Long questions (UNIT 1)

->Operating Systems , structure, services, design

-> PCP

-> Process Scheduling

->Multithreading models

-> Scheduling algorithms ,FCFS,SJF, priority,round robin,multilevel queue, multilevel feedback queue

-> Critical section ,dining philosopher, readers writer problem

-> Deadlock,handling, prevention, banker's algorithm

oufastupdates.com

OPERATING SYSTEMS

An Operating System (OS) is a program that control the execution of application programs and acts as an interface between applications and the computer hardware.

"Operating system is an integrated set of programs that are used to manage the resources (CPU, memory, I/O devices, etc.) and overall operations of computer systems."

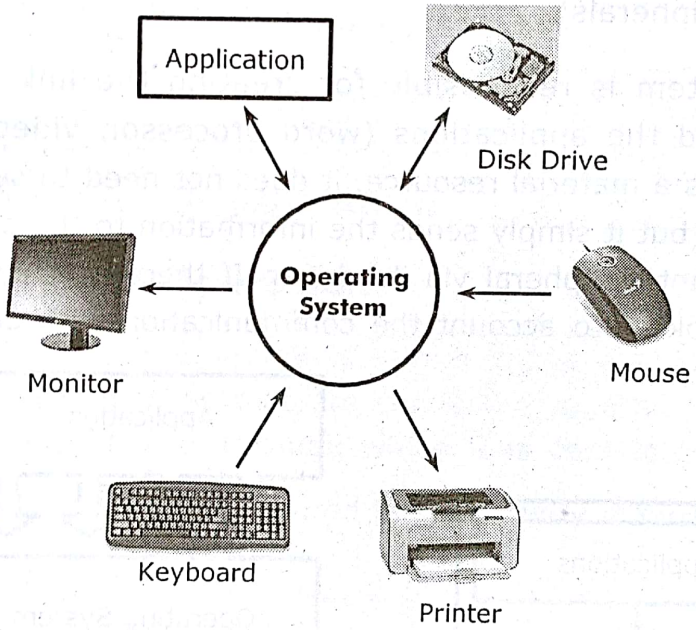


Fig. 1.2.1 Operating System

Operating system is a set of master programs, which is designed to control entire operation of the computer. This is the software that is required in order to run other application software. The operating system can be regarded as a track on which the train runs. As the track acts as a strong link between earth and the train, operating system also acts as a strong link between the hardware and the user of the computer.

To run the train, track has to be installed first, similarly the operating system is required in the system to make it operational. It is loaded automatically into the computer whenever we start the computer. Operating system is general purpose software which is involved in each and every activity of the computer. The operating system decides when and how to process which data, where to store the processed data and from where to supply the data when user requests.

oufastupdates.com

According to this definition, here are the three objectives of operating system,

(1) **Convenience :** An OS make a computer more convenient to use.

(2) **Efficiency :** An OS allows the computer system resources to be used in an efficient manner.

(3) **Ability to Evolve :** An OS should be constructed in such a way as to permit the effective developing, testing and introduction of new system functions without interfering with service.

Why We Need Operating System?

For a computer to be able to operate a computer program (sometimes known as application of software), the machine must be able to perform a certain number of preparatory operations to ensure exchange between the processor, the memory and the physical resources (peripherals).

The operating system is responsible for creating the link between the material resources, the user and the applications (word processor, video game, etc). When a program wants to access a material resource, it does not need to send specific information to the peripheral device but it simply sends the information to the operating system, which conveys it to the relevant peripheral via its driver. If there are no drivers, each program has to recognize and take into account the communication with each type of peripheral.

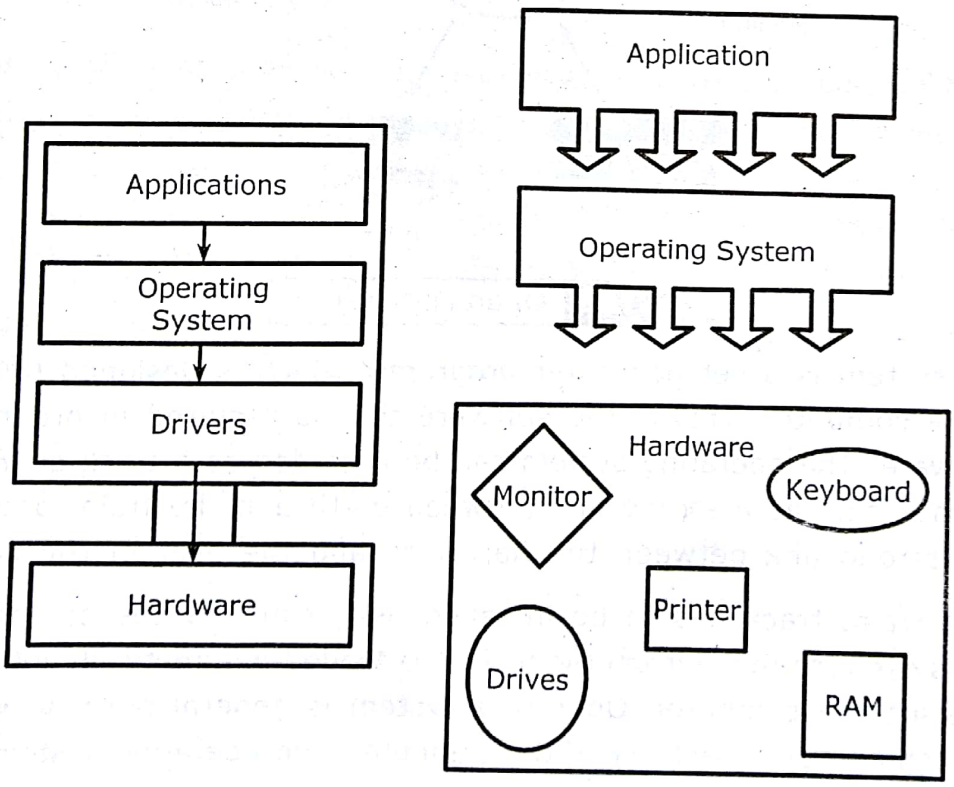


Fig. 1.2.2 The Operating System in a Hierarchy

oufastupdates.com

Fig. 1.4.2 shows the cluster architectures. Clustered system can be categorized into two groups,

(1) **Asymmetric Clustering :** In asymmetric clustering, one machine is in hot stand by mode while the other is running the applications. Hot standby mode monitors the active server and sometimes becomes the active server when the original server fails.

(2) **Symmetric Clustering :** In symmetric clustering mode, two or more than two hosts are running applications and they are monitoring each other.

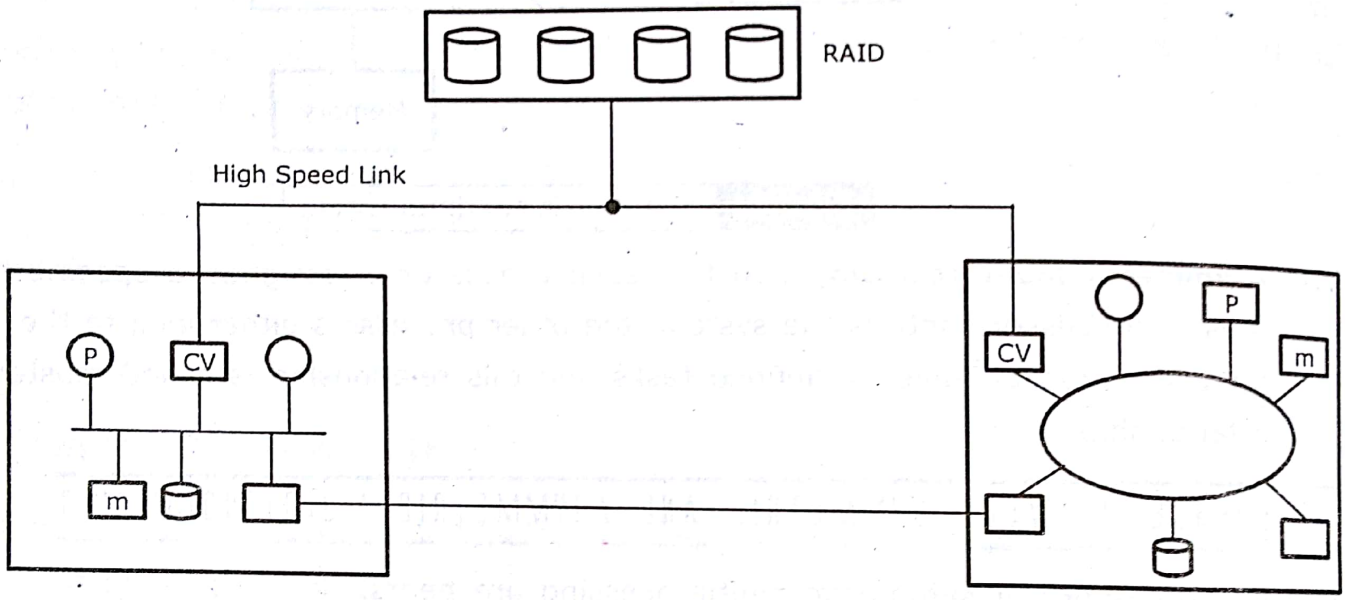


Fig. 1.4.2 Cluster

Parallel clusters and clustering over a WAN (Wide Area Network) is also available in clustering. Parallel clusters allow multiple hosts to access the same data on the shared storage. A cluster provides all the key advantages of distributed systems. A cluster provides better reliability than the symmetrical multiprocessor system.

Cluster technology is rapidly changing. Clustered system use and features should expand greatly as storage area networks. Storage area network allows easy attachment of multiple hosts of multiple storage units.

OPERATING SYSTEM STRUCTURE

All the programs are executed within the environment provided by the operating system. Operating systems, internally differ in their formation as they are organized along various lines. But all of these have some similarities.

The ability to multiprogram is the most significant aspect of operating system. The I/O devices or the CPU cannot be kept busy all the time by a single user. *Multiprogramming* increases CPU utilization by organizing tasks (code and data) so that the CPU always has one to execute.

oufastupdates.com

The idea of this is that operating system keeps several tasks in memory at a time as shown in Fig. 1.5.1. These tasks are the subset of tasks kept in the pool i.e., a place which contains all tasks in the system. The operating system selects a task and starts executing it.

The task that is selected have to wait for the other task such as I/O operations to complete. The CPU is idle in the case of non-multiprogrammed system, but in multiprogrammed system the CPU switches to other task and executes it. As this process needs atleast one task to execute, the CPU is never idle.

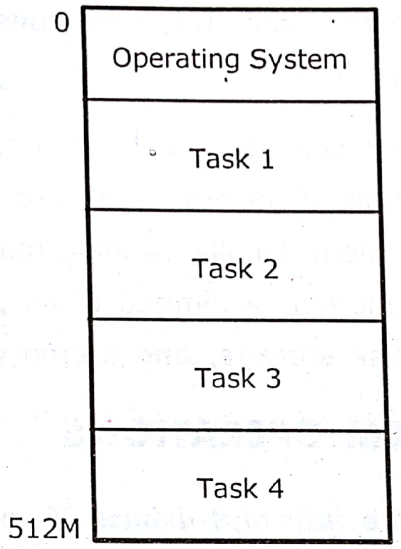


Fig. 1.5.1 Memory Layout for a Multiprogramming System

An environment is provided by the multiprogramming systems in which different resources such as CPU, memory and other peripheral devices are effectively utilized. The *time sharing* systems (also called as *multitasking systems*) are logical extension of multiprogramming. In these systems, the CPU executes multiple tasks by switching among them, but the switches occur so frequently that the users can interact with each program while it is running.

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.

Time-shared operating systems makes use of CPU scheduling and multiprogramming to facilitate each user with a small portion of time shared computer. Each user has atleast one separate program in memory. A program loaded into memory and executing is called a *process*. When a process executes, it typically executes for only a short-time before it either finishes or needs to perform I/O. I/O may be interactive, that is, output goes to a display for the user, and input comes from a user keyboard, mouse, or other device

oufastupdates.com

Since interactive I/O typically runs at "People Speeds," it may take a long-time to complete. Input, for example, may be bounded by the user's typing speed, seven characters per second is fast for people but incredibly slow for computers. Rather than let the CPU sit idle as this interactive input takes place, the operating system will rapidly switch the CPU to the program of some other user.

Time-sharing and multiprogramming require several tasks to be kept simultaneously in memory. Since in general main memory is too small to accommodate all tasks, the tasks are kept initially on the disk in the pool. This pool consists of all processes residing on disk awaiting allocation of main memory.

When the operating system selects a task from the pool, it loads that task into memory for execution. In addition, if several tasks are ready to run at the same time, the system must choose among them. Finally, running multiple tasks concurrently requires that their ability to affect one another be limited in all phases of the operating system, including process scheduling, disk storage, and memory management.

OPERATING SYSTEM OPERATIONS

Modern operating systems are *interrupt-driven*. If no processes are there to execute, then there will be no I/O devices to service and no users to whom to respond. In this situation, the operating system will be idle, waiting for an event to take place.

Events are always signaled by the occurrence of an interrupt or a trap. A *trap* (also called as *exception*) is a software-generated interrupt that occurs by an error, for example, an invalid memory access or divided by zero.

The system's general structure can be defined by the interrupt-driven nature of its operating system. For a particular kind of interrupt, various segments of code in the operating system finds the necessary action that should be taken.

As the software and hardware resources are being shared by the operating system and the users, we have to ensure that an error in a particular program could cause error only in that program that is running. Many processes could be affected by an error in other processes due to sharing.

Without protection against these sort of errors, either the computer must execute only one process at a time or all output must be suspect. A properly designed operating system must ensure that an incorrect (or malicious) program cannot cause other programs to execute incorrectly.

oufastupdates.com

Dual-Mode Operation

For proper operation and correct output, operating system must be protected. The users program and data must be protected from any malfunctioning program. Shared resource also needs some kind of protection.

In dual-mode operation, two separate modes are used for working of operating system. These modes are *user mode* and *kernel mode*. The kernel mode is also called system mode, supervisor mode or privileged mode.

For indicating current mode of the system, mode bit is used in the computer hardware. The mode bit is 0 for kernel and 1 for user. With the mode bit, we are able to distinguish between a task that is executed in user mode or kernel mode.

When the system is executing on behalf of a user application, the system is in user mode and when a user application requests a service from the operating system it must transit from user to kernel mode to fulfill the request which is shown in Fig. 1.6.1.

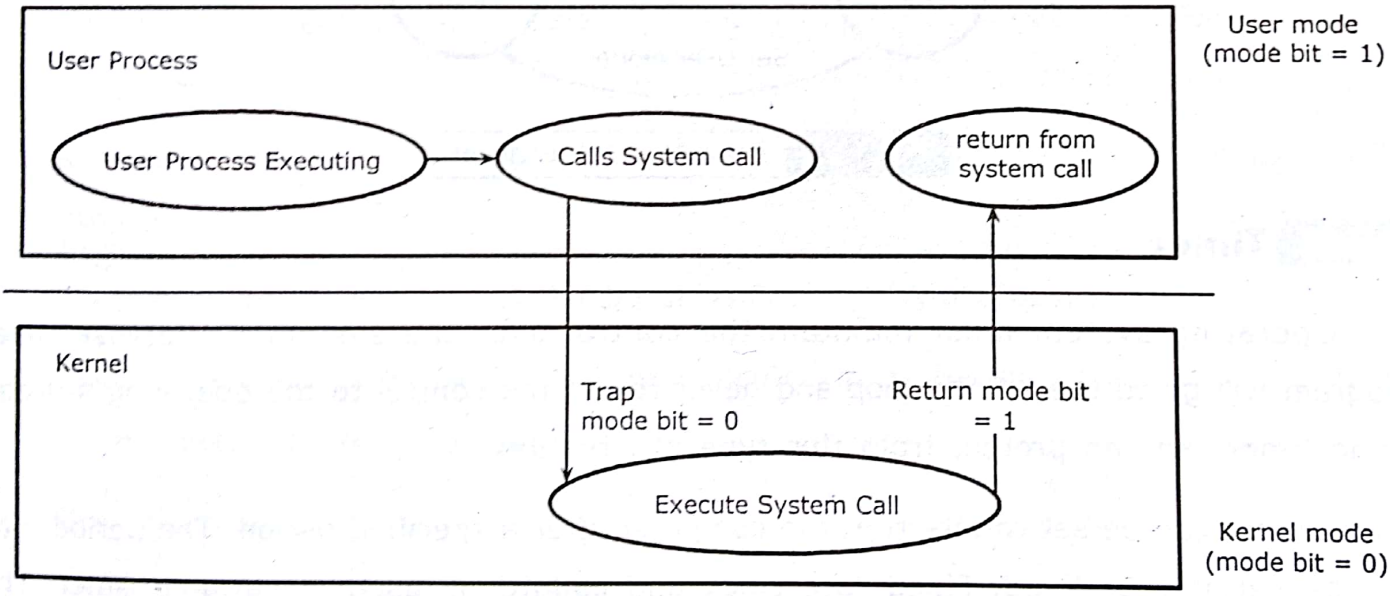


Fig. 1.6.1 Transition From User to Kernal Mode

At the booting time, the hardware starts in the kernel mode, then operating system is loaded. The hardware switches from user mode to kernel mode when interrupts occur. When the operating system gains control of the system, it is in kernel mode.

The dual-mode operation provides the protection to the operating system from unauthorized users. The privileged instructions are executed only in the kernel mode. The computer hardware is not allowed for executing the privileged instructions in other mode, i.e., user mode. If anybody tries to execute the instructions in user mode, it is considered as illegal instruction and also traps it to the operating system.

oufastupdates.com

Software may trigger an interrupt by executing a special operation called a system call. System call is one type of request which is invoked by the user or system. Using privileged instructions, user will interact with the operating system.

This type of request is invoked by user to execute the privileged instructions. As said earlier, this request is called system call. When a system call is executed, it is treated by the hardware as a software interrupt.

MS-DOS has no mode bit so there is no dual mode. Advanced version of Intel CPU i.e., pentium processor supports dual mode operation. For the operating system, Microsoft Windows 2000 and IBM OS/2 provides dual mode operation. These provide greater protection for the operating system. Fig. 1.6.2 shows the dual-mode operation.

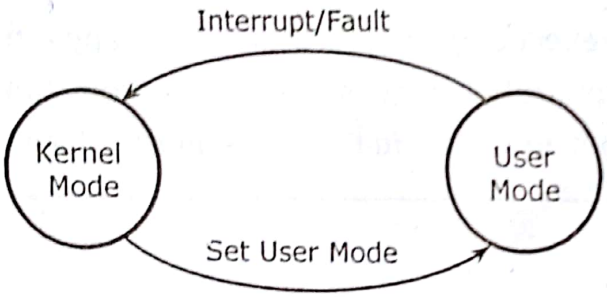


Fig. 1.6.2 Dual-Mode Operation

Timer

Operating system must maintain the control over the system. Otherwise, user program will go to the infinite loop and never return the control to the operating system. Using timer, we can protect from this type of situation,

(1) A timer can be set to interrupt the computer after a specified period. The period may be variable or fixed. Fixed rate clock and counter is used in variable timer. The operating system sets the counter and for every clock ticks, the counter is decremented. An interrupt occurs, when counter reaches to 0.

The operating system will take care while transferring the control to the user. At that time, 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.

(2) The common use of timer is to implement the time sharing. The timer could be set to interrupt every N milliseconds. Each user is allowed to execute for N milliseconds of timer. After completion of timer, next user gets control of the CPU.

oufastupdates.com

PROCESS MANAGEMENT

A process is defined as an instance of a program under execution. Examples of a typical process are a batch job under execution, a time shared user program executing a system task such as SPOOLing etc.

A process cannot simply execute without allocating certain resources. It requires resources such as CPU time, main memory to store its code and data and necessary I/O devices to carry out its job. These resources may either be allocated to a process when it is created or dynamically allocated while it is running.

By definition a program by itself is not a process and it is a passive entity. Because it does not take part in execution, only a copy of the program is loaded into the main memory when requested. Also the program resides in the secondary storage in the form of the content of a file. Considering this, the process may be called as an active entity with program counter pointing to the next instruction to executed and also it requires resource allocation at run time whereas the program does not require any resource.

A process is a measure of unit of work in a system and the system consists of a collection of process.

Therefore, processes may easily be classified into two types,

(1) **System Processes :** System processes executes in a special privileged mode called system mode or kernel mode.

(2) **User Processes :** The user processes executes in the ordinary mode called the user mode.

The process management routines of an operating system are responsible for the following tasks,

- Creation and deletion of user and system processes.
- Monitor, control and record the status of processes in a system.
- Suspend and resume processes.
- Provide techniques for process synchronization, communication and deadlock handling.

oufastupdates.com

OPERATING SYSTEM SERVICES

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

The services provided by operating system are as follows,

(1) **User Interface :** The operating system should provide an interface through which a user can interact. Most of the earlier operating systems provide Command Line Interface (CLI), which uses text commands and a method for entering into the program. All the users are supposed to type their commands through keyboard.

Some systems support batch interface which accepts a file containing a set of commands and executes them. Now-a-days, Graphical User Interface (GUI) is used where a window displays a list of text commands to be chosen by a user through an input or some pointing device.

(2) **Program Execution :** Program execution means running the program, which the programmer has written to get the desired result. The system must be able to load a program into memory and run it. The program must be able to end its execution normally (giving the output) or abnormally (indicating an error).

(3) **I/O Operations :** I/O involves reading or writing data from a disk or a file. When a program is running, it may require I/O which may involve a file or an I/O device. For specific devices, specific functions may be derived. For efficiency and protection, users usually cannot control I/O devices directly. Therefore, the operating system must provide some means to do I/O.

(4) **File-System Manipulation :** The programs may need to read and write files and directories. The file system of the operating system deals with reading and writing of files in the programs. It also deals with creating and deleting files by name.

(5) **Communications :** There are many situations in which one process needs to exchange information with another process. This exchange of information between two processes is known as communication, which can occur in two ways,

(i) The first type of communication takes place between processes executing on the same computer.

(ii) The second type takes place between processes that are executing on different computer network.

(6) **Error Detection :** The operating system should always be aware of errors, which may occur in the CPU and memory hardware (such as power failure or a memory error). The errors may also occur in I/O devices (such as attempt to access an illegal memory location or an arithmetic overflow). For each type of error, the operating system should take appropriate action to ensure correct and consistent computing.

oufastupdates.com

Operating system with multiple users provides following services,

- **Resource Allocation :** If there are more than one user or jobs running at the same time, then resources must be allocated to each of them. Operating system manages different types of resources. Some resources require special allocation code, i.e., main memory, CPU cycles and file storage.
- **Accounting :** Logs of each user must be kept. It is also necessary to keep record of which user uses how much and what kinds of computer resources. This log is used for accounting purposes.

  The accounting data may be used for statistics or for the billing. It also used to improve system efficiency.
- **Protection and Security :** Protection involves ensuring that all access to system resources is controlled. Security starts with each user having to authenticate to the system, usually by means of a password. External I/O devices must be also protected from invalid access attempts.

If a system is to be protected and secure, precautions must be instituted through out it. A chain is only as strong as it's weakest link.

USER OPERATING SYSTEM INTERFACE

For the users to interface with the operating system, there are two approaches. They are,

(1) **Command Interpreter :** This technique allows users to directly enter commands that are to be performed by operating systems. In operating systems such as Windows XP, UNIX command interpreter is a special program that is running when a task is initiated or when user first logs on. The interpreters are known as shells in systems with multiple command interpreters.

Example : In UNIX and LINUX systems, there are seven different shells Bourne shell, Cshell, Bourne-Again shell, the Korn shell etc.

Command interpreters provide the functionality of fetch and execute the next command that is specified by the user. Commands include, create, delete, copy, execute etc.

There are two approaches to implement these commands,

(i) One approach is to execute a command for which the code is in the interpreter. Example when a command is given to create a file, the command interpreter jumps to a section of its code that sets up the parameters and makes the correct system call.

(ii) The second approach is to make use of command to identify a file that is to be loaded into memory and execute.

Example : rm file.txt.

oufastupdates.com

(2) **Graphical User Interface (GUI) :** It allows the user to interface with the operating system via a graphical users interface or GUI. A GUI provides a mouse-based, window-and-menu system as an interface. Instead of giving commands and executing in GUI we can do certain operation by moving the mouse pointer to a particular location and clicking a button.

Most of the UNIX users prefer command interpreter as they often work with shell interfaces. In contrast, windows users prefer GUI environment and almost never use MS-DOS shell interfaces. The choice of using either a command interpreter or Graphical user interface rests with the user.

SYSTEM CALLS

A system call is an interface between the operating system and the process. These system calls are available as assembly language instructions. However, today several high level languages like C allow the processes to make system calls directly.

Whenever a system call is made, the preparatory instructions generated by the compiler load certain registers with the parameters of the system call.

For example, if we want to read the contents of a file XYZ, the required system call may be OPEN, say. The actual appearance of the system call with the parameters specified would be as follows,

OPEN XYZ R : Here XYZ and R are the parameters, where XYZ is the name of the file and R indicates the mode, which stands for 'read'.

Types of System Calls

There are five major categories of system calls, process control, file manipulation, device manipulation, information maintenance and communications.

The system calls are discussed briefly below,

(1) **Process Control :** A program being executed should be able to *end up* either normally or *abort* abnormally. This abnormal termination comes whenever there are bugs in the program. A process executing a specific program may require the output of another program to end up the execution of this first program. Hence, there should be the provision to *load* an *execute* that second program.

We may even want to create a process *terminate* a process and *get the attributes* of an already existing process. We may set the attributes of a process - we may change its priority.

oufastupdates.com

For example, all thee comes under these process management shown in the Fig. 2.3.1.

create process, terminate process
load, execute
end, abort
Get process attributes, set process attributes

Fig. 2.3.1 System Calls Under the Process Management Category

(2) File Management : Consider the case when you want to type some text in a file. Then you need to *open* an empty file, type/write into it, save it and then *close* it. Similarly, you may want to read the contents of a file. You may even write at the end of the existing file i.e., you may want to append to it. All these system calls come under the category of file management.

The same set of calls may be needed for manipulating directories also if the file system is organized into directories.

open, close
read, write, append
save
delete

Fig. 2.3.2 System Calls Under the Process Management Category

(3) Device Management : This block includes system calls made by different files to access certain devices like a printer, speaker etc. For example, if we want to print the contents of a text file, we need to request the printer. We have to ensure that the printer is ready i.e., it is turned ON. Then we check for the sufficiency of the paper. After our job is complete we release the printer.

request device, release device
read device, write device
get device attributes, set device attributes

Fig. 2.3.3 System Calls Under the Device Management Category

(4) Information Maintenance : This block consists of system calls used for maintaining a continuous flow of information between the user and the operating system. We may want to know the system date and time. System calls exist to know the list of files in the system and also to know the number of processes being executed. Provisions are also there to know how many users are currently making use of the resources, the available memory etc.

oufastupdates.com

We can also access the information relating to a particular file or directory like what is the name of that file, when is that file or directory is created, when is it modified, what is the size of that file and so on.

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

Fig. 2.3.4 System Calls Under the Information Maintenance

(5) **Communication :** Two processes on the same computer or on different computers may want to communicate with each other. Before the communication between these two processes takes place, a connection must be opened. The system calls get host id, get process id, open connection, close connection serve to establish and close connection between two processes. Once the connection is established the client and server can send and receive messages.

get host id
Get process id
open connection, close connection

Fig. 2.3.5 System Calls Under the Communications Category

oufastupdates.com

(3) **File Modification :** These are the system programs called text editors and are typically used to create and modify files. Special commands to search contents of files or perform transformations of the text are also available.

(4) **Programming Language Support :** Compilers, assemblers, debuggers and interpreters are commonly provided to the users along with the operating system.

(5) **Program Loading and Execution :** In this after the program is designed, it needs to be loaded into the main memory and executed. The operating system provides system programs as absolute loaders, relocatable loaders, linkage editors and overlay loaders. Debugging systems for higher-level and machine language are as well needed.

(6) **Communications :** These system programs provide the mechanism for creating virtual connections among processes, users and computer systems. Allow users to send messages to one another's screens, browse web pages, send electronic-mail messages, log-in remotely, transfer files from one machine to another etc.

OPERATING SYSTEM DESIGN AND IMPLEMENTATION

Design Goals

The design of an operating system will depend upon the following factors,

(1) Type of system batch, time sharing, single user, multi-user, distributed, real time, etc.

(2) Architecture of the system.

(3) User's expectations convenient to use, easy to learn, easy to use, reliable, safe and fast.

(4) Designer's convenience easy to design, implement and ultimately much easy to maintain.

Even though, we have tried to identify few factors that are involved in the design of the system, it is worth mentioning here that it will always depend much upon the actual task given to the designer, i.e., the actual requirements specified by the customer or company for the design of a domain specific operating system.

Mechanisms and Policies

Mechanism determine how to do something, whereas policy determine what should be done. Let us consider an example of CPU protection, the mechanism is adjusting the timer, and the policy is the value to be set for the timer.

The difference between policy and mechanism is flexibility. Policy can change from time to time i.e., in the above example, the value of timer is different for different types of operating systems. If the operating system is single user, then timer value is different. But, in Apple Mackintosh operating system both policy and mechanism are encoded to enforce a global look and feel to the system.

oufastupdates.com

Implementation

At the implementation level, the concerns are,

(1) Language to be used traditionally, operating systems have been written in assembly language. But nowadays, coding of an operating system is done even in a high level language. The level of language chosen contributes to the attributes of the resulting operating system.

(2) If high level language is used to code an operating system, it can reduce the speed and increase the storage requirements of the resulting operating system. But at the same time it is easier to port (move to some other hardware) the operating system if it is written in a high level language.

(3) On the other hand, an assembly language coding can result in a small sized operating system having high speed but it may not be that easy to understand and debug this code as compared to high level language code.

(4) Algorithms and data structure performance can be improved by incorporating better data structures and algorithms.

(5) Identify the services used mostly services like memory management and CPU scheduling are demanded in higher frequency as compared to others, therefor, the routines written to implement them should be refined for performance purposes.

OPERATING SYSTEM STRUCTURE

An operating system is considerably large and complex program that must be designed and engineered carefully, modified easily and should work properly. Usually it is preferable to have several small components of a system rather than a single or monolithic system. Each component should have a well-defined job and all the components are interconnected to form a single operating system.

The following are the various approaches of operating system design are as follows,

Simple Structure

There are several commercial operating systems which have simple but not well-defined structures. Usually these systems were developed as small, simple and having limited systems but their popularity grew beyond their original scope. One such operating system is MS-DOS which was developed keeping in mind that it should give more functionality within the limited space. Its structure was not divided carefully into modules.

oufastupdates.com

MS-DOS has always experienced the threat of vulnerable and malicious programs which can cause damage to the entire system because of the improper separation between the interfaces and their functionalities. Any application program can access the system or its hardware (CPU, disks etc.,) using the basic I/O routines. No security and protection mechanisms are available. In fact the hardware (i.e., 8088 processor) of that period also provides no hardware protection.

The earlier versions of UNIX also falls in this category.

It divides the system into two parts. They are,

(1) The Kernel (which is also called as heart of UNIX).

(2) The system programs.

The kernel contains several device drivers which interacts with the system directly. Then the problem occurs in developing kernel because it became larger to implement as it has much more functionality.

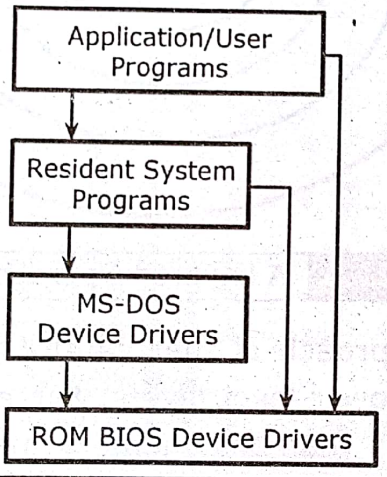


Fig. 2.6.1 Structure of MS-DOS

Layered Approach

In layered approach an operating system is divided into a number of layers or levels. The highest layer (layer N) corresponds to users or application programs and the lowest or bottom layer (i.e., layer 0) corresponds to hardware.

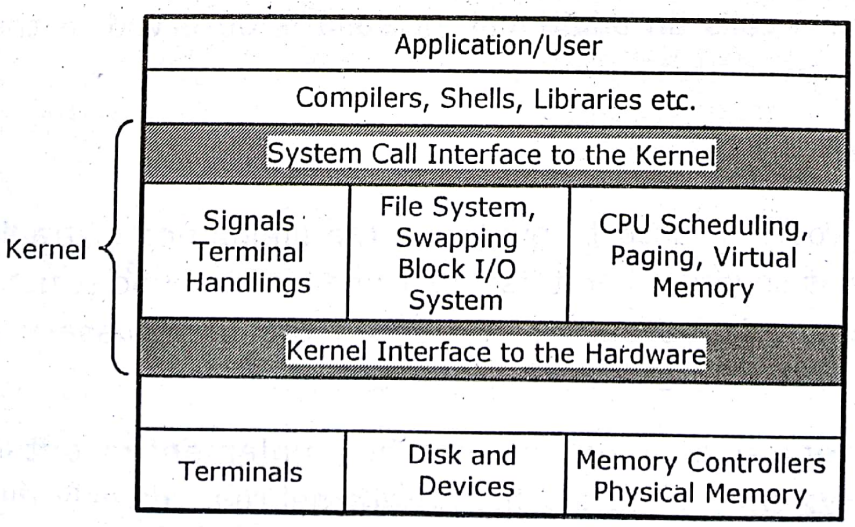


Fig. 2.6.2 Structure of UNIX System

oufastupdates.com

Each layer consists of data structures and operations which are invoked by higher layers. A lower layer provides some services to the upper layer. The advantage of this approach is that construction and debugging becomes simple. As we know the first layer (i.e., layer 0) is nothing but hardware and if we assume that hardware is running correctly, then its services can be used by layer 1. Now, the layer 1 is debugged and if any bug is found it is fixed. The advantage is that the errors can be fixed easily as they lie in that particular layer. Each higher layer simply uses the services of its lower layer without worrying about how these services are implemented (by the lower layer).

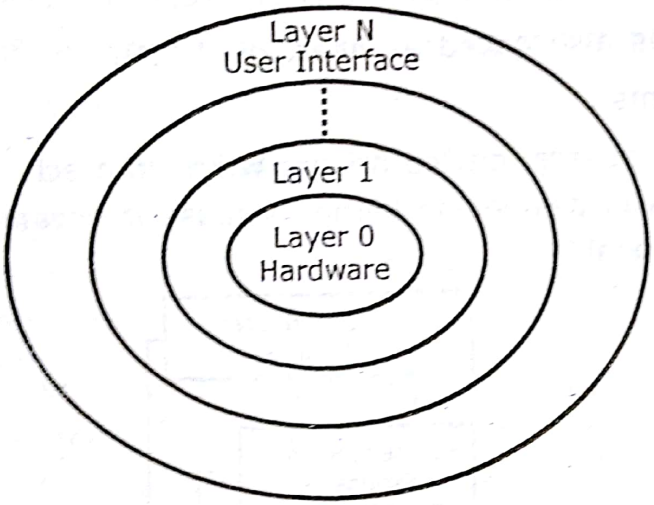


Fig. 2.6.3 A Layered Operating System

The limitation of this approach is that careful pre-planning is needed because a particular layer can use only lower-level layers. For example, the device drivers of hard disk should be at lower layers than the memory management layer, because memory management has to utilize the service provided by device driver of hard disk.

Another major problem in layered approach, is its inefficiency as it increases the overall burden of an operating system.

Consider an example, where a user executes an I/O system call, initially it is cached in I/O layer, which calls memory management layer and ultimately calls the CPU scheduler layer. In each of these calls an additional increase is observed in the system calls and processing time.

Microkernels

Microkernel approach is used to overcome the limitations of traditional UNIX kernel. The kernel in UNIX was so large that it itself became a monolithic structure which is difficult to maintain. Microkernel approach removes all the unnecessary and non-essential components from kernel thereby decreasing its size.

These non-essential components can be implemented outside the kernel as application level or system programs. The microkernel requires only minimal details about the process and memory management.

oufastupdates.com

Its primary function is to provide communication between user programs and various services running inside the user space. For example, a client program and a file server can interact indirectly by sending and receiving messages via microkernel.

The advantage is that it provides flexibility and extensibility. Any new service can be added to user space without modifying kernel. It also increases portability of operating system from one machine to another. It provides security and protection because most of the programs are running at user-level instead of kernel process. The examples of microkernel operating system are Tru64 UNIX, QNX etc.

Module-Based Approach

It is the best methodology of operating system design which uses object-oriented programming techniques. In this approach various components of operating system are implemented as dynamically loadable modules. The kernel consists of core components and dynamic links to load those modules either at runtime or at compile time.

Operating system such as newer version of UNIX, Solaris, LINUX and Mac OSX all use module-based approach.

The Solaris operating system has seven loadable kernel modules as shown in Fig. 2.6.4.

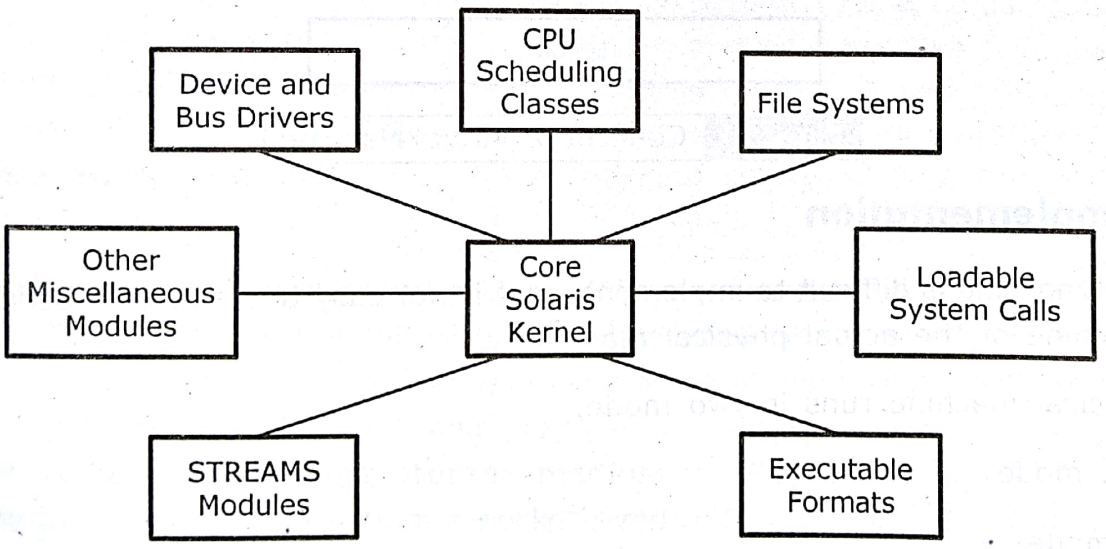


Fig. 2.6.4 Loadable Modules of Solaris Operating System

The kernel consist of core and other services, like device drivers of certain hardware, various FileSystem support etc. These can be added to the kernel dynamically when needed. The approach is similar to layered approach but it is more flexible when compared to it. The core services include the code for communication among them (modules) and loading modules.

oufastupdates.com

PROCESS CONCEPTS

A process is a program in execution. A process defines the fundamental unit of computation for the computer. Components of the process are,

(1) Object program.

(3) Resources.

(2) Data.

(4) Status of the process execution.

Object program i.e., code to be executed. Data is used for executing the program. While executing the program, it may require some resources. Last component is used for verifying the status of the process execution. A process can run to completion only when all requested resources have been allocated to the process. Two or more processes can execute the same program, each using their own data and resources.

The Process

A process can be defined as the work unit in a system. Because a system can have collection of processes, like user process executing use code or operating system process executing system code and so on.

Terms process and program can be used interchangeably.

A process can be program in execution but its execution must be sequentially progressed.

A process contains process stack, temporary data and data section with global variables.

But a program by itself cannot be a process, because program is a passive entity and process is an active entity which means a process in execution involves program counter specifying next instruction to execute and uses resources available in the system.

Process State

With the process execution, its states are also changed and they are defined by the current activity of that particular process.

They may be in any one of the following states,

(1) New : creation of the process.

(2) Running : execution of the instruction.

(3) Waiting : process waiting for the occurrence of some event.

(4) Ready : processor is assigned to a process waiting.

(5) Terminated : process execution finished.

oufastupdates.com

State Diagram

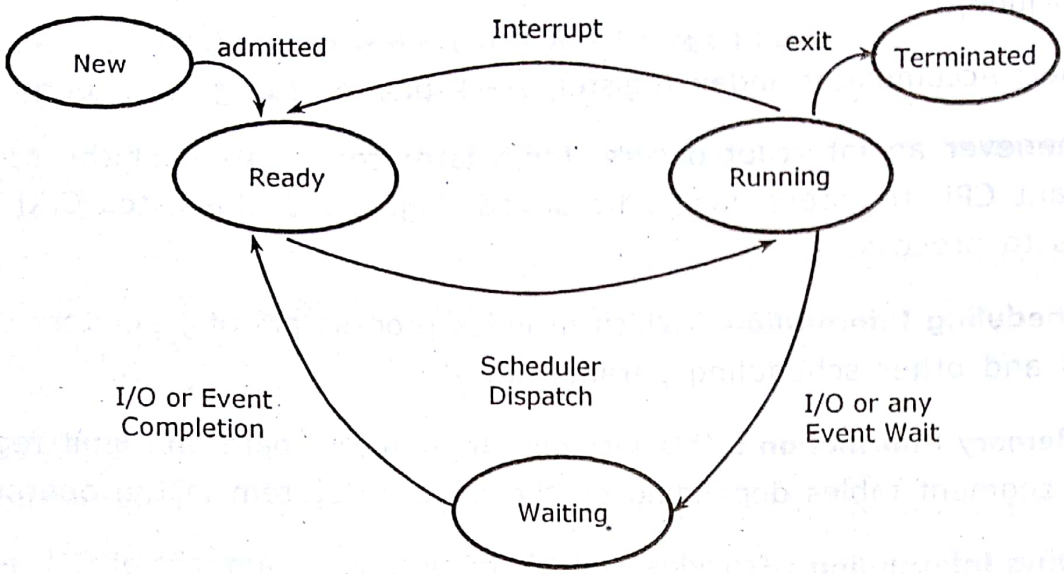


Fig. 3.1.1 State Diagram

Names used to represent the states may vary from operating system to operating system. But the states represented is found in all the systems. And on a processor, only one process can be running at any instant but many processes may be ready and waiting. Fig. 3.1.1 illustrates the state diagram of process.

Process Control Block (PCB)

Each process in a system must be represented by a PCB in the operating system and also called as a task control block. It provides information about mentioned fields with respect to a process. Fig. 3.1.2 in the diagrammatic representation of PCB.

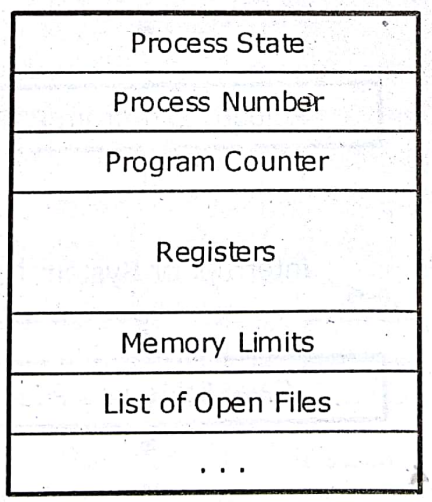


Fig. 3.1.2 Process Control Block

- **Process State :** Giving the information about whether process is ready, running, waiting, halted and other states.
- **Program Counter :** Indicates the address of next instruction in process to be executed.

oufastupdates.com

- **CPU Registers :** The numbers and type differs depending on system architectures, They include,

  Example : Accumulator, index register, stack pointer and general purpose registers.

  Whenever an interrupt occurs, for a later continuation of the process, some important CPU registers has to be saved. Fig. 3.1.3 illustrates CPU switch from process to process.

- **CPU Scheduling Information :** Which includes process priority, pointers to scheduling queues and other scheduling parameters.

- **Main Memory Information :** This includes the value of bare and limit registers, page tables, segment tables depending on the memory system in the operating system.

- **Accounting Information :** Provides the information about amount of CPU and real time used, time limits, account numbers and job/process numbers.

- **I/O Status Information :** Informs about list of I/O devices allocated and list of open files to that process.

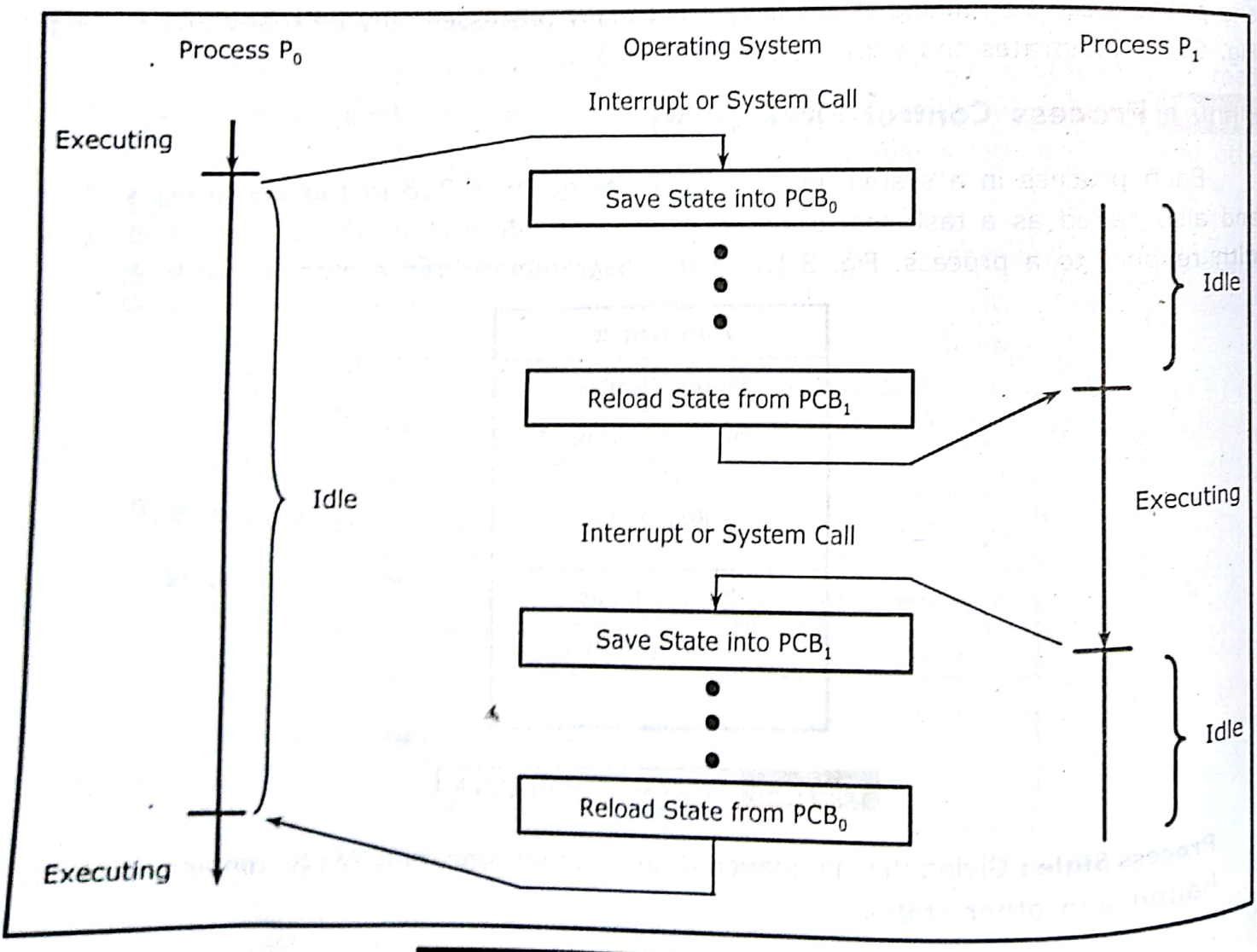


Fig. 3.1.3 CPU Switching Process

oufastupdates.com

PROCESS SCHEDULING

With respect to multiprogramming which maximizes CPU utilization, allowing some processes run all times.

The time sharing helps the CPU to switch among processes very frequently so that user can interact with every program, when it is running.

In such environment, where many processes are considered simultaneously requires a scheduling techniques, so that only one job can be submitted to a processor.

Scheduling Queues

Following are the types of queues,

(1) **Job Queue :** Consists of all the processes in the system. Once processes enters the system, they are placed into a job queue.

(2) **Ready Queue :** Processes which are in main memory, ready and waiting to execute are kept in a list called ready queue. This list is stored as linked list. And the queue's header will contain pointers to the first and last PCB's in the list.

Each PCB consists of a pointer field which points at the next process in the ready queue.

Ready Queue and Various I/O Device Queues

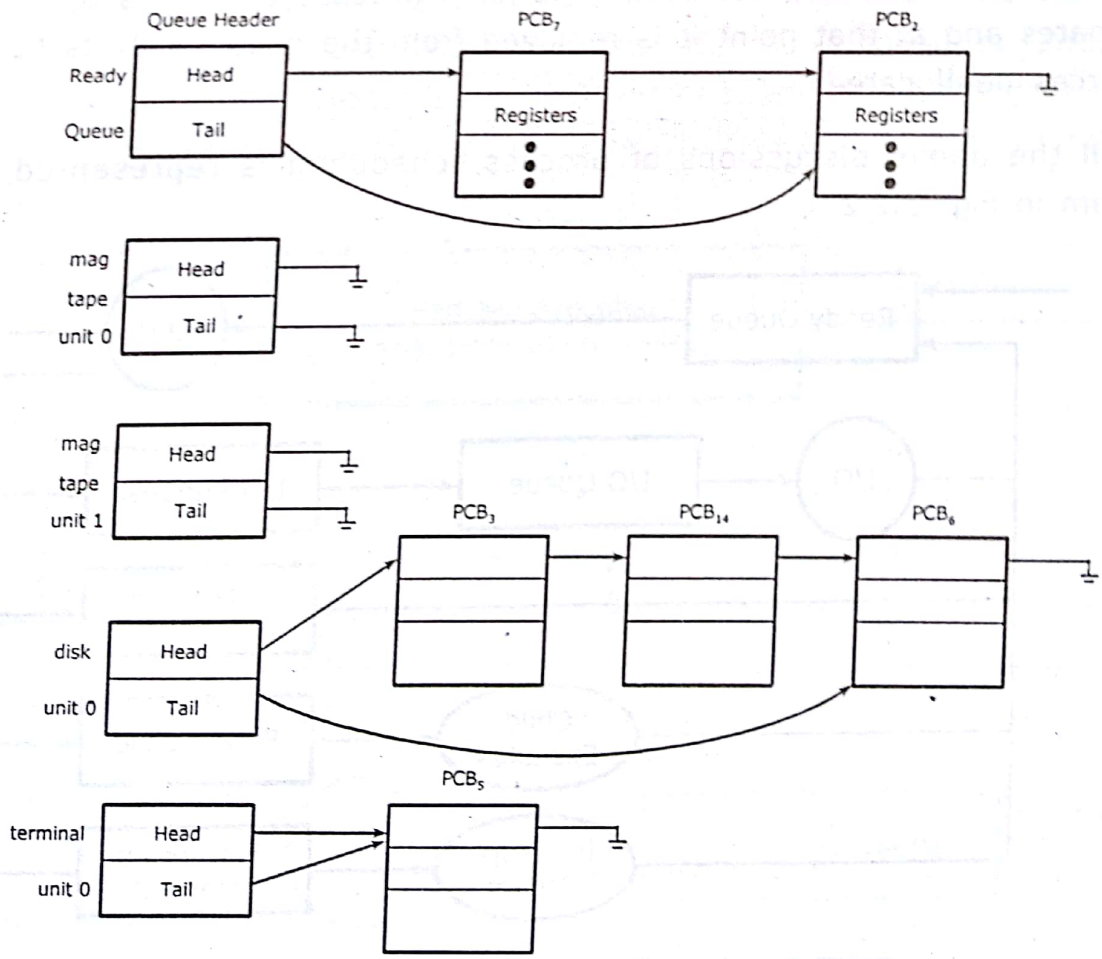


Fig. 3.2.1 The Ready Queue and Various I/O Device Queues

oufastupdates.com

(3) **Device Queue :** A process may quit after executing for some time, due to interrupt or waiting for a particular event occurrence like I/O request where I/O request may request to a shared device (disk).

Since there will be many processes in the system, the disk may be busy with I/O request of some other process, hence it may have to wait for the disk. List of such processes waiting for a particular I/O device is called a device queue.

There will be individual queue for each device.

A new process is initially put in the ready queue and it waits there until it is selected for execution, giving the CPU.

After CPU is allocated and the process is in execution, one of the following events may occur,

(i) The process may issue an I/O request and hence it will be placed in a I/O queue.

(ii) The process could create a new subprocess and waits for its termination.

(iii) The process may be removed forcibly from the CPU, because of an interrupt and placed back in the ready queue.

In the first two cases, the process switches from the waiting state to the ready state and put back into the ready queue. A process continues this cycle until it terminates and at that point it is removed from the queue with its PCB and other resources deallocated.

All the above discussions of process scheduling is represented by queuing diagram in Fig. 3.2.2.

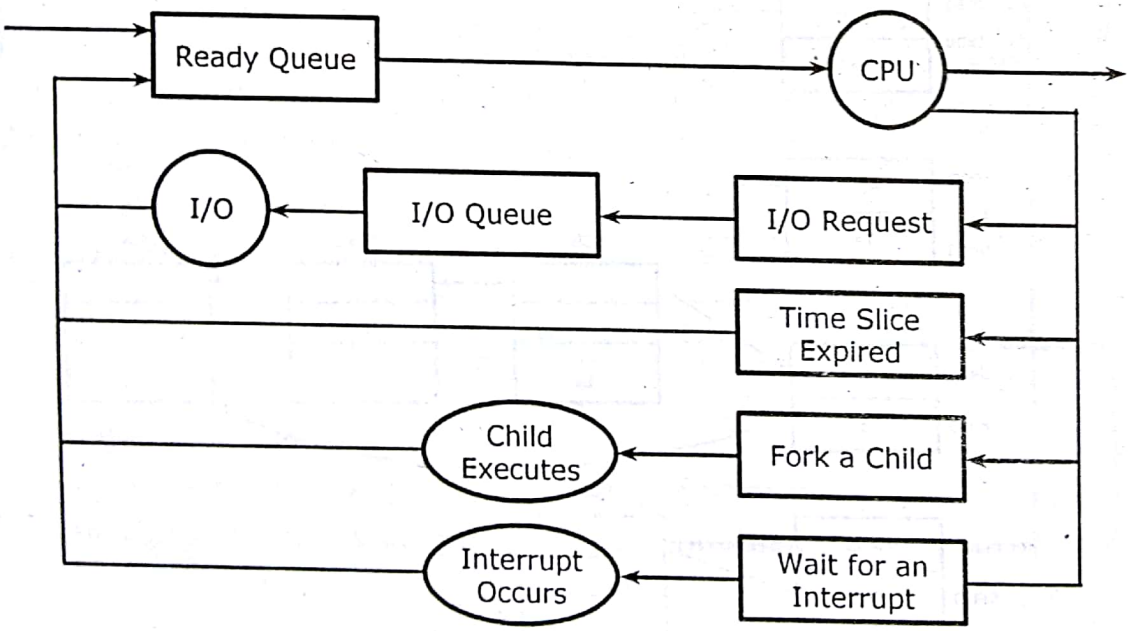


Fig. 3.2.2 Queuing Diagram of Process Scheduling

oufastupdates.com

Schedulers

The processes may shift between various scheduling queues during its time in the system. Hence operating system must select process from these queues in required fashion. This process of selecting processes from the queues is called scheduling.

There are basically 3 types of schedules,

(1) **Short-Term Scheduler :** This is also called *CPU scheduler*. This short-term scheduler selects processes from the list of processes ready to execute and allocates the CPU to them.

Short-term scheduler must select a new process for the CPU very frequently. Typically a process may execute only for few milli seconds, before waiting for an input/output request. And a short-term scheduler executes at least once in every 100 milli secs. Because of this short duration of time between executions, short term scheduler must be very fast.

(2) **Medium-Term Scheduler :** Some of the operating system like time sharing system uses an intermediate level of scheduling called medium-term scheduler, which helps in recovering processes from the memory, thus reducing degree of multiprogramming.

And at sometime later, processes can be reintroduced into memory and its execution can be continued where it is left off. This is called as "swapping', which is useful in swapping processes in and out of memory and may be necessary to improve in process mix and to free up memory to take care of overcommitted available memory.

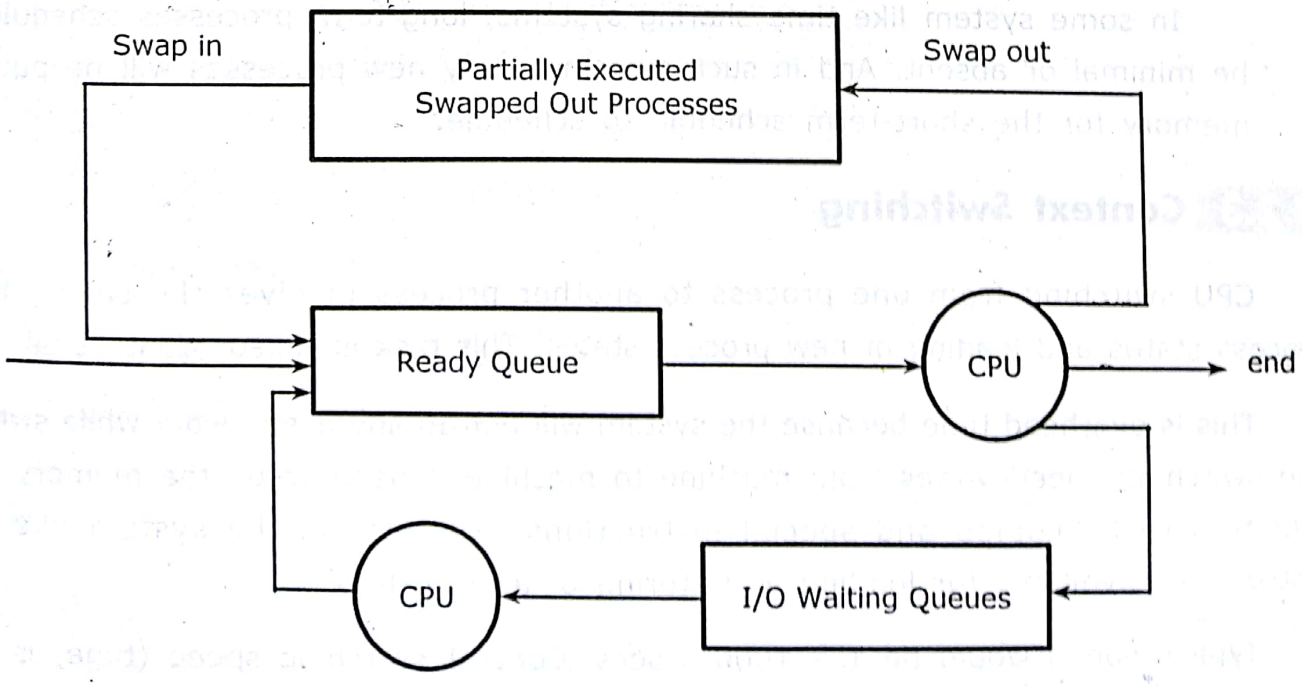


Fig. 3.2.3 Medium Term Scheduling

oufastupdates.com

(3) **Long-Term Scheduler :** Is a batch system, more processes will be submitted and they cannot be executed immediately, hence these processes are spooled onto a mass storage device (disk) and maintained for late execution.

The long-term scheduler selects processes from this pool and loads them into memory for execution.

Long-term scheduler executes less frequently, typically may be minutes between creation of new processes in the system.

Long-term scheduler controls the degree of multi programming that is the number of processes in memory. If the degree of multiprogramming is stable, then the average rate of process creation must be equal to the average departure rate of processes leaving the system. Hence long term scheduler is required to be invoked when a process leaves the system.

Because of the longer interval between executions, the long-term scheduler can afford more time to decide process for execution. But long term scheduler must make a careful selection of jobs, since most of processes can be either input bound or CPU bound.

If all processes are CPU bound, then the device queue will be empty always, and devices are unused, hence system will be unbalanced.

So, a system to achieve best performance, must have to have proper combination of CPU and input bound processes.

In some system like time sharing systems, long-term processes scheduler may be minimal or absent. And in such systems every new processes will be put in the memory for the short-term schedule to schedule.

oufastupdates.com

MULTITHREADED PROGRAMMING

The most prominent use of the concept of thread is an arrangement, in which multiple threads may exist within a single process. More explicitly, this approach is taken in Windows, NT, OS/2, the Sun Version of Unix, is an important operating system known as Mach.

Mach is designed specifically to work in a multiprocessor environment, although it is well suited to a single processor system. In Mach, a task is defined as the unit of protection, or the unit of resource allocation.

The following are associated with tasks,

(1) A virtual address space that holds the task image.

(2) Protected access to processors, other processes (for interprocess communication) files, and I/O resources (devices and channels).

(3) A thread execution state (running, ready etc.).

(4) A saved processor context when not running, one way to view a thread is an independent program counter operating within a task.

(5) An execution stack.

(6) The thread is a static storage for local variables.

(7) Access to the memory and resources of its task, shared with all the other threads in that task.

The key benefits of threads derive from the performance implications. It takes less time to create a new thread in an existing process than to create a brand-new task, less time to terminate a thread, and less time to switch between two threads within the same process. Thus, if there is an application or function that can be implemented as a set of related units of execution, it is far more efficient to do so, as a collection of threads rather than as a collection of separate tasks.

The following are four examples of uses of threads in a single-user multitasking system,

(1) Foreground and Background Work : This is in the sense of directly interacting with the user, not as in foreground and background session.

Example : In a spreadsheet program, one thread could display menus and read user input while another thread executes user commands and updates the spreadsheet.

oufastupdates.com

(2) **Asynchronous Processing :** Asynchronous elements in the program can be implemented as threads.

Example : As a protection against power failure, we can design a word processor to write its RAM buffer to disk once every minute. A thread can be created whose sole job is periodic backup and which schedules itself directly with the operating system, there is no need for fancy code in the main program to provide for time checks or to co-ordinate input and output.

(3) **Speed Execution :** A multithreaded process can compute one batch of data while reading the next batch from a device. On a multiprocessing system, multiple threads from the same process will actually be able to execute simultaneously.

(4) **Organizing Programs :** Programs that involve a variety of activities or a variety of sources and destinations of input and output may be easier to design and implement by using threads.

Benefits of Multithreading

The benefits of a multithreaded programming are,

(1) **Responsiveness :** Multithreading allows a program to continue running, even if part of a process (or) a thread is blocked (or) it is busy performing responsiveness to the user.

Let us consider an example of a web browser, which contains many threads. If one thread is busy in downloading an image, then other thread still allow user interaction.

(2) **Resource Sharing :** All threads belonging to one process share a resource, like memory. The advantage of sharing same address space is that it will take less context switching time.

(3) **Economy :** Allocating memory and resources for creating a process is costly. Since threads belonging to one process share resources like memory etc. i.e., all threads uses same address space, it helps the economy to create and context switch threads. For example, In Solaris 2, operating system, creating a thread is 30 times faster than creating a process and context switching of thread is five times faster than a context switching of a process.

(4) **Utilization of Multiprocessors Architecture :** The benefits of multithreading can be greatly increased in a multiprocessor architecture. Multithreading on a multiprocessor system increases concurrency.

oufastupdates.com

MULTITHREADING MODELS

The main purpose of multithreading model is to make associations or associate threads at the user side and threads at kernel side with one another.

There are three models that can make this association possible. They are,

(1) Many-to-One Model : In this model, associations are made between one thread at the kernel side and many threads at the user side.

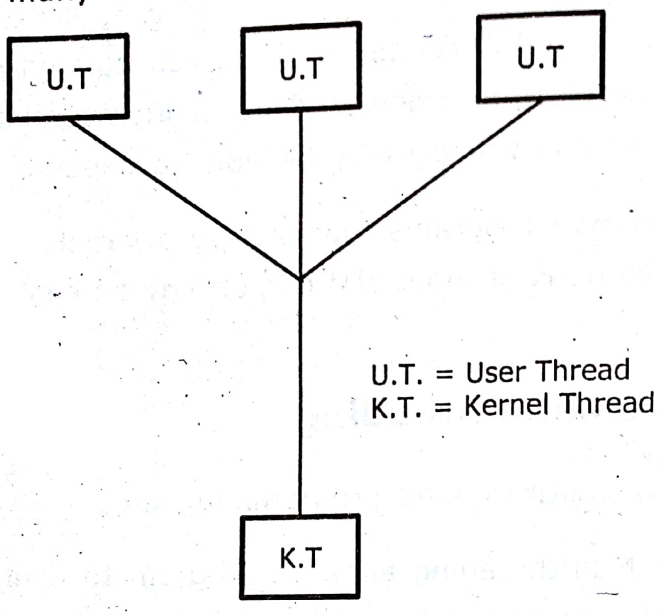


Fig. 4.2.1 Many-to-One Model

(2) One-to-One Model : In this model, associations are made between one thread at the kernel side and one thread at the user side. i.e., suppose there are three kernel and user threads, then one kernel thread is associated with one user thread, then the second kernel thread is associated with another user thread and so on.

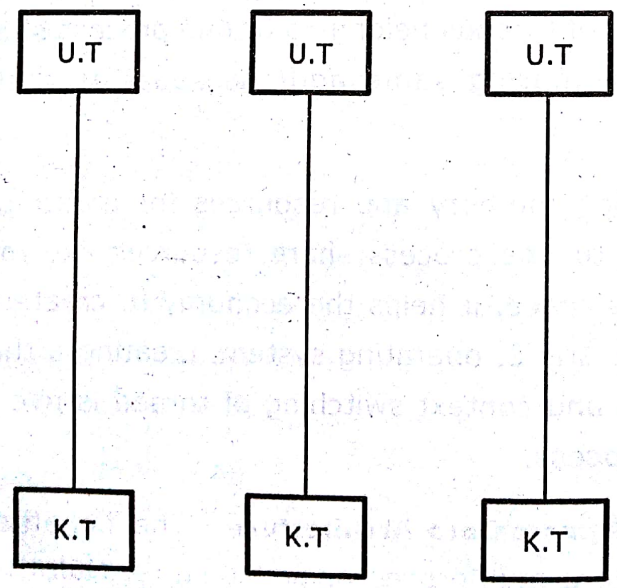


Fig. 4.2.2 One-to-One Model

oufastupdates.com

(3) **Many-to-Many Model :** In this type of model, few threads at the kernel side are associated with more threads at the user side.

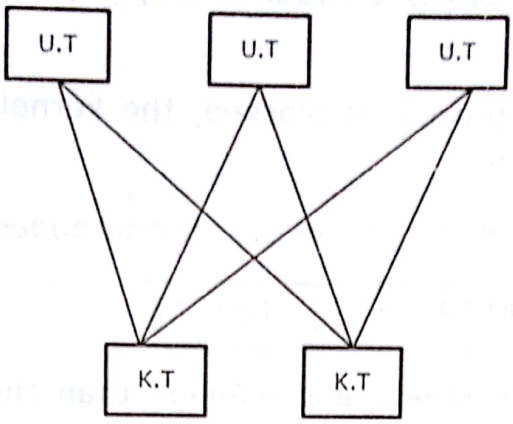


Fig. 4.2.3 Many-to-Many Model

THREAD LIBRARIES

For creating or writing any particular program, we need to have instructions or syntax of how program should be created. Similarly for preparing or creating threads, some instructions or syntax is needed. All these are provided by a special library called thread library.

There are fundamentally two ways of implementing threads,

(1) **User Level Implementation :** The first is to write a user-level library that is substantially self-contained. All of the defining structures and code for the library will be in user space. The vast majority of the library calls will execute entirely in user space and make no more use of system routines than does any other user-level library.

ADVANTAGES OF USER LEVEL THREAD

(1) Thread switching does not require kernel mode privileges.

(2) User level thread can run on any OS.

(3) Scheduling can be application specific.

(4) User level thread are fast to create and manage.

DISADVANTAGES OF USER LEVEL THREAD

(1) In operating system most system calls are blocking.

(2) Multithreaded application cannot take advantage of multiprocessing.

(2) **Kernel Level Implementation :** The second way is to write a library that is inherently a kernel-level implementation. It may define all the same functions as in the first case, but these functions will be completely dependent upon the existence of kernel routines to support them and may well be almost entirely in kernel space. The user-level portion of the library will be relatively small compared to the amount of kernel-level support it requires. The majority of library calls will require system calls.

oufastupdates.com

ADVANTAGES OF USER LEVEL THREAD

(1) Kernel can simultaneously schedule multiple threads from the same process on multiple process.

(2) If one thread in a process is blocked, the kernel can schedule another thread of the same process

(3) Kernel routines themselves can be multithreaded.

DISADVANTAGES OF USER LEVEL THREAD

(1) Generally slower to create and manage than the user threads.

(2) Transfer of control from one thread to another within same process requires a made switch to the kernel.

Both of these methods can be used to implement exactly the same API and they overlap in the kinds of kernel support they require. Some implementations of the POSIX standard are of the first kind, while both OS/2 and Win32 threads are of the second type. When Java is implemented on these OS/2 it inherits the underlying behaviour.

In either case, the programmer will use an API that is implemented by threads library. That library will provide a set of function calls (POSIX has about 50 calls, while Java has a dozen) that is the programmer's sole interface to threads.

Pthreads

Pthreads define an API for thread creation and synchronization. Pthreads refers to the POSIX standard and it is a specification for thread behavior, not an implementation. Numerous systems implement the Pthreads specification including Linux, Solaris, Mac OSX and Tru64 Unix.

All multithreaded programs must make the Pthread function prototypes and symbols available for use. This is accomplished by including the Pthread standard header as,

```
#include<pthread.h>
```

All multithreaded applications must create threads and ultimately destroy them. This is provided in two functions by the Pthreads API,

```
int pthread_create(pthread_at *thread,
pthread_attr_t *attr,
void *(*start_routine) (void *), void *arg);
int pthread_exit(void *retval);
```

The pthread_create function permits the creation of a new thread, while pthread_exit allows a thread to terminate itself.

oufastupdates.com

INTRODUCTION

CPU scheduling is the procedure employed for deciding as to which of the ready processes, the CPU should be allocated. CPU scheduling plays a pivotal role in the basic framework of the operating system owing to the fact that the CPU is one of the primary resources of the computer system.

The algorithm used by the scheduler to carry-out the selection of a process for execution is known as scheduling algorithm. A number of scheduling algorithms are available for CPU scheduling. Each scheduling algorithm influences the resource utilization, overall system performance and quality of service provided to the user. Therefore, one has to reason out a number of criteria to be considered while selecting an algorithm on a particular system.

BASIC CONCEPTS

Since all computer resources are to be scheduled before use, scheduling becomes Operating Systems fundamental function. CPU scheduling is central to Operating Systems design because it is one of the primary computer resources.

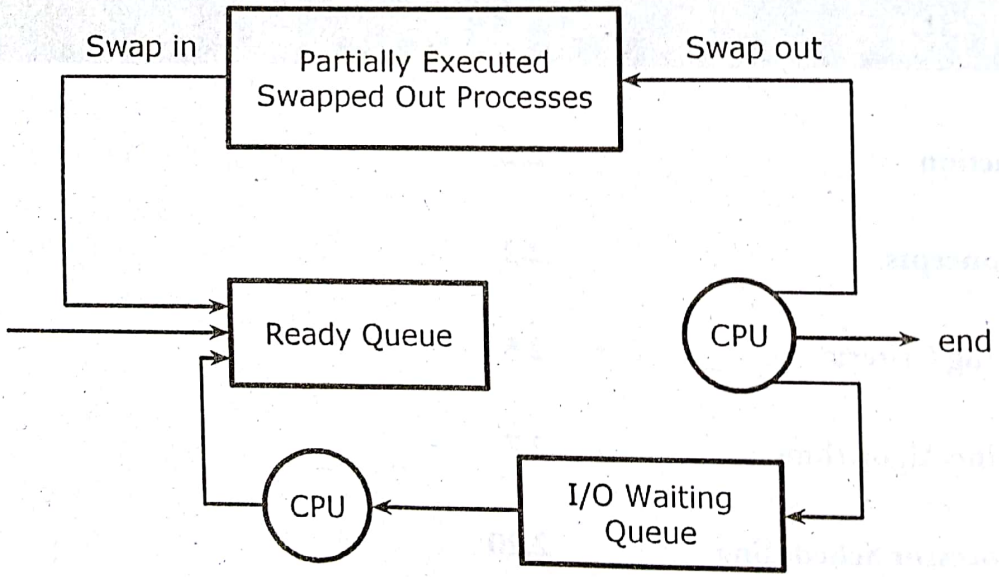


Fig. 5.2.1 CPU Scheduling

The CPU scheduling depends on the following property of processes.

A process execution may consists of a CPU execution and I/O wait cycle and processes alternates between these two. Process execution starts with a CPU burst and that is followed by an I/O burst, which again is followed by CPU burst, and then another I/O burst and so on. The last CPU burst actually ends with a system request to terminate execution rather than with another I/O burst.

oufastupdates.com

CPU I/O Burst Cycle

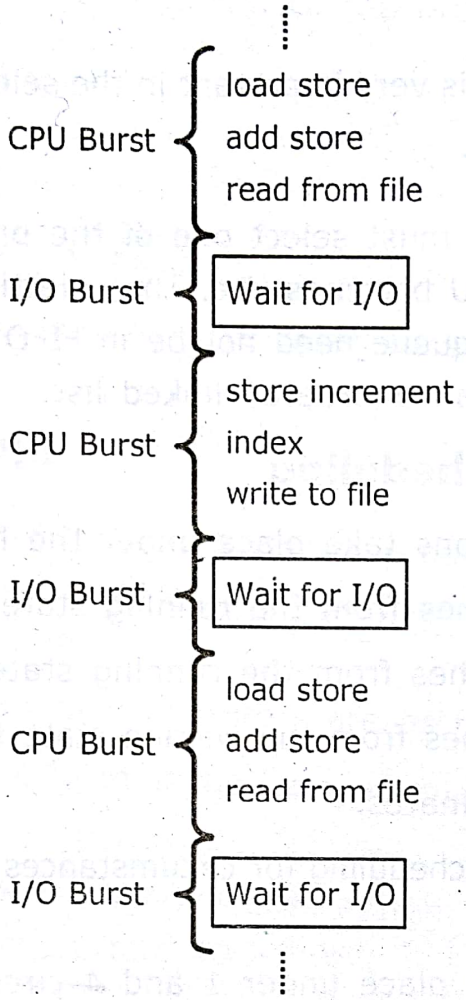


Fig. 5.2.2 Alternating Sequence of CPU and I/O Bursts

CPU burst duration have been measured and it varies largely from system to system and also process-to-process. But frequency curve of them are similar as shown in Fig. 5.2.3.

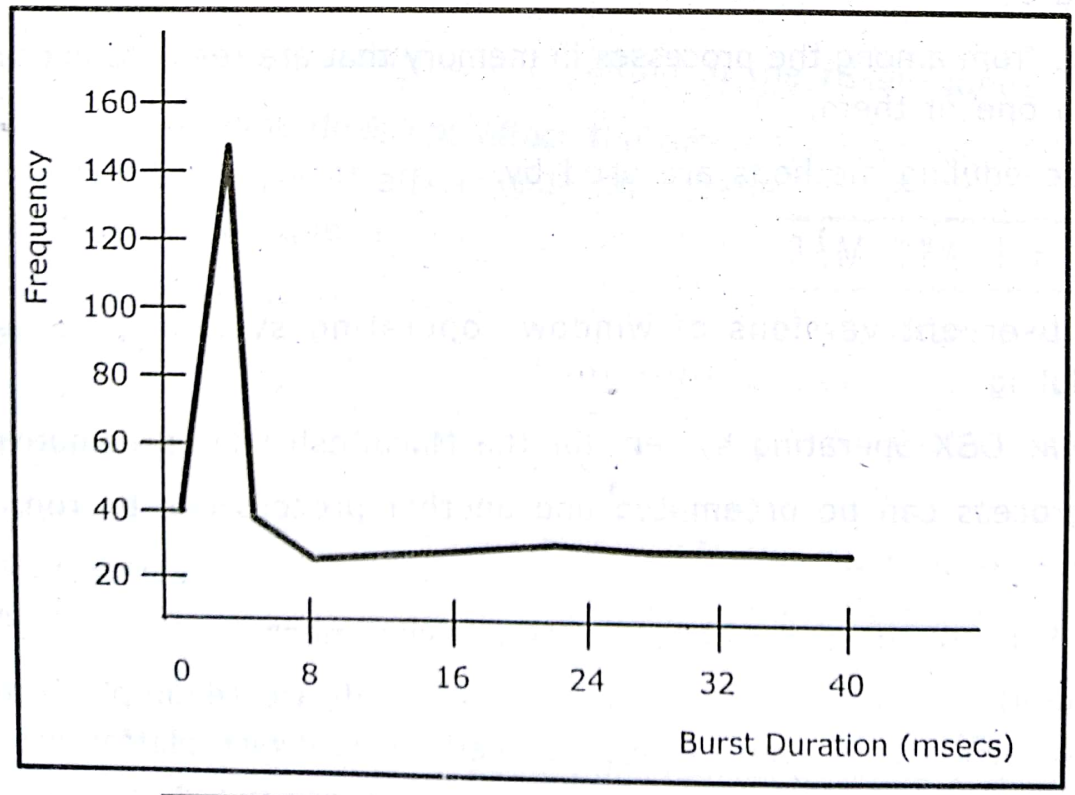


Fig. 5.2.3 Histogram of CPU Burst Durations

ourfastupdates.com

There will be a large number of short CPU bursts and small number of long CPU bursts. An I/O bound program can have many short CPU bursts and CPU bound programs few long CPU bursts.

Hence, job distribution is very important in the selection of CPU scheduling algorithms.

CPU Scheduler

The Operating System must select one of the processes in the ready queue to be executed, whenever the CPU becomes idle. The selection process is carried out by short-term scheduler. The ready queue need not be in FIFO queue, it may be implemented as priority queue, a tree, or an unordered linked list.

Preemptive Scheduling

CPU scheduling decisions take place under the following four circumstance,

(1) When a process switches from the running state to the waiting state I/O request.

(2) When a process switches from the running state to ready state interrupt occurs.

(3) When a process switches from the waiting state to the ready state I/O completion.

(4) When a process terminates.

No choice in terms of scheduling for circumstances 1 and 4, whereas there is a choice for 2 and 3.

If scheduling is taken place under 1 and 4 circumstances, then it is called "non preemptive scheduling", otherwise it is called preemptive type.

In a non preemptive scheduling, once the CPU has been allocated to a process. The process keeps the CPU until it releases CPU either by terminating or by switching to its waiting state.

Selects from among the processes in memory that are ready to execute, and allocates the CPU to one of them.

This scheduling methods are used by,

WINDOWS 3.1 AND MAC

(1) All subservent versions of windows operating system have used preemptive scheduling.

(2) The Mac OSX operating system for the Macintosh uses preemptive scheduling.

The process can be preempted and another process can be run.

UNIX

Example : Scheduling used in MS Windows environment.

This method doesn't require any special hardware (example, timer) needed preemptive type. Hence, can be used on certain hardware platforms.

Preemptive scheduling incurs cost and kernel may be busy when preempte

oufastupdates.com

Dispatcher

Dispatcher module gives control of the CPU to the process selected by the short term scheduler, this involves,

(1) Switching context.

(2) Switching to user mode.

(3) Jumping to the proper location in the user program to restart that program.

(4) Dispatch latency-time it takes for the dispatcher to stop one process and start another running.

SCHEDULING CRITERIA

Different CPU scheduling algorithms have different properties and they may favor one process over another. Hence, while choosing a particular algorithm, properties of various algorithms must be considered.

Criteria used for selecting scheduling algorithm are as follows,

(1) CPU utilization, we know that CPU must be kept as busy as possible and it may range from 0 to 100%.

CPU utilization must be 40% in a lightly loaded system and 90% for heavy loaded systems. In real systems it ranges from 40 – 90%.

(2) *Throughput* is a measure of work done one such measure is the number of processes that are completed per time unit called throughput.

This rate may be one process/work for long processes and 10 processes/see for short processes.

(3) *Waiting Time* is the sum of the time spent waiting in the ready queue.

CPU scheduling algorithms does not affect the amount of time during which a process executes and does I/O. If affects only the amount of time that a process spends waiting in the ready queue.

(4) *Turn around time*, the time taken to execute a process is an important criteria. The interval from the time of submission to the time of completion is the turn around time. And it is the sum of time spent waiting to get into memory, waiting in the ready queue, executing in the CPU and performing I/O.

(5) *Response Time*, this is measure for the time taken from the submission of request until the first response is produced and this measure is called Response Time. And it is the amount of time is takes to start responding and not the time taken to output that response.

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

oufastupdates.com

Types of Scheduling

The CPU scheduling activity is broken down into three separate functions,

(1) **Long-term Scheduling :** The decision to add a program to the pool of processes to be executed.

(2) **Medium-term Scheduling :** The decision to add a process to the pool of processes that are partially or fully in main memory.

(3) **Short-term Scheduling :** This takes place whenever an event occurs that may lead to the interruption of the current process in favour of another.

Examples of such events include,

(i) Clock Interrupts. (ii) I/O Interrupts.

(iii) Operating System Calls. (iv) Signals.

This is the actual decision of which ready process to execute next.

There is one more scheduling other than CPU scheduling called I/O scheduling.

(4) **I/O Scheduling :** The decision as to which process's pending I/O request shall be handled by an available I/O device.

The Fig. 5.3.1 relates the scheduling functions to the process state transition diagram.

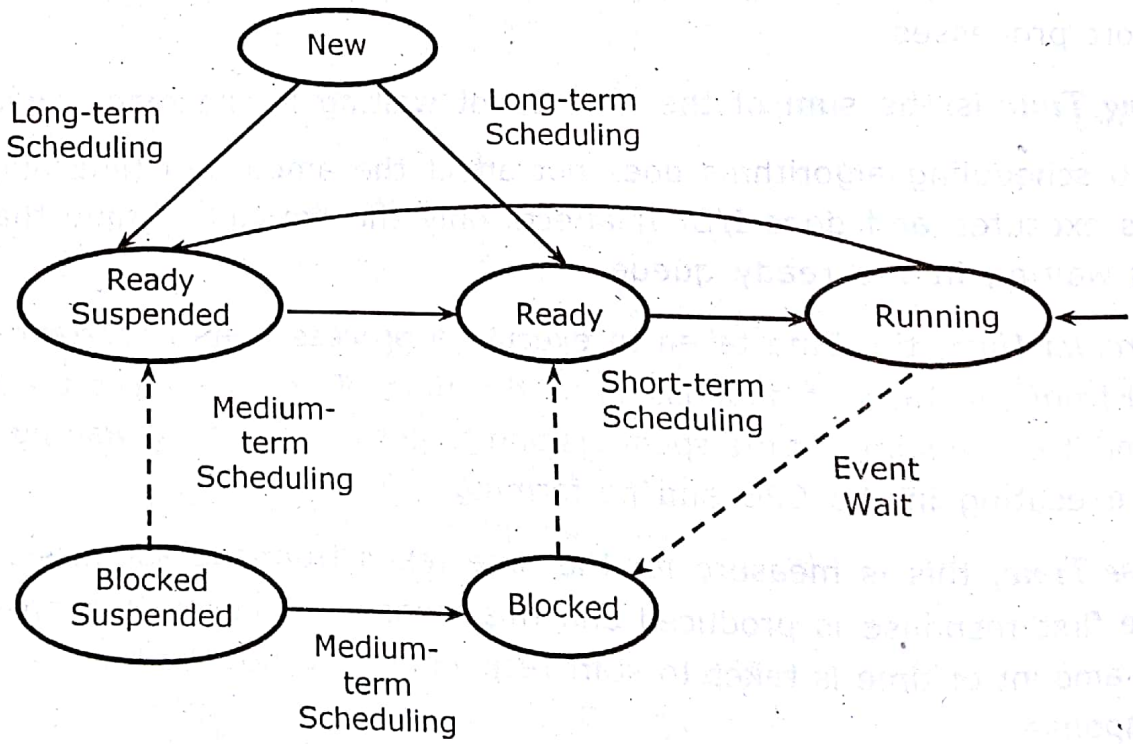


Fig. 5.3.1 Scheduling and Process State Transitions

oufastupdates.com

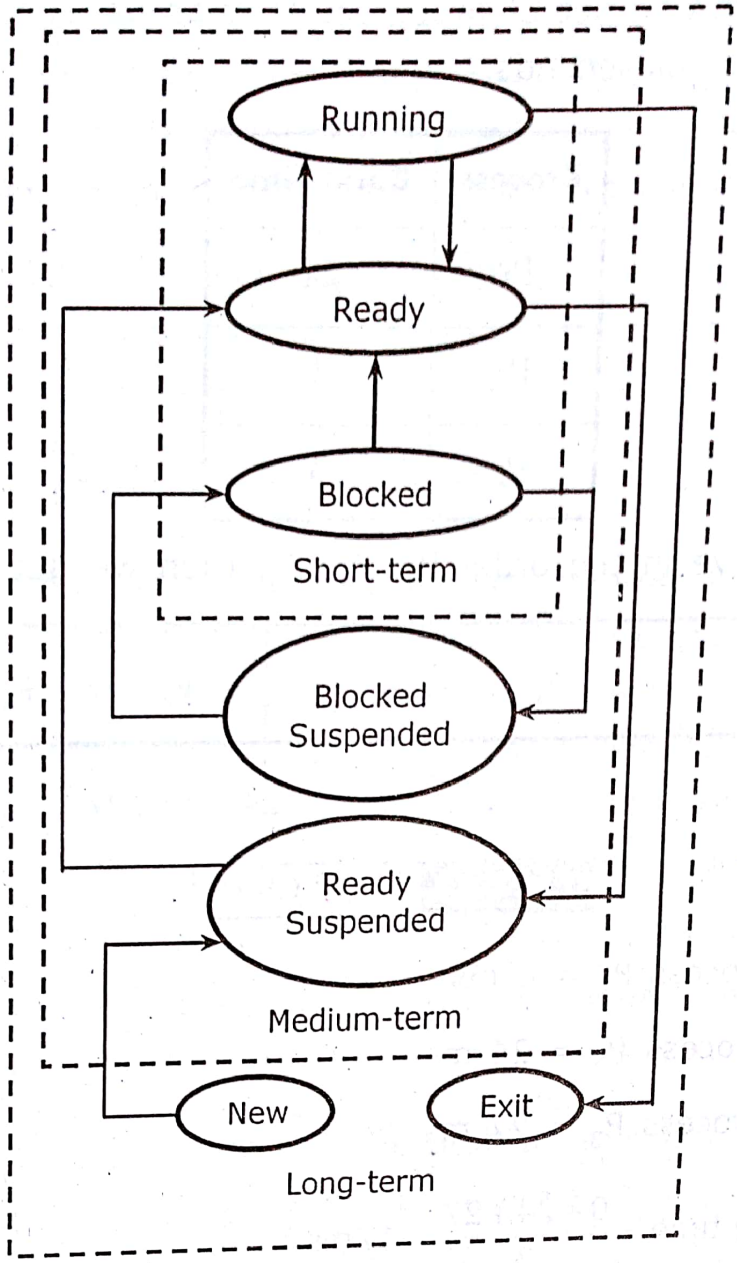


Fig. 5.3.2 Levels of Scheduling

SCHEDULING ALGORITHMS

CPU scheduling deals with the problem of deciding, to which of the processes in the ready queue, the CPU is allocated next.

The different CPU scheduling algorithms are as follows,

First Come First Served Scheduling (FCFS)

This is the simplest algorithm and in this, the process which requests the CPU first is allocated the CPU first. This algorithm is implemented using FIFO queue.

When a process enters ready queue, it Process Control Block (PCB) is linked to the tail of the queue and when CPU is free, it is allocated to the process at the queue head and the running process is then removed from the queue. Hence, FCFS scheduling code is very simple to write and understand.

oufastupdates.com

The average waiting time is FCFS is quite long.

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

Process	Burst Time
P_1	24
P_2	3
P_3	3

If the processes arrive in the order P_1, P_2, P_3, then we get the following result,

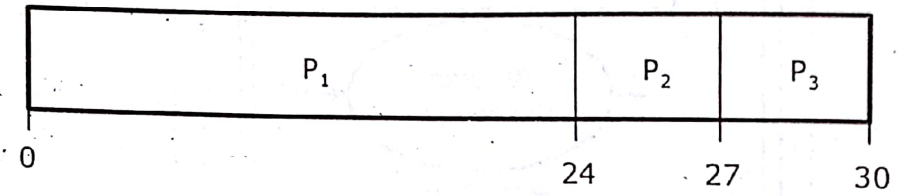


Fig. 5.4.1 Gantt Chart

Waiting time for process P_1 = 0 ms

Waiting time for process P_2 = 24 ms

Waiting time for process P_3 = 27 ms

So, average waiting time $= \frac{0+24+27}{3} = 17$ ms.

If the processes arrive in the order P_2, P_3, P_1, then we get the following Gantt Chart,

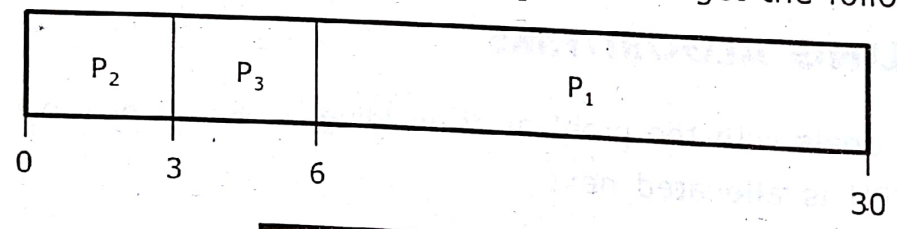


Fig. 5.4.2 Gantt Chart

Waiting time for P_1, P_2, P_3 are now 6ms, 0ms, 3ms respectively.

Average waiting time $= \frac{6+0+3}{3} = 3$ ms.

So, average waiting time varies with the variation in processes CPU burst times.

Another difficulty with FCFS is, it tends to favour CPU bound processes over I/O bound processes. Consider that there is a collection of processes one of which mostly uses CPU and a number of processes which uses I/O devices.

oufastupdates.com

When a CPU-bound process is running, all the I/O-bound processes must wait, which causes the I/O devices to be idle. After finishing its CPU operation, the CPU-bound process moves to an I/O device. Now, all the I/O bound processes, which have very short CPU bursts execute quickly and move back to I/O queues and causes the CPU to sit idle. In this way, FCFS may result in inefficient use of both processor and I/O devices.

Once, the CPU has been allocated to a process, it will not release the CPU until it is terminated or switched to the waiting state. So, this algorithm is non-preemptive. It is difficult to implement for time-sharing systems in which each user get a share of CPU.

Shortest Job First Scheduling (SJF)

Shortest-Job-First (SJF) scheduling is a different approach to CPU scheduling. This algorithm associates with each process the length of the latter's next CPU burst. When the CPU is available, it is assigned to the process that has the smallest next CPU burst. If the two processes have the same length of CPU burst then FCFS scheduling algorithms is followed. As scheduling is done by examining the length of next. CPU burst of a process rather than its total length so it is also called shortest next CPU burst scheduling algorithm.

The key concept of this algorithm is "CPU is allocated to the process with least CPU-burst time". This algorithm is considered to be an *optimal algorithm*, as it gives the minimum average waiting time as a result.

The *disadvantage* of this algorithm is the problem to know the length of time for which CPU is needed by a process. A prediction formula may be used to predict the amount of time for which CPU may be required by a process. Let t_n be the length of nth CPU burst and let τ_{n+1} be our predicted value for the next CPU burst. Defining a parameter 'α' that controls the relative weight of recent and past history in our prediction such that $0 \le \alpha \le 1$ then a formula called *exponential average* as defined as,

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

Where t_n contains our most recent information and τ_n stores the past history in prediction.

If $\alpha = 0$, then $\tau_{n+1} = \tau_n$ $\Rightarrow$ recent history has no effect (current conditions are assumed to be transient)

If $\alpha = 1$, then $\tau_{n+1} = \tau_n$ $\Rightarrow$ in this case only the most recent CPU burst matters.

If $\alpha = 1/2$ then recent and past history are equally weighted.

The SJF Algorithm may be Either Preemptive or Non-Preemptive : The choice arises when a new process arrives in the ready queue while a previous process is executing.

oufastupdates.com

A *preemptive SJF* will preempt this currently executing process and starts the execution of newly entered process while *non-preemptive SJF* will allow the currently executing process to complete its burst time without any interruption in its execution.

Preemptive SJF scheduling is sometimes called Shortest-Remaining-Time-First (SRTF) scheduling.

EXAMPLE PROBLEM 5.1

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

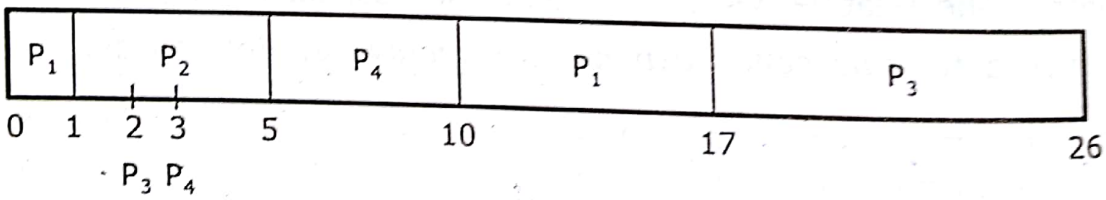
Calculate average waiting time in,

(i) Preemptive SJF scheduling.

(ii) Non-preemptive SJF scheduling.

SOLUTION

(i) Preemptive SJF



At t = 0 ms only one process P_1 is in the system, whose burst time in 8 ms ; starts its execution. After 1 ms, i.e., at t = 1, new process P_2 (burst time = 4 ms) arrives in the ready queue. Its burst time (4 ms) is less than the remaining burst time of P_1 (7 ms) so P_1 is arrive in ready queue but its burst time (9 ms) is larger then the remaining burst time of currently executing process (3 ms). So P_2 is not preempted and it continuous its execution. Again after one ms at t = 3, new process P_4 (burst time = 5 ms) is arrived but again for the same reason P_2 is not preempted until its execution is completed. At t = 5 all the processes have arrive in the system.

oufastupdates.com

Now the processes are executed in SJF scheduling manner, resulting as following,

Waiting time for process P_1 = 0 ms + (10 – 1) ms = 9 ms

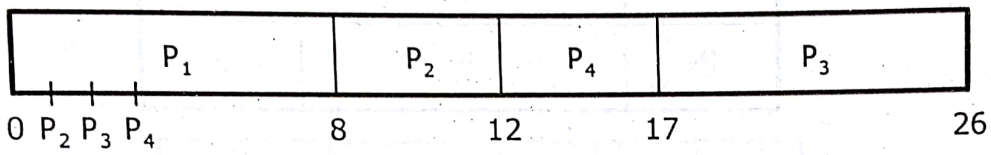
Waiting time for process P_2 = 1 ms – 1 ms = 0 ms

Waiting time for process P_3 = 17 ms – 2 ms = 15 ms

Waiting time for process P_4 = 5 ms – 3 ms = 2 ms

Average wait time $\boxed{\frac{9+0+15+2}{4}} = \boxed{\frac{26}{4}}$ = 6.5 msecs.

(ii) Non-Preemptive SJF



At t = 0 ms only one process is in the ready queue P_1 (burst time = 8 ms). Since in non-preemptive scheme when a process starts its execution then it is not preempted unless its burst time is completed. So, even P_2, P_3 and P_4 are arrived one after the another in the duration of one ms P_1 continues its execution upto 8 ms. After this processes are executed in the accordance of SJF scheduling resulting as following,

Waiting time for process P_1 = 0 ms

Waiting time for process P_2 = (8 – 1) ms = 7 ms

Waiting time for process P_3 = (17 – 2) ms = 15 ms

Waiting time for process P_4 = (12 – 3) ms = 9 ms

$$\therefore \text{ Average waiting time} = \frac{0+7+15+9}{4} \text{ ms}$$

$$= \frac{31}{4} = 7.75 \text{ ms}$$

Priority Scheduling

SJF is a general priority scheduling algorithm. A priority is associated with each process and a process with highest priority is allocated in the CPU. Processes with equal priority are scheduled using FCFS order.

SJF algorithm is a priority algorithm where the priority is the inverse of the predicted next CPU burst. Larger the CPU burst, the lower the priority and vice versa.

Priorities are some fixed range of numbers. Numbers doesn't indicate the priority level i.e., some systems use low numbers to represent low priority and others use low numbers for high priority.

oufastupdates.com

Let us use low numbers to represent high priority.

Example

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

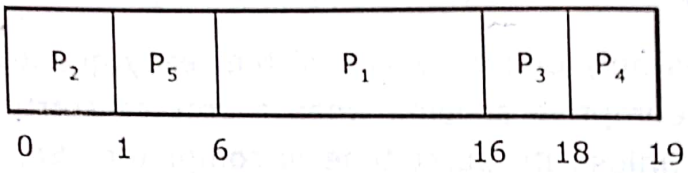


Fig. 5.4.3 Gantt Chart

∴ Average wait time = $P_1 + P_2 + P_3 + P_4 + P_5 / 5$

$$\frac{6+0+16+18+1}{5} = 8.2 \text{ msec}$$

Priorities can be defined externally or internally.

Internally defined priority uses measurable quantities to compute the process priorities. Like, time limits, memory requirements, ratio of I/O burst to CPU burst and number of open files.

While external priorities are set by the external criterias which are external to operating system like for example, Amount and type of funds paid for computer use, importance of process, department sponsoring the work and some other factors.

Priority scheduling can be either preemptive or non-preemptive. When a process arrives at the ready queue, its priority is compared with priority of currently running process.

A pre-emptive scheduling preempts the CPU, if the newly arrived process has high priority than currently running process. Where as, in a non-preemptive scheduling, places the newly arrived process at the head of the ready queue.

The problem with this algorithm is indefinite blocking of process or starvation of processes.

oufastupdates.com

A process which is ready to run but not availed with CPU is considered blocked, waiting for the CPU.

And this algorithm may leave the lower priority processes waiting indefinitely for the CPU.

And in a heavily loaded systems, the high priority process stream may prevent the low priority process from waiting the CPU time.

This may happen because of following two things,

(1) The process may eventually be run.

(2) System may eventually crash and loose all the unfinished low priority processes.

Solution to this problem is *Aging*, it is a technique of increasing gradually the priority of processes which are waiting in the system for CPU, for a long-time.

Round Robin Scheduling

This is designed specially for time sharing systems and is similar to FCFS, but there is preemption in order to switch between processes. It use a small time unit called time quantum or time slice. It varies from 10 – 100 ms.

Ready queue here is considered as circular queue and CPU scheduler goes around the ready queue, allocation CPU to each process for 1 quantum time interval.

This scheduling maintains ready queue in FIFO order of processes. Thus whenever a new process enters the ready queue, is added to queue tail.

The CPU schedular picks the first process from this queue and sets a timer to interrupt after 1 quantum time and dispatches the process.

Following two possible operations may take place,

(1) Process may have a CPU burst of 1 quantum time, then process itself releases CPU voluntarily, allowing schedular to proceed to next process in the ready queue.

(2) Process may have a CPU burst of more than 1 quantum time, then the time goes off, causing an interrupt to O.S. which results in context switch execution and thus placing process at the tail of the ready queue. And then CPU schedular selects next process in the ready queue.

Average wait time in RR scheduling is often long.

oufastupdates.com

Example

Processes	Burst Time (msecs)
P_1	22
P_2	2
P_3	2

Use a time quantum of 4 msecs

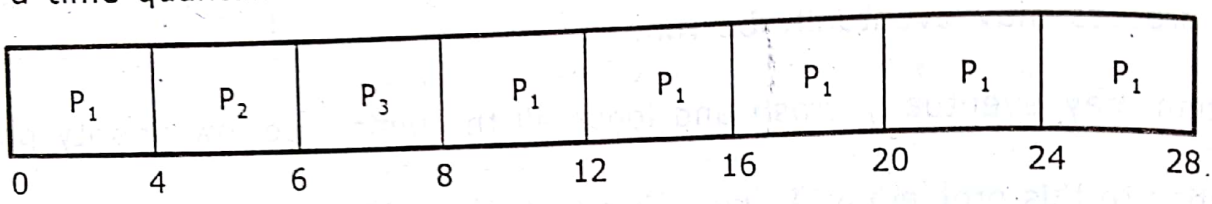


Fig. 5.4.4 Round Robin Process

∴ Average wait time = 4.6 msecs.

Since P_1 gets only first 4 msecs, it requires another 18 msec, and it is preempted after first time quantum and CPU is given to next process in the queue i.e., P_2. But P_2 doesn't require full quantum time, it quits before that quantum expires.

Similarly P_3, quits before the quantum time expires and CPU is then returned to P_1 for required additional time quantum.

In this RR scheduling, no process is allocated the CPU for more than one time quantum. It is preemptive scheduling.

Performance in this scheduling depends more size on time quantum.

If time slice is very large, RR becomes similar to FCFS. If it is too small (typically 1 msecs), then RR approach is called processor sharing.

Multilevel Queue Scheduling

When processes are classified into two different groups, this scheduling techniques can be used for example, whenever a common division has to be made between foreground (interactive) processes and background (batch) processes. These two types of processes requires different response time and hence needs different scheduling techniques. *Fore ground* processes may have priority (externally defined) over back ground processes. A multilevel queue scheduling partitions the ready queue into several separate queues as shown in Fig. 5.4.5.

Where processes are assigned a queue, depending on certain properties of process, like memory size, process type or its priority.

oufastupdates.com

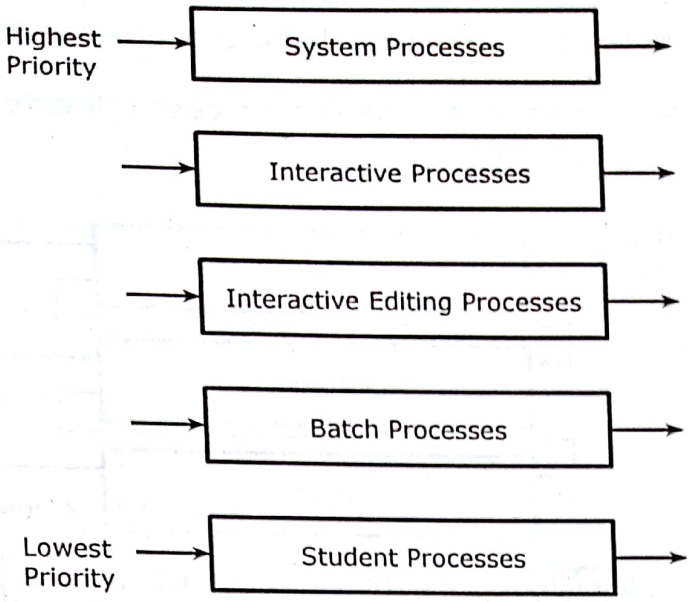


Fig. 5.4.5 Multilevel Queue Scheduling

Each queue can have its own scheduling algorithm. Separate queues can be used for foreground and background processes. And foreground queue may be scheduled using FCFS and background queue may scheduled using RR. But there must be scheduling between queues.

Each queue has absolute priority over lower priority queues. No process in the batch queue, could run unless the other high priority queues are all empty, if an interactive editing process entered the ready queue, while a batch process was running, then batch process would be preempted. Another possibility is to time slice between the queue. Each queue gets a certain portion of CPU time.

Multilevel Feedback Queue Scheduling

In a multilevel queue scheduling, processes are assigned permanently to a queue after entering into the system. Processes do not move between queues. Hence, has an advantage of low scheduling overhead, but is inflexible.

Multilevel feedback queue allows processes to move between queues. The idea here is to separate processes with different CPU burst characteristics.

If a process uses too much CPU time, then it will be placed in a lower priority queue. The processes that waits too long in a lower priority queue may be moved to higher priority queue, which actually prevents starvation (aging).

In general, MLFS algorithm is complex to design, and this scheduler queue is defined using following parameters,

(1) The Number of Queues.

(2) Method used to determine when to upgrade a process to a high priority queue.

(3) Scheduling algorithm for each queue.

oufastupdates.com

(4) Method used to determine when to demote a process to a lowest priority queue.

(5) Method used to determine which queue a process will enter when that process needs service.

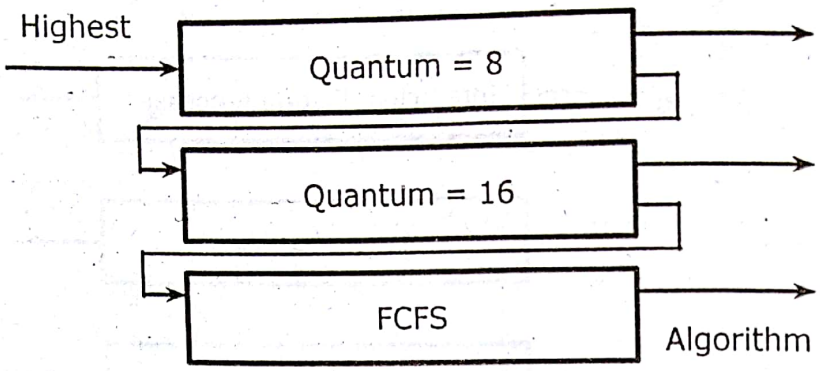


Fig. 5.4.6 Multilevel Feedback Queues

In the above diagram, scheduler first executes all processes in queue 0, then 1 and then 2. A process that arrives for queue 0 will preempt a process in queue 1.

A process entering ready queue is put in queue 0 and a given time quantum of 8 ms. If it does not finish within this time, it is moved to tail of queue 1 and given 16 ms. If it does not complete, it is preempted and put into queue 2. Processes in queue 2 are executed on FCFS basis only when queues 0 and 1 are empty.

Examples Based on Scheduling Algorithms

EXAMPLE PROBLEM 5.1

For the following example calculate average turnaround time and average waiting time for the following algorithms,

(i) FCFS.

(ii) Preemptive SJF.

(iii) Round Robin (1 time unit).

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

oufastupdates.com

SOLUTION

(i) FCFS

- Gantt chart

P_1	P_2	P_3	P_4

0 8 12 21 26

- Waiting time

Process	Waiting Time
P_1	$0 - 0 = 0$
P_2	$8 - 1 = 7$
P_3	$12 - 2 = 10$
P_4	$21 - 3 = 18$

- $\textit{Average waiting time} = \frac{0+7+10+18}{4} = \frac{35}{4} = 8.75$ msecs

- Turnaround time = Burst time + Waiting time.

Process	Turnaround Time
P_1	$8 + 0 = 8$
P_2	$4 + 7 = 11$
P_3	$9 + 10 = 19$
P_4	$5 + 18 = 23$

- $\textit{Average turnaround time} = \frac{8+11+19+23}{4} = \frac{61}{4} = 15.25$ msecs

(ii) Preemptive SJF

- Gantt chart

P_1	P_2	P_4	P_1	P_3

0 1 5 10 17 26

oufastupdates.com

- Waiting time

Process	Waiting Time
P_1	$0 + (10 - 1) = 9$
P_2	$1 - 1 = 0$
P_3	$17 - 2 = 15$
P_4	$5 - 3 = 2$

- $\textit{Average waiting time} = \frac{9+0+15+2}{4} = \frac{26}{4} = 6.5$ msecs

- Turnaround time

Process	Turnaround Time
P_1	$8 + 9 = 17$
P_2	$4 + 0 = 4$
P_3	$9 + 15 = 24$
P_4	$5 + 2 = 7$

- $\textit{Average turnaround time} = \frac{17+4+24+7}{4} = \frac{52}{4} = 13$ msecs

(iii) Round Robin (1 time unit)

- Gantt chart

P_1	P_2	P_3	P_4	P_1	P_2	P_3	P_4	P_1	P_2	P_3	P_4	P_1	P_2	P_3	P_4

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

P_1	P_2	P_3	P_4	P_1	P_2	P_3	P_4	P_1	P_2

17 18 19 20 21 22 23 24 25 26

- Waiting time

Process	Waiting Time
P_1	16
P_2	9
P_3	15
P_4	11

- $\textit{Average waiting time} = \frac{16+9+15+11}{4} = \frac{51}{4} = 12.75$ msecs

oufastupdates.com

- Turnaround time

Process	Turnaround Time
P_1	$8 + 16 = 24$
P_2	$4 + 9 = 13$
P_3	$9 + 15 = 24$
P_4	$5 + 11 = 16$

- $$\textit{Average turnaround time} = \frac{24+13+24+16}{4} = \frac{77}{4} = 19.25 \text{ msces}$$

EXAMPLE PROBLEM 5.2

For the processes listed below, draw a Gantt chart using priority scheduling. A larger priority number has higher priority,

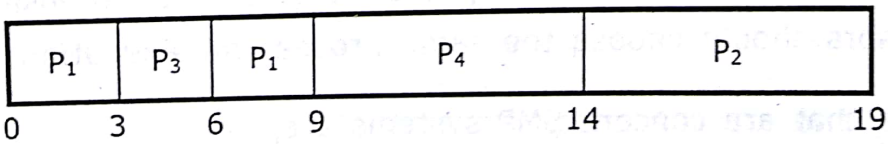
(i) Preemptive.

(ii) Non-preemptive.

Process	Arrival Time	Burst Time	Priority
P_1	0.0	6	4
P_2	3.0	5	2
P_3	3.0	3	6
P_4	5.0	5	3

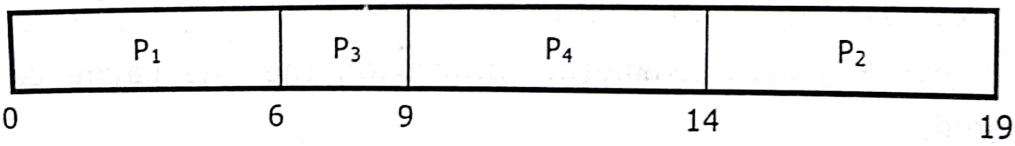
SOLUTION

(i) Preemptive Priority : Gantt chart



oufastupdates.com

(ii) Non-preemptive Priority : Gantt chart



SYNCHRONIZATION

An unordered execution of cooperating processes may result in data inconsistency. To understand the concept, consider two cooperating processes P_1 and P_2 that update the balance of an account in a bank. The code segment for the processes is given in Table 6.1.1.

Table 6.1.1 Code Segment for Processes P_1 and P_2

Process P_1	Process P_2
Read Balance	Read Balance
Balance = Balance + 1000	Balance = Balance - 400

Suppose that the balance is initially 5000, then after the execution of both P_1 and P_2, it should be 5600. The correct result is achieved if P_1 and P_2 execute one by one in any order either P_1 followed by P_2 or P_2 followed by P_1. However, if the instructions of P_1 and P_2 are interleaved arbitrarily, the balance may not be 5600 after the execution of both P_1 and P_2. One possible interleaving sequence for the execution of instructions of P_1 and P_2 is given in Table 6.1.2.

Table 6.1.2 Possible Interleaved Sequence

Process P_1	Process P_2	Balance
Read Balance		5000
	Read Balance	5000
Balance = Balance + 1000		6000
	Balance = Balance – 400	4600

The above interleaved sequence results in an inconsistent balance, that is, 4600. If the order of last two instructions is interchanged, the balance would be 6000 (again, inconsistent). Note that a situation where several processes sharing some data execute concurrently and the result of execution depends on the order in which the shared data is accessed by the processes is called *race condition*.

To avoid race conditions or such inconsistent situations, some form of synchronization among the processes is required which ensures that only one process is manipulating the shared data at a time. One common way to synchronize the processes is signalling, in which the process generates signals to allow other processes to manipulate the shared data.

oufastupdates.com

THE CRITICAL SECTION PROBLEM

A critical section is a segment of code present in a process in which the process may be modifying or accessing common variables or shared data items. The most important thing that a system should control is that, when one process is executing its critical section, it should not allow other processes to execute in its critical sections.

Before executing critical section the process should get permission to enter its critical section from the system. This is called an entry section. After that process executes its critical section and comes out of it this is called exit section. Then, it executes the remaining code called remainder section. Fig. 6.2.1 shows the general structure of a process. Here the entry section 2 exit section are highlited by using boxes to show the important segments of the code.

```
while(1 )
{
        ----------
        ----------
        ----------
        [ entry section ]
        ----------
        ----------
        ----------   } Critical Section
        ----------
        ----------
        ----------
        [ Exit Section ]
        ----------
        ----------   } Remainder Section
        ----------
        ----------
}
```

Fig. 6.2.1 Structure of a Process

oufastupdates.com

REQUIREMENTS FOR CRITICAL SECTION PROBLEM SOLUTION

The following are the important properties that should be satisfied by critical section implementation or solution.

(1) **Mutual Exclusion :** When a process P1 is in its critical section, then no other process can be executing in their critical section.

(2) **Progress :** When a critical section is not in use and other processes is requesting for it, then it should be granted to only that process which is not executing in its remainder section enter its own critical section.

(3) **Bounded Waiting :** A limit or bound is fixed for each process to enter critical section beyond which it is not allowed to enter in critical section.

There are two general ways for handling critical sections in operating systems. They are,

(1) **Preemptive Kernel :** It allows a kernel mode process to be preempted (i.e., interrupted) during execution.

(2) **Non-preemptive Kernel :** It does not allow a kernel mode process to be preempted during execution, the process will execute until in exits kernel mode or voluntarily leaves control of the CPU. This approach is helpful in avoiding race conditions.

The preemptive kernel is used in real-time system, where a process executing in kernel mode can also be preempted which makes the Kernel more responsive. Windows XP, Windows 2000 and traditional Unix are non-preemptive Kernels whereas Linux of Kernel version 2.6 is preemptive kernel.

oufastupdates.com

Dining-Philosophers Solution Using Monitors

This solution to the dining-philosophers uses monitors, and the restriction that a philosopher may only pick up chopsticks when both are available. There are also two key data structures in use in this solution,

(1) **Enum {thinking, hungry, eating} state[5] :** A philosopher may only set their state to eating i.e., state [i] = EATING, when neither of their adjacent neighbours is eating,

(state[(i + 1) % 5] != EATING && state[(i + 4) % 5] != EATING).

(2) **Condition self[5] :** This condition is used to delay a hungry philosopher who is unable to acquire chopsticks.

In the following solution philosophers share a monitor dp and eat, using the following sequence of operations,

(i) dp.pickup( ) - Acquires chopsticks, which may block the process.

(ii) eat.

(iii) dp.putdown( ) - Releases the chopsticks.

```
monitor dp
{
    enum {THINKING, HUNGRY, EATING} state[5];
    condition self[5];
    void pickup (int i)
```

oufastupdates.com

```
{
    state[i] = HUNGRY;
    test(i);
    if(state[i] != EATING)
    self[i].wait( );
}
void putdown(int i)
{
    state[i] = THINKING;
    test((i + 4)%5);
    test((i + 1)%5);
}
void test(int i)
{
        if((state[(i + 4)%5] != EATING) &&
                        (state[i] == HUNGRY) && (state[(i + 1)%5] != EATING))
    {
        state[i] = EATING;
        self[i].signal( );
    }
}
initialization_code( )
{
    for(int i=0; i<5; i++)
    state[i] = THINKING;
}
}
```

oufastupdates.com

Fig. 6.4.4 A Monitor Solution to the Dining-Philosophers Problem

Readers-Writers Problem

Consider a situation where
we have a file shared
between many people.

If one of the people tries
editing the file, no other
person should be
reading or writing at the
same time, otherwise
changes will not be visible
to him/her.

However if some person
is reading the file, then
others may read
it at the same time.

Precisely in OS we
call this situation
as the readers-writers problem

Problem conditions :

One set of data is shared among a number of processes

Once a writer is ready, it performs its write. Only one writer may write at a time

If a process is writing, no other process can read it

If at least one reader is reading, no other process can write

Readers may not write and only read

oufastupdates.com

Solution when Reader has the Priority over Writer

Here priority means, no reader should wait if the share is currently opened for reading.

Three variables are used: mutex, wrt, readcnt to implement solution

semaphore mutex, wrt; // semaphore mutex is used to ensure mutual exclusion when readcnt is updated i.e. when any reader enters or exit from the critical section and semaphore wrt is used by both readers and writers
int readcnt; // readcnt tells the number of processes performing read in the critical section, initially

Functions for Semaphore :

– wait() : decrements the semaphore value.

– signal() : increments the semaphore value.

Writer process:

Writer requests the entry to critical section.
If allowed i.e. wait() gives a true value, it enters and performs the write. If not allowed, it keeps on waiting.
It exits the critical section.

```
do {
    // writer requests for critical section
    wait(wrt);

    // performs the write

    // leaves the critical section
    signal(wrt);

} while(true);
```

Reader process:

Reader requests the entry to critical section.
If allowed:
it increments the count of number of readers inside the critical section. If this reader is the first reader entering, it locks the wrt semaphore to restrict the entry of writers if any reader is inside.
It then, signals mutex as any other reader is allowed to enter while others are already reading.
After performing reading, it exits the critical section. When exiting, it checks if no more reader is inside, it signals the semaphore “wrt” as now, writer can enter the critical section.
If not allowed, it keeps on waiting.

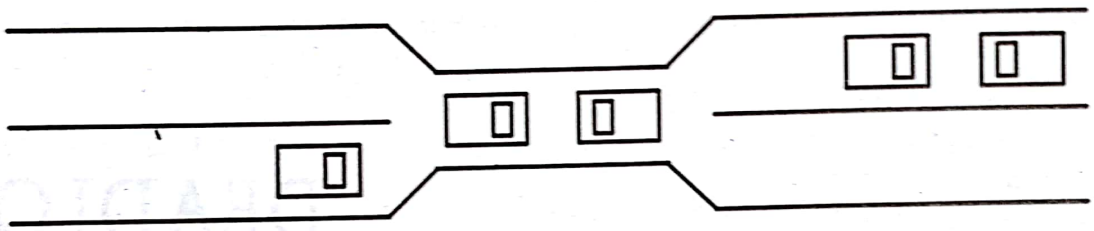
```
do {

   // Reader wants to enter the critical section
   wait(mutex);

   // The number of readers has now increased by 1
   readcnt++;

   // there is atleast one reader in the critical
section
   // this ensure no writer can enter if there is even
one reader
   // thus we give preference to readers here
   if (readcnt==1)
      wait(wrt);

   // other readers can enter while this current
reader is inside
   // the critical section
   signal(mutex);

   // current reader performs reading here
   wait(mutex);   // a reader wants to leave

   readcnt--;

   // that is, no reader is left in the critical
section,
   if (readcnt == 0)
       signal(wrt);         // writers can enter

   signal(mutex); // reader leaves

} while(true);
```

oufastupdates.com

Thus, the semaphore ‘wrt‘ is queued on both readers and writers in a manner such that preference is given to readers if writers are also there. Thus, no reader is waiting simply because a writer has requested to enter the critical section.

DEADLOCK

Deadlock means "*a lock having no keys*". A set of processes is deadlocked, if each of them waits for an event that can be caused only by processes in the set. Thus, each processes waits for an event that cannot occur.



- Traffic only in one direction.
- Each section of bridge can be viewed as a resource.
- If a deadlock occurs, it can be resolved if one car backs up (preempt resources and rollback).
- Several cars may have to backed up if a deadlock occurs.
- Starvation is possible.

Fig. 7.1.1 Bridge Crossing Example

Many process compete for the allocation for a finite number of resources in any multiprogramming environment. When a process is created it requires certain number of resources or when it is executing it may need other resources, which are made available by the operating system.

But if the resources are not available at that time, the requesting process has to wait, and in turn it may enter into a wait state. A waiting process may not change its state again, because there may be a situation in which the resource requested may be held by other waiting processes. This is known as a *Deadlock situation*.

Once the system reaches deadlock is situation the system will hang indefinitely. Therefore, the study of deadlocks in operating systems is very important.

oufastupdates.com

Necessary Conditions

A deadlock situation may occur if the following four conditions hold simultaneously in a system.

Table 7.3.1 Conditions for Resource Deadlock

Condition	Explanation
Mutual exclusion (Non-sharable resources)	Here in mutual exclusion at least one resource is held in a non-sharable mode, i.e., one or more resources in a system cannot be shared simultaneously by different processes. Therefore, only one process is allowed to use the resource at a time. If in case other processes request the same resource which is currently in use, the requesting of processes must be delayed until the resource is released.
Hold and wait	This condition states that there exists a process that is holding at least one resource and is waiting to get additional resources, which are currently being held by other process. For example, at the time of creating a process, it requires the allocation of certain type of resources. When it is ready to execute it may require CPU as a resource. That means it is holding some resources (such as memory or I/O devices) and is waiting for the allocation of other resource (such as CPU) to continue its task.
No preemption	This condition allows a process to hold resources indefinitely until it releases on its own. That is the resources cannot be preempted under any circumstance. The resource has to be released voluntarily by the process holding it.
Circular wait	In this condition there exists a set of process $P_0, P_1, P_2, \ldots P_n$ waiting for resource allocation. The peculiarity of these waiting processes is that P_0 is waiting on the resource that in currently held by P1, P_1 is waiting on P_2, P_3 on P_4 and P_n on P_0. This form of waiting creates a circle.

It is very much required that all the above stated conditions must occur simultaneously to get into deadlock situation. Deadlocks can be more easily be understood with the help of graphs, which are discussed in the next section.

Resource Allocation Graph

Resource allocation graph is used to describe the deadlock. It is also called *system resource allocation graph*. Graph consists of a set of vertices (V) and set of edges (E). All the active processes in the system denoted by $P = \{P_1, P_2, \ldots P_n\}$ and set consisting of all resource type in the system is denoted by $R = \{R_1, R_2, R_3, \ldots R_m\}$. Request edge is an edge from process to resource and denoted by $P_i \rightarrow R_j$. An assignment edge is an edge from resource to process and denoted by $R_j \rightarrow P_i$. Holding of resource by process is denoted by assignment edge. Requesting of resource by process is denoted by request edge.

oufastupdates.com

Let us consider,

$$P_1 \rightarrow R_1 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$$

There is a cycle but no deadlock. Because the process P_4 may release its instance of resource type R_2. That resource can then be allocated to P_3, breaking the cycle.

METHODS FOR HANDLING DEADLOCKS

There are in principle three methods for handling deadlock,

(1) Use a protocol to ensure that the system will never enter a deadlock.

(2) Allow the system to enter deadlock and then recover from it.

(3) Ignore the problem altogether as is done in the UNIX operating system.

Deadlock prevention and deadlock avoidance algorithms fall under the first category. In deadlock prevention, it is ensured that at least one of the necessary conditions for deadlock does not hold. In deadlock avoidance, the information regarding the resources a process will request and use is given in advance. Using this information for each process and the resources currently available, the operating system decides whether the current request can be satisfied or not. If both the above schemes are not used, the system may enter a deadlock.

In that case, an algorithm to detect a deadlock and an algorithm to recover from deadlock is used. If none of the above methods of handling deadlocks is used, the system may enter a deadlock and the system may hang.

In that case, brute force approach of resetting the system can be used. This approach is viable in systems where the possibility of occurrence of a deadlock is less. Instead of spending time in running the above algorithms, the system can be allowed to enter a deadlock and then break the deadlock by resetting the system.

DEADLOCK PREVENTION

Methods for preventing deadlock are of two classes,

(1) **Indirect Method :** An indirect method is to prevent the occurrence of one of the three necessary condition i.e., mutual exclusion, hold and wait and no preemption.

(2) **Direct Method :** A direct method is to prevent the occurrence of a circular wait.

Mutual Exclusion

Mutual exclusion condition must be applied only to non-sharable resources such as printer. Hence, sharable resources do not require mutual exclusion condition at all and therefore they are not involved in the deadlock state.

For example, we can consider a read-only file in multiprogramming environment where many users may share the information. Therefore, a process never need to wait for a sharable resource. But still it is not possible to prevent deadlocks by denying mutual exclusion condition.

oufastupdates.com

Hold and Wait

To prevent hold and wait condition the operating system must ensure that, whenever a process requests a resource it will not hold any other resources simultaneously.

There are two possible solutions by which we can achieve this,

(1) Whenever a process requests resources, all the resources are allocated to it before it executes. This can be done by allowing resource requests are to be processed before any other system calls.

(2) The second solution is to allow a process to request resources only when it does not have any resources. This implies that whenever a process gets some resources it should immediately use them and before it makes additional request, it must release all the resources currently held.

Consider an example of a process which copies data from tape drive to the disk file, sorts it, and prints. In the initial stage the process requires tape drive and disk drive as resources until it copies. Once the copying task is complete it must release both the resources before it makes an addition request for a printer.

DISADVANTAGES

(1) Resource utilization is very less.

(2) Starvation may be possible.

(3) Allows the process to request resources only when process has non (resource).

No Preemption

This condition is also caused by the nature of the resource. This condition can be prevented in several ways,

(1) If a process holding certain resources is defined a further request. That process must release its original resources and if necessary request them again, together with additional resource.

(2) If a process requests a resource that is currently held by another process, the operating system may preempt the second process and require it to release its resources.

(3) In general, sequential I/O devices cannot be preempted.

(4) Preemption is possible for certain types of resources, such as CPU and main memory.

oufastupdates.com

Circular-Wait

In this we can try to break the circular wait condition by considering an ordering of resource types in the system. This requires that each resource type is allocated an unique integer value and whenever a process request the resource, it can be processed in the increasing order.

For example, consider the integer values assigned to the resource type as,

f(tape-drive) = 0

f(disk-drive) = 2

f(Promotes) = 9

Where "f" is any function which given the integer value. In the beginning, a process may request any resource R_0. When it requests next time the resource type R_j, then $f(R_j) > f(R_i)$, there by maintaining the request order. If many instances of the same resource type is requested, one single request may be issued.

DEADLOCK AVOIDANCE

An alternative solution in deadlock management is deadlock avoidance. In this method an additional information about a process and its resource utilization will be obtained well be advance. The system takes a decision before it allocate the new resources to the requesting processes by considering the resources currently available, the resources currently allocated to each process, future requests and releases of resources by each process.

If the request cannot be processed immediately the system make the process to wait for resource allocation in future thus avoiding a possible deadlock. The question is how the system will know the advance information regarding the resources. This can be solved by allotting each process to declare its maximum number of resources of each type including the current requests as well as the future requests.

Then the system will be in a better position to decide and allocate the resources to the requesting processes, without deadlock situation.

Safe State

The state of a system is said to be safe if the system is able to allocate resources to each process in some predefined order and still avoid a deadlock situation. The safe sequence or ordering of allocation of resources is more important for the system to be in safe state.

oufastupdates.com

The Fig. 7.6.1 shows clearly the safe or unsafe state of a system.

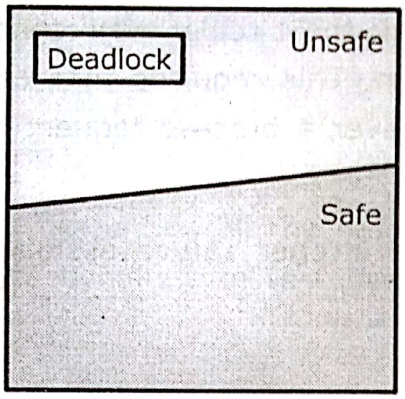


Fig. 7.6.1 Safe/Unsafe State Diagram

The system to be in safe state it must follow a safe sequence for resource allocation. For example, consider the processes P_1, P_2 ... P_n is a safe sequence if for each P_i the request can be satisfied by using currently available resources, along with the resources already held by all P_j where $j < i$.

If in case the request cannot be processed immediately for P_i, it has to wait until all P_j finish their tasks. And also when P_i terminates $P_i + 1$ can avail the resources. This may be repeated for other processes also. Therefore if there is no such safe sequence then the system will be in unsafe state.

The unsafe state does not necessarily mean a deadlock situation. The system in unsafe state may enter into deadlock situation at any moment. Hence, we can conclude that all unsafe state are not deadlocks, also as long as, the operating system is in safe state it can fight and avoid unsafe states.

To understand the safe or unsafe state of a system and safe resource request sequence let us consider an example. Assume that a system has 13 tape drives, and processes P_0, P_1 and P_2 are in the system. The processes maximum and current need of tape drives are 11, 5, 6 and 4, 3, 2 respectively. Hence, the number of free drives available are 4, as shown in Table 7.6.1.

Table 7.6.1 Free Drives

	Max request	Current requst	Available
P_0	11	4	4
P_1	5	3	
P_2	6	2	

oufastupdates.com

To determine the safe sequence we should start with the next need of a process which is less then or equal to the available. From the Table 7.6.1 it is P_1 because it requires only 2 more tape drives out of 4 tape drives available. After completing the task it returns all the 5 tape units. Now, the latest availability of resources increases to 7. Now for the next allocation P_0 and P_2 are competing. P_0 requires 7 and P_2 requires 4. Here, the better choice would be P_2 because still 3 tape drives are free after allocation. If P_0 were to be selected the availability would come down to 0. Therefore, the safe sequence is $\{P_1, P_2, P_0\}$.

Example : Consider another example of having 13 drives altogether. The maximum and current needs are 11, 4, 10 and 5, 3, 2. With this the safe resource is $\{P_1, P_0, P_2\}$. But if P_0 request for an additional tape drive the system moves to unsafe state and the deadlock may appear.

oufastupdates.com

Banker's Algorithm

The major disadvantage of resource allocation graph is to handle the multiple instances of resources. Therefore a separate algorithm has been devices known as Banker's algorithm.

It is called as Banker's algorithm as by Banking system to make sure that bank never enter in deadlock state and is never allocates its available cash to satisfy the need of its customer. Whenever a new process enters a system it must declare the maximum need of resource so it is easy to keep track of processes and resources.

(1) Available : Vector of length M indicates the number of available resources.

(2) Maximum : Matrix n × m which defines the maximum demand of each process.

(3) Allocation : n × m matrix define the number of resources of each type currently allocated to each process.

If allocation (i, j) = k

P_i currently allocated to K instants of resources of R_j.

(4) Need : n × m matrix indicates the remaining resources need of each process Need [i, j] = max [i, j] – Allocation [i, j].

ASSUMPTIONS

Let P and Q be arrays of length m then we say that $x \le y$ only if $x_i \le y_i \ \forall\ i = 1, 2, \ldots n$. If P = (2, 6, 5, 3) and Q = (1, 0, 4, 2) then Q < P if $Q \le P$ and $Q \ne P$.

Thanks for downloading

oufastupdates.com