame number from the TLB add offset and retrieve the physical address.
- when a particular page is not found in the TLB buffer (TLB miss) and then the page number to frame mapping is searched in the page table in the main memory.
- TLB implementation
  - TLB can only hold limited entries – page replacement policies should be thought of
  - Some TLBs are wired down (kernel code) – meaning they remain in TLB forever.
  - Some TLBs have Address specific IDs which allows identification of each process uniquely and we check the protection of the process address space.
  - This is done by ensuring that the ASID of requesting process is matching the ASID of virtual page. If not matching, then it is considered as TLB miss. TLB has to be flushed (erased) to ensure that the next executing process does not use the wrong translation information
  - The percentage of time a particular page is found in TLB is known as hit ratio (expressed as %)
  - TLB miss is calculated as 100 – hit ratio.

Numerical

Given hit ratio = 80%.

Time taken to search TLB = 20ns

Time taken to access page Table in main memory = 100ns

me taken to access the desired byte in main memory = 100ns
find the Effective Access Time (EAT)?

Effective Access Time is calculated based on the probability of finding the page in the TLB or in main memory

$$EAT = \frac{80}{100} \times (20+100) + \frac{20}{100} \times (120+100)$$

$$= \frac{80}{100} \times 120 + \frac{20}{100} \times 220$$

$$= 140 \text{ nanoseconds}$$

If hit ratio = 98%.

$$\frac{98}{100} \times 120 + \frac{2}{100} \times 220$$

$$= 122 \text{ nanoseconds}$$

Protection

How to ensure protection in a paged environment.

Memory protection in paged environment is accomplished by protection bits associated with each frame. These bits are kept in the page table.

One bit can define a page as rw page or a read only page.

Every reference to memory goes through page table. The OS has to check

① whether address is valid

② If valid, check the protection bits depending upon the kind of access request

③ eg: a read only page has the read only bit set. If any process requests to write on it, it has to be trapped to OS.

Valid / Invalid bit is used to check whether it is legal page or not.

valid bit = 1 (valid)
= 0 (Invalid).

The OS will check the reference is valid or not. It also checks the protection bits before it grants the access.

Eg) 14 bit address space (0 to 2^{14}) (0 to 16383)
Consider a program which should use only 0 to 10468
If page size = 2KB, ie 2×1024 ($2^1 \times 2^{10}$). then we have page references from 0 to 5 (5×2048) = 10240 as valid references. But page 5 till 10468 should be valid and remaining should not be given access.

$2048 \overline{)10240}^{5}$

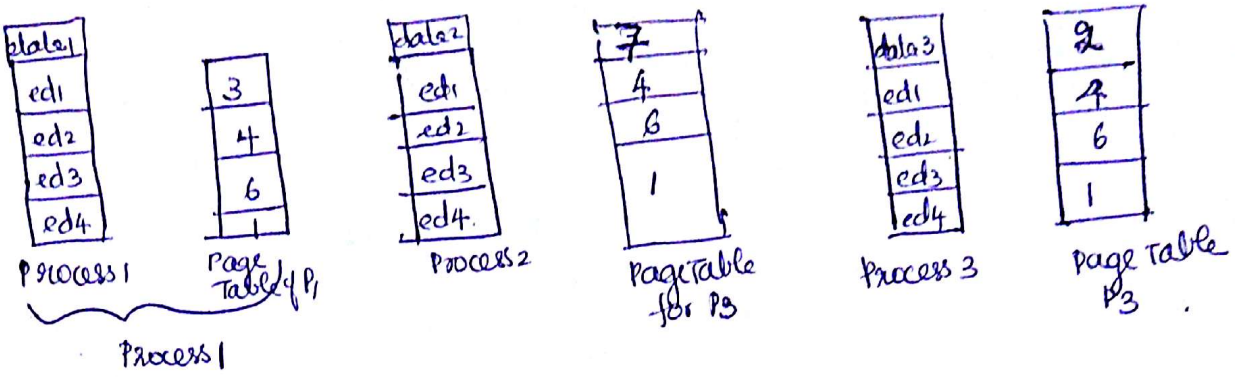
10468 − 10240 = 228 bytes of page 5 → valid access
remaining bytes → invalid.

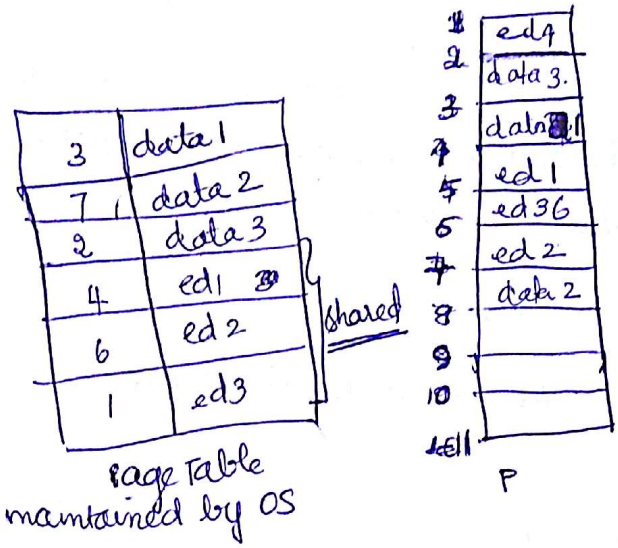
Hence page 5 we give valid access but we check for protection bits. one more way to solve the problem is to keep track using PTLR (Page Table Length Register)

Shared pages

- One of the main advantage of paging is sharing of common code is possible.
consider a system that supports 40 users, each of them requires 150 KB of code and 50KB data space. If we are not sharing then $150 \times 40 + 40 \times 50 = 8000$ KB is required. If we allow sharing, $150 + 40 \times 50 = 2150$ KB of memory is only required.
Compilers, runtime libraries, database systems etc only one copy need to be maintained.
Since paging supports the sharing of the code, the memory can be used for loading other programs

PTO





//only one copy to be maintained - and this copy is shared
But OS should take care of logical address space such that common code page no to frame no mapping is same

Structure of page Table

Some of the common techniques for structuring the page Table

(i) Hierarchical paging
(ii) Hashed page Table
(iii) Inverted Page Table.

(i) Hierarchical Paging

Most modern computer systems support large logical address space (2^{32} or 2^{64}). In such an environment, ~~the~~ if size of each page is 4KB ie 2^{12}
Then no: of entries in page Table in a 32 bit representation will be $2^{32}/2^{12}$ or $(2^{20}) \rightarrow 1$ million entries.
Assume each entry takes 4 bytes each process needs 4MB for page Table alone.

Two disadvantages:-

* Size of page Table is Huge
* More time taken to search page Table for entry

Solutions:

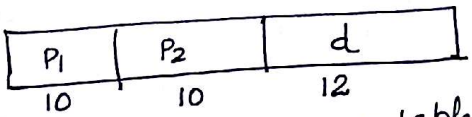
① Divide the page table into smaller divisions.

We achieve divisions in various ways

* 2 level paging
* 3 level paging
* 4 level paging etc.

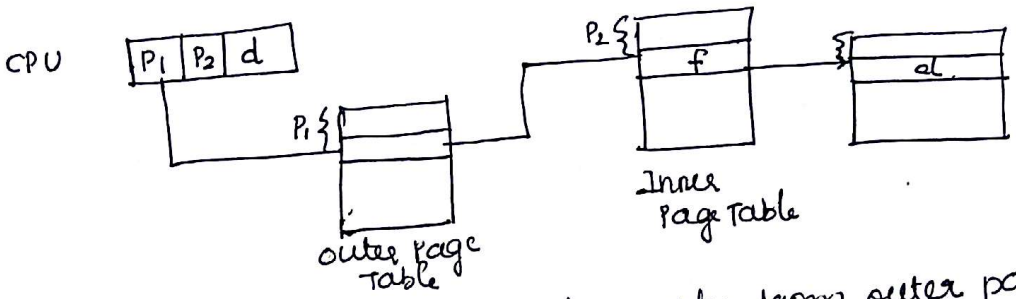
Consider a 32 bit logical address space.

logical address is divided as 20 bits and offset consisting of 12 bits. Further the 20 bits are again divided into 10 bits each as shown below



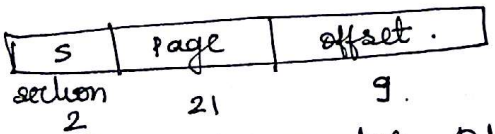
where

P_1 is the index to outer page table and P_2 for the inner page Table



Because address translation works from outer page Table towards inward it is also known as forward - mapped page Table.

VAX architecture supported 2 level paging -



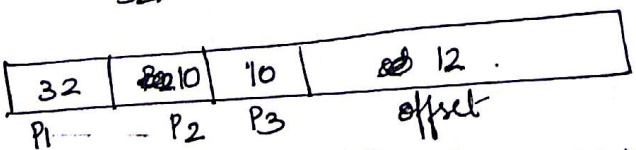
s designates section number, p index to page table & offset.

In a 64 bit machine, even 2 level scheme makes the search of page table tedious. Hence we go for 3 level scheme

64 bit → 42 10 12 → outer page Table: 2^{42} entries which is very huge

32 10

- Hence we further divide to 4 level paging.



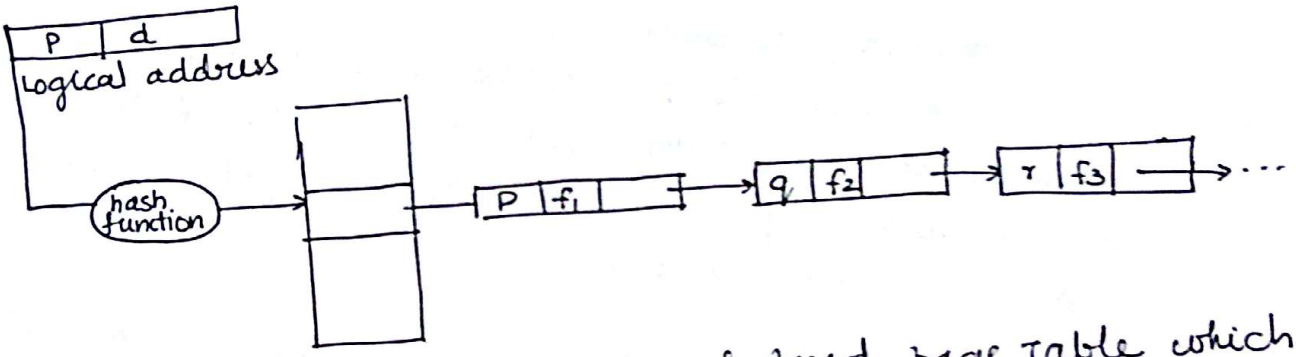
Here outer page table 2^{32} bytes in size

SPARC architecture uses 3-level scheming and 32 bit

Motorola 68030 architecture supports 4 level paging

ii) Hashed Page Tables

- A common technique used for handling pages whose size is greater than 32 bits.
- Here for every generated virtual address, a hash function is applied which points to an entry in hash table.
- For every entry in hash table, there is a linked list of elements containing 3 fields - pageno, frameno & next pointer.
- whenever the virtual address is hashed on to hash table, we traverse the linked list to find a match for the page number and we retrieve frame no from there.

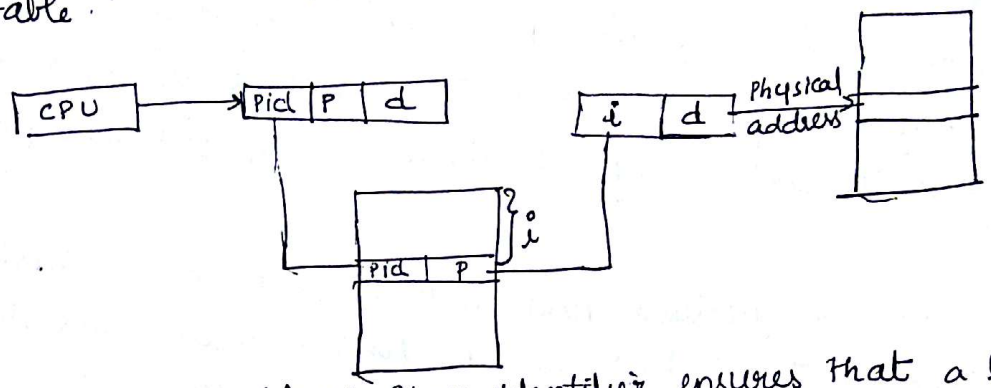


A variation of this scheme is the clustered page table which are similar to hashed table, but each entry refers to a set of pages rather than a single page.
Clustering page tables are particularly useful for sparse address spaces, where the memory references are scattered & non-contiguous throughout memory.

(iii) Inverted Page Table

Each process has associated page table and all the page used by process with mapping on to physical frame are managed by the OS. But the time taken to search such a page table containing millions of entries is so huge. Thus we use inverted page table.

In inverted page table, for each real page in main memory, an entry is maintained in the page Table.
Each entry consists of virtual address of page stored in that real memory location, with information regarding which process owns the page.
Thus only one entry per real page is maintained in the page table.



storing the address space identifier ensures that a logical page for a particular process is mapped to the corresponding physical frame.

< process id , pagenumber , offset)

Process id assumes role of address space identifier. The page no is located at an offset equal to the frame no. Thus we find the frame no & add offset to get physical address in main memory

Adv:
Decrease amt of memory required to store page table.

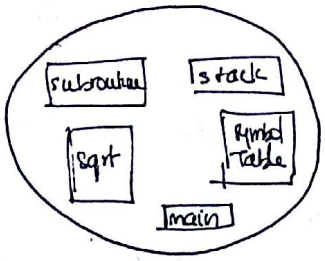
Disadv
① Time taken to search — because inverted page table is sorted as per physical address, but look up happens as per logical address

To alleviate the problem, we can make use of hash table to limit the search-time. TLB will be consulted before hash table and inverted page Table.

② Difficulty in implementing shared pages
- Inverted page table contains only one entry for each physical frame. Hence it will be difficult to implement shared pages which contains multiple virtual address. Sol:- Allow only one virtual address to hold mapping

SEGMENTATION

- A memory management technique where there is a clear seperation of logical address and physical address.
- The users view of a program is as containing various segments like main program, subroutine, error handling function etc.

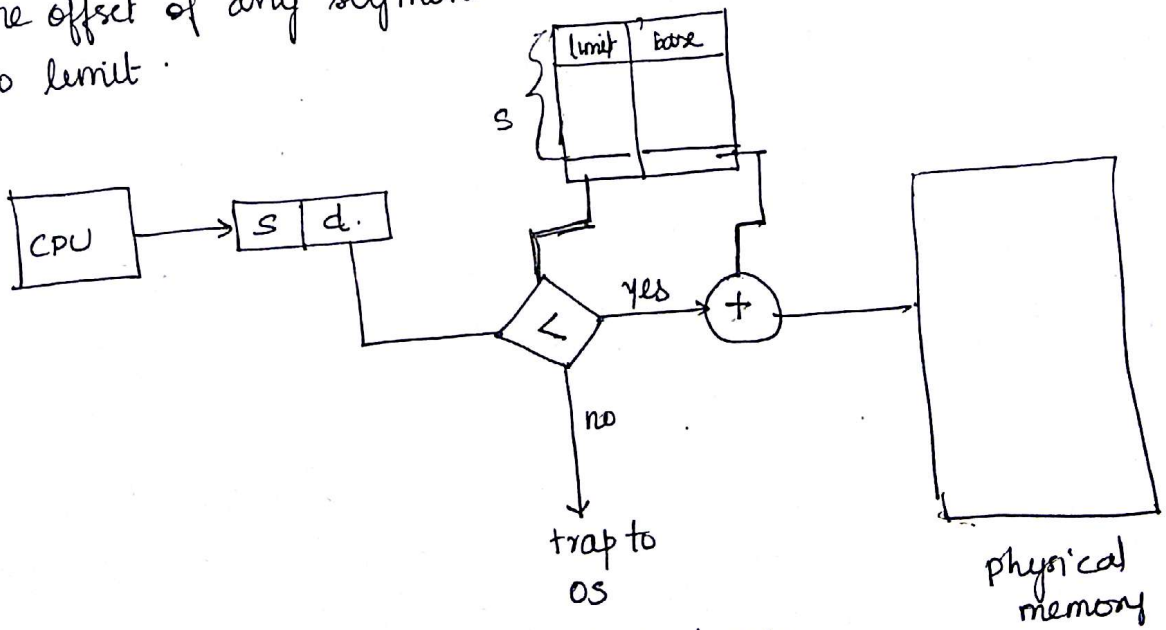


User view

- In this scheme a logical address space consists of various segments of varying sizes.
- virtual address consists of segment no, offset

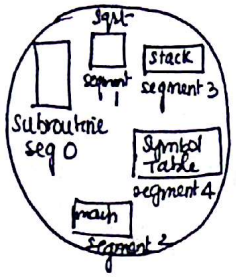
Hardware

The mapping of segment to actual physical memory is done with the help of segment Table.
Each entry of the segment Table consists of base and the limit. the segment base contains the starting physical address, whereas limit specifies length of segment.
The offset of any segment will be any value between 0 to limit.

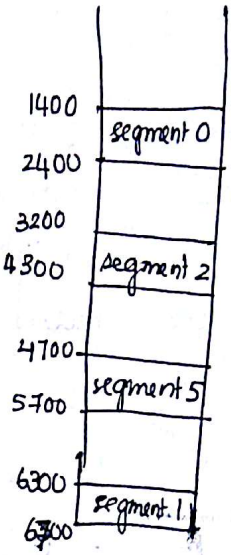


Segmentation Hardware

Example of segmentation



	limit	base
0	1000	1400
1	400	6300
2	1100	3200
3	1000	4700.



Suppose the segment chosen is Segment 2

Its base address : 3200 & limit is 1100.

Whenever a virtual address comes the offset d is compared with limit say (200 < 1100) ? yes then base address is added to offset to get desired byte. Else the OS is intimated about the illegal accss.

CHAPTER 9 VIRTUAL MEMORY MANAGEMENT

Virtual memory is a technique that allows the execution of processes that are not completely in main memory. One major advantage of this scheme is that programs larger than main memory also can be executed. Virtual memory abstracts main memory into extremely large, uniform array of storage, seperating logical memory as veived from physical memory. Virtual memory also allows processes to share files easily and to implement shared memory.

To implement vertual memory, it is not easy and it substantially decrease performance if used carelessly.

BACKGROUND:-

For a process to be executed the program should be loaded into main memory. There are two ways of implementing it

(i) Entire logical address to be loaded into main memory. This can be done with the help of dynamic loading, but requires careful programming techniques and precautions by the programmer.

(ii) Load only the modules to be executed rather than the whole process itself. This takes off the limitation that the size of the program to be less than the physical memory

Programs often have code to handle unusual conditions – error conditions which occurs seldom and mostly in practice this code is never executed.

- Arrays, lists, tables etc are often allocated more space than it actually occupies.
- Certain options or features of a program are used very rarely.

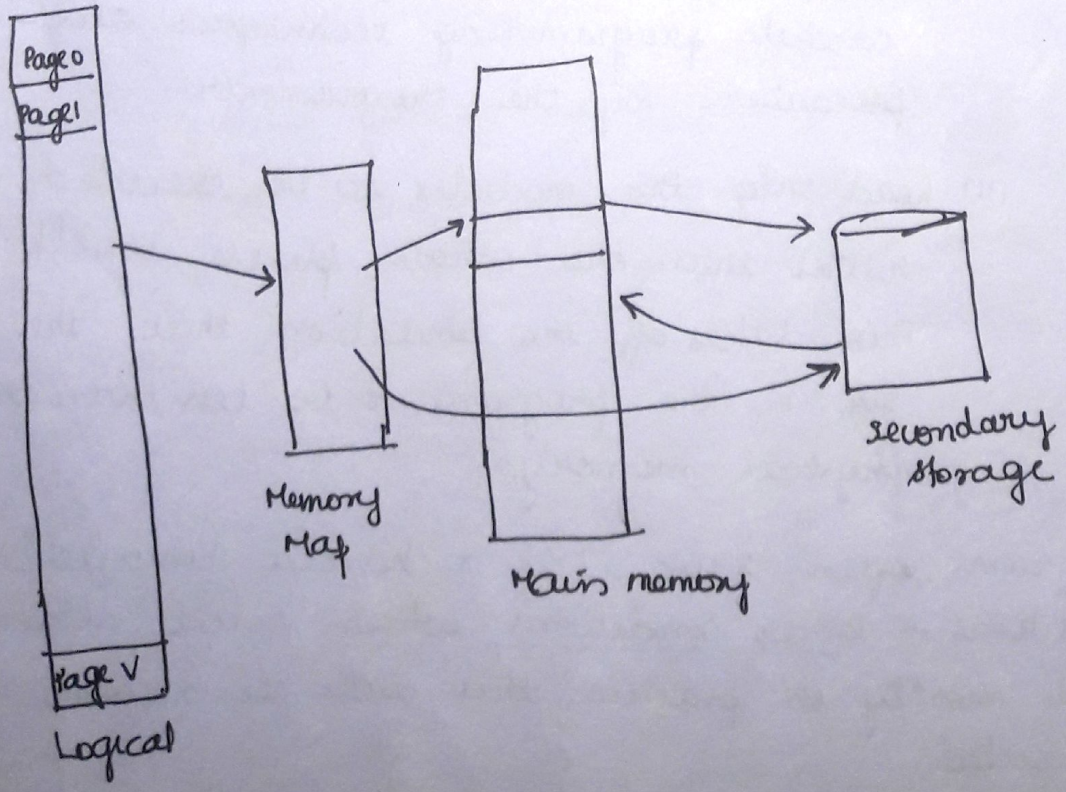
This technique of executing a program, when it is partially loaded into memory is known as virtual memory

Advantages of virtual memory

* A program is no longer constrained by the size of main memory. Users will be able to write programs without limiting or knowing size of the main memory.

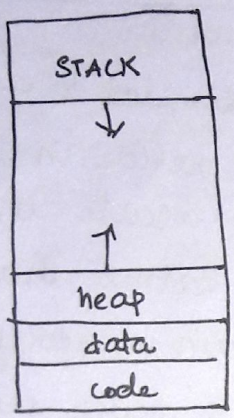
*) concurrency can be improved as some other programs can be loaded at the same time - thereby increasing CPU utilization and thorough put, but with no increase in response time & turn around time

* Less I/O needed to load a swap each user program into memory, so each program would run faster.



Virtual memory involves seperation of logical memory from physical memory. It makes the programmers work much easy as they can write programs without bothering about size of physical memory

The virtual address space of a process refers to the logical or virtual view of how a process is stored in memory.



Virtual address space.

- We allow stack to grow downward and heap to grow upward.
- The large blank space between the heap & part is part of virtual address space.
- Holes (blank space where pages can be swapped in.

virtual address spaces that include holes are also known as sparse address space.

Benefits of ~~logical~~ virtual address space:

(1) seperation of logical and physical address space
(2) system libraries can be shared among various processes through mapping of shared object into virtual address space.
(3) Virtual memory enables processes to share memory. Virtual memory allows one proces to create a region of memory that can be shared with another space
(4) Virtual memory can allow pages to be shared during process creation using fork() system call.

DEMAND PAGING

Demand paging is a strategy which loads only some pages initially as they are needed. This technique is known as demand paging and is commonly used in virtual memory systems

With demand-paged virtual memory, pages are only loaded when they are demanded during program execution. Pages that are never accessed are never loaded into physical memory.

A demand paging system is similar to a paging system with swapping where processes reside in secondary memory (disk) and when we want to execute the process we load it into main memory. Rather than swapping the entire process into the main memory, we use a lazy swapper ie a page is never swapped into main memory until it is called / referenced.

A swapper manipulates entire process whereas a pager is concerned with the individual pages of a process.

BASIC CONCEPTS

- When a process is to be swapped in, the pager has to decide which pages to be brought into main memory. Only necessary pages are brought into main memory as we can reduce swap time and efficiently we can use physical address space.

- For this we need to distinguish between those pages which are already there in main memory and those residing on disk. The valid - Invalid bit scheme in the page table can be used for this purpose.

  if valid bit = 1 ⇒ both legal and present in main memory
  valid bit = 0 ⇒ either illegal or the page is currently not in main memory.

- If all the pages required for execution are there in main memory, the process execution proceeds normally.
- But if the process tries to access some page that is not currently in main memory, then it causes a page fault trap. This trap is to intimate the OS about its failure in bringing a page required for execution.

STEPS IN HANDLING PAGE FAULT

(1) whenever a page reference comes, we check the internal table in the PCB, whether the page reference is a valid or invalid memory reference.

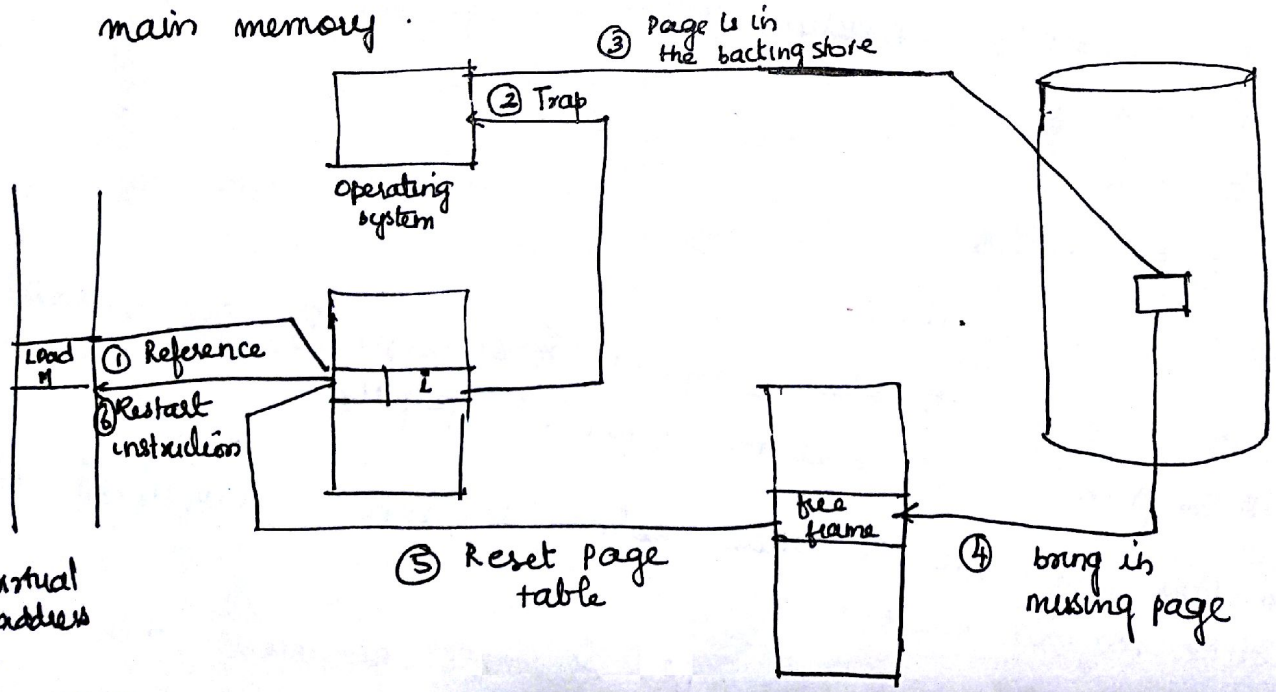
(2) If the reference is invalid, we terminate the process. But if it was a valid page and if its is not present in main memory, we now page it in

(3) we find a free frame by taking one from the free frame list if available

(4) we schedule a disk operation to read the desired page into newly allocated frame.

(5) when disk read is complete, we update the internal page table entry to indicate that page is in memory.

(6) we restart the instruction that was interrupted by the trap. The process can now access the page as though it was in main memory.



When we start executing a process with no pages in main memory we call it as pure demand paging. The rule of pure demand paging is that a page is never brought into memory unless required by the process. Sometimes for executing one instruction, several page faults might be required which can result in unacceptable system performance. But programs tend to have locality of reference which helps in reasonable performance with demand paging

HARDWARE SUPPORT FOR DEMAND PAGING

(1) Page Table :- This table has the ability to mark an entry invalid through a reference of valid/invalid as well as protection bits

(2) Secondary memory :- Holds pages that are not present in main memory. A high speed disk, also known as swap device, and the section of disk used for this purpose is known as swap space.

One of the crucial requirement with demand paging is its ability to restart any instruction after a page fault. Because we save the state (Registers, condition code, instruction counter) of the interrupted process when the page fault occurs, we must be able to restart the instruction exactly the same place and state except that now page is available in main memory.

Consider the following eg.

(1) Fetch & decode instruction ADD
(2) Fetch A
(3) Fetch B
(4) Add A and B
(5) Store sum in C.

if we fault at while trying to store in (5), the entire steps from 1 to 5 will be repeated again.

Another challenge is that what happens to modified values when paging happens

Sol :- Microcode computes and attempts to access both ends of both blocks. - if any modifications have to happen it

should happen before page fault occurs

Sol ②:- use temperory registers to hold overwritten locations values. If there is a page fault, old values are written back into memory before the trap occurs.

Performance of Demand Paging.

Effective access time for a demand paged environment is given as

$$(1-P) * ma + P * \text{page fault time}.$$

(P) → probability of page fault

(1-P) → probability of no page fault

ma → memory access Time. $(0 \le P \le 1)$.

Sequences of actions following page fault

① Trap to OS
② save user registers and process state
③ Determine that it was a page fault
④ check if the page reference was legal & find location of page on disk
⑤ Issue a read from the disk to free frame
   (a) wait in a queue until read request is serviced
   (b) wait for device seek / lateney time
   (c) begin transfer of the page to a free frame
⑥ while waiting, allocate CPU to some other process.
⑦ Receive interrupt from I/O sub system (disk).
⑧ save registers & process state of process in execution
⑨ Determine interrupts was from disk
⑩ correct page table & other information such that desired page is in main memory.
⑪ wait for CPU to be allocated to this process again
⑫ Restore the user registers, process state and new page table and resume the interrupted

PTO

page sharing.

instruction.

Summarizing there are 3 major components for the page fault service time

① Time taken to service the page fault interrupt
② Read in the page
③ Restart the process.

① & ③ task can be reduced with careful coding (100 μs)
Page switch time around 8 ms for entire page fault service time

Problem:-

if average page fault service time is 8 ms, memory access time is 200 ns, then effective memory access time in nano seconds is _____.

let p → probability of page fault

$$\begin{aligned} Eat &= p * 8 \times 10^6 + (1-p)200 \\ &= 8000000\,p + 200 - 200\,p \\ &= 200 + 7999800\,p \end{aligned}$$

NOTE:- Effective access time is directly proportional to page-fault rate.

Two challenges in demand paging

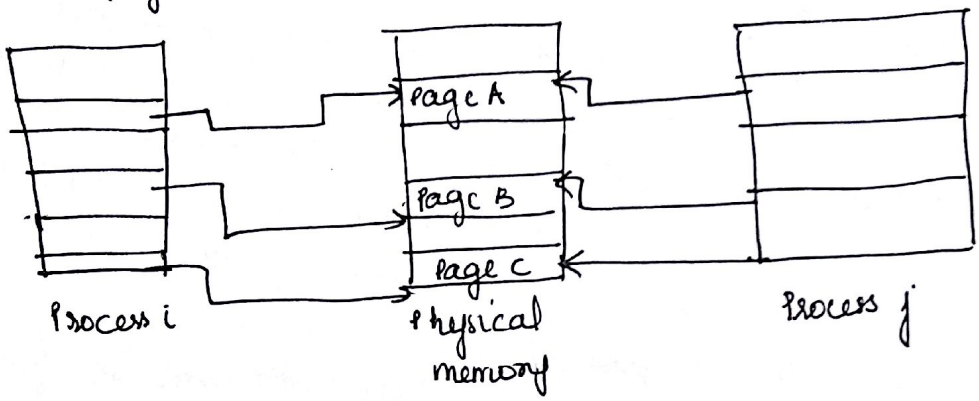
(1) Reduce the page fault rate
(2) Handling the overall use of swap space.
- option1 – Disk I/o to swap space is faster than to file system
- option2 – Demand pages from file system initially, but write on to swap space.

copy on write

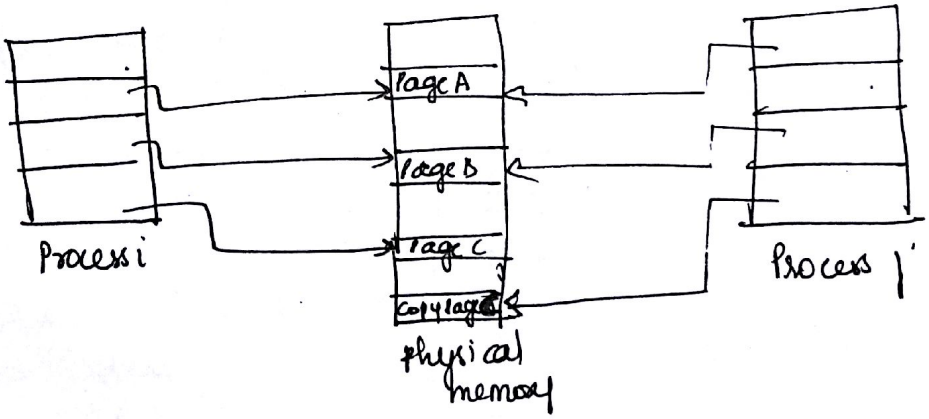
Demand paging is used when reading a file from disk into main memory. when a process creation happens on a fork() system call, it can initially by pass the need for demand paging using a technique similar

to demon process page sharing.
This technique provides rapid process creation and minimizes the number of new pages that must be allocated to the newly created process.

fork() system call creates a child process as duplicate of the parent. All the pages of parent are duplicated for the child, but as many subprocesses can be created at the same time, a variation technique known as copy on write which works by allowing the parent & the child process to initially share some pages. Some of the pages are marked as copy on write pages meaning that if either process writes to a shared page, a copy of the shared page is created.



If the child process wants to modify a page marked as copy on write, then only it will create the copy of the page. The child process then modifies in the copied page & not the page belonging to parent.
All unmodified pages are shared by parent & child.
All OS like windows XP, Linux, solaris etc uses this technique



After process j modifies page C

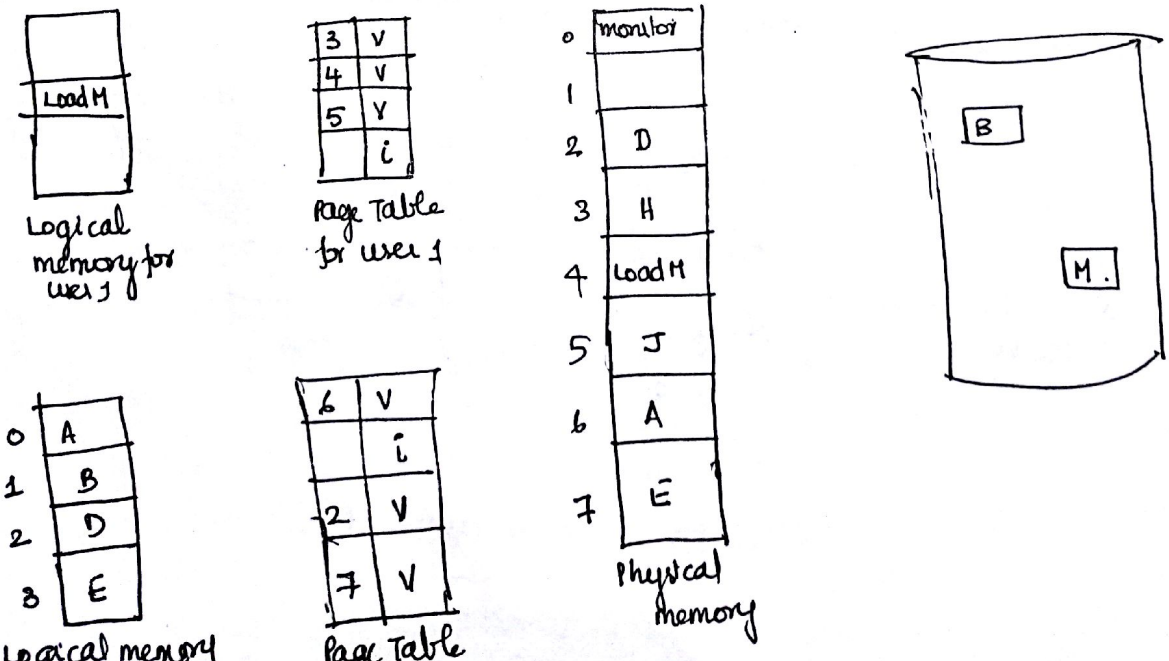
when a copy on write page demand comes, a free frames has to be found from pool of free frames.. Operating systems typically fill these demand using a technique known as zero fill on demand. Zero fill on demand pages have been zeroed out before being allocated, erasing previous contents.

Several versions of UNIX and solaris, makes use of a technique known as virtual memory fork() (Vfork()). Here the parent process is suspended while the child executes. Since Vfork() does not support copy on write pages, the parent will be able to retrieve only modified pages by child once it resumes. This technique is used only when there is not much modification

PAGE REPLACEMENT

Whenever a page is first referred by a process, page fault occurs. Thus it is very important to decide which all pages to be kept in main memory while execution. Keeping the entire process, might even load routines which might never be called, thus causing wastage of resource. To increase the CPU utilization we can increase the degree of multiprogramming. If we overallocate process into memory, there will not be free frames available and then the new pages have to be swapped in after swapping out some of the pages which are not required in main memory (page Replacement).

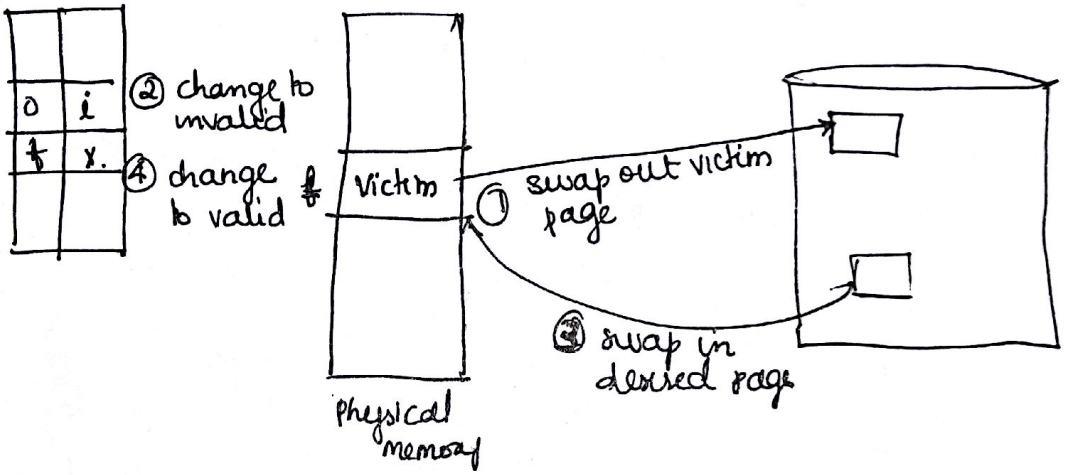
Need for page Replacement



The need for page replacement arises when there are no free frames available to swap in desired page. If there are no free frames available, then we have to make a frame free by swapping out a page that is not currently required.

PAGE FAULT SERVICE ROUTINE WITH PAGE REPLACEMENT

1. Find location on disk abt desired page
2. Find free frame
   a) If free frame available – swap in the desired page into it
   b) If no free frame is available – use page replacement algorithm to select victim frame.
   c) Write victim page on to disk and swap in the new page.
3. Once the desired page is read in, modify the page table entries



Page Replacement.

If there are no free frames, then 2 page transfers (swap out) and swap in are required, which effectively doubles page fault service time and the effective access time accordingly.

We can reduce this overhead by using a modify bit (dirty bit). The modify bit is set only if a page is modified, else it remains the same. Thus if the page is not modified ie modify bit = 0, then we don't have to write the page on to disk if it is already there. this technique applies only for read only pages.

This scheme reduces the eat and page fault source time by one-half.

With demand paging a page is never brought into memory until called for. If again, the page brought into is selected as victim page, again page fault can occur. Thus with demand paging we must have frame allocation algorithms and page replacement algorithms.

The string of memory references is called a reference string. For a given page size, we need to consider only the page no rather than the entire address.

Eg) Let memory address sequence be
0100, 0432, 0101, 0612, 0102, 0103, 0104, 0101, 0611, 0102, 0103, 0104, 0101, 0610, 0102, 0103, 0104, 0101, 0609, 0102, 0105

If page size = 100 bytes
the reference string can be found as.
01, 04, 01, 06, 01, 01, 01, 01, 06, 01, 01, 01, 01, 06, 01, 01, 01, 01, 06, 01, 01

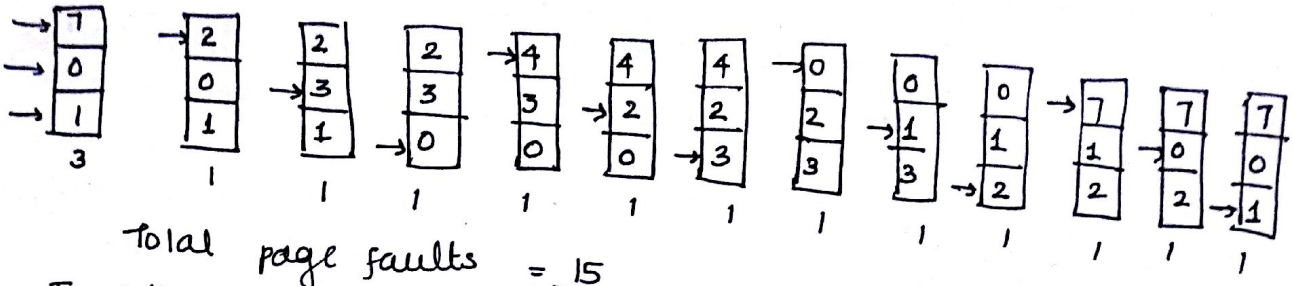
To execute a page replacement algorithm, we need to know
① page no
② No of free frames available.
③ Logic for selection of victim page.

(1) FIFO PAGE REPLACEMENT ALGORITHM

- simplest page replacement algorithm
- Replacement based on when a page is brought into memory. The page which was first brought into memory is selected as victim page.
- we create a FIFO queue to hold all pages in memory we replace a page at the head of queue and when we bring in a page and put at the tail of the queue.

consider the reference string : 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

Available frames : 3



Total page faults = 15

Initially, none of the pages 7, 0, 1 in memory → Hence 3 Page faults
To bring in page 2 → Replace page 7 as it arrived first
To bring in page 3 → Replace page 0.
To bring in page 0 → Replace page 1 and so on.

Advantages

(1) Easy to understand and program.
(2) Fair mechanism

DISAdvantages

(1) Performance not always good as it does not consider whether
(2) page will be required in near future
(2). can cause Beladys Anomaly.

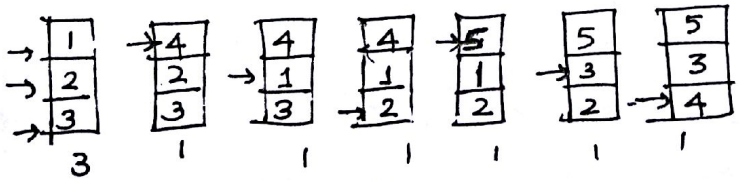
Beladys Anomaly

A condition where the no: of page faults increase when the no: of free frames increases, contrary to the concept that more available frames lesser the page fault

eg) Consider the reference string

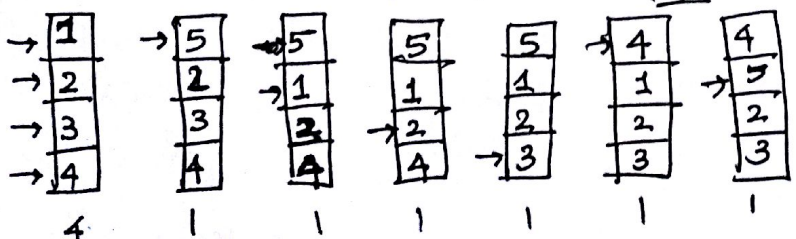
1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

Case 1: No of available frames = 3



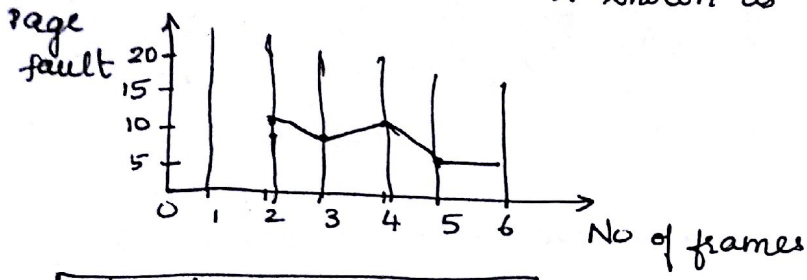
No: of page faults = 9

Case 2 :- consider available free frames = 4



Nos of page faults = 10

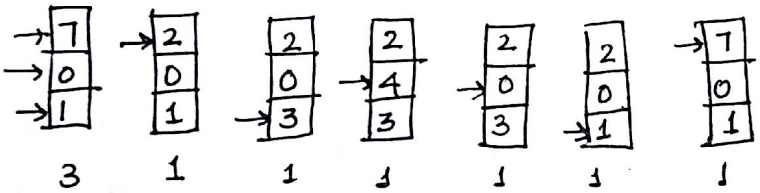
Here the page fault increased as the no: of available frames increases - This is known as Beladys Anomaly.



Optimal Page Replacement

- It has the lowest page fault rate of all algorithms.
- Doesnot exhibit Beladys Anomaly.
- called as OPT or MIN
- Replace the page that will not be used for the longest period of time.

- Eg: 7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1 as reference string
  no: of available frames : 3



To bring in Page 2 – see future reference – the victim page is one that will not be used for longest time out of 7, 0, 1, 7 is not refered in near future – so replace 7

To bring in page 3 – 0 & 2 are required in immediate future – so choose 1 as victim

To bring in page 4 → out of 2, 3, 0, 0 is not refered for long time

To bring in page 0 → 4 is not used in future

To bring in page 1 → out of 2 0 3 3 is not refered in immediate future

To bring in 7 → Replace 2

No of page faults = 9

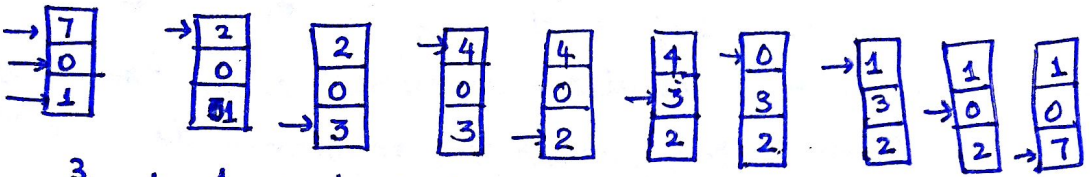
Disadvantage:- Difficulty to predict future references.

LRU page Replacement Algorithm

- In this algorithm we use the recent past as an approximation of near future.
- Here we replace a page that has not been used for a longer period of time (LRU → Least Recently used)
- For implementing this algorithm, we should know when a page was last accessed

Eg) Reference string :- 7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0

No of frames :- 3



3 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 = 12

12 page faults

Advantage :-

- Better than FIFO (Results in Lesser Page fault)
- Better than optimal, where future knowledge was required.
- No Beladys Anomaly

Disadvantages :-

Require substantial Hardware assistance -
Difficulty in implementation with respect to an order for the frame defined by last time of use.

Two implementations are feasible.

(1) Counters :- Each page table entry - time of reference field. Everytime the page is referred, a counter is incremented for clock and the contents of clock register are copied to the time of use field in the page Table entry for that page.

We then replace a page with the least reference value or least time value.

- Requires search of entire page table to find the LRU and a write to memory for each memory access.
- Requires the times to be maintained even if the page tables are changed.

(ii) Stack :- Another implementation technique where the most recent page is kept in top of stack and the least recently used page in the bottom. Since the entries required to be removed from middle, better to implement as doubly linked list with a head & a tail pointer. Removing a page and putting it at top requires changing of pointers - hence it is expensive, but there is no search cost as the tail pointer points to the bottom.

LRU Approximation Page Replacement :-

In LRU approximation algorithms, we make use of reference bit. Initially the reference bit value for all pages is cleared to zero and if a page is referred then it is made to one. Thus we know which all pages are referred but not the order. Reference bits are associated for each page in the page table.

ADDITIONAL REFERENCE BITS ALGORITHM.

We can gain some more knowledge using additional reference bits. Every page table entry can be made to hold a 8 bit value to indicate the references made. Initially bits = 00000000. when a reference is made, it is set to 1. Thus 10110001 is a most recent page than 01100110. The page with the least value

is chosen as the LRU page.
The no: of bits can be varied of course, and is selected based on hardware available to make the updation as fast as possible. In the reference bits method extreme case entire page is set 0 and then page is given a second chance.

Second chance Algorithm

- The basic algorithm of second chance algorithm is a FIFO replacement algorithm.
- when a page is chosen as the victim, we inspect its reference bit - If value = 0 → we proceed to replace the page.
  = 1 → we give the page a second chance & proceed to next page in queue to check its reference bits.
- Thus a page which is given a second chance is not replaced until all the pages which are given a second chance are replaced.
- Thus a page which is most frequently will have the referenced bit set again, so that it will ~~be~~ not be replaced.

Implementation of second chance Algorithm

- Use circular queue.
- The pointer indicates which is the next page to be replaced.
- when a frame is needed, it will advance till it finds a page with reference bit = 0. once a victim page is found, it replaces that page.
- The worst case for this algorithm is the complete circular queue traversal, in the process of finding victim page when all bits are set as 1.

ENHANCED SECOND CHANCE ALGORITHM

- In this algorithm, we classify the pages into 4 classes based on the values of two bit values - modify bit and reference bit

(1) (0,0) ⇒ modify bit = 0 and reference bit = 0.
This is the best page to be replaced - not modified, not referenced

(2) (0,1) ⇒ modify bit = 0 and reference bit = 1
A page which is not modified, but referenced

(3) (1,0) ⇒ Modify bit = 1 and reference bit = 0
A page which is modified - has pending I/O, but not refered

(4) (1,1) ⇒ modify bit = 1 and reference bit = 1
A page which is modified, has pending I/O as well as referenced recently. Worst page to be replaced.

In this algorithm, rather than checking only the reference bit, we check the class group - we replace the page which is encountered in the lowest non empty class.

NOTE:- The circular queue have to be scanned so many times before we decide the victim.

COUNTING BASED PAGE REPLACEMENT

In this class of algorithms, we keep a counter of the number of references that have been made to each page. There are two widely used schemes

(1) The least frequently used (LFU) page Replacement

- Here we replace a page with the smallest count.
- Smallest count indicates - least no: of times a page is refered (Not actively used)

Problem arises when a page is heavily used initially & later not referenced at all. Since it has large count, it will not be chosen as the victim.

Solution :- Shift the count right by 1 bit at regular intervals forming an exponentially

(2) The MFU (Most frequently used) :- based on the argument that page with smallest count is probably just now brought into and hence it is yet to be used. Hence replace a page with largest count.

PAGE BUFFERING ALGORITHMS:

- Some systems keep a pool of free frames. when a page fault occurs, a victim frame is chosen as before. The desired page is then read into free frame from the pool before the victim is written out. By doing this we are not waiting for the page to be written out.

- An expansion of this idea is to maintain a list called as modified page list. This list keeps track of the pages which are modified and whenever the paging device is idle, it writes out the pages to the secondary storage. This scheme increases the probability of finding a clean page.

- Another modification is to keep a pool of free frames and to remember the mapping of pages to frames. If the frame contents are not modified, that page can be ~~directly swapped out~~ chosen as victim page. No write out to disk happens as the page is clean (not modified)

VAX/VMS system uses FIFO page buffering algorithm.

UNIX OS some versions use second chance algorithm

Applications & page Replacement

Applications accessing data through OS virtual memory perform worse if OS does not provide ~~or~~ buffering.

eg) Database applications provides its own I/O & memory management as application knows better than OS which does for general purpose.

eg2) Datawarehouses which perform massive sequential disk reads, followed by computations and writes, MFU is preferred over LRU.

Because of this problem, the OS gives special programs the ability to use a disk partition as a sequential array of logical blocks, without any file system data structures. This is called as raw I/O.

Raw I/O bypasses all file system services, such as file I/O demand paging.

Allocation of Frames

If we have 93 frames and two processes what are the various ways of allocating frames.

(1) Equal allocation

In this method, the total no: of available frames are equally divided between the processes. So in the above example, each process gets equal share ie $93/2 = 46$ frames each.

Advantage :-

- Fair mechanism
- easy to implement

Disadvantage

- If one process requires 50 and other requires only 25, with equal allocation it is not possible

(2) **Proportional allocation**

- In proportional allocation the total no: of available frames are proportionally divided

eg: Total available free frames : 15

P_1 requires : 10

P_2 requires : 8

No: of frames allocated to P_1 : $\frac{10}{(10+8)} \times 15$

$= \frac{10 \times 15}{18} = \frac{150}{18} = 8$ frames

No of frames allocated for P_2 : $\frac{8}{(10+8)} * 15$

$= \frac{8}{18} \times 15 = \frac{120}{18} = 6$ frames

ie No of frames allocated $a_i = \frac{s_i \times m}{S}$

a_i = allocation for Process i
s_i = Requirement of process i
S = Total requirement of all process
m = no: of available frames

(3) **Priority allocation**

Here based on the priority of the process, the frame allocation will be done. The higher the priority, a process has more frames will be allocated to it.

Advantage

- High priority process might be allocated frames quickly so that priority is considered

Disadvantage

- Low priority process can suffer from starvation - a problem where the low priority process are always deprived of frames.
- Not fair allocation.

(4) **Global Vs Local Allocation**

We can classify page replacement algorithms into two broad categories

(1) Global Allocation

Here frames are allocated in a global way ie Pages of other process can also be replaced to make free frames. ie one process can take frames from any of the free frames available throught system even it is in other process space

(2) Local Allocation

Here the frames are chosen only local to the process. If need comes, we first check in the frames available local to the process. Here we do not globally replace page neither chose free frame allocated to some other process.

Global allocation can cause starvation of process, if the same low priority process are chosen again & again. With this allocation, a process ~~can't~~ might take only frames allocated to other process. A process will not be able to control its page fault rate

In Local allocation, a process have to choose frames local to it only. It can control the page fault rates by running a suitable algorithm which can minimize the page fault rate

But when considering the throughput of system as a whole, higher throughput is achieved using global allocation rather than local allocation.

When should we allocate the frames?

It is always better to allocate atleast minimum no: of frames required for the execution of process to decrease the page fault raised by process.

The minimum no: of frames is defined by computer architecture and maximum no: is defined by the available physical memory

Non Uniform memory Access (NUMA)

In systems with multiple CPUs, a given CPU can access some sections of memory faster than others. These performance differences are caused by how CPUs are connected on the system. Systems in which memory access time vary significantly are collectively known as Non uniform memory access systems (NUMA)

Managing which page frames are stored at which locations significantly affect performance in NUMA systems. The goal is to have memory frame to be allocated as close as to the CPU (ie which causes minimum latency). Solaris solves the problem by creating an lgroup entity in the kernel to keep track of last CPU on which process ran.

THRASHING

If no: of frames for a low priority process falls below minimum number required by computer architecture, we must suspend that process. Else it results in a condition called Thrashing

Thrashing is a condition where more time is spend in paging rather than in execution

If a process does not have enough frames,

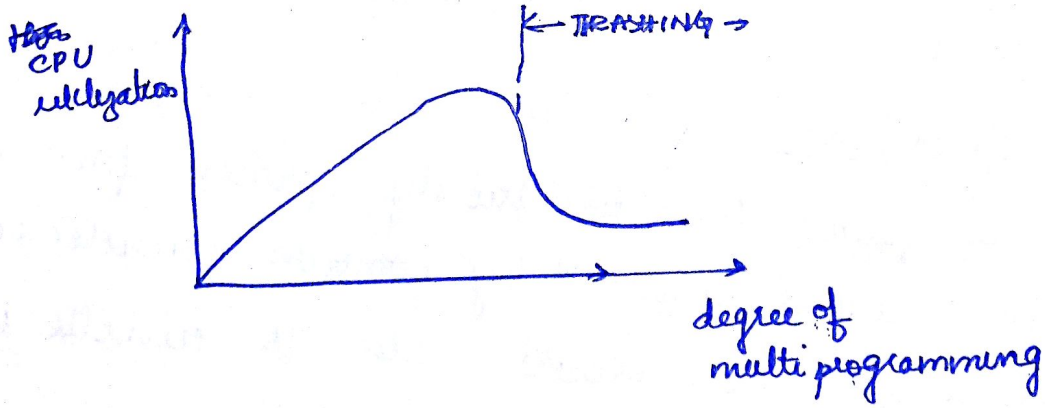
it results in page faults and it will call page replacement algorithm. when the OS sees the CPU to be idle, it tries to increase the degree of multiprogramming by adding more process. Those process also tries to ~~flag~~ compete for frames resulting in more performance degradation.

Cause of Thrashing

OS monitors ~~thes~~ CPU utilization. when CPU is idle due to paging activity it tries to add new process. A global page replacement algorithm is used, it replaces pages without regard to which process it belongs to. No process can control page fault rate in global replacement policies. This can result in increased paging activity and decreased CPU utilization.

The CPU scheduler sees decrease in CPU utilization and increases the degree of multiprogramming.

The new process tries ~~hee~~ to get started by taking frames from running process causing more page faults lowering the performance. This cycle causes the system throughput to be too low.



↑ Thrashing → ↑ page fault → ↑ Effective access time.

when thrashing is observed, we must decrease the degree of multiprogramming

ways to limit thrashing

① use local replacement algorithm than global
- No process can steal frames from other process
- can control page fault rate.
- problem not entirely solved as there will be a long queue of process waiting and effective access time will increase.

② knowledge about how many frames a process needs.

various techniques are used

(i) working set strategy :-

starts by looking at how many frames a process is actually using. This approach defines the locality of reference.

②. The locality model states that ~~when~~ a process moves from one locality to another, when it executes.

- Locality is a set of pages actively used together.

A program is composed of different localities which may overlap

- eg) when a function call happens it changes its locality & when it exits it moves its locality to calling fn locality.

Thus if we give enough frames to satisfy requests of one locality, page fault does not happen till change of locality happens.

Working set Model

- working set model is based on the assumption of locality.
- This parameter uses a parameter Δ, to define the working set window
- The idea is to examine the most recent Δ pages. The set of pages in the most recent Δ page references is the working set.
- If the page is active in use it will be in working set, and not used for long time - drop out from working set after Δ time units of its last reference.

Eg) $\Delta = 10$ Working set at time $t_1 = \{1, 2, 3, 6, 7\}$

t_2 $\{3, 4\}$.

- Accuracy of working set depends upon selection of Δ.

If Δ is too small → not encompass the entire locality

If Δ is too large - encompasses several localities.

- The working set size WSSi is computed as $D = \Sigma WSSi$,

D = Total demand for frames

WSSi → working set size

If D > no: of frames available - thrashing will occur

- once Δ is selected, the working set model is simple.
- The OS monitors working set required for each process & allocates enough frames to provide its working set size if there are enough and extra frames - another process can be initiated.

If the $\Sigma WSSi > m$, OS will select one process to be suspended & more frames are reallocated to other process.

- Working set model ~~prev~~ limits thrashing thereby increasing the CPU utilization

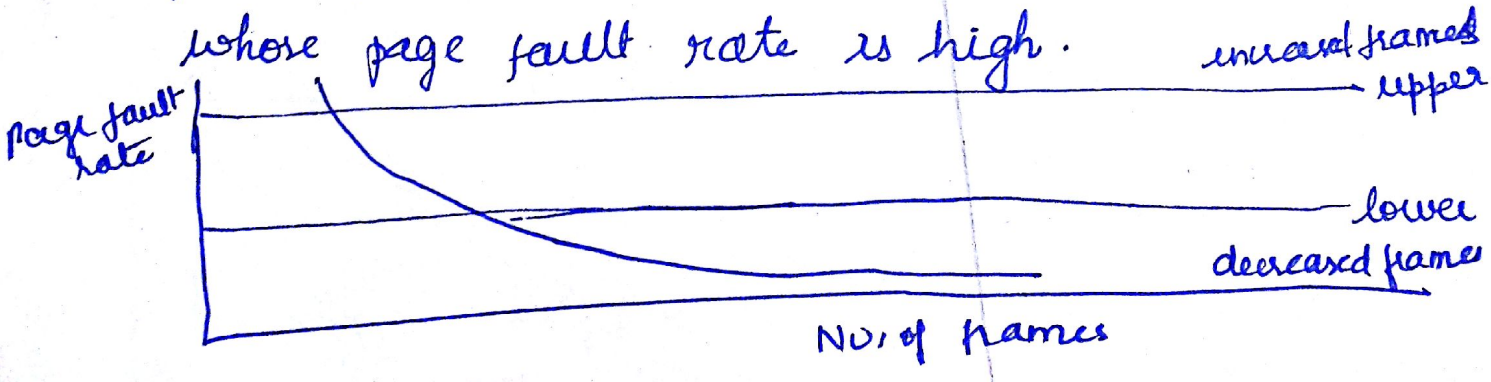
To improvise on the working set model, we can increase the no: of history bits & the frequency of interrupts

Page Fault Frequency

- Another strategy to control thrashing.
- Thrashing has high page fault rate and we attempt to ↓ page fault rate.
- when page fault rate is too high, it implies that process needs more frames and if it is too low, the process might have more frames
- An upper & Lower bound is established based on the desired page fault rate.

  If the actual page fault > desired page fault rate - we allocate the process another frame

  if the actual page fault rate < desired page fault rate then, we remove a frame from the process.
- Thus we directly measure & control page fault rate to prevent thrashing.
- Suspension of process happens if page fault rate increases & if no free frames are there. The freed frames are then distributed to processes whose page fault rate is high.

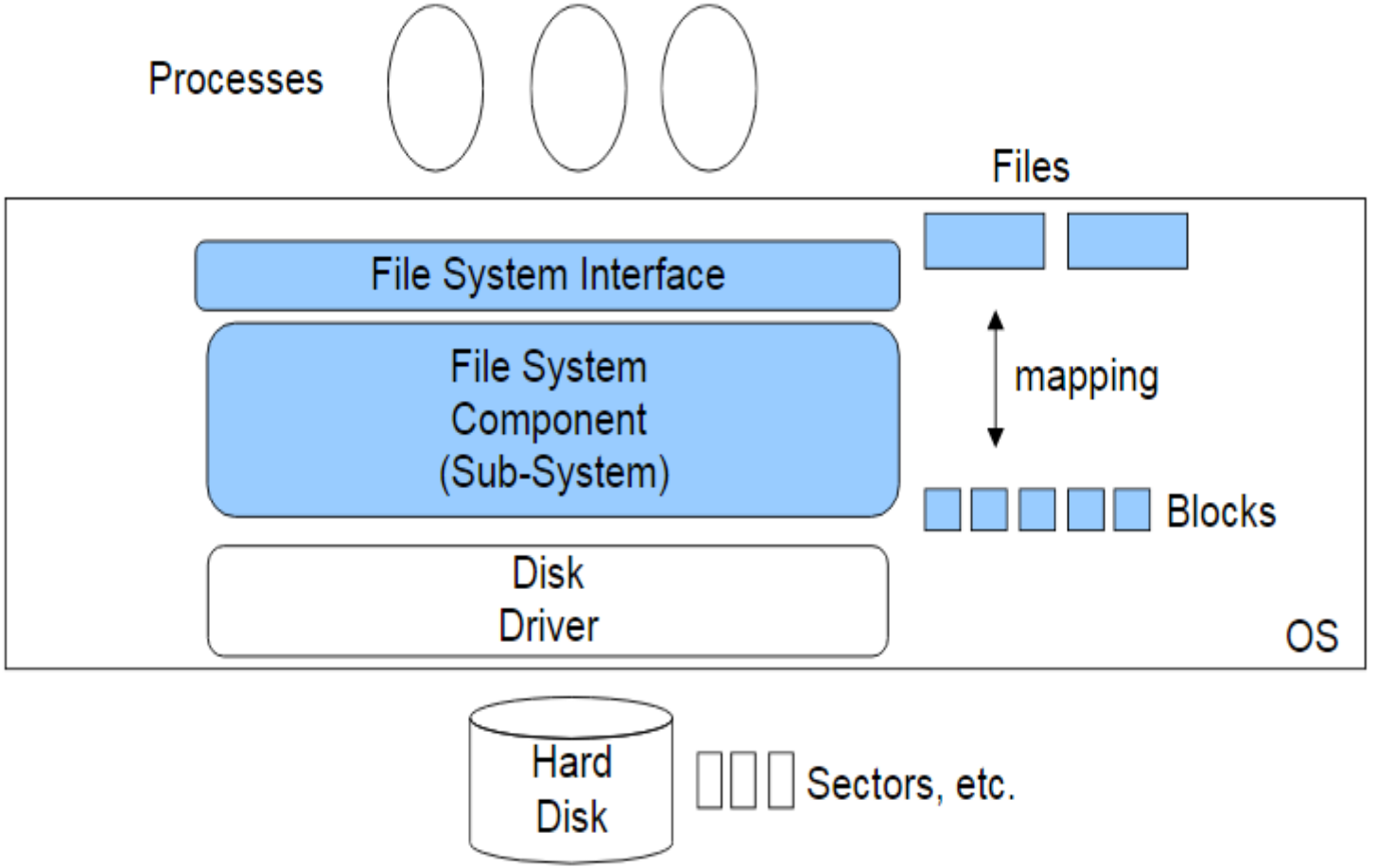


MODULE 5
File System Interface and Implementation

FILE CONCEPT

- Provides a mode of permanent storage of information.
- Information in a file is related in some way.
- File
  - A file is a named collection of related information that is recorded in a secondary storage
- Types
  - Data
    - numeric
    - character
    - binary
- Files can hold any kind of information
  - text, binary, application-defined structures, etc.

FILE CONCEPTS



FILE STRUCTURE

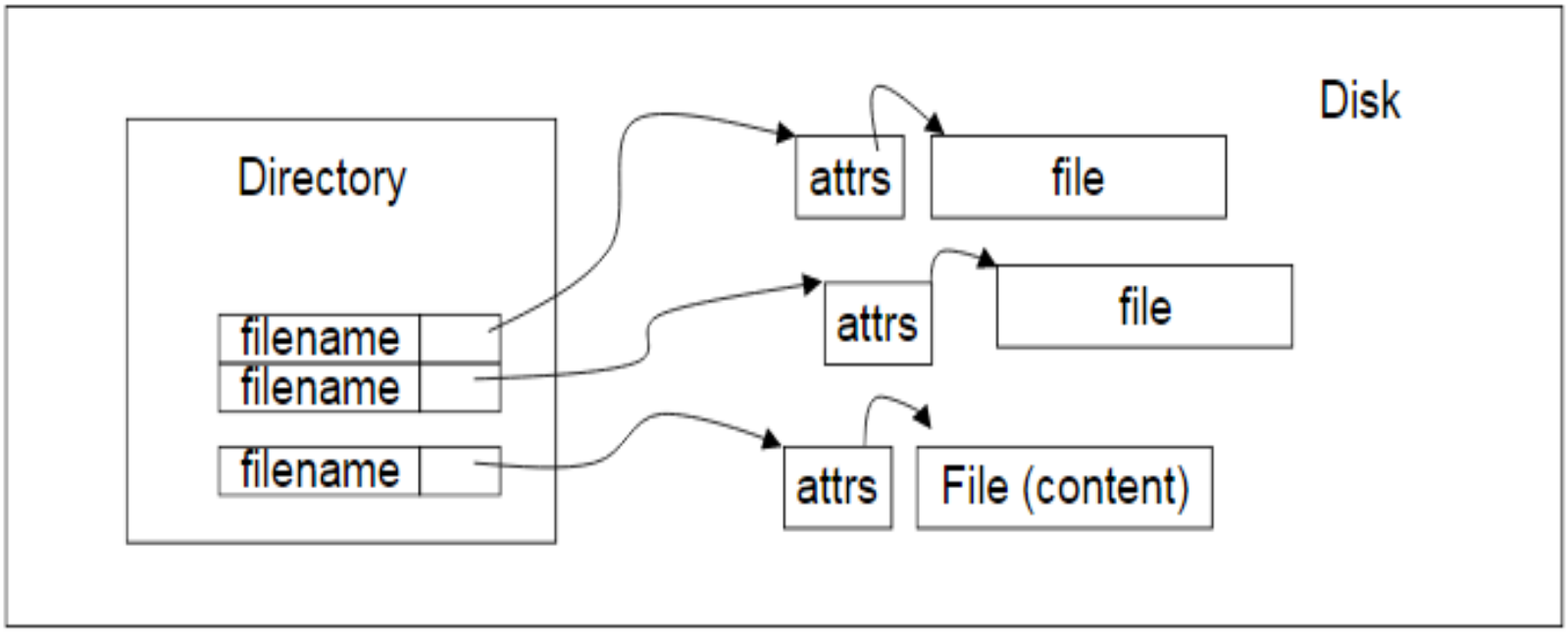
- None - sequence of words, bytes
- Simple record structure
  - Lines
  - Fixed length
  - Variable length
- Complex Structures
  - Formatted document
  - Relocatable load file
- Can simulate last two with first method by inserting appropriate control characters
- Who decides:
  - Operating system
  - Program

FILES AND DIRECTORIES

- There are two basic things that are stored on disk as part of the area controlled by the file system

  - files (store content)

  - directory information (can be a tree): keeps info about files, their attributes or locations



FILE ATTRIBUTES

- **Name** – only information kept in human-readable form
- **Identifier** – unique tag (number) identifies file within file system
- **Type** – needed for systems that support different types
- **Location** – pointer to file location on device
- **Size** – current file size
- **Protection** – controls who can do reading, writing, executing
- **Time, date, and user identification** – data for protection, security, and usage monitoring
- Information about files are kept in the directory structure, which is maintained on the disk

FILE OPERATIONS

- File is an **abstract data type**
- **Create**
- **Write**
- **Read**
- **Reposition within file**
- **Delete**
- **Truncate**

BASIC OPERATIONS

- *Open(F_i)* – search the directory structure on disk for entry F_i, and move the content of entry to memory
- *Close(F_i)* – move the content of entry F_i in memory to directory structure on disk
- **COMMON OPERATIONS**
  - **Appending – adding** new information to the end of an existing file
  - **Renaming –** an existing file

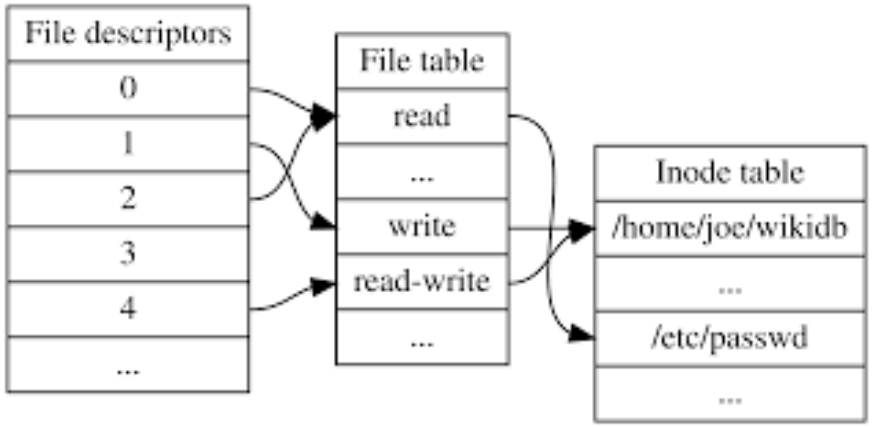
FILE OPEN

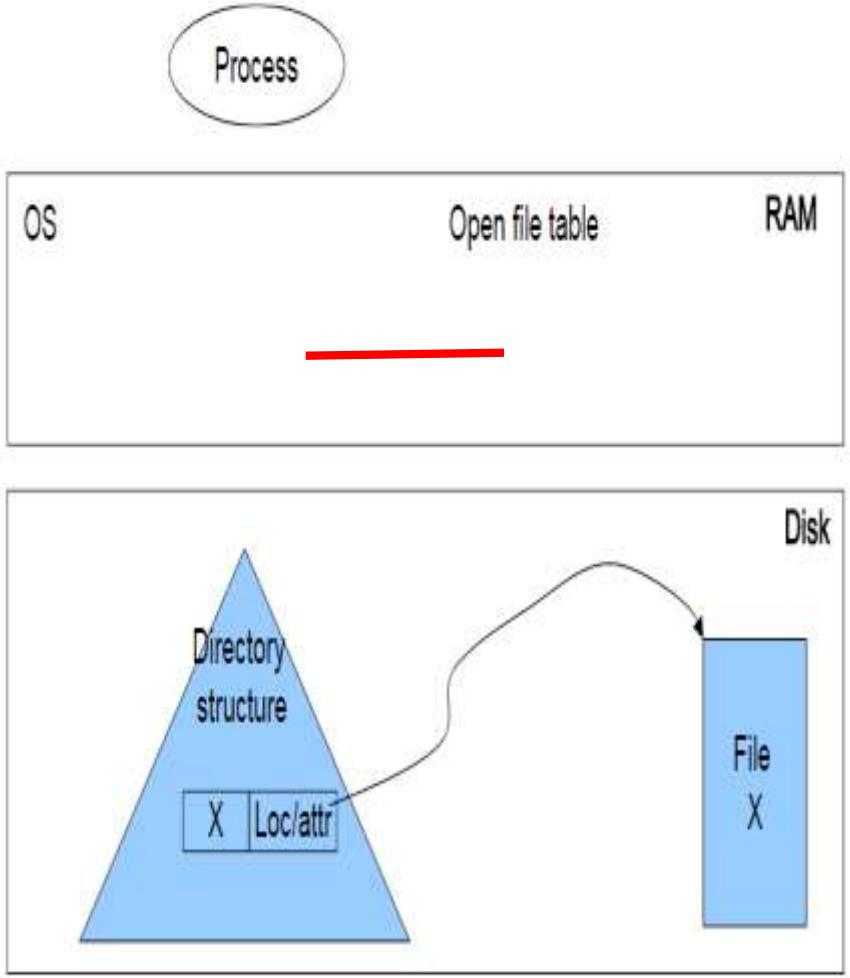
Most of the file operations involves searching in the directories for the entry associated with the file name

To avoid constant searching, **OS keeps a small table called Open-file table**

Open-File Table

- Contains information about all open files





- When a file operation is requested, the file is specified via an index into this table, so no searching is required
- When a file is no longer being actively used, it is closed by the process and the OS removes its entry from the open-file table
- Create and delete are system calls that work with closed rather than with open files

Open(), Close()

Implicitly done

Open()→ Done implicitly when the first reference to it is made

Close()→ Closed automatically when job or program that opened the file terminates

Explicitly done (using system call)

Open()→ takes a file name & search the directory, copying the directory entry into the open-file table

Can also accept access mode information – create, readonly, writeonly etc.

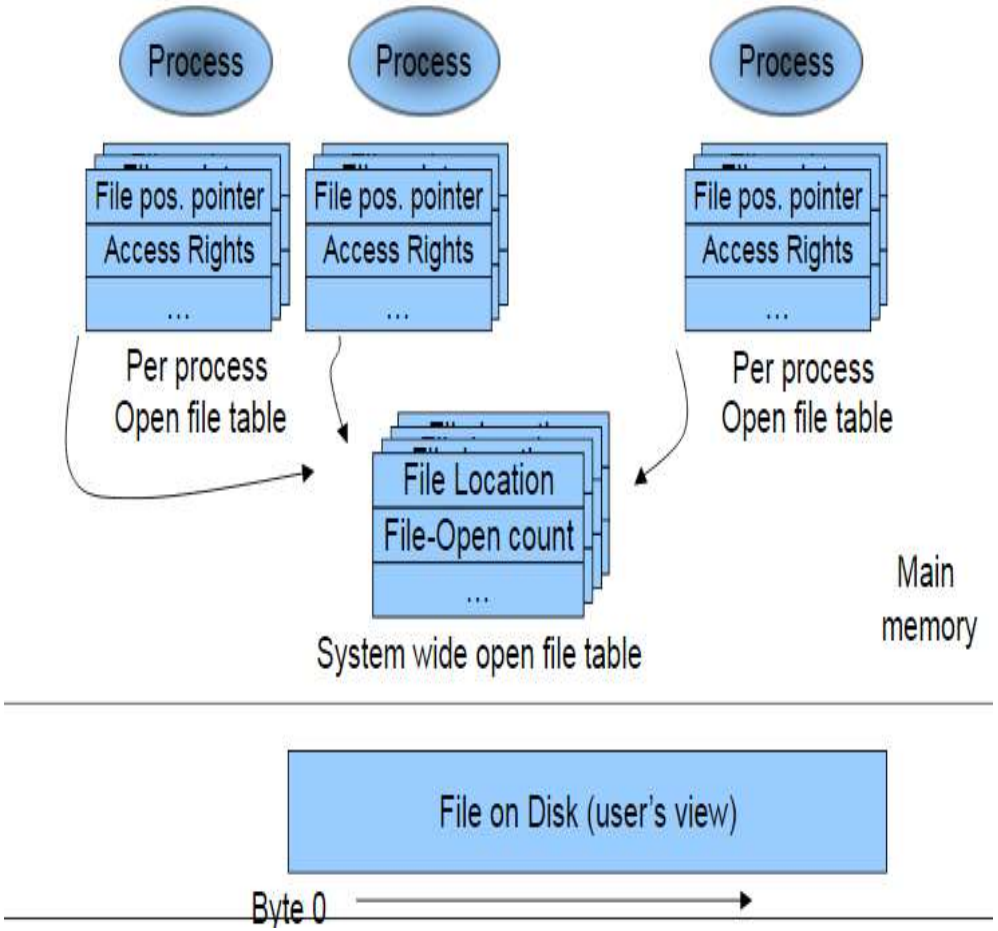
- **Multi-process Environment**
  - Several processes may open the file simultaneously
  - Several different applications open the same file at same time
- So, OS uses 2 levels of **internal table**

i) Per-Process Table

- Traces all files the process has opened
- Store information regarding the use of the file by that process
- Access rights, accounting information can also be included

ii) System-wide Table

- Contains process - independent information such as, location of file on disk, access dates, file-size



OPEN FILE

- Several pieces of data are needed to manage open files:
  - **File Pointer**: pointer to last read/write location, per process that has the file open
  - **File-Open Count**: counter of number of times a file is open – to allow removal of data from open-file table when last processes closes it
  - **Disk Location of the File**: cache of data access information
  - **Access Rights**: per-process access mode information

FILE LOCKING

- Some OS provide facilities for locking an open file (or section of a file)
- File lock allow
  - One process to lock a file and prevent other processes from gaining access to it
- Useful for files that are shared by several processes
  - Ex: System log file

- **Types of File Locks**
- **Shared Lock (akin reader lock)**
  - **Several process** can acquire the lock concurrently
- **Exclusive Lock (akin writer lock)**
  - **Only one process** at a time can acquire such a lock
- Some OS provides only exclusive lock
- **File-locking Mechanism**
- OS may provide either **mandatory or advisory file-locking mechanism**
- **Mandatory (locking scheme is mandatory)**
  - If a lock is mandatory, then once a process acquires an exclusive lock, the OS will prevent other process from accessing the locked file
- **Advisory Lock (Optional)**
  - The OS will not prevent other process from accessing the locked file
- **Windows adopts mandatory locking, UNIX adopts advisory locking**
- **Use of file locks:**
  - Enforces precautions
  - Ensures that two or more processes do not become involved in **deadlock** while trying to acquire file locks

FILE TYPES – NAMES, EXTENSIONS

file type	usual extension	function
executable	exe, com, bin or none	ready-to-run machine-language program
object	obj, o	compiled, machine language, not linked
source code	c, cc, java, pas, asm, a	source code in various languages
batch	bat, sh	commands to the command interpreter
text	txt, doc	textual data, documents
word processor	wp, tex, rtf, doc	various word-processor formats
library	lib, a, so, dll	libraries of routines for programmers
print or view	ps, pdf, jpg	ASCII or binary file in a format for printing or viewing
archive	arc, zip, tar	related files grouped into one file, sometimes com-pressed, for archiving or storage
multimedia	mpeg, mov, rm, mp3, avi	binary file containing audio or A/V information

FILE STRUCTURE

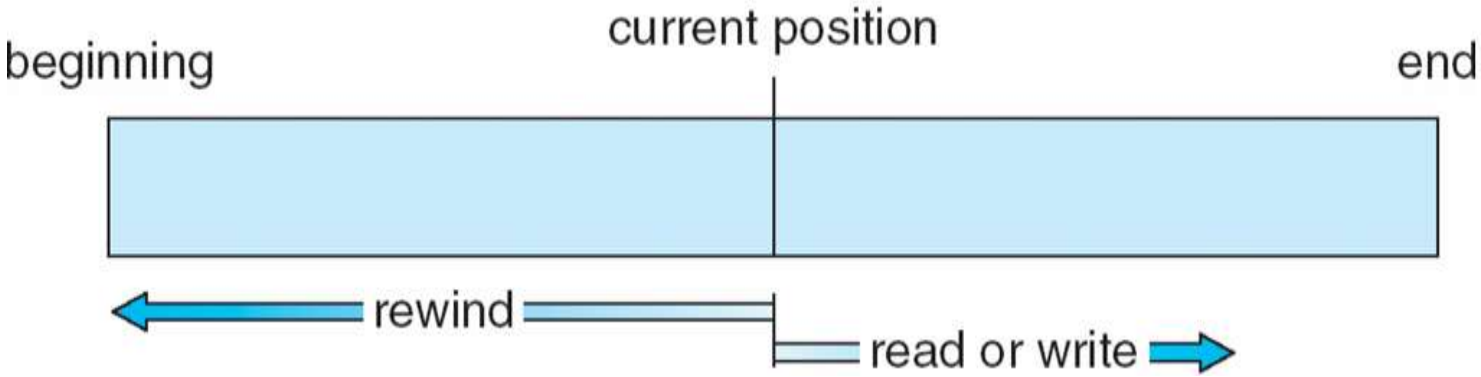
- Indicates the internal structure of the file.
- Files must confirm to the required structure, that is understood by the OS
- **Ex:**
  - **exe file** → with the specific structure, OS will determine whether in memory to load the file and at what location the first instruction is
- **Disadvantage**
  - Cumbersome to maintain the different structures, if the OS maintains multiple file structure

ACCESS METHOD

1. Sequential Access
2. Direct Access
3. Other Access Methods

- **Sequential Access**
  - Simplest access method
  - Information in the file is processed in order, one record after the other
  - Ex: Editors, Compilers
  - Program may be able to skip forward or backward n records for some integer n

read next
write next
reset (to the beginning)
no read after last write
(rewrite)



SEQUENTIAL ACCESS FILE

- **Direct Access**
  - Another method is **direct access or relative access**
  - A file is made up of fixed-length logical records that allow programs to read and write records rapidly in no particular order
  - For direct access, the file is viewed as a numbered sequence of blocks or records.
  - Thus we may read block 14, then read block 53, then write block 7
  - There are no restrictions on the order of reading or writing for a direct-access file

- For direct-access method, the file operations must be modified to include the block number as a parameter.

read *n (rather than read next)*
write *n (rather than write next)*
position to *n*
read next
write next
rewrite *n*

n = relative block number

essential for database systems

SIMULATION OF SEQUENTIAL ACCESS ON A DIRECT-ACCESS FILE

sequential access	implementation for direct access
reset	*cp* = 0;
read next	*read cp*; *cp* = *cp* + 1;
write next	*write cp*; *cp* = *cp* + 1;

Example of Index and Relative Files

Index → contains pointers to the various blocks

- To find a record in the file, first search the index and then use the pointer to access the file directly and to find the desired record

-Binary Search is made on the index

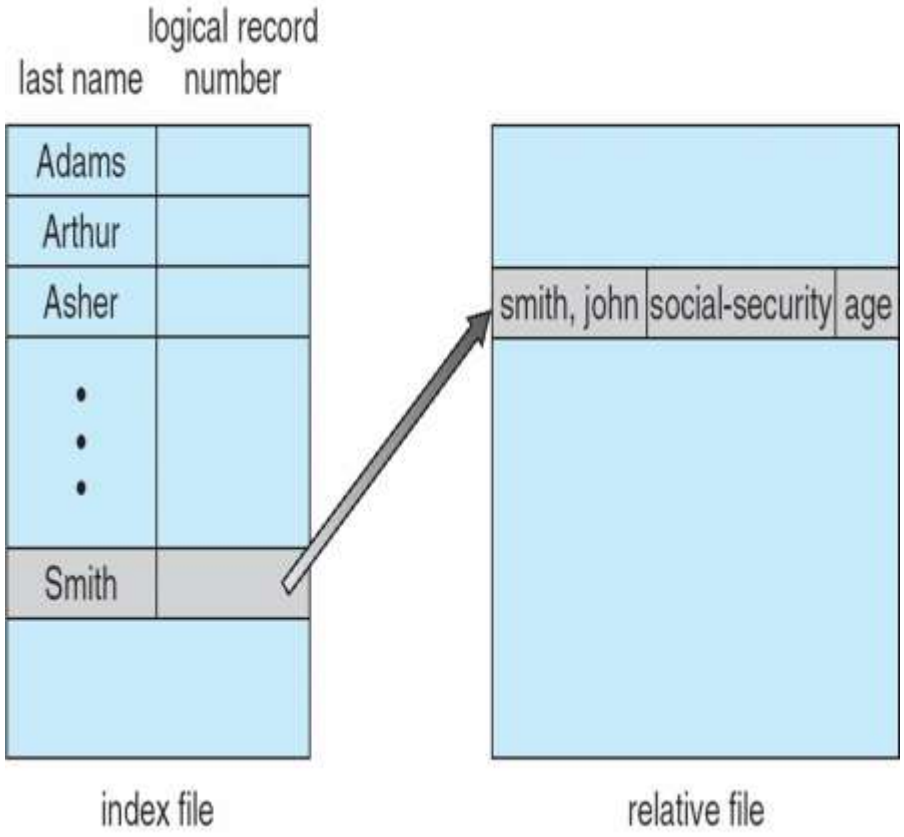
Disadvantage:

For large files, the index file itself may become too large to be kept in memory

Solution

Create index for the index file

Primary index file contains the pointers to the secondary index files, which would point to the actual data items



PROTECTION

PROTECTION

- File owner/creator should be able to control:
  - what can be done
  - by whom

- **Types of access**
  - Read
  - Write
  - Execute
  - Append
  - Delete
  - List

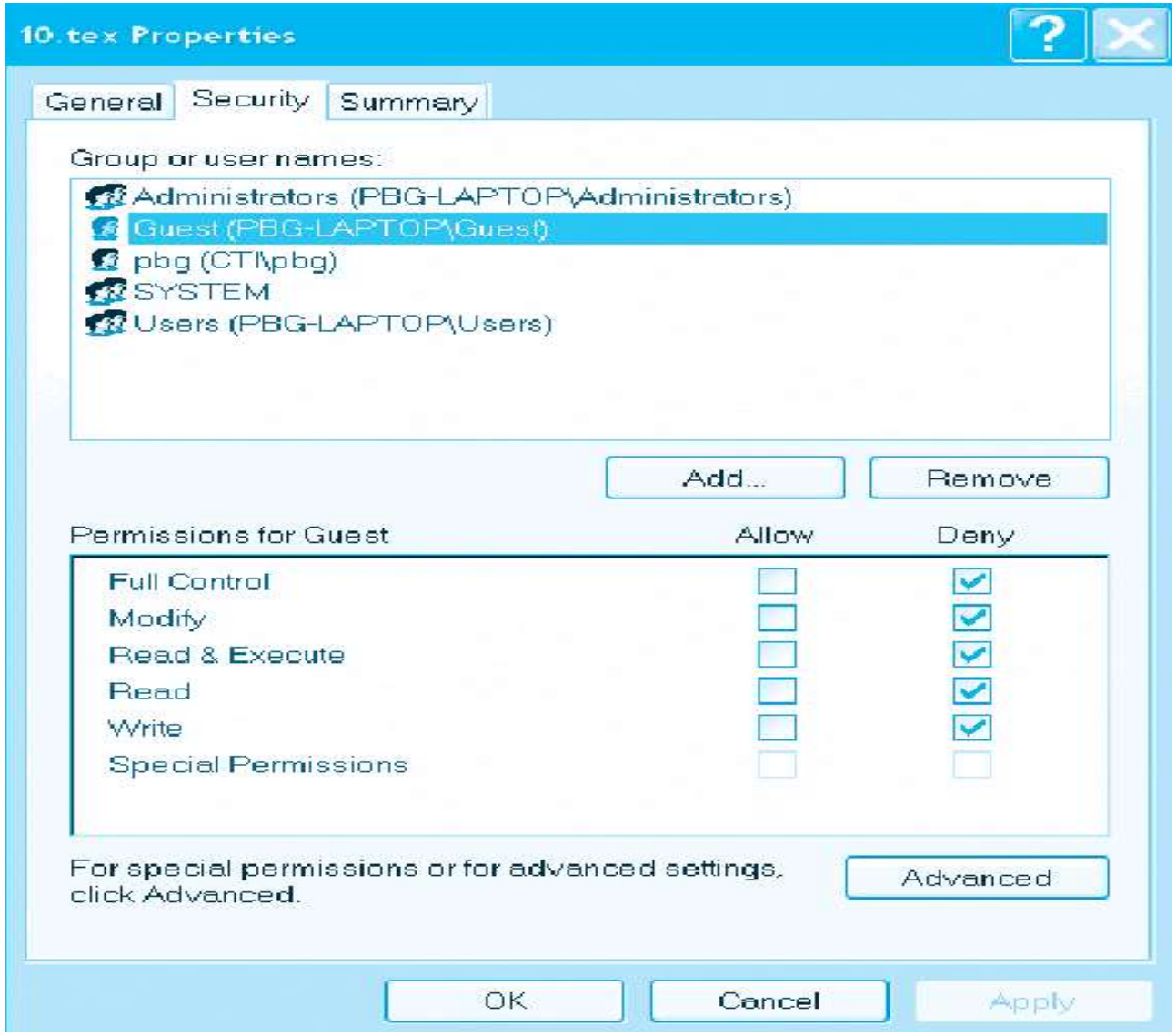
ACCESS LIST AND GROUPS

ACCESS CONTROL

- **Scheme Implemented**
  - Identity dependent access is associated with each file and directory called as **Access-Control List (ACL)**
- **Mode of Access**
  - read, write, execute
- **Three classes of users**

			RWX
a) **owner access (creator)**	7	⇒	1 1 1
b) **group access (Shared user)**	6	⇒	RWX 1 1 0
c) **public access (all other users)**	1	⇒	RWX 0 0 1

Windows XP Access-control List Management



A Sample UNIX Directory Listing

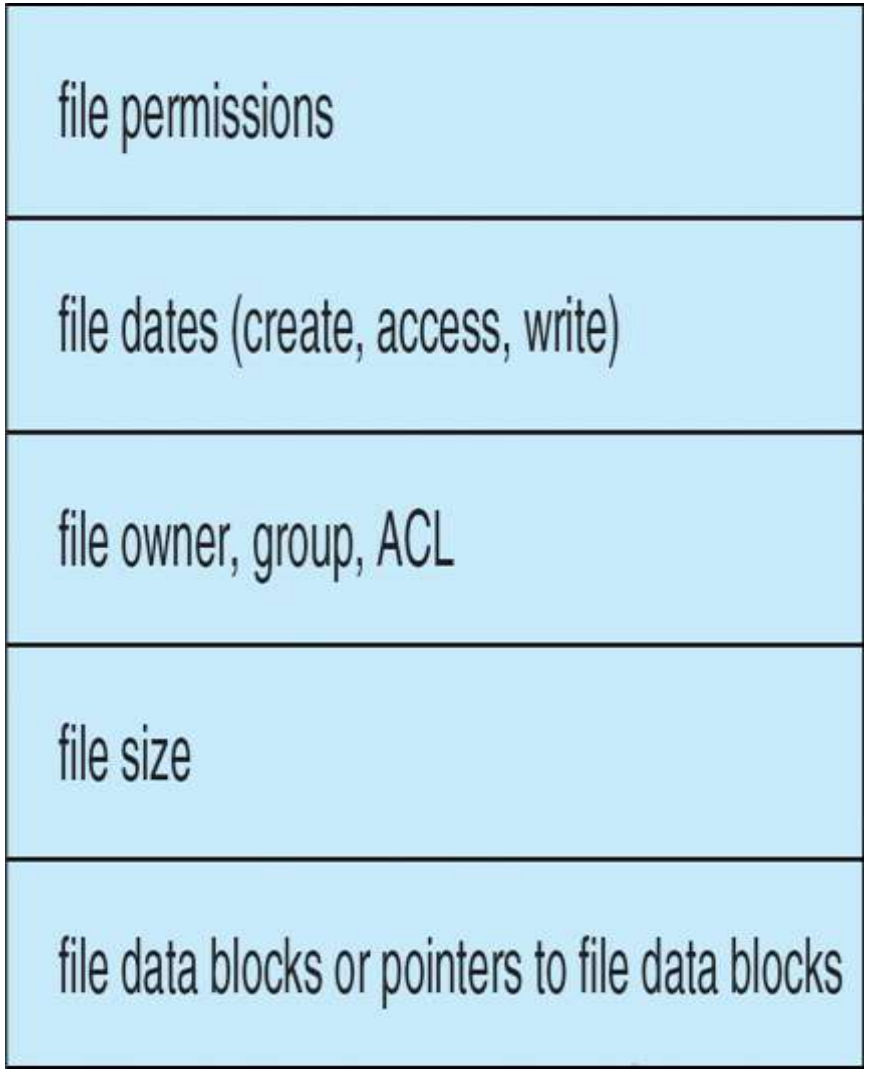
-rw-rw-r--	1 pbg	staff	31200	Sep 3 08:30	intro.ps
drwx------	5 pbg	staff	512	Jul 8 09.33	private/
drwxrwxr-x	2 pbg	staff	512	Jul 8 09:35	doc/
drwxrwx---	2 pbg	student	512	Aug 3 14:13	student-proj/
-rw-r--r--	1 pbg	staff	9423	Feb 24 2003	program.c
-rwxr-xr-x	1 pbg	staff	20471	Feb 24 2003	program
drwx--x--x	4 pbg	faculty	512	Jul 31 10:31	lib/
drwx------	3 pbg	staff	1024	Aug 29 06:52	mail/
drwxrwxrwx	3 pbg	staff	512	Jul 8 09:35	test/

Partition and mounting

- **In-memory information**
  - Used for both file-system management and performance improvement via caching
  - Data are loaded at mount time, updated during file-system operations and discarded at dismount
- **Types of Structures used**
  - **In-Memory Mount Table**
    - Contains information about each **mounted volume**
  - **In-Memory Directory-Structure**
    - Cache, holds **the directory information** of recently accessed directories
  - **System-wide Open-File Table**
    - Contains a copy of the **FCB of each open file**, as well as other information
  - **Per-Process Open-File Table**
    - Contains a **pointer** to the appropriate entry in the **system-wide open-file table**
  - **Buffers**
    - Hold file-system blocks when they are **being read from disk or written to disk**

A Typical File Control Block (FCB)

- **Creating a new file:**
  - Application Program calls the logical file system
  - **Logical file system knows the format of the directory structure**
  - It allocates a new FCB.
- **File Control Block (or inode)** – Storage structure consisting of information about a file



- Open() → Call passes a file name to the logical file system
  - Searches the **system-wide open-file** table to see if the file is already in use by another process
  - If so, a **per-process open-file table entry is created** pointing to the existing system-wide open-file table
  - If file not already open, the **directory structure** is **searched** for the given file name
    - Once file is found, **FCB is copied** into a system-wide open-file table in memory
    - Next, an **entry is made in per-process open-file table**, with a pointer to the entry in the system-wide open-file table

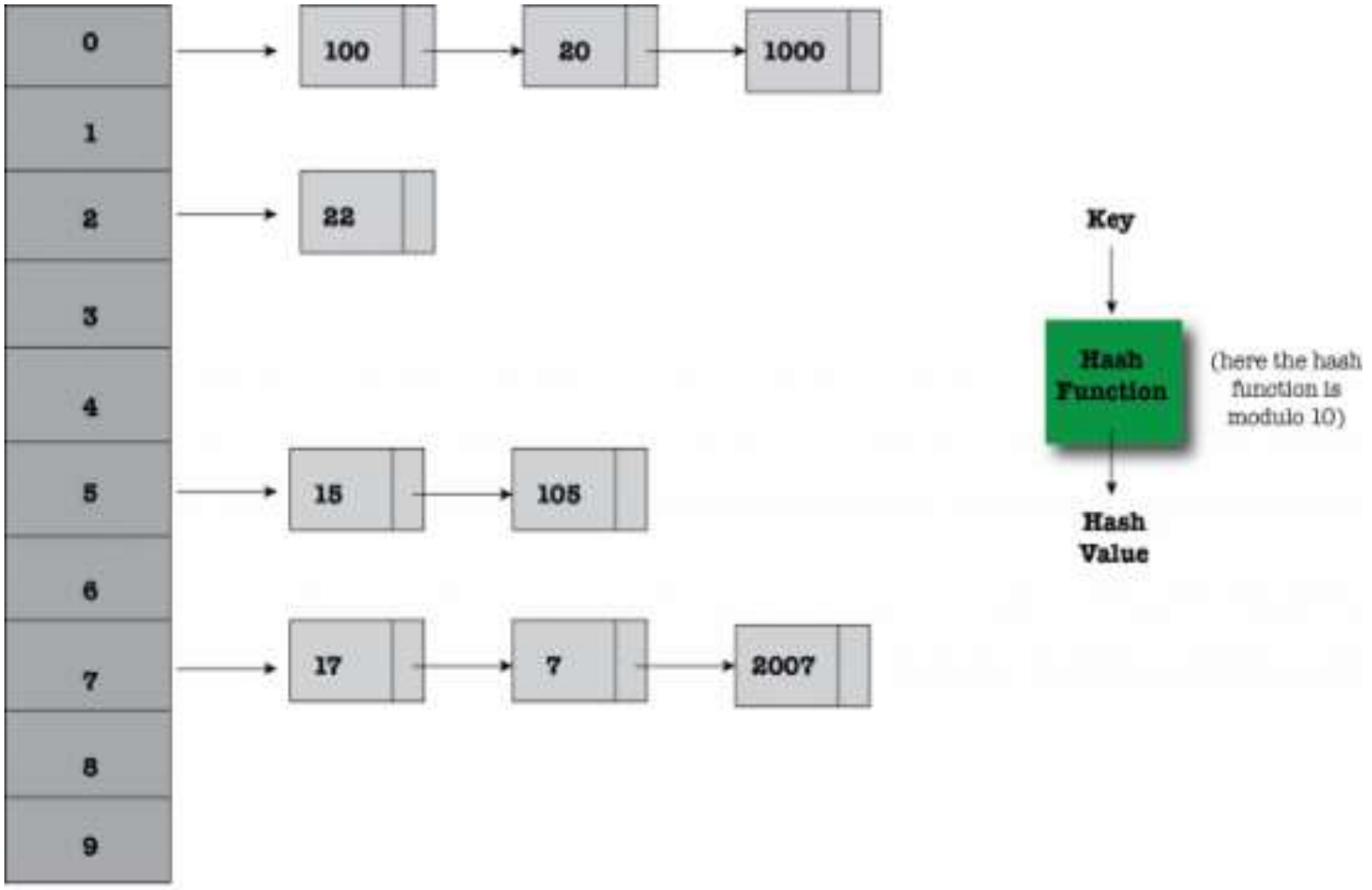
DIRECTORY IMPLEMENTATION

Two Methods

Linear List

Hash Table

- **Linear List** of file names with pointer to the data blocks.
- **Creating a new file**
  - First **search** the directory, ensure **no existing file** has the **same name**
  - Then **add** the **new entry** at the end of the directory
- **Deleting an existing file**
  - **Search** the directory for the existence of the named file
  - **Release** the **space allocated** to it
- **Advantage**
  - simple to program
- **Disadvantage**
  - **time-consuming** to execute as it uses **linear search** to search a file for its existence



A Hash Table

Chained-overflow hash table

- **Hash Table** – linear list with hash data structure
  - **Advantage**: decreases directory search time
  - **Collisions** – situations where two file names hash to the same location
  - **Disadvantage**
    - Fixed size
    - Depends on hash function
  - **Alternative Approach**
    - Chained-overflow hash table can be used
    - Each hash entry can be a linked list instead of individual value

ALLOCATION METHODS

QUESTIONS

Explain different disk space allocation method with an example. (8 Marks)

Name the different file allocation methods. Explain the linked allocation of file implementation with merits and demerits. (8 Marks)

With supporting diagrams, explain linked and indexed method of allocating disk space. (8 Marks)

What is a file? Explain the different allocation methods. (10 Marks)

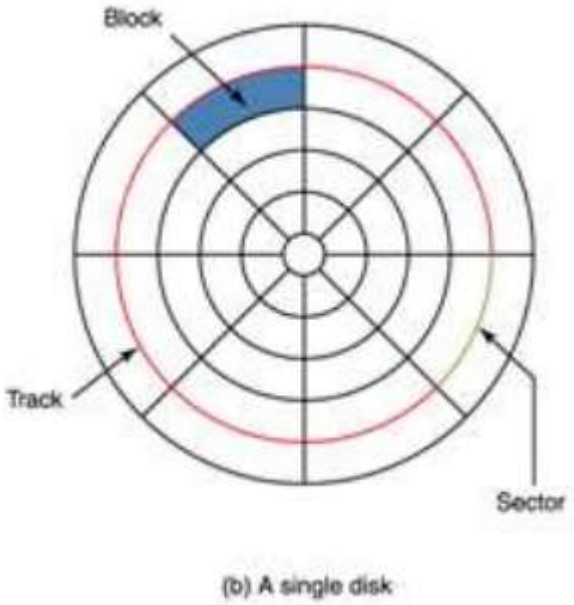
What are the three methods for allocating disk space? Explain with suitable example. (12 Marks)

Compare contiguous and linked allocation methods for disk space. (05 Marks)

- **Allocation Method**
  - refers to how **disk blocks are allocated for files**
- Methods of allocating disk space

ALLOCATION METHOD

Contiguous Allocation



(b) A single disk

a contiguous sequence of sectors from a single track
data is transferred between disk and main memory in blocks

sizes range from 512 bytes to several kilobytes:

Non-Contiguous Allocation

Linked Allocation
Indexed Allocation

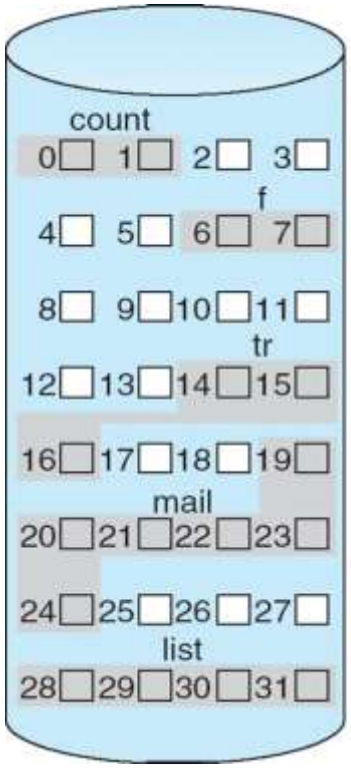
CONTIGUOUS ALLOCATION

- Each file occupies a set of contiguous blocks on the disk
- **Example:**

  File is n blocks long and starts at location b, then, it occupies block

  b, b+1, b+2, … , b+n-1
- Directory entry consists of addresses of :
  - Starting block
  - Length
- Strategy used to find the free hole:
  - First Fit
  - Best Fit
- **Advantage:**
  - Simple – only starting location (block #) and length (number of blocks) are required

    Used in CDROMs, DVDs



Disk

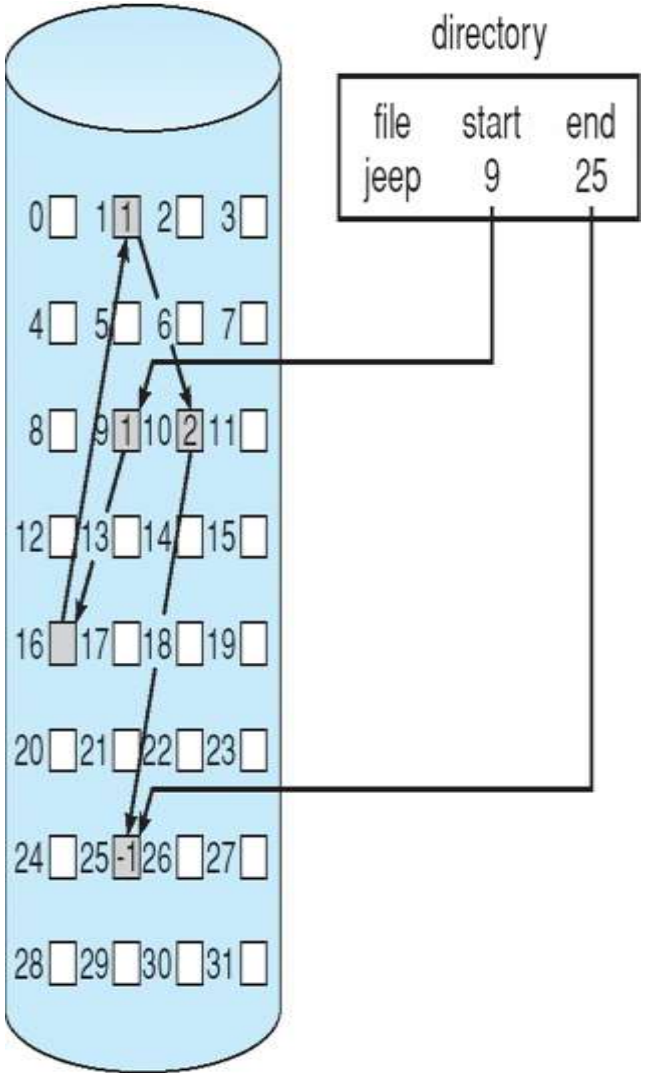
directory

file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

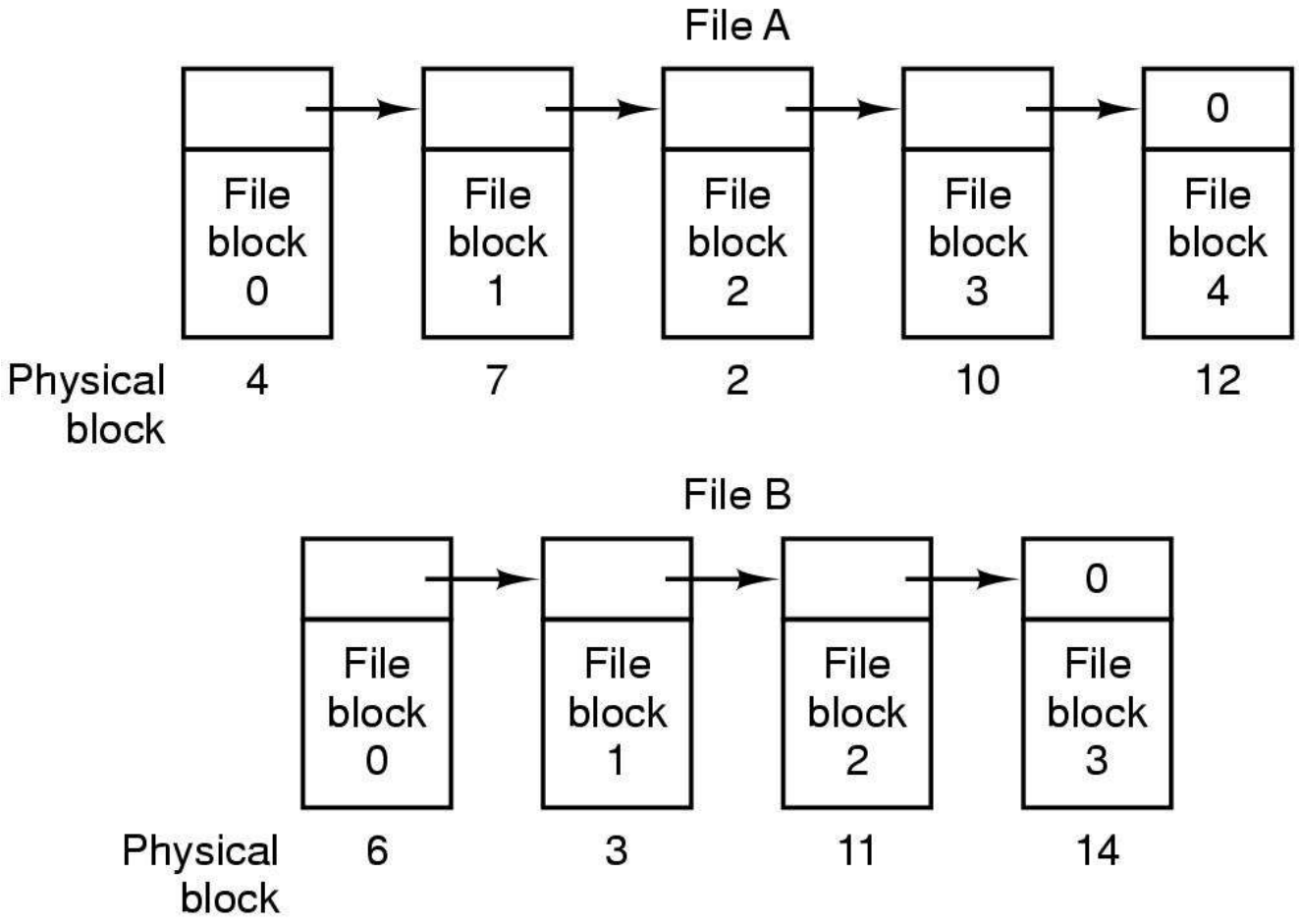
- **Disadvantage:**
  - Wasteful of space (dynamic storage-allocation problem)
  - Suffers from **external fragmentation**
    - Files **allocated and deleted**, the free **disk space is broken into little pieces**
    - **Solution:**
      - **Compaction Technique**, but expensive
  - Complex to **determine how much space** is needed for a file
    - When creating a file, total amount needed for a file must be found and allocated
    - Allocating too **little space** : File cannot grow
    - Allocating too **large space**: Space wasted

LINKED ALLOCATION

- Solves the problem of contiguous allocation
- Each **file** is a linked list of disk blocks: **blocks** may be **scattered anywhere** on the **disk**.
- The directory contains a **pointer** to :
  - the **first block** of the file and
  - **last block** of the file
- Also each block contains **pointers to the next block**, which are not made available to the user.
- **Ex: File for five blocks:**
  - Start at block 9, continues to block 16, then to block 1, then 3 and 25



block = | pointer |

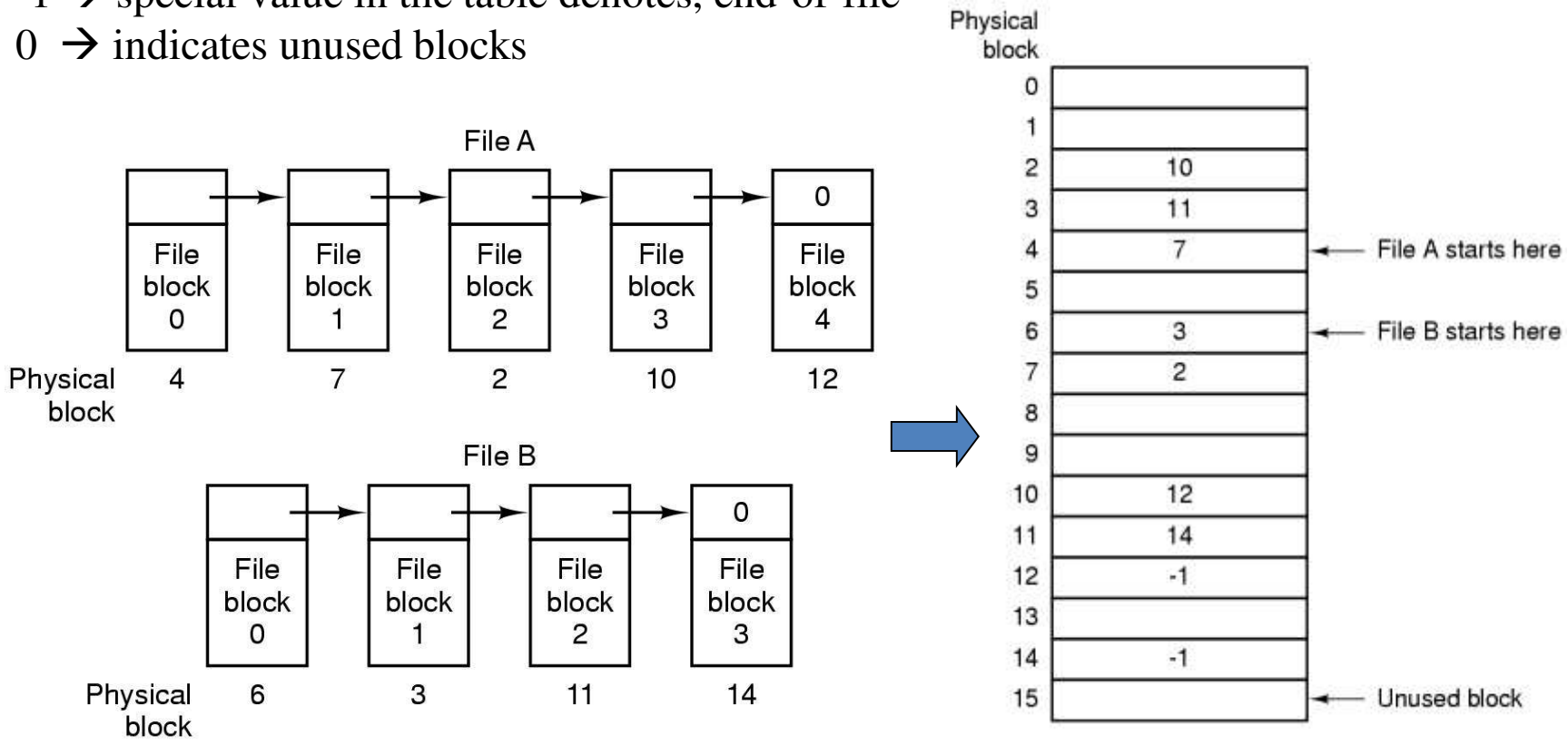

File A
File block 0
File block 1
File block 2
File block 3
0
File block 4
Physical block
4
7
2
10
12
File B
File block 0
File block 1
File block 2
0
File block 3
Physical block
6
3
11
14

- **Create a new file**
  - Create a new entry in the directory, with 2 pointers, 1 to first block, initialize it to nil
  - File is initially empty
- **Read a file**
  - Simply follow the file pointer from block to block
- **Write to a file**
  - Find a free block, written to and linked to the end of the file
- **Pros**
  - No space lost to external fragmentation
  - Disk only needs to maintain first block of each file
  - Size of the file need not to be declared
- **Cons**
  - Random access is costly
  - Overheads of pointers. Ex. Size of a block is 512 bytes, and if pointer occupies 4 bytes, then user sees block of 508 bytes only
  - Loss/damage of file pointer, leads to incorrect access

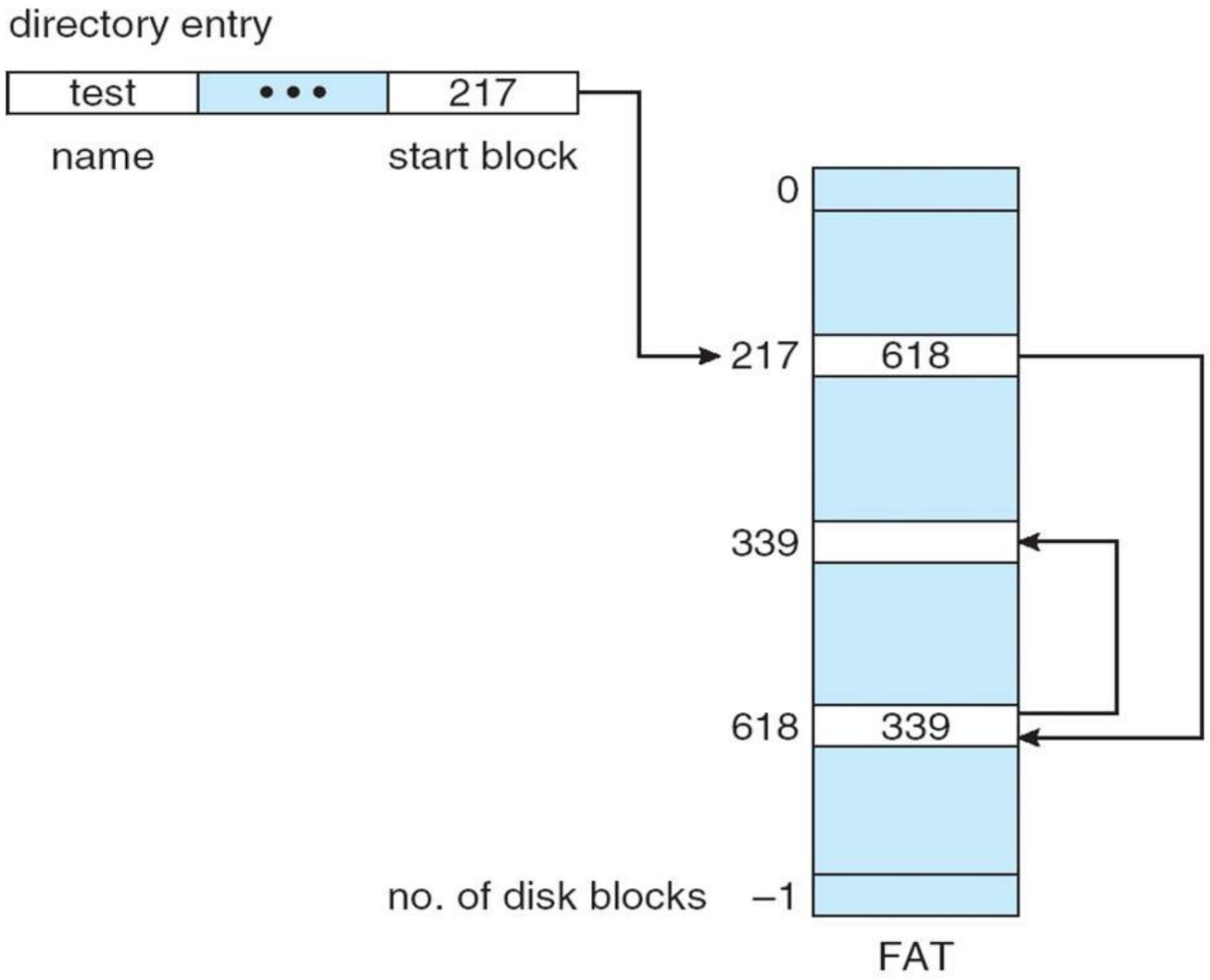
MS-DOS FILE SYSTEM

- Implement a linked list allocation using a table
  - Called File Allocation Table (FAT)
  - Take pointer away from blocks, store in this table

-1 → special value in the table denotes, end-of-file
0 → indicates unused blocks

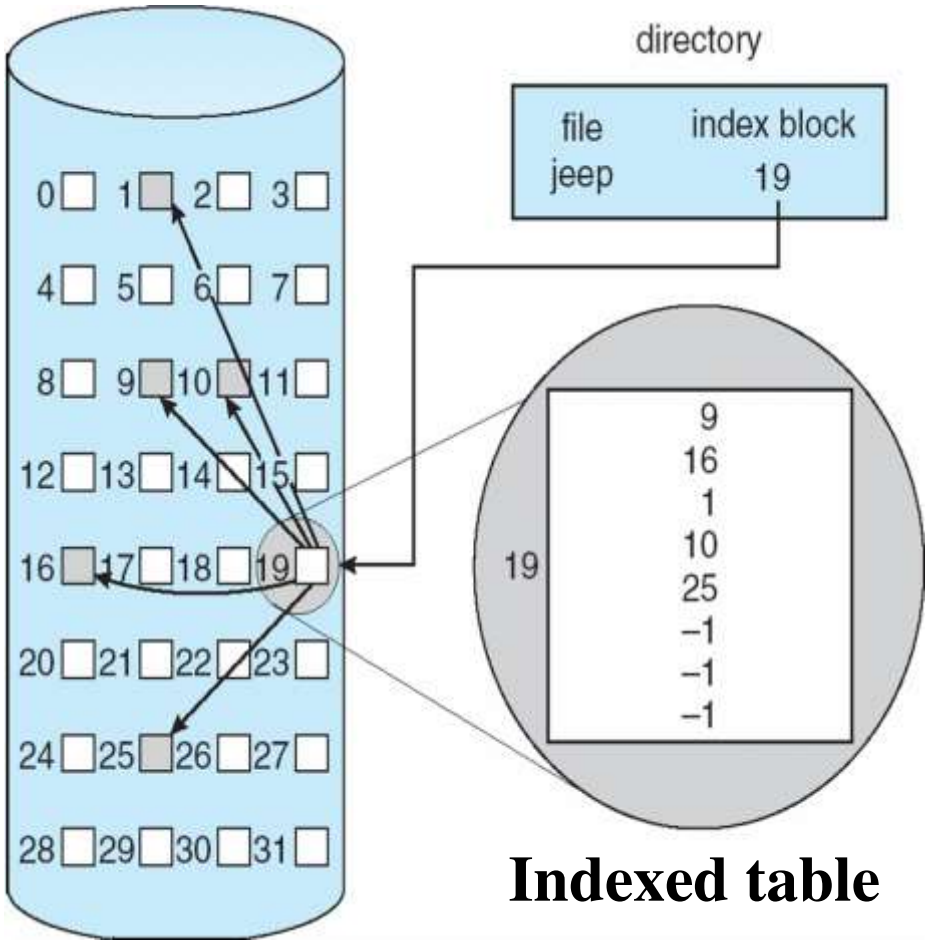


File-Allocation Table

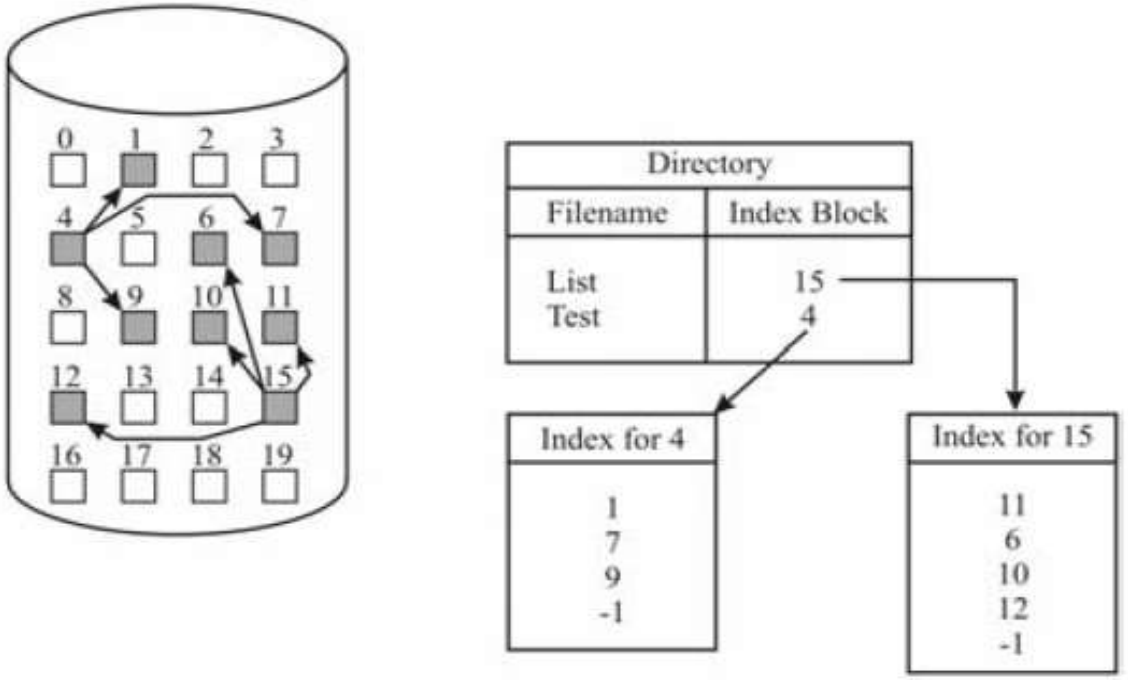


INDEXED ALLOCATION

- Solves external fragmentation and size declaration problem
- Brings all pointers together into the *index block.*
- Logical view.



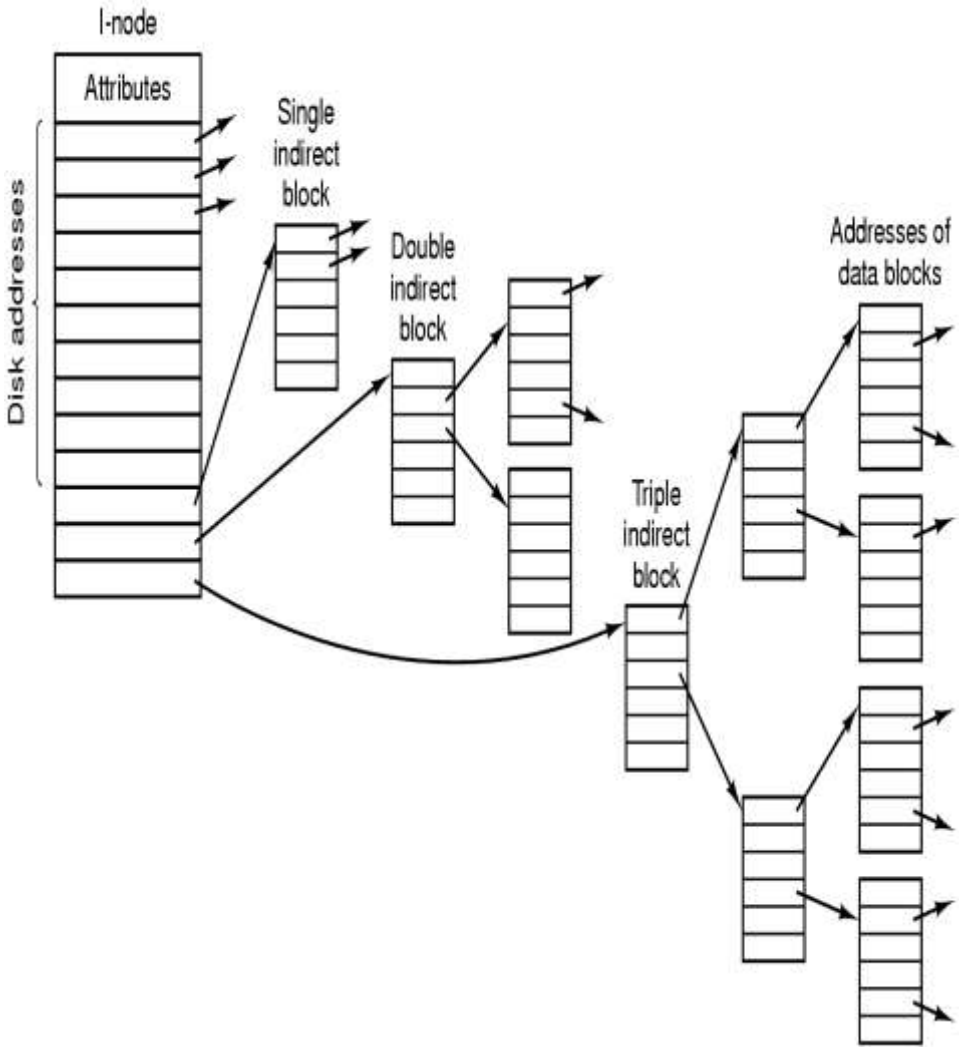
- Creating a new file
  - Set pointer in the index block to nil
  - When i^{th} block is written, a block is obtained from the free-space manager, and its address is put in the i^{th} index-block entry
- **Every file will have an index block**, so it must me as small as possible
- If index block is too small, it can't handle enough pointers

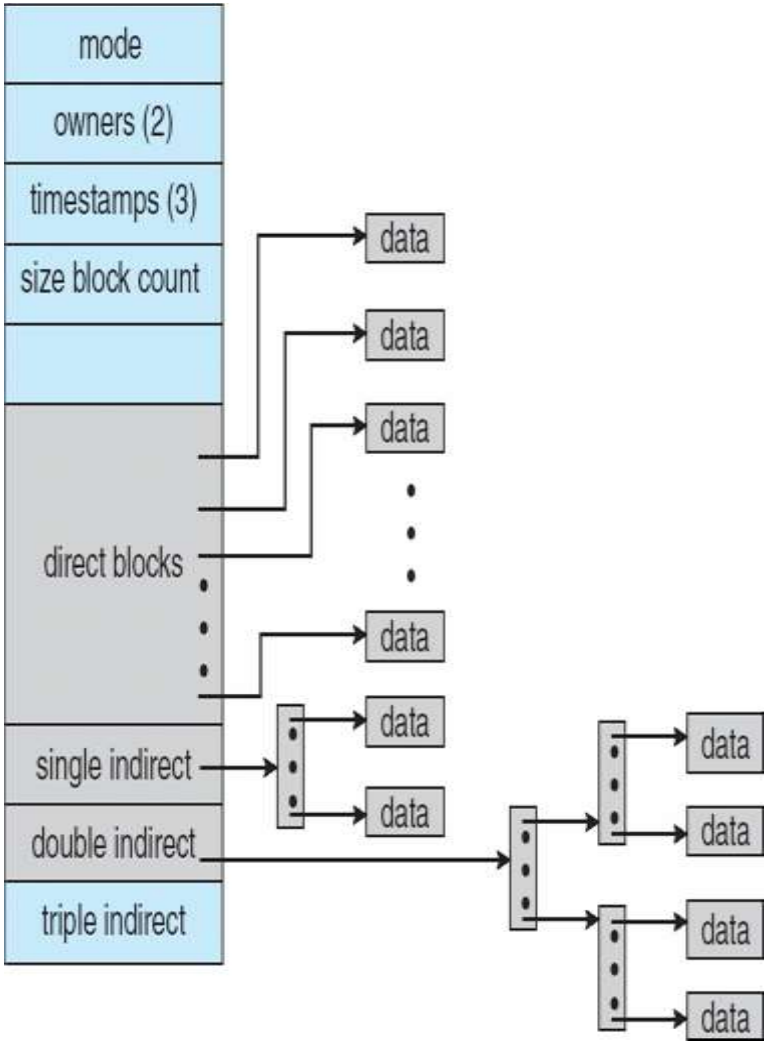


- Mechanisms used for solving this purpose: ***small file large file pblm***
  - **Linked Scheme**
    - For a large file, link several index blocks together.
  - **Multilevel Index**
    - First level index block can point to the set of second level index blocks, which in turn point to the file block
  - **Combined Scheme**
    - **Direct Blocks →**
      - Contains addresses of the blocks that contain data of a file
      - Can be used for small files
    - **Indirect Blocks →**
      - First points to a single indirect block, contains an index block (contains no data), but the addresses of the block that contains data
      - Double indirect block – Contains address of the block containing actual data
      - Triple indirect block – Last pointer, containing address of the triple indirect block

Combined Scheme: UNIX (4K bytes per block)

Direct blocks:
15 pointers of index blocks





FREE SPACE MANAGEMENT

QUESTIONS

- **December 2013/Jan 2014**

  How is free space managed? Explain. (5 Marks)

- **June/July 2013**

  Explain the different approaches to managing free space on disk storage. (10 Marks)

- **June/July 2014**

  What do you mean by free space list? With suitable example, explain any two methods of implementation of free space list. (8 Marks)

- **June/July 2015**

  Explain bit vector free-space management technique. (05 Marks)

- **December 2015**

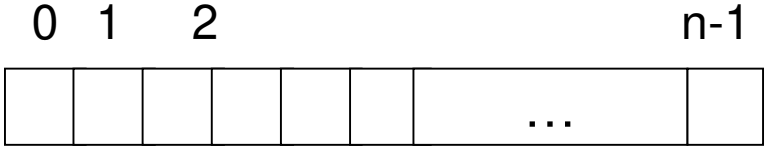
  Write a note on any four different methods for managing free space. (6 Marks)

- **Need for Free-Space Management**
  - Disk space is **limited**, we need to **reuse the space** from deleted files for new files, if possible
- **Free-Space List**
  - Used to keep track of free disk space, system maintains free-space list
- **Free-Disk Blocks**
  - Free-space list records all free disk blocks – those that are not allocated to some file or directory
- **On Creating a File**
  - Search the free-space list for the required amount of space and allocate that space to the new file
  - This space is then removed from the free-space list
- **On Deleting a File**
  - The disk-space is added to the free-space list

- Free-Space Management Approaches
  - Bit Map or Bit Vector
  - Linked List
  - Grouping
  - Counting

- **Bit Map or Bit Vector**
  - Each block is represented by 1 bit

- Bit vector (*n* blocks)



bit[*i*] = { **1 ⇒ block[*i*] free**; **0 ⇒ block[*i*] occupied** }

- **Example**
  - Consider a disk where blocks 2,3,4,5,8,9,10,11,12,13,17,18,25,26 and 27 are free and rest of the blocks are allocated
  - Free-space bit map would be

    001111001111110001100000011100000…

- **Technique for finding the first free block**
  - Check the disk space sequentially for checking whether the value is not 0
  - The first non-0 word is scanned for the first 1 bit
- **Advantage of this approach**
  - Simple
  - Efficient in finding contiguous free blocks on disk
- **Disadvantage**
  - Requires extra space

- **Linked List free-space on disk**
  - Link all the free disk blocks
  - Keep the pointer to the first free block in a special location of the disk
- **Example**

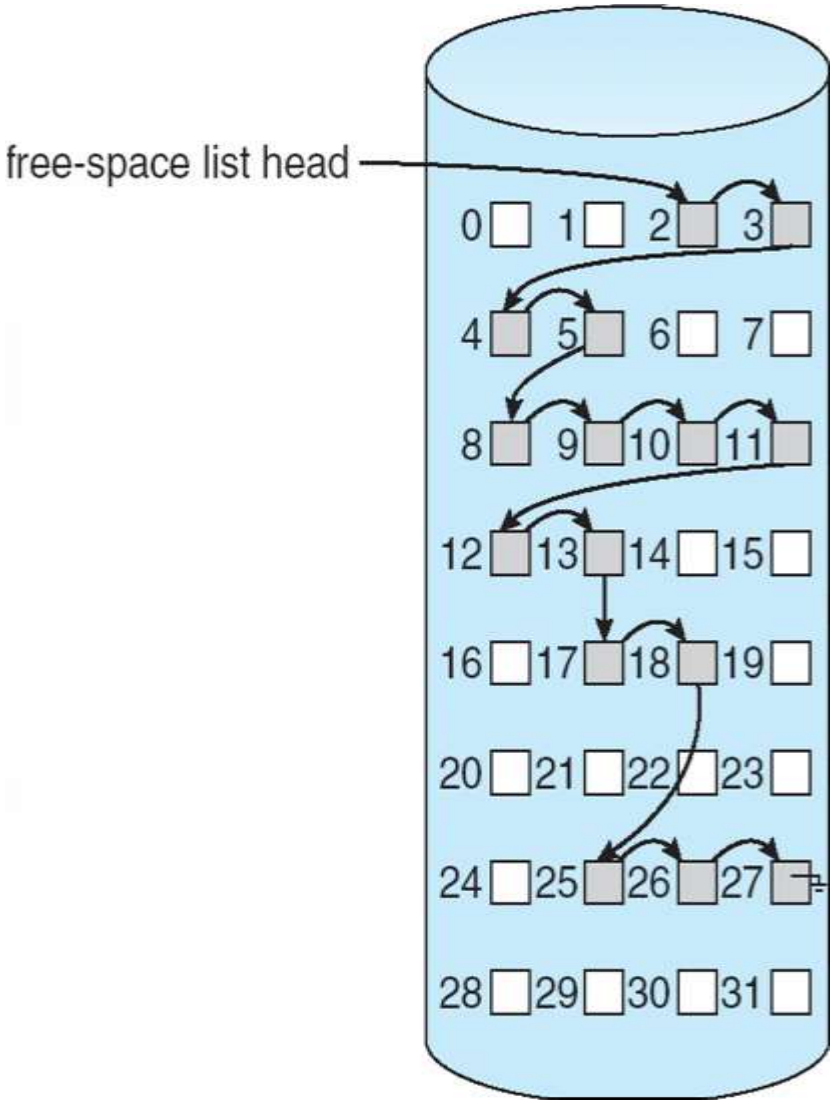
  Consider a disk where blocks

  2,3,4,5,8,9,10,11,12,13,17,18,

  25,26 and 27 are free and rest

  of the blocks are allocated

-Block 2 points to block 3, block 3 points to block 4, so on…

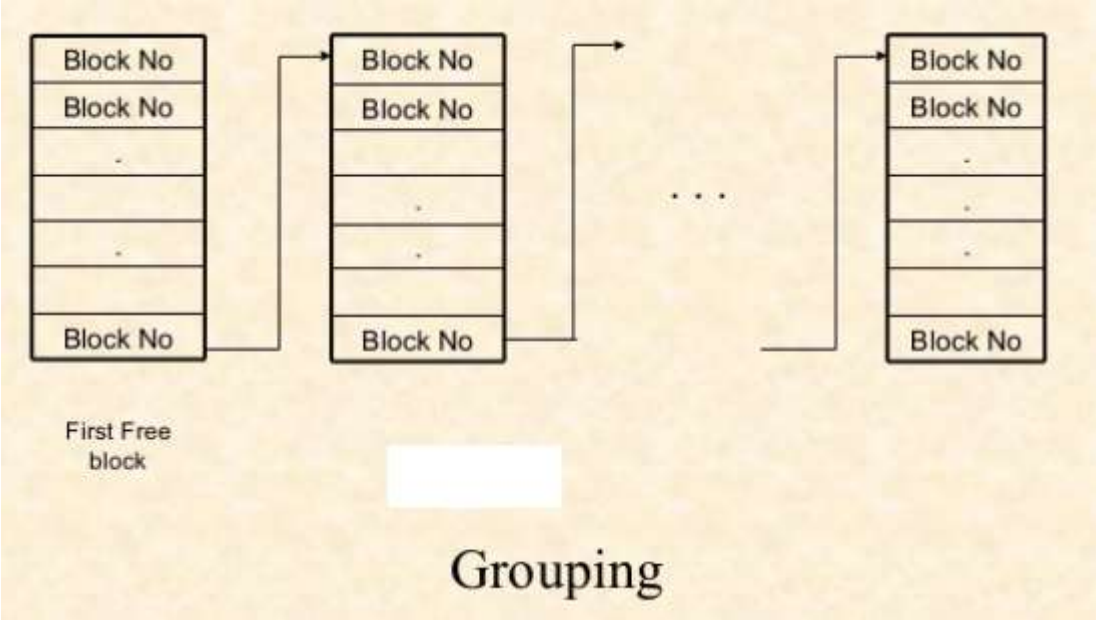


- **Advantage**
  - No waste of space
- **Disadvantage**
  - Scheme is not efficient
  - As it needs to traverse and read each block, which takes too much of I/O time
  - Cant get contiguous space easily

- **Grouping**
  - Stores the **addresses** of n free blocks in the first free blocks
  - The first n-1 of these blocks are actually free
  - The last block contains the addresses of another n free blocks and so on

Advantage

Address of large number of free blocks can be found easily



Grouping

- **Counting**
  - Several contiguous blocks may be allocated or freed simultaneously
  - Keep the list of n free disk addresses in the first free block and the number (n) of free contiguous blocks that follow the first block
  - Each entry in the disk space consists of a disk address and count

The free-space list can contain pairs `(block number, count)`