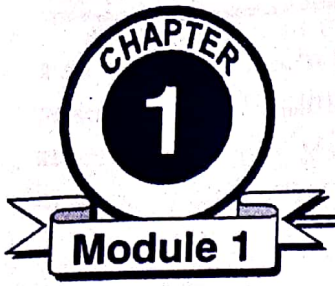


Introduction

Syllabus :

Introduction to Operating System, Objectives and Functions of O.S., OS Services, Special purpose systems, Types of OS, System Calls, Types of system calls, Operating system structure, System Boot.

1.1 Introduction to Operating System :

MU - Dec. 2014, May 2015, Dec. 2015

What is OS ?

An operating system is system software which manages, operates and communicates with the computer hardware and software. To complete the execution user program need many resources. The main Job of the operating system is to provide resources and services to the user program or user program so without a computer operating system, a computer would be useless.

Definition :

- An operating system acts as an interface between the user and hardware of the computer and also controls the execution of application programs.
- Operating system is also called as resource manager.

Need of Operating System :

Basically operating systems perform tasks, for example identifying input from the input devices such as keyboard, mouse etc and sending output to the output devices such as monitor, printer etc and keeping track of files and directories on the disk, and controlling peripheral devices such as secondary storage devices, printers, scanners, audio mixer. The heart of a computer system

is a processing unit called CPU. A computer system should make it available and possible for a user's user program to use the processing unit. The OS makes memory available to a user program when required. Similarly, user programs need use of input facility to communicate with the user program. This is often in the form of a key board, or a mouse or even a joy stick.

1.2 Objectives and Functions of Operating Systems :

MU - Dec. 2014, May 2015, Dec. 2015

Following are the three objectives of Operating System.

1. Convenience
2. Efficiency
3. Ability to evolve

- **Convenience :** Computer system can be conveniently used due to operating system.
- **Efficiency :** An operating system permits the computer system resources to be used in an efficient manner.
- **Ability to evolve :** The operating system is built in a such manner that, it allows the efficient development, testing, and introduction of new system functions without interfering with service.

Functions of Operating System :

Operating system consists of many components. Each component of the operating system has its own set of defined inputs and outputs. Different components of OS perform precise tasks to offer the overall functionality of the operating system. The most important functions of the operating system are as follows :

1. Process Management
2. Memory Management
3. File Management
4. Device Management
5. Protection and Security
6. User Interface or Command Interpreter
7. Booting the Computer
8. Performs basic computer tasks

- **Process Management :** The process management activities involves :
  1. To provide control access to shared resources like file, memory, I/O and CPU.
  2. Control the execution of user applications.
  3. Creation, execution and deletion of user and system processes.
  4. Resume a process execution or cancel it.
  5. Scheduling of a process.
  6. Synchronization, interposes communication and deadlock handling for processes.
- **Memory Management :** The memory management activities handled by operating system are :
  1. Allocation of memory to the processes.
  2. Free the memory from process after completion of execution.
  3. Reallocation of memory to a program after used block becomes free.
  4. Keep track of memory usage by the process.
- **File Management :** The file management activities of operating system consist of :
  1. Creation and deletion of files and directories.
  2. Provide access to files and allocation of storage space for files.
  3. Maintain back-up of files.
  4. File Security.
- **Device Management :** The device management tasks include :
  1. To open, close and write device drivers,
  2. Communicate, control and monitor the device driver.
- **Protection and Security :** The resources of the system are protected by the operating system. User authentication, file attributes such as read, write, encryption, and back-up of data are used by operating system to give necessary protection.
- **User Interface or Command Interpreter :** Operating system offers an interface between the computer user and the computer hardware. The user interface is a set of commands or a graphical user interface through which the user interacts with the applications and the hardware.

- **Booting the Computer :** The process of starting or restarting the computer is known as booting. If computer is switched off completely and if turned on then it is called cold booting. A warm booting is the process of using the operating system to restart the computer.
- **Performs basic computer tasks :** The different peripheral devices such as the mouse, keyboard and printers are managed by operating system. Now days most operating systems are plug and play which means any device will automatically be detected and configured without any user interference.

1.3 Evolution of Operating Systems :

Following are the developments that have occurred in computing facility in the last four to five decades. In the 1960s, people were using mainframe computer system as a computing facility.

- The mainframe computer system would be normally kept in a computer center with a controlled environment. The users used to prepare their program as a deck of punched cards with encoded list of program instructions. Among the deck header cards which were "job control cards" would specify compilers needed to compile the program.
- The operators at the computer system would group the job as per programming languages. The jobs which required long processing time were classified as long jobs and which had short processing time was classified as "short jobs".
- The entire processing was batch processing on the basis of set of jobs. There was no user interaction in processing the jobs. The processor would remain busy at a time in processing one program.
- If any I/O required then processor would wait as processor speed is high with compare to I/O speed. As a consequence of this processor utilization was poor and would remain idle for most of the time. In this scenario only one program was kept in the memory at the time of execution.
- So it was needed to have better utilization of processor and multiple programs in memory at a time so that multiple users would connect to the system.
- In the decade of 1970s, the operating systems designers developed the concept of multiprogramming. In multiprogramming, physical memory was divided in many partitions. Each partition was holding exactly one program and operating system was residing in exactly one partition.

- Along with this policy of memory allocation, a policy was needed to switch the processor from one program to other. It has solved the problem of processor remaining idle while I/O for program is going in.
- This is what the operating system provided leading to the high throughput. A system still was operating in batch processing mode.
- In the early decade of 1980s, system designers added a feature to give interactive access to the system. In this period time sharing systems came in to picture.
- Idea behind the time sharing is to let the use fill that all system resources is given to his program while execution. To achieve it, each user program was allocated a slice of time of processor.
- In this allocated time, user program partly complete the execution. When time slice ends, the processor switches to other program one by one and again come back to first program in very short time.
- This gives the illusion to user that, the system is only available to him alone. In the decade of 1970-80, there was an exceptional growth in bulk storage. It has helped to realize the concept to propose the idea of extended storage.
- Again the concept of "virtual storage" helped to utilize this extended storage as an enhanced address space. By swapping in the active part of the program in memory from disk and swapping out suspended program to the disk supported to enhance the concept of multiprogramming. This led to the large number of users using the system.
- In the mid of 1970 decade, Massachusetts Institute of Technology (M.I.T.) developed first time sharing system called Compatible Time Sharing System (CTSS).
- It was used on IBM-7094 machine. Next was MULTICS developed jointly by M.I.T., General Electric and Bell Laboratory. It was in early 1970s Bell Laboratory scientists developed well known OS called UNIX.
- In decade of 1980s microcomputers came in scene and CP/M was first operating system for this platform. It was developed on Intel 8080 in 1974. MP/M, the version of CP/M, was designed as multiuser, timesharing with real time capabilities.

- After the arrival of IBM PC based on Intel 8086, PC-DOS of IBM and MS-DOS was developed. After the arrival of IBM 286, IBM and Microsoft developed OS/2 for 286 and 386-machines. It was multiuser system.
- After the arrival 386 and 486-based computers, to facilitate and reduce the development time, Microsoft developed Graphical User Interface (GUI) based MS-WINDOWS. It was used on top of DOS to provide user friendly GUI.
- After arrival of Pentium processors, Microsoft developed multiuser Windows NT and single user Windows 95, then to Windows 98 and Windows XP.
- After this, Windows 2000 was developed as a single user operating system for desktop users and as a multiuser system for business users to combine the both streams.
- Now Linux is emerged as more successful operating system which is a variant of Unix. Table 1.1 summarizes this evolution of operating system.

Table 1.1

Sr. No.	Decade	Machines	Operating systems and features
1.	1970	Mainframes	MULTICS. Multiuser, timesharing.
2.	1980	Minicomputers	UNIX. Multiuser, timesharing and networked.
3.	1990	Desktop Computers	DOS, MS-WINDOWS, Wndows-95, Windows 98, Windows XP (for desktop users), Windows NT, UNIX/Linux etc. Multiuser, timesharing, networked, clustered.
4.	2000	Handheld Computers	Windows XP (for desktop user), Windows 2000, Linux, Multiprocessor, Multiuser and networked.

1.4 Operating System Services :

Following are the six services provided by operating system for efficient execution of users application and to make convenience to use computer system :

1. User Interface
2. I/O Operations
3. Communications
4. Program Execution
5. File System Manipulation
6. Error Detection

- **User Interface :** Almost all operating systems have a user interface (UI). Two fundamental approaches for users to interface with the operating system are command-line interface and graphical user interface or GUI. Command interpreter executes next user-specified command. A GUI provides a mouse-based window and menu system as an interface.
- **Program Execution :** The operating system provides an environment to run users programs efficiently. The resources needed to the programs to complete execution are provided by operating system ensuring optimum utilization of computer system. Memory allocation and deallocation, processor allocation and scheduling, multitaskin, etc functions are performed by operating system. The operating system has all rights of resource management. User program does not given these rights.
- **I/O Operations :** Each program requires cannot produce output without taking any input. This involves the use of I/O. During execution, program requires to perform I/O operations for input or output. Until I/O is completed, program goes in waiting state. I/O can be performed in three ways i.e. programmed I/O, Interrupts driven I/O and I/O using DMA. The underlying hardware for the I/O is hidden by operating system from users. These I/O operations make it convenient to execute the user program. Operating system provides this service to user program for efficient execution.
- **File System Manipulation :** Program takes input, processes it and produces the output. The input can be read from the files and produced output again can be written into the files. This service is provided by the operating system. All files are stored on secondary storage devices and manipulated in main memory. The user does not have to worry about secondary storage management. Operating system accomplishes the task of reading from files or writing into the files as per the command specified by user. Although the user level programs can provide these services, it is better to keep it with operating system. The reason behind this is that, program execution speed is fundamental to I/O operations.
- **Communications :** During execution it is required that, the processes running on the same machines or different machines needs to communicate. If the communication is across different machines, messages transit through network. It can be achieved through user programs by customizing it to the

hardware that helps in message transmission. Users programs can be customized by offering the service interface to operating system to achieve communication among processes in distributed system. Operating system kept this service with it to relieve the user program from taking care of communications.

- **Error Detection :** To avoid the malfunctioning of the complete system, the operating system constantly monitors the system for detecting the errors. If these privileges would have been given to user programs, most of the user program time would have elapsed in this activity leading to performance degradation. Operating system needs to perform serious tasks during this process. Serious task includes deallocation of many resources such as processor, memory etc. Again these tasks are too critical to be handed over to the user programs due to possibilities of disturbing the normal operation of operating system.

1.5 Special Purpose Systems :

Apart from general purpose computer system, some classes of computer systems have limited functions and its objectives are deal with limited computation domains. These are :

1.5.1 Real-Time Embedded Systems :

- Embedded computers are the most common form of computers. These devices are used in every field of our life, from car engines and manufacturing robots to VCRs and microwave ovens. These devices perform very specific tasks. These embedded systems differ significantly. These can be general-purpose computers, running standard operating systems such as UNIX or Linux with special-purpose applications to implement the functionality. Some of the systems are hardware devices with a special-purpose embedded operating system providing just the functionality desired.
- The embedded systems are becoming more multifaceted and complex today. Also these systems will affect our life with more involvement. This means they will bear more and more responsibilities on their shoulders to solve real time problems to make our life easier. So real time operating system needs to be effective to manage more complex real time applications.

- Real time operating systems must respond quickly. These systems are used in an environment where a large number of events (generally external) must be accepted and processed in a short time. Real time processing necessitates quick dealing and characterized by providing instant response. For example, a measurement from a petroleum refinery indicating that temperature is getting too high and might demand for immediate attention to avoid an explosion.
- In real time operating system swapping of programs from primary to secondary is not frequent. Most of the time, processes remain in primary memory in order to provide quick response, therefore, memory management in real time system is less demanding compared to other systems. The primary functions of the real time operating system are :
  - Management of the CPU and other resources to fulfill the requirements of an application.
  - Synchronization with and responding to the system events.
  - Efficient movement of data among processes and carrying out coordination among these processes.
- In addition to these primary functions, there are certain secondary functions that are not compulsory but are included to enhance the performance. These are :
  - To provide an efficient management of RAM.
  - To provide an exclusive access to the computer resources.
- Few more examples of real time processing are :
  1. Airlines reservation system.
  2. Air traffic control system.
  3. Systems that provide immediate updating.
  4. Systems that provide up to the minute information on stock prices.
  5. Defense application systems like as RADAR.

1.5.2 Multimedia Systems :

Apart from conventional data, now days operating system should be able to handle multimedia data. Multimedia data comprises audio and video files as well as conventional files. Multimedia data should be delivered to the application in defined time constraint. E.g. Video frames must be streamed as 25 frames per second. Multimedia depicts a broad range of applications that are in well-liked use today. These comprise audio files such as MP3 DVD movies, video conferencing, and short video clips of movie previews or news stories downloaded over the Internet.

Multimedia applications may also comprise broadcasting over the World Wide Web. For example, broadcasting of speeches or sporting events. Multimedia applications include combination of both audio and video. For example, a movie may consist of separate audio and video tracks. Multimedia applications are delivered to desktop personal computers and directed toward smaller devices such as PDAs and cellular telephones as well. The design of operating system should also support for requirements of multimedia systems.

1.5.3 Handheld Systems :

- Personal Digital Assistants (PDAs), for example palm and pocket-PCs and cellular telephones are the examples of handheld systems. Several of such system uses special-purpose embedded operating systems .These devices are having small size and have a small amount of memory, slow processors, and small display screens. Physical memory size of these devices is between 512 KB and 128 MB. Therefore it is a job of operating system and applications to manage memory efficiently.
- A second concern to developers of handheld devices is the speed of the processor used in the devices. Handheld devices require faster processors with compare to PC. Faster processors need more power. So it is obvious that, handheld devices require large battery size and will occupy more space for battery. Therefore most handheld devices use smaller, slower processors that consume less power. As a result, the operating system and applications must be designed not to toll the processor.
- The final issue deal with program designers for handheld devices is I/O. A short of physical space restricts input methods to small keyboards, handwriting recognition, or small screen-based keyboards. The small display

screens also provide constraints on output options. The size of display for a handheld device limits to 3 inches square with compares to 30 inches in PC. Therefore reading e-mail and browsing web pages must be reduced into smaller displays. A solution to this problem is to deliver small set of web page and display called as web clipping.

1.6 Types of Operating System :

The classification of the different operating systems is given below along with a few examples of operating systems that fall into each of the categories. Many computer operating systems will fall into more than one of the following categories.

(1) Multi-user	(2) Multiprocessing	
(3) Multitasking	(4) Multithreading	(5) Real time

1. **Multi-user :**
   - In multiuser operating system multiple users can work on same computer simultaneously. Some operating systems allow hundreds or even thousands of parallel users working on the system.
   - Following are some examples of multi-user operating systems. Linux, UNIX, Windows 2000, Windows.NET.
2. **Multiprocessing :**
   - Multiprocessing operating system supports for multiple CPU in a system. It supports for allocation of different CPU to multiple threads of the program. Due to parallel execution of the threads, efficiency and performance can be achieved. Following are some examples of multiprocessing operating systems.
   - Linux, UNIX, Windows 2000.
3. **Multitasking :**
   - Operating system which supports for multitasking, permits for execution of many software processes at the same time. Multiple tasks are handled concurrently.

- Following are some examples of multitasking operating systems.
  - UNIX
  - Windows 2000.

4. **Multithreading :**
   - A single program can have multiple threads that can run in parallel. The operating system supporting for multithreading can divide the program in multiple threads that can run in parallel.
   - Such operating systems that would fall into this category are:
     - Linux
     - UNIX
     - Windows 2000.
5. **Real Time :**
   - For Real time operating systems time constrains are a key parameter.
   - If the completion of particular task should happen in given time constraints or action completely must take place at a certain instant or within a certain range.
   - Real time operating systems must respond quickly.
   - All above OS like UNIX, Windows are not real time OS.

1.7 System Calls :

MU - May 2015, Dec. 2015

- The interface between OS and user programs is defined by the set of system calls that the operating system offers. System call is the call for the operating system to perform some task on behalf of the user's program. Therefore system calls make up the interface between processes and the operating system.
- The system calls are functions used in the kernel itself. From programmer's point of view, the system call is a normal C function call. Due to system call, the code is executed in the kernel so that there must be a mechanism to change the process mode from user mode to kernel mode.
- In the UNIX operating system, user applications do not have direct access to the computer hardware. Applications first request to the kernel to get hardware access and access to computer resources.

- During execution when application invokes a system call, it is interrupted and the system switches to kernel space. The kernel then saves the process execution context of interrupted process and determines purpose of the call. The kernel warily makes sure that the request is valid and that the process invoking the system calls has sufficient privilege. Some of the system calls can only be invoked by a user with super user privilege.
- If the whole thing is fine, the kernel processes the request in kernel mode and can access the device drivers in charge of controlling the hardware. The data of the calling process can be read and modified by kernel, as it has access to memory in user space. But, it will not execute any code from a user application, for clear security reasons. When the kernel finishes the processing of request, it restores the process execution context that was saved when the system call was invoked. After this, control returns to the calling program which persists executing.

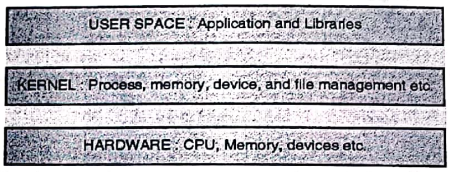


Fig. 1.1 : The kernel

As shown in Fig. 1.1 the kernel is a central module of most computer operating systems. Its main responsibilities are to :

- **Act as a standard interface to the system hardware :** For example, while reading a file, application does not have to be aware of the hard-drive model or physical geometry as kernel provides abstraction layer to the hardware.
- **Manage computer resources :** As several users and programs shares machine and its devices, access to those resources must be synchronized. The kernel implements and ensures a fair access to resources such as the processor, the memory and the devices.

- **Put into effect isolation between processes :** The kernel guarantees that one process cannot corrupt the execution environment of another process. Any process cannot access the memory that the kernel has allocated to other process.
- **Implement multitasking :** Each process gets the false impression that it is running without interrupted on its own processor. Actually, several processes compete constantly for system resources : the kernel keeps switching the active process for each processor behind the scene

There are two ways for the process to switch from the user mode to the kernel mode. These are :

- A user process can explicitly request to enter in kernel mode by issuing a system call.
- During the execution of user process kernel can take over to carry out some system housekeeping task.

The kernel mode is both a software and hardware state. Modern processors offer a advantaged execution mode, called as Supervisor Mode in which only kernel runs. The privileged operations are such as modifying special registers, disabling interrupts, accessing memory management hardware or computer peripherals. If it is not in supervisor mode, the processor will reject these operations.

System calls take place in different ways, depending on the computer in use. Apart from the identity of the desired system call, more information is needed. The precise type and amount of information differ according to the particular operating system and call. Consider the example of getting the input. We may require specifying the source, which can be file or device. We also need to specify the address and length of the memory buffer into which the input should be read. The device or file and length may be implicit in the call.

Parameters can be passed to the operating system by following three different ways :

- Pass the parameters in registers.
- If there are more parameters than registers, then the parameters are generally stored in a block, or table, in memory, and the address of the block is passed as a parameter in a register. This is the approach taken by Linux and Solaris.

- Program can place the parameters onto the stack which then popped off the stack by the operating system.
- Some operating systems favor the block or stack method, because those methods do not limit the number or length of parameters being passed.

1.8 Types of System Calls :

MU - May 2015, Dec. 2015

System calls are categorized in five groups. These are :

1. Process control
2. Device manipulation
3. Communications
4. File manipulation
5. Information maintenance

Sr. No.	Group	Examples
1.	Process control	end, abort, load, execute, create process, terminate process, get process attributes, set process attributes, wait for time, wait event, signal event, allocate and free memory
2.	File manipulation	create file, delete file, open, close, read, write, reposition, get file attributes, set file attributes
3.	Device manipulation	request device, release device, read, write, reposition, get device attributes, set device attributes, logically attach or detach devices
4.	Information maintenance	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
5.	Communications	create, delete communication connection, send, receive messages, transfer status information, attach or detach remote devices.

1.9 Operating System Structure :

1.9.1 Monolithic Systems :

This type of operating system can be treated as having no structure. The operating system is constructed as a set of procedures and each procedure can call any other ones whenever needed. Each procedure in the system has a precise interface in terms of parameters and results. Every procedure can call any other one, if the called one offers some useful computation that the calling procedure needs. In this technique, compilation of all the individual procedures or files containing the procedures is done. After this, linker is used to bind them all together into a single object file. Every procedure is able to be seen to every other procedure so there is no information hiding.

A little structure is possible to impose in monolithic systems. The system calls offered by the operating system are requested by putting the parameters in a well-defined place (e.g., on the stack) and then executing a trap instruction. Due to this instruction machine gets switched from user mode to kernel mode and transfers control to the operating system. The operating system then fetches the parameters and find out which system call is to be executed.

This organization suggests a basic structure for the operating system :

- A main program that calls the demanded service procedure.
- A collection of service procedures that perform the system calls.
- A collection of utility procedures that assist the service procedures.

In this representation, there is single service procedure for each system call which takes care of it. The utility procedures perform things that are desired by several service procedures, such as fetching data from user programs. This separation of the procedures into three layers is shown in Fig. 1.2.

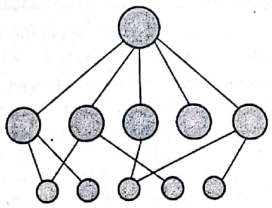


Fig. 1.2 : A simple structuring model for a monolithic system.

1.9.2 Layered Systems :

Another approach of organizing the operating system is as a hierarchy of layers, each one constructed upon the one below it. The first system built in this way was the THE system developed at the Technische Hog school Eindhoven in the Netherlands by E. W. Dijkstra (1968) and his students. As shown in Fig. 1.3 the system had total 6 layers.

- **Layer 0 :** This layer was responsible to handle allocation of the processor, context switching when interrupts occurred or timers expired. This layer was responsible to offer the basic multiprogramming of the CPU.
- **Layer 1 :** The memory management is handled by this layer. It was responsible to allocate space for processes in main memory. If there is no space in main memory, then it also allocates space on a 512K word drum used for holding parts of processes (pages). The layer 1 software was concerned of making needed pages were brought into memory whenever they were required.
- **Layer 2 :** Handled communication between each process and the operator console. Above this layer each process effectively had its own operator console.
- **Layer 3 :** The management of I/O devices and buffering the information streams to and from them was handled by this layer. On top of layer 3 each process could be able to deal with abstract I/O devices with good properties, instead of real devices with many peculiarities.

- **Layer 4 :** The user programs were resided in this layer. They did not have to be concerned about process, memory, console, or I/O management.
- **Layer 5 :** The system operator process was located in layer 5.

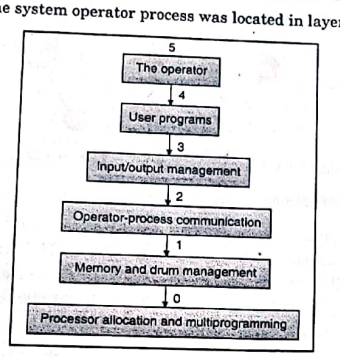


Fig. 1.3 : Structure of the THE operating system

The layered approach is one of the ways to make system modular. In layered approach shown in Fig. 1.4, the operating system is broken up into a number of layers. The bottom layer which is layer 0 , is the hardware and the highest layer is layer N, which is the user interface. An operating system layer is an implementation of an abstract object made up of data and the operations that can manipulate those data. A characteristic operating system layer consists of data structures and a set of routines that can be invoked by higher-level layers. The same layer can too invoke operations on lower-level layers. The advantages of layered approach are that:

- It keeps much better control over the computer and over the applications that utilizes that computer.
- Implementers have more liberty in changing the internal workings of the system and in creating modular operating systems.
- Ease of construction and debugging.

Limitations of layered approach :

- The major trouble with the layered approach is to define various layers properly. As a layer can use only lower-level layers, cautious planning is necessary.
- Layered implementation inclined to be less efficient than other types.

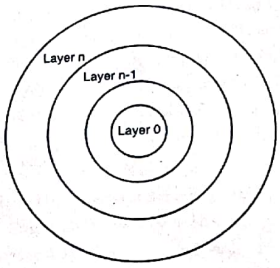


Fig. 1.4 : A layered operating system

1.9.3 Virtual Machines :

The early releases of OS/360 were strictly batch systems. But, due to requirement, many 360 users, decided to have timesharing, so various groups, both inside and outside IBM decided to write timesharing systems for it. The TSS/360, the first time sharing system, arrived late and it was so big and slow that few sites converted to it. It was finally neglected. And it its development cost was very high around some $50 million (Graham, 1970). But a group at IBM's Scientific Center in Cambridge, Massachusetts, produced a radically different system that IBM eventually accepted as a product, and which is now widely used on its remaining mainframes.

This system, originally called CP/CMS and later renamed VM/370 (Seawright and MacKinnon, 1979), was based on an astute observation :

- A timesharing system provides
- Multiprogramming and
- An extended machine with a more convenient interface than the bare hardware.

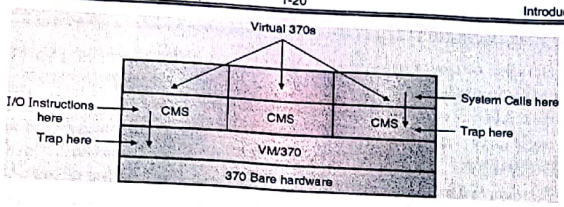


Fig. 1.5

The essence of VM/370 is to completely separate these two functions. The heart of the system, known as the **virtual machine monitor,** runs on the bare hardware and does the multiprogramming, providing not one, but several virtual machines to the next layer up, as shown in Fig. 1.5

However, unlike all other operating systems, these virtual machines are not extended machines, with files and other nice features. Instead, they are *exact* copies of the bare hardware, including kernel/user mode, I/O, interrupts, and everything else the real machine has.

1.9.4 Client-Server Model :

- As a large part of the traditional operating system code moved into a higher layer CMS, the VM/370 got a lot of simplicity. However, VM/370 itself is still a complex program because simulating a number of virtual 370s in their entirety is a tough and complex task.
- A trend in modern operating systems is to take the idea of moving code up into higher layers even further and remove as much as possible from kernel mode, leaving a minimal **microkernel.**
- The usual approach is to implement most of the operating system in user processes. To request a service, such as reading a block of a file, a user process (**client process**) sends the request to a **server process,** which then does the work and sends back the answer.
- In this model, shown in Fig 1.6, all the kernel handles the communication between clients and servers.
- By splitting the operating system up into parts, each of which only handles one facet of the system, such as file service, process service, terminal service, or memory service, each part becomes small and manageable.

- As all servers run as user-mode processes, and not in kernel mode, they do not have direct access to the hardware.
- As a consequence, if a bug in the file server is triggered, the file service may crash, but this will not usually bring the whole machine down. Another advantage of the client-server model is its adaptableness to use in distributed systems.
- The communication of client with a server takes place by sending messages. When client sends a message to server, it is not necessary for client to know whether the message is processed in its own local machine, or whether it was sent to a server on a remote machine in network.
- Either server is present on client machine or on a remote machine in network, from clients point of view it appears to be : a request was sent and a response came back.

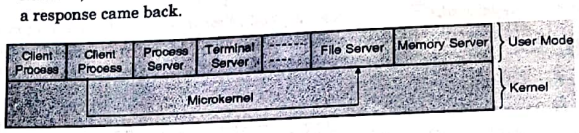


Fig. 1.6 : Client-Server Model

1.9.5 Monolithic Kernel vs. Microkernel :

MU - May 2015, Dec. 2015

- The majority of CPUs support at least two modes of operation i.e. kernel mode and user mode. In **kernel mode,** all instructions are allowed to be executed, and the entire memory and set of all registers is accessible throughout the execution.
- On the contrary to this, in **user mode**, memory and register access is restricted. The CPU gets switched to kernel mode while executing operating system code. The only means to switch from user mode to kernel mode is through system calls as implemented by the operating system.
- The operating system can be put in full control as system calls are the only basic services an operating system provides. The operating system can also be put in full control as hardware helps to restrict memory and register access.
- If virtually entire operating system code is executed in kernel mode, then it is a monolithic program that runs in a single address space. The

disadvantage of this approach is that it is difficult to change or adapt operating system components without doing a total shutdown and perhaps even a full recompilation and reinstallation.

- Monolithic operating systems have drawbacks from the viewpoint of openness, software engineering, reliability, or maintainability.
- If the operating system is organized into two parts, it can provide more flexibility. In the first part, set of modules for managing the hardware is kept which can uniformly well be executed in user mode.
- For example, memory management module keeps track allocated and free space to the processes. It is required to execute in kernel mode at the time when registers of the Memory Management Unit (MMU) are set.
- A small **microkernel** contains only code that must execute in kernel mode. It is the second part of the operating system. Actually, a microkernel require only contain the code for, context switching, setting device registers, manipulating the MMU, and capturing hardware interrupts.
- Microkernel also contains the code to pass system calls to calls on the proper user level operating system modules, and to return their results. Following is the organization of operating system with this approach.

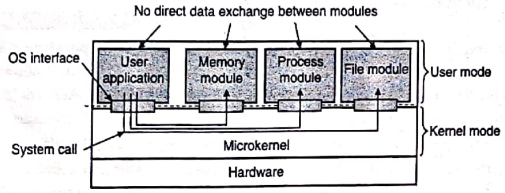
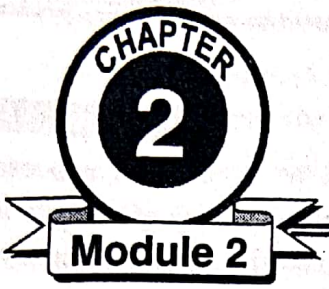


Fig. 1.7 : Separating applications from operating system code through a microkernel

1.10 System Boot :

- Unless the operating system is available in memory to use by hardware computer cannot be ready to use. The process of starting a computer by loading the kernel in main memory is known as *booting* the system.
- A small program called as the **bootstrap program** or **bootstrap loader** finds the kernel, loads it into main memory, and starts its execution.

- On some computer systems, for example PCs, follows two-step process. In first step simple bootstrap loader obtains a more complex boot program from disk, and this complex program then loads the kernel.
- After a CPU receives a reset event, for example, when it is powered up or rebooted, the instruction register is loaded with a predefined memory location, and execution starts there.
- At this location initial bootstrap program resides. This program is in **Read-Only Memory (ROM).** When system starts up the RAM initially is in unknown state. ROM is suitable because it needs no initialization and cannot be infected by a computer virus.
- The bootstrap program runs diagnostics to decide the state of the machine. If the diagnostics pass, the program can carry on with the booting steps. It also initializes all parts of the system, from CPU registers to device controllers and the contents of main memory.
- It then starts the operating system. Several systems like cellular phones, PDAs, and game consoles store the complete operating system in ROM. If the operating systems are small in size, storing it in ROM is appropriate for, simple supporting hardware, and rugged operation.
- A difficulty with this approach is that changing the bootstrap code needs changing the ROM hardware chips. This problem is solved by some system using **Erasable Programmable Read-Only Memory (EPROM).**
- EPROM is read only apart from when clearly given a command to become writable.
- The characteristics of ROM go down somewhere between those of hardware and those of software. Therefore all forms of ROM are also known as **firmware.**
- For large size operating systems like Windows, Mac OS X, and UNIX or for systems that change often, the bootstrap loader is stored in ROM, and the operating system is on disk.
- In this case, the bootstrap runs diagnostics and has a small piece of code that can read a single block at a fixed location (say block zero) from disk into memory and execute the code from that **boot block.**



Process Management

Syllabus :

Process concept : Operations on process, Process scheduling : basic concepts, scheduling criteria, scheduling algorithms, Pre-emptive, Non-pre-emptive, FCFS, SJF, SRTN, Priority based, Round Robin, Multilevel Queue scheduling, Operating System Examples.

Synchronization : Background, the critical section problem, Peterson's Solution, Synchronization Hardware, Semaphores, classic problems of Synchronization: The Producer Consumer Problem: Readers writers problem, Semaphores, Dinning Philosopher Problem.

2.1 Introduction :

- Process manager implements the process management functions. In multiprogramming, single CPU is shared among many processes. If many processes remain busy in completing I/O, CPU is allocated to only one process at a given point of time.
- Here some policy is required to allocate CPU to process, called as CPU scheduling. If multiple users are working on the system, operating system switches the CPU from one user process to other.
- User gets the illusion that only he or she is using the system. Process synchronization mechanism is required to ensure that only one process should use critical section.
- Process communication, deadlock handling, suspension and resumption of processes and creation and deletion of the processes etc are some of the activities performed in process management.

2.1.1 Process :

MU - Dec. 2014

- A program or application under execution is called as process. A process includes the execution context. A program resides on the disk. On disk it does not require any resources. A program gets executed in main memory. So it should be transferred from disk to main memory.
- To complete execution, program needs many resources and competes for it. Now it becomes process. From the computation context point of view, a process is defined by CPU state, memory contents and execution environment.
- A CPU state is defined by the content of the various registers such as Instruction Register (IR), Program Counter (PC), Stack Pointer (SP) and general purpose registers.
- A small amount of data is stored in CPU registers. Memory contains program code and its predefined data structures. Heap is reserved memory area for dynamically allocation of memory to the program at run time.
- In stack, program local variables are allocated and return values of function call are stored. Some register values are also saved in stack.
- Execution environment includes open files, communication channels to other processes etc. following are the components of the process :
  - The object code that is to be executed.
  - Resources required by the program to complete the execution.
  - The data on which program will operate.
  - Program execution state.

2.2 Context Switch :

- When CPU switches from one process to other, a context switch occurs. A context switch is the switching of the CPU (Central Processing Unit) from one process or thread to another.
- When context switch occurs, operating system restores the information of currently executing process for the later use. When CPU again gets allocated to the same process, the restored information is used to resume the execution.

- The context of a process is represented in the Process Control Block (PCB) of a process. The information needs to be restored includes address of the next instruction to be executed (program counter), CPU register contents, pointers to the memory allocated to the process, scheduling information, changed state, I/O state, accounting information etc.
- While context switch, system does not perform any useful work. So context switch is pure overhead on the system.
- The speed of context switching depends on the number of registers that must be copied, memory speed. So context switch speed varies from system to system.

2.3 Operations on Processes :

2.3.1 Process Creation :

Processes are created because of the following four principal events. These are :

- System initialization. When an OS is booted, typically several processes are created.
- Execution of a process creation system call by a running process. Often a running process will issue system calls to create one or more new processes to help it do its job. Creating new processes is particularly useful when the work to be done can easily be formulated in terms of several related, but otherwise independent interacting processes.
- A user request to create a new process. In interactive systems, users can start a program by typing a command or (double) clicking an icon.
- Initiation of a batch job. Here users can submit batch jobs to the system (possibly remotely). When the OS decides that it has the resources to run another job, it creates a new process and runs the next job from the input queue in it.

During the execution process can create new process by using *create process* system call. The process which creates new process is called a **parent** process, and the new processes are called the **children** of that process. Newly created processes may in turn create other new processes, creating a **tree** of processes. In UNIX or the Windows family of operating systems processes are identified by unique **process identifier** (or **pid),** which is typically an integer number. The *ps* command is used in UNIX to obtain the listing of processes. By using command

ps -el complete information for all processes currently active in the system can be obtained.

Process requires certain resources like CPU time, memory, files, I/O devices to complete its task. The subprocess also needs these resources. Subprocess can get its needed resources directly from the operating system, or it may be forced to a subset of the resources of the parent process. The parent can share the resources among its children or it can divide the resources to allocate to its children. If child process is restricted to a subset of the parent's resources overloading the system by creating too many subprocesses can be avoided. The initialization data may be passed along by the parent process to the child process after creation of the process.

After creation of a new process, two possibilities exist related to execution :

- The parent carries on executing in parallel with its children.
- The parent waits until some or all of its children have terminated.

There are also two possibilities related to the address space of the new process :

- The child process may have same program and data as the parent. That is it is duplicate of parent.
- The child process has a new program loaded into it.

2.3.2 Process Termination :

- When process finishes the execution of last statement, it terminates. After this it request the operating system to delete it by using the *exit ()* system call. Just then, the process may return an integer to its parent process by using *wait()* system call.
- Then all allocated resources to the process like physical and virtual memory, open files, and I/O buffers are freed by the operating system. Termination can happen in other situation also.
- By executing the suitable system call, any process can terminate the another process. Normally, parent of the process to be terminated, executes this system call.
- Otherwise, users could randomly kill each other's jobs. It is necessary that a parent requires knowing the identities of its children. Thus, when a process creates a new process, the identity of the child process is passed to the parent.

- Any of the child processes execution can be terminated by child for a diversity of reasons, such as these :
  - If the allocated resource usage is exceeded by child. (The parent must have a means to examine the state of its children.)
  - The task assigned to the child is no longer required.
  - The parent is exiting, and the OS does not permit a child to carry on if its parent terminates.
- Some of the systems, together with VMS, do not permit a child to exist if its parent has terminated. In these systems, if a process terminates either normally or abnormally, then all its children must also be terminated called as cascading termination.
- It is usually initiated by the operating system. In UNIX, process can be terminated by using the exit() system call; its parent process may wait for the termination of a child process by using the wait() system call.
- The wait() system call returns the process identifier of a terminated child with the intention that the parent can tell which of its probably many children has terminated.
- If the parent terminates, however, all its children have assigned as their new parent the *init* process. Thus, the children still have a parent to collect their status and execution statistics.

2.4 Process Control Block :

MU - Dec. 2014

- Any process is identified by its process control block (PCB). PCB is the data structure used by the operating system to keep track on the processes. All the information associated with process is kept in process control block. There is separate PCB for each process.
- Traffic controller module keeps track on the status of the processes. PCB's of the processes which are in the same state (ready, executing, waiting etc) are linked together giving a specific name to the list such as ready list.
- If many processes wait for the same device, then PCBs of all these processes are linked together in chain waiting for that device. If device becomes free then traffic controller checks the PCB chain to see if any process is waiting for the device.

- If the processes are available in that device chain, then again it is placed back to ready state to request that device again. A typical process control block is shown in Fig. 2.1.

Pointer	Current State
Process ID	
Priority	
Program counter	
Registers	
Accounting	
Memory allocations	
Event information	
List of open files	
. . .	

Fig. 2.1 : Process control block

- **Pointer :** This field points to another process control block. Pointer is used for maintaining the scheduling list.
- **Current state :** Current process state may be new, ready, executing, waiting etc as described above.
- **Process ID :** Identification number of the process. It is assigned by operating system.
- **Priority :** Priority of the process.
- **Program counter :** It stores the address of the next instruction to be executed for this process. It is the address from which execution resumes after context switch.
- **Registers :** It indicates general purpose register, index registers stack pointers and accumulators etc. number of register and type of register totally depends upon the computer architecture. The state information of these should be stored after interrupt. This information is again used by the process to resume the execution.
- **Accounting :** Information for calculating the process's priority relative to other processes.

- This may include accounting information about resource use so far. It also includes amount of CPU time and real time used, time limits, process numbers and so on.
- **Memory allocation :** This information may include the value of base and limit register. It includes paging, segmentation related information depending on the memory system used. Address space allocated to the process etc.
- **Event information :** For a process in the blocked state this field contains information concerning the event for which the process is waiting.
- **List of open files :** Files opened by the process.

After creation of the process, loader or linker provides information. As per this provided information, hardware registers and flags are set. Whenever that process is blocked, the contents of the processor register are usually saved on the stack and the pointer to the related stack frame is stored in the PCB. In this way, the hardware state can be restored when the process is scheduled and resume execution again.

2.5 Process States and Process State Transition Diagram :

MU - May 2015, Dec. 2015

- During execution, process changes its state. Process state shows the current activity of the process. Process state contains five states. Each process remains in one of these five states. There is a queue associated with each state of the process.
- Process resides on that queue as per the state in which it resides. The states are listed below.

1. New state
2. Ready state
3. Execution state
4. Waiting (blocked) state
5. Terminated state

1. **New state :** The new process being created.
2. **Ready state :** A process is ready to run but it is waiting for CPU being assigned to it.
3. **Executing state :** A process is said to be in running state if currently CPU is allocated to it and it is executing.
4. **Waiting (blocked) state :** A process can't continue the execution because it is waiting for event to happen such as I/O completion. Process is able to run when some external event happens.
5. **Terminated state :** The process has completed execution.

The process state transition diagram is shown in Fig. 2.2.

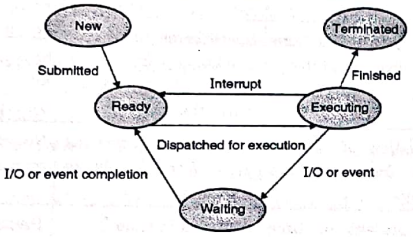


Fig. 2.2 : Process state transition diagram

- When the process is created, it remains in new state. After the process admitted for execution, it goes in ready state. A process in this state, wait in the ready queue. Scheduler dispatches the ready process for execution i.e. CPU is now allocated to the process.
- When CPU is executing the process, it is in executing state. After context switch, process goes from executing to ready state. If executing process initiates an I/O operation before its allotted time expires, the executing process voluntarily give up the CPU.
- In this case process transit from executing to waiting state. When the external event for which a process was waiting happens, process transit from waiting to ready state. When process finishes the execution, it transit to terminated state.

2.6 Process Scheduling :

2.6.1 Scheduling Queues and Schedulers :

- When the process enters into the system, they are put by system in the queue called a job queue. All the processes in the system reside in job queue. When processes are ready for the execution and wait for CPU, they kept in ready queue.
- The operating system also has other queues. When processes wait for particular device, they are kept in respective devices queue.

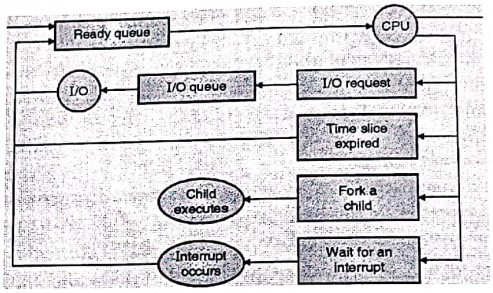


Fig. 2.3 : Queuing-diagram representation of process scheduling

- After allocation of CPU to the process start the execution. While executing the process, one of the numbers of events could occur.
- Until the process finishes the execution, it travels between various scheduling queues. The operating system selects the processes from queues based on some policy.
- This selection is carried out by the program called as scheduler. There are three types of schedulers to schedule a process. These are :

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

2.6.1.1 Long-term Scheduler :

- When programs are submitted to the system for the purpose of processing, long term scheduler comes to know about it. Then its job is to choose processes from the queue and place them into main memory for execution purpose.
- CPU bound processes require more CPU time and less I/O time until execution completes. On the contrary, I/O bound processes use up more time in doing I/O and require less CPU time for computation.
- The primary responsibility of the long term scheduler is to supply a reasonable mix of jobs, such as I/O bound and CPU bound.
- Degree of multiprogramming is high if more number of processes are in memory for execution. It also controls the degree of multiprogramming.
- If the degree of multiprogramming is steady, then the average rate of new process creation and average departure rate of processes leaving the system must be equal.
- The long term scheduler is not present in timesharing operating systems. When process state transition take place from new to ready, then there long term scheduler come into picture for scheduling purpose.

2.6.1.2 Short-term Scheduler :

- Processes which are in ready queue wait for CPU. A short term scheduler chooses the process from ready queue and assigns it to the CPU based on some policy. These policies can be First Come First Served (FCFS), Shortest Job First (SJF), priority based and round robin etc, Main objective is increasing system performance by keeping the CPU busy.
- It is the transition of the process from ready state to running state. Actual allocation of process to CPU is done by dispatcher. Short term scheduler is faster than long tern scheduler and should be invoked more frequently compare to long term scheduler.

2.6.1.3 Medium-term Scheduler :

- If the degree of the multiprogramming increases, medium-term scheduler swap out the processes from main memory. The swapped out processes again swapped in by medium-term scheduler.

- This is done to control the degree of multiprogramming or to free up a memory. This is also helpful to balance the mix of different processes, some time sharing operating system have this additional scheduler.

2.6.1.4 Comparison of Three Schedulers :

Sr. No.	Long-term Scheduler	Short-term Scheduler	Medium-term Scheduler
1.	Selects processes from the queue and loads them into memory for execution.	Chooses the process from ready queue and assigns it to the CPU.	Swap in and out the processes from memory.
2.	Speed is less than the short term scheduler.	Speed is very fast and invoked frequently than long term scheduler.	Speed is in between both short term scheduler and long term scheduler.
3.	Transition of process state from New to Ready.	Transition of process state from Ready to Executing.	No process state transition.
4.	Not present in time sharing system.	Minimal in time sharing system.	Present in time sharing system.
5.	Supply a reasonable mix of jobs, such as I/O bound and CPU bound	Select a new process to allocate to CPU frequently.	Processes are swapped in and out for balanced process mix.
6.	It controls degree of multiprogramming through placing processes in ready queue.	It has control over degree of multiprogramming as it allocates processes to CPU for execution.	Reduce the degree of multiprogramming by swapping the processes in and out.
7.	It is also called as job scheduler.	It is also called as CPU scheduler.	

2.7 Scheduling Algorithms :

2.7.1 Scheduling Criteria :

- In multiprogramming a number of programs can be in memory at the same time. Processes perform I/O operations in the normal course of computation. Since I/O operations ordinarily require more time to complete than do CPU instructions, multiprogramming systems allocate the CPU to another process whenever a process invokes an I/O operation.
- Short term scheduler allocates the processes to CPU as per some policy called as scheduling algorithms. The main aim of the scheduling is to improve performance by keeping CPU busy all the time.
- Criteria for the for performance evaluation of the scheduling strategy is :
  - **CPU Utilization :** It is amount of time CPU remains busy.
  - **Throughput :** Number of jobs processed per unit time.
  - **Turnaround time :** Time elapsed between submission of job and completion of its execution.
  - **Waiting time :** Processes waits in ready queue to get CPU. Sum of times spent in ready queue is waiting time.
  - **Response Time :** Time from submission till the first response is produced.
  - **Fairness :** Every process should get fair share of the CPU time.

2.7.2 Non-Preemptive Vs Pre-emptive Scheduling :

Following are the two types of scheduling algorithms :

- **Non-Preemptive :** Non-preemptive algorithms are designed so that once a process is allocated to CPU, it does not free CPU until it completes its execution.
- **Preemptive :** Preemptive algorithms allow taking away CPU from process during execution. If highest priority process arrives in the system, CPU from currently executing low priority process is allocated to it. It ensures that always highest priority process will be executing.

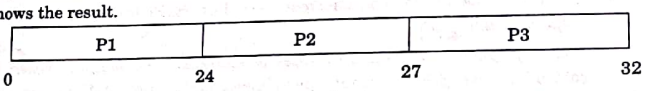
Following are the scheduling algorithms :

2.7.3 First In First out (FIFO) :

- This is a Non-preemptive scheduling algorithm. FIFO strategy allocates the CPU to processes in the order of their arrival. This algorithm treats ready queue as FIFO. A process does not give up.
- CPU until it either terminates or performs I/O. If the longer job assigned to CPU then many shorter jobs has to wait. As long processes can hold the CPU, this algorithm gives fewer throughputs.
- This algorithm can be easily implemented using FIFO queue. In time sharing system every user should get the CPU time at regular interval. So FIFO algorithm is not useful in such situations. FIFO algorithm is inappropriate for interactive systems; large fluctuations in average turnaround time are possible.
- If we assume that arrival time is zero. Consider the following example :

Process	Burst time
P1	24
P2	03
P3	05

Assume that processes arrive in the order P1, P2, and P3.The Gantt chart shows the result.



Turnaround time for P1 = (24 – 0) = 24

Turnaround time for P2 = (27 – 0) = 27

Turnaround time for P3 = (32 – 0) = 32

Average Turnaround time = (24 + 27 + 32) / 3 = 27.67

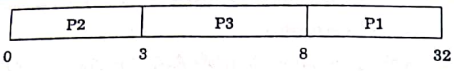
Waiting time for P1 = 0

Waiting time for P2 = 24

Waiting time for P3 = 27

Average Waiting time = (0 + 24 + 27) / 3 = 17

- If the order of arrival is changed and considered as P2, P3, P1, the Gantt chart shows the result.



Turnaround time for P2 = (3 – 0) = 3

Turnaround time for P3 = (8 – 0) = 8

Turnaround time for P1 = (32 – 0) = 32

Average Turnaround time = (3 + 8 + 32) / 3 = 14.33

Waiting time for P2 = 0

Waiting time for P3 = 3

Waiting time for P1 = 8

Average Waiting time = (0 + 3 + 8) / 3 = 3.66

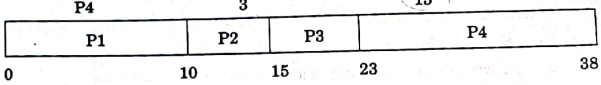
- The result shows that there is significant reduction in average turnaround time and average waiting time. The result varies as order of job arrival varies.

2.7.4 Shortest Job First (SJF) :

- Shortest Job First (SJF) may be preemptive or non preemptive. Reordering the jobs so as to run the shortest jobs first (SJF) improves the average response time.
- Ready queue is maintained in order of increasing job lengths. When a job comes in, insert it in the ready queue based on its length. SJF minimizes the average wait time because it gives service to less execution time processes before it gives service to large execution time processes.
- While it minimizes average wait time, it may punish processes with high execution time. If shorter execution time processes are in ready list, then processes with large service times tend to be left in the ready list while small processes receive service.
- In extreme case, where the system has little idle time, processes with large execution time will never be served. This total starvation of large processes may be a serious problem of this algorithm.
- Consider the example discussed for FCFS algorithm. In this example it is assumed that SJF is non preemptive. If the order of arrival is P2, P3, P1, the order of execution time will be 3, 5, and 24. There is significant reduction in average turnaround time and average waiting time.

- Consider again the following example of non preemptive shortest job first (SJF) algorithm.

Process	Arrival time	Burst time
P1	0	10
P2	1	5
P3	2	8
P4	3	15



Turnaround time for P1 = (10 – 0) = 10

Turnaround time for P2 = (15 – 1) = 14

Turnaround time for P3 = (23 – 2) = 21

Turnaround time for P4 = (38 – 3) = 35

Average Turnaround time = (10 + 14 + 21+ 35) / 4 = 20

Waiting time for P1 = 0

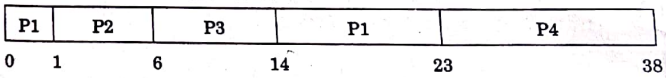
Waiting time for P2 = (10 – 1) = 9

Waiting time for P3 = (15 – 2) = 13

Waiting time for P4 = (23 – 3) = 20

Average Waiting time = (0 + 9 + 13 + 20) / 4 = 10.5

- In preemptive Shortest Job First (SJF), if newly arrived process has less execution time with compare to currently executing process, then the CPU will be given newly arrived process. The preemptive Shortest Job First (SJF) is called as **Shortest Remaining Time First Scheduling (SRTF).** Consider the above example for preemptive Shortest Job First (SJF). The Gantt chart shows the result.



- P1 arrives at time 0, so get the CPU as it is only process in the queue. At time 1, process P2 arrives having execution time 5 (less than remaining time 9 of p1). So P1 is preempted and P2 is scheduled. At time 2, process P3 arrives having execution time 8 which is greater than remaining time of P2.
- So P2 will complete the execution. After P2, process P3 will execute as its execution time is less than P1 (9) and P4 (15). Process P1 has remaining execution time 9. So it is scheduled next. Finally P4 will complete the execution.

Turnaround time for P1 = (23 – 0) = 23

Turnaround time for P2 = (6 – 1) = 5

Turnaround time for P3 = (14 – 2) = 12

Turnaround time for P4 = (38 – 3) = 35

Average Turnaround time = (23 + 5 + 12 + 35) /4 = 18.75

Waiting time for P1 = (14 – 1) = 13

Waiting time for P2 = (1 – 1) = 0

Waiting time for P3 = (6 – 2) = 4

Waiting time for P4 = (23 – 3) = 20

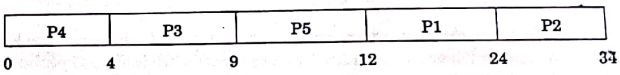
Average Waiting time = (13 + 0 + 4 + 20) / 4 = 18.75

2.7.5 Priority Scheduling :

- In priority scheduling priority number (integer) is associated with each process. The CPU is allocated to the process having the highest priority. Smallest integer is considered as highest priority.
- In some system, largest number can be considered as a highest priority. Priorities can be defined either internally or externally. Internally defined priorities use some measurable quantity or quantities to compute the priority of a process.
- For example, time limits, memory requirements, the number of open files, and the ratio of average I/O burst to average CPU burst have been used in computing priorities.

- External priorities are set by criteria outside the OS, such as the importance of the process, the type and amount of funds being paid for computer use, the department sponsoring the work, and other, often political, factors.
- Preemptive and non preemptive SJF is a priority scheduling where priority is the shortest execution time of job. In this algorithm, low priority processes may never execute. This is called **starvation**.
- Solution to this starvation problem is **aging**. In aging as time progresses, increase the priority of the process so that lowest priority processes gets converted to highest priority gradually.
- Consider the following example for priority scheduling. The Gantt chart shows the result.

Process	Burst time	Priority
P1	12	4
P2	10	5
P3	5	2
P4	4	1
P5	3	3



Turnaround time for P1 = (24 – 0) = 24

Turnaround time for P2 = (34 – 0) = 34

Turnaround time for P3 = (9 – 0) = 9

Turnaround time for P4 = (4 – 0) = 4

Turnaround time for P5 = (12 – 0) = 12

Average Turnaround time = (24 + 34 + 9 + 4 + 12) / 5 = 16.6

Waiting time for P1 = 12

Waiting time for P2 = 24

Waiting time for P3 = 4

Waiting time for P4 = 0

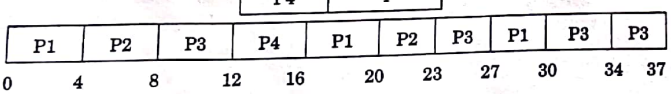
Waiting time for P5 = 9

Average Waiting time = (12 + 24 + 4 + 0 + 9) / 5 = 9.8

2.7.6 Round Robin Scheduling :

- Round Robin Scheduling is designed especially for time-sharing systems where many processes get CPU on time sharing basis. In this algorithm, a small unit of time called time quantum is defined.
- CPU is allocated to each process for this time quantum period of time. To implement this scheduling, ready queue treated as FIFO queue. The new processes go to the tail of queue and each time CPU chooses the process from head of queue.
- When time quantum expires, context switch occurs and CPU switches to other process which is scheduled next. The time quantum is fixed and then processes are scheduled such that no process get CPU time more than one time quantum.
- If process is executing and request for I/O the process goes in waiting (blocked) state. After the completion of I/O, process again gets added at the tail of ready queue. The time quantum should not be very small or very large.
- If the time quantum is very large, the algorithm will behave just like FCFS. A smaller time quantum increases context switches leading to performance degradation (less throughput) as context switches elapses more time.
- Consider the following example. If we use a time quantum of 4 milliseconds then the Gantt chart shows the result.

Process	Burst time
P1	11
P2	7
P3	15
P4	4



Turnaround time for P1 = (30 – 0) = 30

Turnaround time for P2 = (23 – 0) = 23

Turnaround time for P3 = (37 – 0) = 37

Turnaround time for P4 = (16 – 0) = 16

Average Turnaround time = (30 + 23 + 37 + 16) / 4 = 26.5

Waiting time for P1 = 0 + (16 – 4) + (27 – 20) = 19

Waiting time for P2 = 4 + (20 – 8) = 16

Waiting time for P3 = 8 + (23 – 12) + (30 – 27) = 22

Waiting time for P4 = 12

Average Waiting time = (19 + 4 + 22 + 12)/4 = 14.25

2.7.7 Multilevel Queue Scheduling :

- Sometimes it is necessary to categorize the processes into different groups. For example, separation is made between interactive processes and batch processes.
- The response-time need of these two processes can be dissimilar. So these processes can have different scheduling requirement. Also, interactive processes may have higher priority over batch processes.
- A multilevel queue scheduling algorithm divides the ready queue into a number of separate queues. Depending on the property, processes are permanently allocated to one queue. These properties includes such as memory size, process priority, or process type.
- Scheduling algorithm for different queue can be different. The Interactive processes queue might be scheduled by an RR algorithm, while the batch processes queue is scheduled by an FCFS algorithm.
- Also, there must be scheduling among the queues, which is usually implemented as fixed-priority preemptive scheduling.
- Consider multilevel queue scheduling algorithm with five queues, listed below in order of priority:
  1. System processes
  2. Interactive processes
  3. Interactive editing processes
  4. Batch processes
  5. Student processes

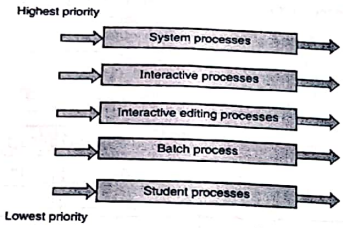


Fig. 2.4 : Multilevel queue scheduling

- Every queue has absolute priority over lower-priority queues. Unless and until higher priority queues become empty, processes in lowest priority queue cannot run.
- For example, processes in interactive editing process queue could not run unless the queues for system processes, interactive processes, were all empty. If a batch process entered the ready queue while a student process was running, the student process would be preempted.
- In this algorithm, every queue gets a definite portion of the CPU time, which it can then schedule among its different processes.
- For example, in the interactive - batch queue example, the interactive queue can be given 80 percent of the CPU time for RR scheduling among its processes, whereas the batch queue receives 20 percent of the CPU to give to its processes on an FCFS basis.

2.7.8 Multilevel Feedback-Queue Scheduling :

- In multilevel queue scheduling processes are not allowed to transfer from one queue to the other. Also processes are permanently assigned to a particular queue. This arrangement has the benefit of low scheduling overhead, but it is inflexible.
- The multilevel feedback-queue scheduling algorithm permits the processes to move between queues. If the processes are CPU-bound, then it use more

CPU time and it will be transferred to a lower-priority queue. This idea keeps I/O-bound and interactive processes in the higher-priority queues. Processes waiting for longer period of time in a lower-priority queue may be transferred to a higher-priority queue. This form of aging prevents starvation.

- Consider the example of a multilevel feedback-queue scheduler with three queues, queue-0, queue-1 and queue-2. Initially the scheduler starts executing all processes in queue-0.
- Once all the processes in queue-0 are executed, and queue-0 becomes empty, it will execute processes in queue-1. In the same way, processes in queue-2 will only be executed if queues 0 and 1 are empty.
- A process arriving for queue-1 will preempt a process in queue-2. In the same way process that arrives for queue-0 will preempt a process in queue-1. A process from the ready queue is placed in queue-0.
- If the process does not finish within time quantum of 8 in queue-0, it is moved to the tail of queue-1. The process at the head of queue-1 is given a quantum of 16 milliseconds if and only queue-0 is empty.
- If it does not finish, it is preempted and is placed into queue-2. Processes in queue-2 are run on an FCFS basis but are run only when queues 0 and 1 are empty.
- If the process with a CPU burst of 8 milliseconds or less then this scheduling algorithm gives highest priority to it. This process will immediately get the CPU, finish its CPU burst, and go off to its next I/O burst.
- Processes that require above 8 but below 24 milliseconds are also served speedily, although with lower priority than shorter processes. Long processes automatically go down to queue 2 and are served in FCFS order with any CPU cycles left over from queues 0 and 1.

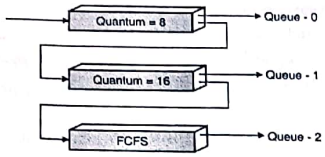


Fig. 2.5 : Multilevel feedback queues

Example 1 : Consider the following set of processes, with the length of the CPU burst given in milliseconds:

Process	Burst Time	Priority
P1	10	3
P2	1	1
P3	2	3
P4	1	4
P5	5	2

The processes are assumed to have arrived in the order P1, P2, P3, P4, P5 all at time 0. Draw four Gantt charts that illustrate the execution of these processes using the following scheduling algorithms: FCFS, SJF nonpreemptive, priority (a smaller priority number implies a higher priority), and RR (quantum = 1) and also calculate turnaround time average waiting time. **MU - Dec. 2014**

Solution :

1) FCFS

P1	P2	P3	P4	P5

0 10 11 13 14 19

Arrival time for all processes is 0.

Process	Priority	Finish time	TAT	Waiting time
P1	3	10	10	0
P2	1	11	11	10
P3	3	13	13	11
P4	4	14	14	13
P5	2	19	19	14
Average			13.4	9.6

2) SJF (Nonpreemptive)

Arrival time for all processes is 0.

P2	P4	P3	P5	P1

0 1 2 4 9 19

Process	Priority	Finish time	TAT	Waiting time
P1	3	19	19	9
P2	1	1	1	0
P3	3	4	4	2
P4	4	2	2	1
P5	2	9	9	4
Average			7	3.2

3) SJF (Preemptive)

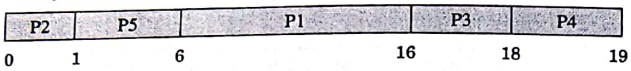
Since all processes arrive at same time 0, the solution is same as nonpreemptive SJF.

P2	P4	P3	P5	P1

0 1 2 4 9 19

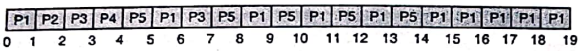
Process	Priority	Finish time	TAT	Waiting time
P1	3	19	19	9
P2	1	1	1	0
P3	3	4	4	2
P4	4	2	2	1
P5	2	9	9	4
Average			7	3.2

4) Priority Scheduling (Nonpreemptive)



Process	Priority	Finish time	TAT	Waiting time
P1	3	16	16	6
P2	1	1	1	0
P3	3	18	18	16
P4	4	19	19	18
P5	2	6	6	1
Average			13.4	8.2

5) Round Robin (time quantum = 1)



Process	Priority	Finish time	TAT	Waiting time
P1	3	19	19	9
P2	1	2	2	1
P3	3	7	7	5
P4	4	4	4	3
P5	2	14	14	9
Average			9.2	5.4

Example 2 : Calculate average waiting time and average turnaround time for the following situation. Assume quantum of 2 ms.

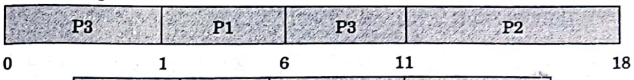
Process	Burst Time	Priority	Arrival time
P1	5	1	1
P2	7	3	5
P3	6	2	0

(i) Preemptive SJF (ii) Preemptive priority (iii) RR

Solution :

(i) Preemptive SJF

Following is the Gantt chart



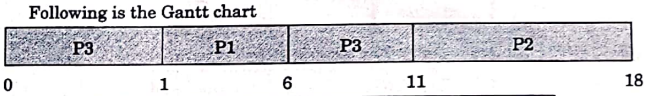
Process	Priority	TAT	Waiting time
P1	1	(6 – 1) = 5	(1 – 1) = 0
P2	3	(18 – 5) = 13	(11 – 5) = 6
P3	2	(11 – 0) = 11	(6 – 1) = 5
Average		9.6	3.66

Average TAT = (5 + 13 + 11)/3 = 9.6

Average waiting time = (0 + 6 + 5) /3 = 3.66

(ii) Preemptive priority

Following is the Gantt chart



Process	Priority	TAT	Waiting time
P1	1	(6 – 1) = 5	(1 – 1) = 0
P2	3	(18 – 5) = 13	(11 – 5) = 6
P3	2	(11 – 0) = 11	(6 – 1) = 5
Average		9.66	3.66

Average TAT = (5 + 13 + 11)/3 = 9.6

Average waiting time = (0 + 6 + 5)/3 = 3.66

(iii) RR

Following is the Gantt chart

P3	P1	P3	P1	P2	P3	P1	P2	P2	P2

0 2 4 6 8 10 12 13 15 17 18

Process	Priority	TAT	Waiting time
P1	1	(13 – 1) = 12	(2 – 1) + (6 – 4) + (12 – 8) = 7
P2	3	(18 – 5) = 13	(8 – 5) + (13 – 10) = 6
P3	2	(12 – 0) = 12	(4 – 2) + (10 – 6) = 6
Average		**12.33**	**6.33**

Average TAT = (12 + 13 + 12)/3 = 12.33

Average waiting time = (7 + 6 + 6)/3 = 6.33

Example 3 : Consider the four processes P1, P2, P3 and P4 with length of CPU burst time. Find out average waiting time and average turnaround time for the following algorithm.

(i) FCFS

(ii) RR (slice = 4 ms)

(iii) SJF

Process	Arrival time	Burst time
P1	0	8
P2	1	4
P3	2	9
P4	3	5

Solution :

(i) FCFS

Following is the Gantt chart

P1	P2	P3	P4

0 8 12 21 26

Process	TAT	Waiting time
P1	(8 – 0) = 8	0
P2	(12 – 1) = 11	(8 – 1) = 7
P3	(21 – 2) = 19	(12 – 2) = 10
P4	(26 – 3) = 23	(21 – 3) = 18
Average	15.25	8.75

Average TAT = (8 + 11 + 19 + 23)/4 = 15.25 ms.

Average waiting time = (0 + 7 + 10 + 18)/4 = 8.75 ms.

(ii) RR (slice = 4 ms)

Following is the Gantt chart

P1	P2	P3	P4	P1	P3	P4	P3

0 4 8 12 16 20 24 25 26

Process	TAT	Waiting time
P1	20	(16 – 4) = 12
P2	(8 – 1) = 7	(4 – 1) = 3
P3	(26 – 2) = 24	(8 – 2) + (20 – 12) + (25 – 24) = 15
P4	(25 – 3) = 22	(12 – 3) + (24 – 16) = 17
Average	18.25	11.75

Average TAT = (20 + 7 + 24 + 22)/4 = 18.25 ms.

Average waiting time = (12 + 3 + 15 + 17)/4 = 11.75 ms.

(iii) Preemptive SJF

Following is the Gantt chart

P1	P2	P4	P1	P3

0 1 5 10 17 26

Process	TAT	Waiting time
P1	(17 – 0) = 17	(10 – 1) = 9
P2	(5 – 1) = 4	(1 – 1) = 0
P3	(26 – 2) = 24	(17 – 2) = 15
P4	(10 – 3) = 7	(5 – 3) = 2
Average	13	6.5

Average TAT = (17 + 4 + 24 + 7)/4 = 13 ms.

Average waiting time = (9 + 0 + 15 + 2)/4 = 6.5 ms.

Example 4 : Suppose that the following process arrive for execution at time indicate

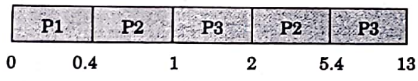
Process	Arrival time	Burst time
P1	0.0	8
P2	0.4	4
P3	1.0	1

Calculate average waiting time and average turnaround time using SRTF and SJF.

Solution :

(i) Preemptive SJF is SRTF (Shortest Remaining Time Next)

Following is the Gantt chart



Process	TAT	Waiting time
P1	(13 – 0) = 13	(5.4 – 0.4) = 5
P2	(5.4 – 0.4) = 5	(2 – 1) = 1
P3	(2 – 1) = 1	(1 – 1) = 0
Average	6.33	2

Average TAT = (13 + 5 + 1)/3 = 6.33 ms.

Average waiting time = (5 + 1 + 0)/3 = 2 ms.

(ii) Non - preemptive SJF

Following is the Gantt chart

Process	TAT	Waiting time
P1	(8 – 0) = 8	0
P2	(13 – 0.4) = 12.6	(9 – 0.4) = 8.6
P3	(9 – 1) = 8	(8 – 1) = 7
Average	9.53	5.2

Average TAT = (8 + 12.6 + 8)/3 = 9.53 ms.

Average waiting time = (0 + 8.6 + 7)/3 = 5.2 ms.

Example 5 : Consider the following set of processes having their CPU burst time (in millisecond).

Process	CPU Burst time	Arrival time
P1	10	0
P2	5	1
P3	2	2

Calculate average waiting time using following CPU scheduling algorithms.

(1) FCFS (2) SJF

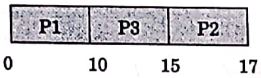
(3) Priority scheduling having priority range from 1 to 3, respectively for process P1 = 3, P2 = 2, P3 = 3 as given

(4) RR (slice = 2)

Solution :

(1) FCFS

Following is the Gantt chart



Process	Waiting time
P1	0
P2	(10 – 1) = 9
P3	(15 – 2) = 13
Average	7.33

Average waiting time = (0 + 9 + 13)/3 = 7.33 ms.

(2) Preemptive SJF

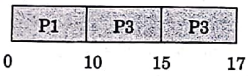
Following is the Gantt chart

Process	Waiting time
P1	(8 – 1) = 7
P2	(4 – 2 – 1) = 1
P3	(2 – 2) = 0
Average	2.67

Average waiting time = (7 + 1 + 0)/3 = 2.67 ms.

(3) Priority scheduling having priority range from 1 to 3, respectively for process P1 = 3, P2 = 2, P3 = 3 as given.

Following is the Gantt chart

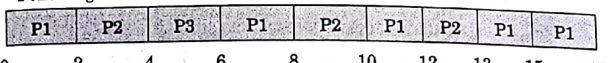


Process	Waiting time
P1	0
P2	(10 – 1) = 9
P3	(15 – 2) = 13
Average	7.33

Average waiting time = (0 + 9 + 13)/3 = 7.33 ms. (Same as FCFS)

(4) RR (slice = 2)

Following is the Gantt chart



Process	Waiting time
P1	(6 – 2) + (10 – 8) + (13 – 12) = 7
P2	(2 – 0) + (8 – 4) + (12 – 10) = 7
P3	(4 – 2) = 2
Average	5.33

Average waiting time = (7 + 7 + 2)/3 = 5.33 ms.

Example 6 : Consider the following set of processes.

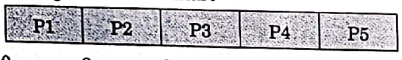
Process	CPU Burst time	Arrival time
P1	3	0
P2	5	1
P3	2	2
P4	5	3
P5	5	4

Calculate average waiting and turnaround time for each algorithm.

(1) FCFS (2) SJF (3) RR (slice = 2)

Solution :

(1) FCFS : Following is the Gantt chart



Process	TAT	Waiting time
P1	(3 – 3) = 0	0
P2	(8 – 1) = 7	(3 – 1) = 2
P3	(10 – 2) = 8	(8 – 2) = 6
P4	(15 – 3) = 12	(10 – 3) = 7
P5	(20 – 4) = 16	(15 – 4) = 11
Average	9.2	5.2

Average turnaround time = (0 + 7 + 8 + 12 + 16)/5 = 9.2 ms.

Average waiting time = (0 + 2 + 6 + 6 + 11)/5 = 5.2 ms.

(2) SJF

Following is the Gantt chart

Process	TAT	Waiting time
P1	(3 – 3) = 0	0
P2	(10 – 1) = 9	(5 – 1) = 4
P3	(5 – 2) = 3	(3 – 2) = 1
P4	(15 – 3) = 12	(10 – 3) = 7
P5	(20 – 4) = 16	(15 – 4) = 11
Average	8.6	4.6

Average turnaround time = (0 + 9 + 3 + 12 + 16)/5 = 8.6 ms.

Average waiting time = (0 + 4 + 1 + 7 + 11)/5 = 4.6 ms.

(3) RR (slice = 2)

Following is the Gantt chart

Process	TAT	Waiting time
P1	(11 – 0) = 11	(10 – 2) = 8
P2	(18 – 1) = 17	(2 – 1) + (11 – 4) + (17 – 13) = 12
P3	(6 – 2) = 4	(4 – 2) = 2
P4	(19 – 3) = 16	(6 – 1) + (13 – 8) + (18 – 15) = 11
P5	(20 – 4) = 16	(8 – 4) + (15 – 10) + (19 – 17) = 11
Average	12.8	8.8

Average turnaround time = (11 + 17 + 4 + 16 + 16)/5 = 12.8 ms.

Average waiting time = (8 + 12 + 2 + 11 + 11)/5 = 8.8 ms.

Example 7 : Assume that you have following jobs to execute with one processor

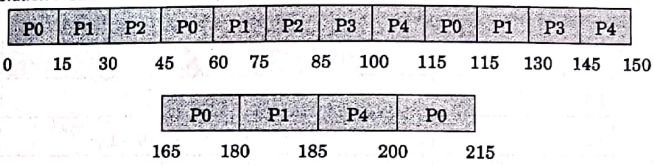
Job	CPU Burst time	Arrival time
0	75	0
1	50	10
2	25	10
3	20	80
4	45	85

Suppose system uses round robin with quantum of 15.

(i) Draw Gantt chart

(ii) Find average wait and turnaround time

Solution : The Gantt chart is,



Process	TAT	Waiting time
P0	(215 – 0) = 215	(45 – 15) + (115 – 60) + (165 – 30) + (200 – 180) = 140
P1	(185 – 10) = 175	(15 – 10) + (60 – 30) + (130 – 75) + (180 – 145) = 125
P2	(85 – 10) = 75	(30 – 10) + (75 – 45) = 50
P3	(150 – 80) = 70	(85 – 80) + (145 – 100) = 50
P4	(200 – 85) = 115	(100 – 85) + (150 – 115) + (185 – 165) = 70
Average	130	87

Average turnaround time = (215 + 175 + 75 + 70 + 115)/5 = 130 ms.

Average waiting time = (140 + 125 + 50 + 50 + 70)/5 = 87 ms.

Example 8 : Use following Scheduling algorithms to calculate ATAT and AWT for the following process

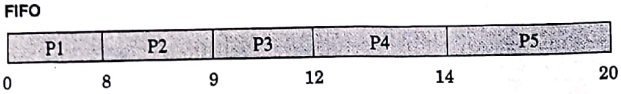
(i) FCFS (ii) Preemptive and non-preemptive SJF

(iii) Preemptive priority.

Process	Arrival time	Burst time	Priority
P1	0	8	3
P2	1	1	1
P3	2	3	2
P4	3	2	3
P5	4	6	4

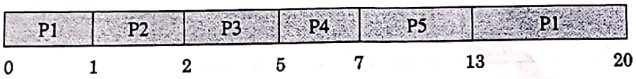
Solution :

(I) FIFO



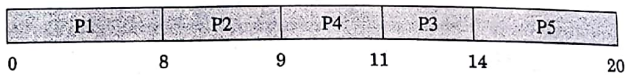
Process	Priority	Finish time	TAT	Waiting time
P1	3	8	(8 – 0) = 8	0
P2	1	9	(9 – 1) = 8	(8 – 1) = 7
P3	2	12	(12 – 2) = 10	(9 – 2) = 7
P4	3	14	(14 – 3) = 11	(12 – 3) = 9
P5	4	20	(20 – 4) = 16	(14 – 4) = 10
Average			(53/5) = 10.6	(33/5) = 6.6

(II) SJF(Preemtive)



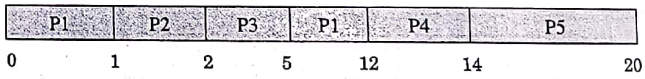
Process	Priority	Finish time	TAT	Waiting time
P1	3	20	(20 – 0) = 20	(13 – 1) = 12
P2	1	2	(2 – 1) = 1	(1 – 1) = 0
P3	2	5	(5 – 2) = 3	(2 – 2) = 0
P4	3	7	(7 – 3) = 4	(5 – 3) = 2
P5	4	13	(13 – 4) = 9	(7 – 4) = 3
Average			(37/5) = 7.4	(17/5) = 3.4

(III) SJF(Nonpreemtive)



Process	Priority	Finish time	TAT	Waiting time
P1	3	8	(8 – 0) = 8	0
P2	1	9	(9 – 1) = 8	(8 – 1) = 7
P3	2	12	(14 – 2) = 12	(11 – 2) = 9
P4	3	14	(11 – 3) = 8	(9 – 3) = 6
P5	4	20	(20 – 4) = 16	(14 – 4) = 10
Average			(52/5) = 10.4	(32/5) = 6.4

(iv) Priority (Preemptive)



Process	Priority	Finish time	TAT	Waiting time
P1	3	20	(12 – 0) = 12	(5 – 1) = 4
P2	1	2	(2 – 1) = 1	(1 – 1) = 0
P3	2	5	(5 – 2) = 3	(2 – 2) = 0
P4	3	7	(14 – 3) = 11	(12 – 3) = 9
P5	4	13	(20 – 4) = 16	(14 – 4) = 10
Average			(43/5) = 8.6	(23/5) = 4.6

2.8 Operating System Examples :

Following are the description of scheduling policies of the Solaris, Windows XP, and Linux operating systems. It is significant to keep in mind that we are describing the scheduling of kernel threads with Solaris and Linux. Since Linux does not differentiate between processes and threads, the term *task* is used while discussing the Linux scheduler.

2.8.1 Solaris Scheduling :

In Solaris thread scheduling is done on the basis of priority. There are four types of scheduling, which are classified, in order of priority :

- Real time
- System
- Time sharing
- Interactive

Each class contains different priorities and different scheduling algorithms.

- Time sharing is the default scheduling class for a process. The scheduling policy for time sharing dynamically changes priorities and multilevel feedback queue is used to allot time slices of different lengths.
- For the higher priority, the time slice is smaller and for the lesser priority, the time slice is larger. Interactive processes normally have a higher priority and CPU-bound processes have a lower priority.
- This scheduling policy provides good response time for interactive processes and good throughput for CPU-bound processes. The interactive class uses

the same scheduling policy as the time-sharing class, but it gives windowing applications a higher priority for better performance.

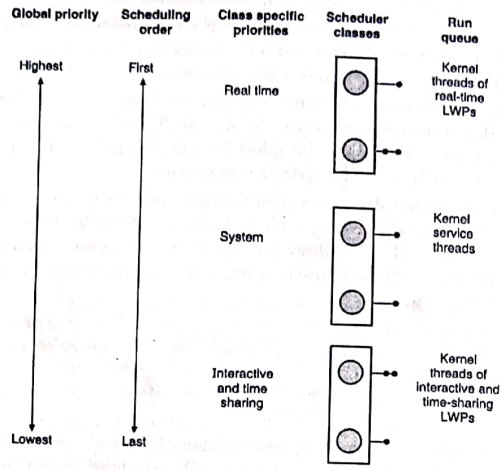


Fig. 2.6 : Solaris scheduling

- Solaris dispatch table for interactive and time-sharing threads contains the following fields. These two scheduling classes comprises 60 priority levels :
  - **Priority :** It is the class-dependent priority for the time-sharing and interactive classes. A higher integer shows a higher priority.
  - **Time quantum :** It shows the time quantum for the related priority. If the priority is lowest (priority 0) then the time quantum is highest (200 milliseconds). For the highest priority (priority 59) the lowest time quantum is (20 milliseconds).
  - **Time quantum expired :** If thread has used its whole time quantum without blocking then this field indicates new priority of this thread. Such threads are CPU-bound and have their priorities lowered.

- **Return from sleep :** If thread comes back from sleep state then this field indicates the priority of such thread.

- Two new scheduling classes for Solaris 9 are **fixed priority** and **fair share.** Both fixed-priority class and time-sharing class threads have same priority range. But, their priorities are not dynamically adjusted.
- In order to make scheduling decisions, fair-share scheduling class uses CPU **shares** in place of priorities. Solaris make use of system class to run kernel processes, for example, the scheduler and paging daemon. Once recognized, the priority of a system process does not vary.
- Systems class does not contain user processes running in kernel mode. The system class is held in reserve for kernel use. Threads within real-time class are assigned the highest priority. This assignment permits a real-time process to have a guaranteed response from the system within a bounded period of time.
- A real-time process will run before any process belonging to any other class. Usually, however, a small number of processes are the member of the real-time class.

2.8.2 Windows XP Scheduling :

- A priority-based, preemptive scheduling algorithm is used by Windows XP to schedule threads. The Windows XP scheduler makes sure that the highest-priority thread will always run.
- In Windows XP kernel, dispatcher handles the scheduling. A thread selected by the dispatcher will execute until following conditions occurs :
  - It is preempted by a higher-priority thread
  - It terminates
  - Its time quantum ends
  - It calls a blocking system call, such as for I/O.

A lower-priority running thread will be preempted if a higher-priority real-time thread becomes ready for execution. The dispatcher decides the order of thread execution on the basis of 32-level priority scheme. Priorities are separated into two classes. These are :

- **The variable class :** Includes threads having priorities from 1 to 15.
- **Real-time class :** Includes threads with priorities ranging from 16 to 31.

- A thread running at priority 0 that is used for memory management.

There is separate queue used by dispatcher for every scheduling priority. Dispatcher passes through the set of queues from highest priority queue to lowest priority queue. The traversal continues until it finds a thread that is ready to run. The dispatcher will execute a special thread called the **idle thread** if there is no ready queue.

There is a correlation among the numeric priorities of the Windows XP kernel and the Win32 API. The Win32 API classifies a number of priority classes to which a process can belong. These includes :

- REALTIME-PRIORITY_CLASS
- HIGH-PRIORITY_CLASS
- ABOVE_NORMAL.PRIORITY_CLASS
- NORMAL-PRIORITY_CLASS
- BELOW.NORMAL-PRIORITY_CLASS
- IDLE-PRIORITY_CLASS

REALTIME-PRIORITY_CLASS priorities are constant but, the priority of a thread belonging to any classes except REALTIME-PRIORITY_CLASS can change. Every priority class contains relative priority. Relative priority values are :

- TIME-CRITICAL
- HIGHEST
- ABOVE-NORMAL
- NORMAL
- BELOW-NORMAL
- LOWEST
- IDLE

The priority of each thread is based on the following :

- Its priority class.
- Its relative priority in that class.

In addition, each thread has a base priority which represents a value in th priority range for the class the thread belongs to. By default, the base priority i the value of the NORMAL relative priority for that specific class.

The base priorities for each priority class are :

- REALTIME-PRIORTTY_CLASS-24
- HIGH-PRIORITY_CLASS-13
- ABOVE-NORM AL.PRIORJTY_CLASS-10
- NORMAL-PRIORITY_CLASS-8
- BELOW.NORMAL-PRIORITY_CLASS-6
- IDLE-PRIORITY_CLASS-4

- All processes have usually membership of the NORMAL.PRIORITY_ CLASS. A process will be a member of this class if and only if the parent of the process was of the IDLE-PRIORITY_CLASS or unless another class was specified at the time process was created.
- The initial priority of a thread and base priority of the process the thread belongs to is equal. Thread gets interrupted after its time quantum finishes. At this moment, if the thread is in the variable-priority class, its priority is lowered.
- Threshold to lower the priority is base priority. If the priority is lowered and thread is CPU bound then CPU utilization of this thread decreases. As soon as a variable-priority thread becomes free from a wait operation, the dispatcher increases the priority.
- The level of increase in priority depends on what the thread was waiting for; for instance, a thread that was waiting for keyboard I/O would get a large increase, whereas a thread waiting for a disk operation would get a moderate one. This policy offer better response times to interactive threads that are using the mouse and windows.

2.8.3 Linux Scheduling :

All the versions of Linux bellow version 2.5 was supporting a variant of the conventional UNIX scheduling algorithm. With this traditional UNIX scheduler, two types of trouble was :

- It does not give sufficient support for SMP systems.
- If number of tasks on the system grows, scaling was not supported.

In version 2.5, Linux kernel have a scheduling algorithm that runs in constant time-known as O(1) despite the number of tasks on the system. The new scheduler also offers improved support for SMP, together with processor affinity and load balancing. It also provides fairness and support for interactive tasks.

The Linux scheduler is a preemptive, priority-based algorithm having two separate priority ranges :

- A real-time range from 0 to 99.
- A nice value ranging from 100 to 140.

Mapping of above two ranges are done in a global priority scheme where in lower integer values denote higher priorities.

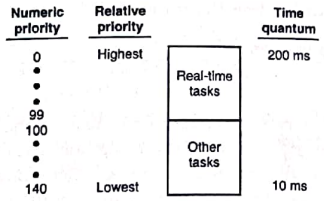


Fig. 2.7 : Relationship between priorities and time slice length

- To the higher-priority tasks longer time quantum is allocated and to the lower-priority tasks shorter time quantum is assigned. The relation between priorities and time-slice length is shown in Fig. 2.7
- If the task is runnable then it is eligible for execution on the CPU if and only if it has time left over in its time slice. If a task has used up its time slice, it is assumed expired and is not eligible for execution again until all other tasks have also used up their time quanta.
- The kernel keeps a record of all runnable tasks in a data structure. As SMP is supported by kernel, each processor preserves its own run queue and schedules itself independently. Every run queue holds two priority arrays: the active array contains all tasks with time remaining in their time slices, and the expired array contains all expired tasks. Each of these priority arrays contains a list of tasks indexed according to priority.
- The scheduler selects the task with the highest priority from the active array for execution on the CPU. In multiprocessor systems, all processors are scheduling the highest-priority task from its own run queue structure.

- When all tasks have worn out their time slices (when active array is empty), the two priority arrays are exchanged; the expired array becomes the active array, and vice versa.
- Static priorities are assigned to real-time tasks. Remaining other tasks has dynamic priorities that are based on their *nice* values plus or minus the value 5. The interactivity of a task decides whether the value 5 will be added to or subtracted from the *nice* value.

2.9 Synchronization :

Background :

- In a single-processor multiprogramming system, processes are not executed concurrently. In order to get the appearance of concurrent execution a fixed time slot is allocated to each process. After utilization of this time slot, CPU gets allocated to other process. Such switching of CPU back and forth between processes is called as **context switch**.
- At a time single process gets executed so parallel processing cannot be accomplished. Also there is a definite amount of overhead drawn in switching back and forth between processes. Apart from above limitations, interleaved execution offers major benefits in processing efficiency and in program structuring.
- In a multiple processor system, interleaving and overlapping the execution of multiple processes is achievable. Both interleaving and overlapping correspond to basically diverse modes of execution and present diverse problems. In reality, both interleaving and overlapping can be treated as illustration of concurrent processing and both the techniques address the similar problems.
- The comparative speed of execution of processes depends on activities and behavior of other processes, how the operating system handles interrupts, and the scheduling policies of the operating, system.

The difficulties that arise are :

1. Global resources sharing : For example, suppose two processes use the same global variable simultaneously and both carry out read and write operations on that variable, then various reads and writes execution ordering is serious.
2. Optimal management of the resources is not easy for the operating system.

3. Programming error tracing is not easy as results are typically nondeterministic and not reproducible.

In a single processor multiprogramming system a single user is supported. The user can switch from one application to another, and each application uses the same keyboard for input and the same screen for output because each application needs to use the procedure echo, it makes sense for it to be a shared procedure that is loaded into a portion of memory global to all applications. Thus, only a single copy of the echo procedure is used, saving space.

Consider the example :

```
void echo()
{
    chinput = getchar();
    choutput = chinput;
    putchar(choutput);
}
```

In order to have efficient and close interaction among processes, sharing of primary memory among processes is useful. Consider the following sequence :

1. Initially process P1 calls the echo procedure. The getchar() returns its value and stores it in input variable chinput. Process P1 gets interrupted straight away after returned value gets stored in chinput. At this instant, the most recently entered character, m, is stored in variable chinput.
2. After P1 gets interrupted, there is turn of process P2 and it call up the echo procedure. Process P2 inputs and displays the single character n on the monitor.
3. Now again, process P1 resumes its execution and the value m gets overwritten in the variable chinput and as a result it gets lost. The variable chinput holds value n, which is transferred to variable choutput and displayed.
4. In this way, the first character m is lost and the second character n is displayed two times. All this happens due to shared global variable, chinput. If after updating the shared global variable, one process is interrupted, another process may modify the variable before the previously interrupted

process can use its value. In above example both P1 and P2 are allowed to use shared global variable chinput.

However, if only one process at a time is allowed to be in procedure, above discussed sequence would result in the following :

(i) Initially process P1 calls the echo procedure. The getchar() returns its value and stores it in input variable chinput. Process P1 gets interrupted straight away after returned value gets stored in chinput. At this instant, the most recently entered character, m, is stored in variable chinput.

(ii) After P1 gets interrupted, there is turn of process P2 and it call up the echo procedure. At this situation, process P1 is in suspended state and it is still inside the echo procedure. The process P2 is not allowed to enter inside the echo() procedure. Therefore, P2 is suspended until P1 comes out of the echo procedure.

(iii) Later on, process P1 is resumes and finishes the execution of echo procedure. The output displayed is the character m.

(iv) When P1 comes out of echo() procedure, P2 becomes active. Now process P2 can successfully call the echo() procedure.

Therefore it is necessary to use shared global variable by only one process at a time.

2.10 Race Condition :

MU - Dec. 2014

A race condition takes place when more than one process write data items so that the final result depends on the order of execution of instructions in the multiple processes. Consider two processes, P1 and P2, which share the global variable "x". At the time of execution, process P1 updates "x" to the value 1 and at the time of execution of another process, P2 updates "x" to the value 2. Thus, the two tasks are in a race to write variable "x". In this example the process that modifies x value lastly decides the final value of "x".

Role of operating system :

The operating system should be capable of keeping track of the different processes.

- Allocation and reclamation of various resources should be done by operating system.

- Each process's data and physical resources should be protected by operating system against unintentional interfering by other processes.
- The operations performed by the process and the output that it generates should not depend on the speed of execution compare to the speed of other concurrent processes.

Process Interaction can be defined as :

- The processes not aware of each other
- The processes in a straight way or indirectly aware of each other

The conflict occurs between the concurrently executing processes, if these processes compete for the use of the same resource. More than one process may require the same resource while they are executing. The existence of the particular process is not known to other process. The competing processes do not pass the information to each other.

2.11 Critical Section Problem :

MU - Dec. 2014, May 2015

As discussed previously, it is necessary to find out some means to disallow more than one process from reading and writing the shared data at the same time. The portion of the program where the shared memory is accessed is referred as the *Critical Section*. In order to avoid race conditions and faulty results, one must be able to recognize the codes in *Critical Sections* in each thread. The typical properties of the code that comprises *Critical Section* are as follows :

- These codes reference one or more variables in a "read-update-write" way. At the same time, any of those variables maybe changed by another thread.
- These codes modify one or more variables that can be referenced in "read-update-write" fashion by another thread.
- Any part of data structure used by the codes can be modified by another thread.
- Codes modify any part of a data structure that is currently in use by another thread.

When one process is currently executing shared modifiable data in its critical section, no other process is to be permitted to execute in its critical section. Hence, the execution of critical sections by the processes is mutually exclusive in time.

Critical section is a code where only one process at a time can be executing. Critical section problem is design an algorithm that allows at most one process into the critical section at a time, without deadlock. Solution of the critical section problem must satisfy mutual exclusion, progress, bounded waiting.

Any process directly cannot enter in critical section. First process has to obtain permission for its entry in its critical section. The segment of code which implements this appeal of process is the entry section. When process comes out of the critical section after completing its execution there, it has to execute exit section. The rest of the code is the remainder section.

Any solution to the critical-section problem must satisfy the following three necessary conditions :

1. Mutual exclusion 2. Progress 3. Bounded waiting

1. **Mutual exclusion :** If one process is executing in its critical section, then other processes should not be executing in their critical sections.
2. **Progress :** If any process is not executing in its critical section and some processes desire to enter their critical sections, then only those processes that are not executing in their remainder sections should take part in the decision on which will enter its critical section next, and this selection cannot be delayed indefinitely.
3. **Bounded waiting :** There should be bound, on the number of times that other processes are permitted to enter their critical sections after a process has made a request to enter its critical section and before that request is approved.

Semaphore and **monitor** are the solutions to achieve mutual exclusion. Semaphore is a synchronization variable that tasks on positive integer values. Binary are those that only have two values 0 or 1.

Hardware does not provide the semaphore. The critical section problem can be solved by using semaphores. Like semaphore, a monitor also solves critical section problem. It is a software component which contains one or more procedures, an initialization sequence and local data.

Following are the components of monitors :

- Shared data declaration
- Shared data initialization
- Operations on shared data
- Synchronization statement

2.12 Mutual Exclusion :

MU - May 2015, Dec. 2015

Formally speaking, while one process executes the shared variable, all other processes desiring to do so at the same moment should be kept waiting; when that process has finished executing the shared variable, one of the processes waiting to do so should be allowed to proceed. In this fashion, each process executing the shared data (variables) excludes all others from doing so simultaneously. Only one process should be allowed to execute in critical section. This is called Mutual Exclusion.

The mutual exclusion is put into effect only if processes access shared changeable data. If during execution, the operations of the processes do not conflict with each other, they should be permitted to proceed in parallel.

Requirement of mutual exclusion :

- It is obligatory to enforce a mutual exclusion: If many processes want to execute in critical section then only one process should be allowed to execute in that critical section among all processes that have critical sections for the same resource or shared object.
- If the process halts in its remainder section (other than critical section) then it is allowed to do so without interfering with other processes.
- If process is waiting to enter in critical section then it should not be delayed for an indefinite period: that is it should not lead to deadlock or starvation.
- If the critical section is empty (any process not executing in critical section), then if any process that requests entry to its critical section must be allowed to enter without delay.
- Assumptions about relative process speeds or number of processors are not made.
- Any process will not remain inside its critical section for indefinite time. It should stay there for a finite time only.

Mutual exclusion conditions :

If we could arrange matters such that no two processes were ever in their critical sections simultaneously, we could avoid race conditions. We need four conditions to hold to have a good solution for the critical section problem (mutual exclusion).

- No two processes may at the same moment inside their critical sections.
- No assumptions are made about relative speeds of processes or number of CPUs.
- No process should outside, its critical section should block other processes.
- No process should wait arbitrary long to enter its critical section.

2.13 Peterson's Solution :

MU - May 2015, Dec. 2015

- Peterson's solution involves only two processes. It gives a good means of algorithmic explanation of work out the critical-section problem and exemplifies some of the difficulties concerned in designing software that addresses the requirements of mutual exclusion, progress, and bounded waiting.
- The two processes *Pi and Pj* perform alternate execution between their critical sections and remainder sections (remaining code of the process). Peterson's solution requires the two processes to share two data items :

```
int x;
boolean y[2];
```

- The variable x decides who will enter in its critical section. if x==i, then process *Pi* is permissible to execute in its critical section. The y array is used to specify whether process *is ready* to enter its critical section. For example, if y[i] is true, this value indicates that *Pi* is ready to enter its critical section.
- In order to enter the critical section, process *Pi* first sets y[i] equal to true and then sets variable x to the value j. *Pi* sets x==j because, if the other (*Pj*) process desires to enter the critical section, it can do so. If both processes attempt to enter simultaneously, x will be set to both i and j at approximately the same time. Only one of these assignments will last; the other will occur but will be overwritten straight away.


```
do {
    y [i] = TRUE;
    x= j;
    while (y[j] && x== j);

    critical section

    y[i] = FALSE;

    remainder section
} while (TRUE);
```

The final value of x decides which of the two processes is permitted to enter its critical section first. We can prove that solution is correct by showing :

- Mutual exclusion condition is satisfied.
- The progress requirement is satisfied.
- The bounded-waiting requirement also achieved.

Mutual exclusion :

- Every *Pi* go into its critical section if and only if either y[j] == false or x==i. Also note that, if both *Pi* and *Pj* runs in critical sections simultaneously, then y[0] == y [1] ==true. It implies that *Pi* and *Pj* could not have successfully executed their while statements at about the same time, since the value of variable x can be either i or j but cannot be both.
- Hence, one of the processes say, *Pi* must have successfully executed the while statement, whereas *Pj;* had *to* execute at least one additional statement ("x== j").
- However, at that time, y [j] == true and x == j, and this condition will continue as long as *Pi* is in its critical section. Therefore we can conclude that mutual exclusion is preserved.

Progress and bounded waiting :

- Process *Pi* can be prohibited from entering the critical section only if it gets stuck in the while loop with the condition y [j] ==true and x== j; this loop is the only one possible.
- If *Pj* is not ready to go into the critical section, then y [j] ==false, and *Pi* can go into its critical section. If *Pj* has set y[j] to true and is also running in its

while statement, then either variable x == i or x == j. Depending on the value of x either *Pi* or *Pj* will enter in the critical section.

- If x == i, then *Pi* will enter the critical section. If x== j, then *Pj* will enter the critical section. However, once *Pj* exits its critical section, it will reset y[j] to false, allowing *Pi* to enter its critical section.
- If *Pj* resets y [j] to true, it must also set x to i. Thus, since *Pi* does not change the value of the variable x while executing the while statement, *Pi* will enter the critical section (progress) after at most one entry by *Pj* (bounded waiting).

2.14 Synchronization Hardware :

MU - May 2015

- The simple hardware instructions that are available on many systems can be used successfully in solving the critical section problem. Hardware features can make any programming task easier and get better system efficiency.
- If we restrict the interrupts from taking place while a shared variable was being modified, in uniprocessor environment the critical-section problem could be solved.
- Many systems have special instructions called Test and Set Lock called as TSL instruction. It is having format: "TSL ACC, X". In this instruction ACC is accumulator register and X is symbolic name of memory location. X holds character to be used as flag.
- TSL is indivisible instruction means that it cannot be interrupted in between. After instruction gets executed, following action take place.
  - Content of X is copied to ACC
  - Content of X becomes N

During execution of above two steps, TSL instruction is not interrupted. If we assume value of X=N initially. The meaning of N is critical region is not free and F means it is free. Consider the following steps to enter in critical region.

1. TSL ACC, X (Content of X is copied to ACC and Content of X becomes N)
2. CMP ACC, "F" (See if critical region is free)
3. BU 1 (If critical region not free then loop back.)
4. Return (Return to caller and enter)

Following are the steps for exiting from critical region :

1. MOV X, "F" (F gets copied to X)
2. Return (Return to caller)

Consider the two processes *Pi* and *Pj*.

- Assume initially scheduler schedules *Pi* and X = "F". So step 1 makes ACC= F and X= N. Here Pi after executing step 4, prepare to enter CR (critical region).
- *Pj* gets scheduled due to context switch before *Pi* enters the CR. *Pj* executes step 1 and X = N. In step 2, *Pj* fails the comparison and loop backs. It cannot execute step 3. So *Pj* cannot enter in CR as *Pi* is already in its CR.
- Consider *Pi* is again scheduled and executing in its CR. Even though context switch occurs and *Pj* gets scheduled at this moment it cannot enter in CR as ACC and X values are N (busy waiting).
- After completion of execution in CR, *Pi* executes step 5 to exit the CR ans X = "F".
- If *Pj* scheduled after this, it execute step 1 and ACC becomes F as X was F in above step. Now due to step 1 by *Pj* ACC = "F" and X= "F".
- Step 2 is executed by *Pj* and succeeds and it enters the CR.

This solution satisfies all conditions mutual exclusion, progress and bounded waiting. Since special hardware is required cannot be generalized to all machines. Also due to busy waiting, it is not efficient solution.

2.15 Semaphores :

MU - May 2015, Dec. 2015

E.W. Dijkstra (1965) abstracted the key notion of mutual exclusion in his concepts of semaphores.

The solutions of the critical section problem represented in the section are not easy to generalize to more complex problems. To avoid this complicatedness, we can use a synchronization tool called as **semaphore**. A semaphore S is an integer variable that, apart from initialization, is accessed only through two standard atomic operations : wait and signal. These operations were firstly termed P (for wait) and V (for signal).

Definition :

- A semaphore S is integer variable whose value can be accessed and changed only by two operations wait (P or sleep or down) and signal (V or wakeup or up). Wait and signal are atomic operations.
- Binary semaphores do not assume all the integer values. It assumes only two values 0 and 1. On the contrary, counting semaphores (general semaphores) can assume only nonnegative values.
- The wait operation on semaphores S, written as wait(S) or P(S), operates as follows :

```
wait(S): IF  S > 0
         THEN S: = S - 1
         ELSE (wait on S)
```

The signal operation on semaphore S, written as signal(S) or V(S), operates as follows :

```
signal(S): IF (one or more process are waiting on S)
           THEN (let one of these processes proceed)
           ELSE S: = S +1
```

- The two operations wait and signal are done as single indivisible atomic operation. It means, once a semaphore operation has initiated, no other process can access the semaphore until operation has finished. Mutual exclusion on the semaphore S is enforced within **wait(S)** and **signal(S)**.
- If many processes attempt a wait(S) at the same time, only one process will be permitted to proceed. The other processes will be kept waiting in queue. The implementation of wait and signal promises that processes will not undergo indefinite delay. Semaphores solve the lost-wakeup problem.

Disadvantages of semaphore :

1. They are essentially shared global variables.
2. Access to semaphores can come from anywhere in a program.
3. There is no control or guarantee of proper usage.
4. There is no linguistic connection between the semaphore and the data to which the semaphore controls access.
5. They serve two purposes, mutual exclusion and scheduling constraints.

2.16 Producer Consumer Problem :

- In producer-consumer problem assumes that buffer is bounded buffer. This means that there is a finite numbers of slots are available in a buffer. While producing the items, if the buffer is full the producer process should be suspended. In the same way, while consuming the items from buffer, if it becomes empty, consumer should be suspended. It is also necessary to ensure that only one process at a time manipulates a buffer to avoid race conditions or lost updates.
- Sleep-wake up system calls is used in this situation to avoid race condition. Producer process should go to sleep when buffer is full and consumer process should wake up when producer will put data in buffer. In the same way, consumer goes to sleep until the producer puts some data in buffer and wakes up to consume this data.
- Here two processes, producer and consumer share a common, fixed-size (bounded) buffer. The producer puts data into the buffer and the consumer takes data out.
- Difficulty arises when the producer wants to put a new data in the buffer, but there is no space in buffer. That is, it is already full. Way out to this problem is that, producer goes to sleep and to be awakened when the consumer has removed data.
- It may also happen that, consumer wants to take out data from the buffer but buffer is already empty. Way out to this problem is that, consumer goes to sleep until the producer puts some data in buffer and wakes consumer up.

The code for the producer process can be modified as follows :

```
while (true)
{
/* produce an item in nextProduced variable */
while (counter == ARRAY_SIZE)
; /* do nothing */
array[in] = nextProduced;
in = (in + 1) % ARRAY-SIZE;
counter++;
```

The code for the consumer process can be modified as follows :

```
while (true)
{
while (counter == 0)
/* do nothing */
nextConsumed = array [out];
out = (out + 1) % ARRAY_SIZE;
counter--;
/* consume the item in nextConsumed variable*/
```

- Producer and consumer functions are accurate if considered separately, they may not work correctly when executed concurrently. This approach also leads to same race conditions. we have seen in earlier approaches.
- Race condition can occur due to the fact that access to 'counter' is unconstrained. The essence of the problem is that a wakeup call, sent to a process that is not sleeping, is lost.

Producer consumer problem using semaphore :

- In previous section we have seen to producer-consumer problem. The solution to producer-consumer problem uses three semaphores namely, full, empty and mutex.
- The semaphore 'full' is used for counting the number of slots in the buffer that are full. The 'empty' for counting the number of slots that are empty and semaphore 'mutex' to make sure that the producer and consumer do not access modifiable shared section of the buffer simultaneously.

Initialization :

- Set full buffer slots to 0.

  i.e., semaphore full = 0.
- Set empty buffer slots to N.

  i.e., semaphore empty = N.
- For control access to critical section set mutex to 1

  i.e., semaphore mutex = 1.


```
Producer ( )
while (true)
    produce-Item ( );
    P (empty);
    P (mutex);
    enter-Item ( );
    V (mutex);
    V (full);
Consumer ( )
while (true)
    P (full)
    P (mutex);
    remove-Item ( );
    V (mutex);
    V (empty);
    consume-Item (Item)
```

2.17 Reader/Writers Problem :

MU - May 2015, Dec. 2015

- While defining the reader/writers problem, it is assumed that, many processes only read the file (readers) and many write to the file (writers). File is shared among a many number of processes. The conditions that must be satisfied are as follows :
  - Simultaneously reading of the file is allowed to many readers.
  - Writing to the file is allowed to only one writer at the same time.
  - Readers are not allowed to read the file while writer are writing to the file.
- In this solution, the first reader accesses the file by performing a down operation on the semaphore file. Other readers only increment a counter, read count. When readers finish the reading, counter is decremented.
- When last one ends by performing an up on the semaphore, permitting a blocked writer, if there is one, to write. Suppose that while a reader is reading file, another reader comes along. Since having two readers at the same time is not a trouble, the second and later readers can also be allowed if they come down.
- After this assumes that a writer wants to perform write on the file. The writer cannot be allowed to write the file, since writers must have restricted access, so the writer is suspended. The writer will suspended until no reader is reading the file. If a new reader arrives continuously with very short

interval and perform reading, the writer will never obtain the access of the file.

- To stop this circumstance, the program is written in a different way: when a reader comes and at the same time a writer is waiting, the reader is suspended instead of being allowed reading immediately.
- Now writer will wait for readers that were reading and about to finish but does not have to wait for readers that came along after it. The drawback of this solution is that it achieves less concurrency and thus lower performance. Following is the solution given.

```
typedef int semaphore;
semaphore mutex = 1;/* controls access to 'readcount' */
semaphore file = 1;/* controls access to file */
int readcount = 0;
void reader(void)
{
    while (TRUE) { /* repeat evermore */
    down(&mutex); /* get exclusive access to 'readcountc' */
    readcountc = readcountc + 1;/* one reader more now */
    if (readcount == 1) down(&file); /* if this is the first reader... */
    up(&mutex);/* release exclusive access to 'readcount' */
    read_file() /*read the file */
    down(&mutex);/* get exclusive access to 'readcount' */
    readcount = readcount - 1; /* one reader fewer now */
    if (readcount == 0) up(&db);/* if this is the last reader... */
    up(&mutex);/* release exclusive access to 'readcount' */
    use_data_read();/* noncritical region */
    }
}
void writer(void)
{
    while (TRUE) {/* repeat evermore */
    think_up_data(); /* noncritical region */
    down(&file);/* get exclusive access */
    write_file(); /* write the file */
    up(&file); /* release exclusive access */
    }
}
```


2.18 Dinning Philosopher Problem :

MU - Dec. 2014, Dec. 2015

- The problem is stated as : The work of five philosophers is thinking and eating. A table is laid for them and all agreed that only food that contributed to their thinking efforts was spaghetti. Each philosopher requires two forks to eat spaghetti.
- The provision for eating is on a round table with a big serving pot of spaghetti, five plates, and five forks. One plate is given to one philosopher. A philosopher sits at assigned place at the table. He has to use two forks on either side of the plate. He takes and eats some spaghetti.
- Algorithm should be designed which will allow the philosophers to eat. The algorithm must satisfy mutual exclusion (two philosophers should not use the same fork at the same time) while avoiding deadlock and starvation. This problem exemplifies basic problems in deadlock and starvation.
- The dining philosopher's problem can be seen as example of problems related to coordination of shared resources, which may occur when an application includes concurrent threads of execution. In the same way, this problem is a typical test case for assessing approaches to synchronization.

Solution using semaphores :

Following is the solution using semaphores. Each philosopher picks up first the fork on the left side and then the fork on the right side. After the philosopher is finished eating, the two forks are replaced on the table.

- This solution, unfortunately, leads to deadlock: If all of the philosophers are hungry simultaneously, all will pick up the fork on their left, and right side fork will not be there. In this unseemly position, all philosophers starve.
- Additional five forks (a more clean solution!) will solve the problem or use of just one fork to eat the spaghetti can be a solution. Following program shows the solution.

```
/* program diningphilosophers */
semaphore fork [5] = {1};
int i;
void philosopher (int i)
{
    while (true) {
    think();
    wait (fork[i]);
    wait (fork [(i+1) mod 5]);
```

```
        eat();
        signal(fork [(i+1) mod 5]);
        signal(fork[i]);}
}
void main()
{
        parbegin (philosopher (0), philosopher (1),
        philosopher (2), philosopher (3),
        philosopher (4));
}
```

- In another strategy consider the helper who only allows four philosophers at a time into the dining room. With at most four seated philosophers, at least one philosopher will have access to two forks. Following is the solution using semaphore.

```
/* program diningphilosophers */
semaphore fork[5] = {1};
semaphore room = {4};
int i;
void philosopher (int i)
{
        while (true) {
        think();
        wait (room);
        wait (fork[i]);
        wait (fork [(i+1) mod 5]);
        eat();
        signal (fork [(i+1) mod 5]);
        signal (fork[i]);
        signal (room); }
}
void main()
{
        parbegin (philosopher (0), philosopher (1),
        philosopher (2), philosopher (3),
        philosopher (4));
}
```