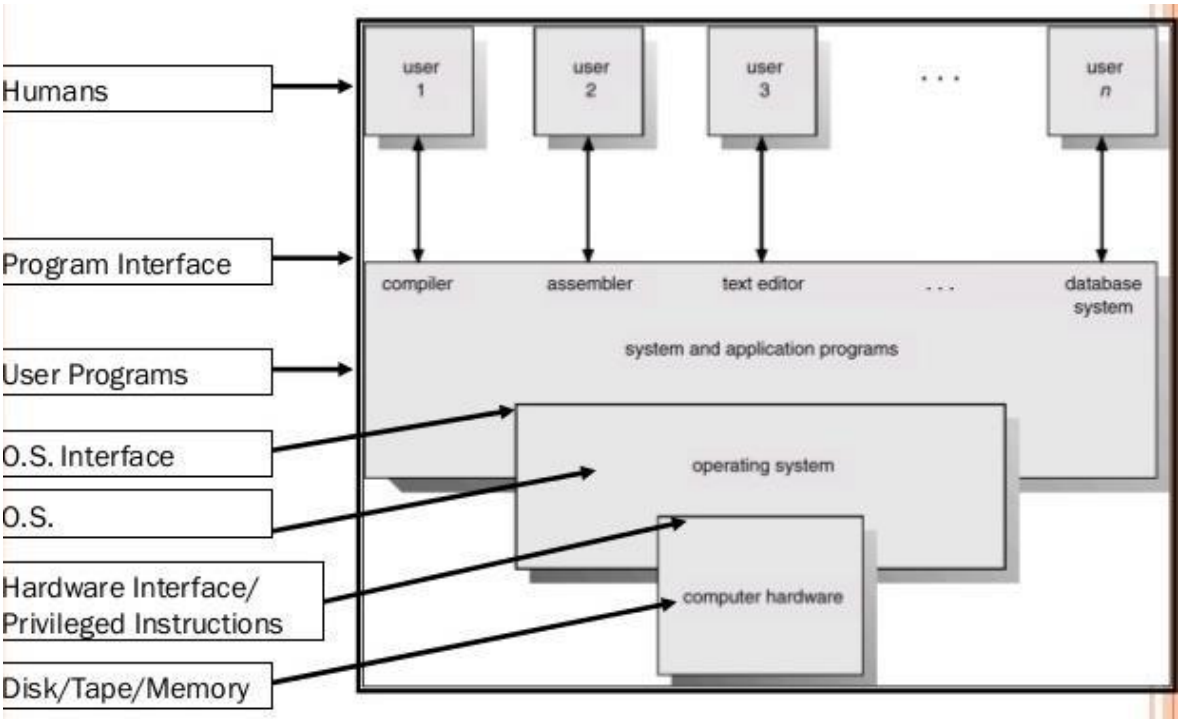
UNIT-I:Introduction: Operating-System Structure, Operating-System Services, User and Operating-System Interface, System Calls, Types of System Calls.

COMPUTER SYSTEM AND OPERATING SYSTEM OVERVIEW

Operating System is a set of programs that acts as an interface between the user of computer and the hardware of computer. An OS makes a computer more convenient to use. An OS allows the computer system resources to be used in efficient manner. An OS should be constructed in such a way as to permit the effective development, testing and introduction of new system function without interfering new services.



1. OS FUNCTIONS

1. **User interface:** Users interact with application programs and computer hardware through a user interface. Almost all operating systems today provide a windows-like Graphical
   Command line interface (CLI), which uses text command. E.g., DOS, UNIX.
   Command: copy file1 file2 (DOS)
   cp file1 file2 (UNIX)
   Batch Interface (BI), in which commands and directives to control those commands are entered into files, and those files are executed.
   Graphical User Interface (GUI), in which graphic objects called icons are used to represent commonly used features. E.g, Windows-95,Windows-98, Windows-xp, GNOME and KDE on LINUX.
2. **Program development**
   OS provides services such as editors and debuggers to assist the programmer in creating programs. These services are in the form of utility programs that are supplied with the OS (not part of OS) and are referred to as application program development tools.

3. **Program execution**

   Executable program (instructions and data) must be loaded into main memory and allocates all the resources required for the program to execute. The OS handles these scheduling duties for the user.

4. **Access to Input/output devices**
   Each I/O device has own set of instructions for operation. The OS provides uniform interface to all devices so that programmer can access a device as a file.
5. **Controlled access to files**
   OS must understand not only nature of I/O device (disk drive, tape drive) but also the file structure(sequential file, index sequential file, random files). In case of system with multiple users OS may provide protection mechanism to control access to the files.

6. **Error detection and response**
   The errors can be internal or external hardware errors, such as a memory error, or device failure; and various software errors such as divide by zero. OS must provide a response that clears the error condition with the least impact on running applications. The response may range from ending the program that caused the error, to simply report the error to the applicant.
7. **Accounting**
   We want to keep track of which users use how much and what kinds of computer resources. This record keeping may be used for accounting.
8. **Interpreting the commands**
   In UNIX shell is called as an interpreter. Shell takes command from the user interprets it, expands it and finally it will see the command is executed but it will not execute command. UNIX provides three type of shells and they are korn shell, born shell and c shell.
9. **Memory management**
   Allocating memory to the process
   De-allocating the memory
   Protection (One process should not enter into the other process area)
10. **Process management**
    Creation of process
    Termination of process
    Process switching over

11. **Inter Process communication**
    In multi task operating systems more than one process can be loaded into memory. Operating system facilitates to communicate among the process on a system.

Types of Operating System

- DOS (Disk Operating System)
- UNIX
- LINUX
- Windows
- Windows NT

EVOLUTION OF OPERATING SYSTEM

Serial Processing:

The programmer interacted directly with the computer hardware; there was no OS. These computers were run from a console consisting display lights, toggle switches, some form of input device and printer. Programs in machine code were loaded via input device (e.g card reader), if error halted the program, the error condition was indicated by lights. If the program proceeded to normal completion, the output appeared on printer.

Main Problems:

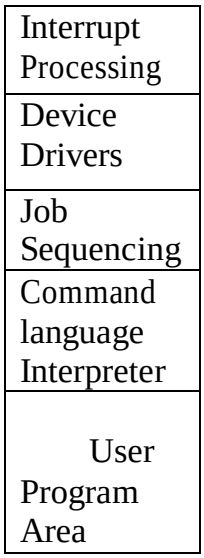
A user might sign up for an hour and finish in 45 minutes; this would result in wasted computer processing time. On the other hand, the user might run into problems, not finish in allotted time.

Setup time:

A single program, called a job, could involve loading the compiler plus source program into memory, saving the compiled program (object code) and then loading and linking together the object program and common functions. Each of these steps could involve mounting and dismounting tapes or setting up card decks. Thus, a considerable amount of time was spent just in setting the program to run.

Simple Batch System:

Earlier computers were very expensive, so the wasted time due to scheduling and setup time in serial processing was unacceptable. In simple batch, user need not have direct interaction with the processor. A piece software known as MONITOR will be used as operating system. Each program is constructed to branch back to monitor when it completes processing, at which point the monitor automatically begins loading next program.



Memory Layout for Resident Monitor

The monitor performs a scheduling function: A batch of jobs is queued up, and jobs are executed as rapidly as possible, without intervening time. With each job, instructions are included in a primitive form of **Job Control Language (JCL).**

```
$JOB
$FTN
  •
  •          Fortran Instructions
  •
  •
$LOAD
$RUN
  •
  •          Data
  •
$END
```

$JOB indicates the beginning of the job. The monitor reads $FTN line and loads the appropriate language compiler from its mass storage (usually tape). The compiler converts source code into object code and stores it on tape. When monitor reads $LOAD, it loads the object code into memory (in the place of compiler) and transfers control to it.

Advantage:

1) Reduced manual intervention compared to serial processing.

Problems:

1) The speed of I/O device is very slow compared to speed of CPU and whenever job goes for I/O the CPU will be idle. CPU utilization is very less.
2) Only one JOB (program) will be loaded into memory.

Multi Programmed Batch System:

The speed of I/O device is very slow compared to speed of CPU. In simple batch systems whenever job goes for I/O the CPU will be idle since only one job is loaded into memory.

In ***multi programmed batch systems*** more than one JOB is loaded into memory. When one JOB needs to wait for I/O, the processor switches to other JOB, which is not waiting for I/O.
In multiprogramming, more than one program lies in the memory. For example, we open word, excel, access and other applications together but while we type in word other applications such as excel and access are just present in main memory but they are not performing any task or work.

In multiprogramming with two programs, If JOB A goes for I/O, CPU will automatically goes for JOB B. If we observe the above figure, CPU utilization in multiprogramming (fig: 2b) with two processes has increased compared to uniprogramming (Fig: 2a).

In multiprogramming with three programs, If JOB A goes for I/O, CPU will automatically goes for JOB B and JOB B goes for I/O CPU runs JOB C. The CPU utilization in multiprogramming with three programs has increased compared to multiprogramming with two programs. So, CPU utilization and resource utilization increases with the number of programs in main memory.

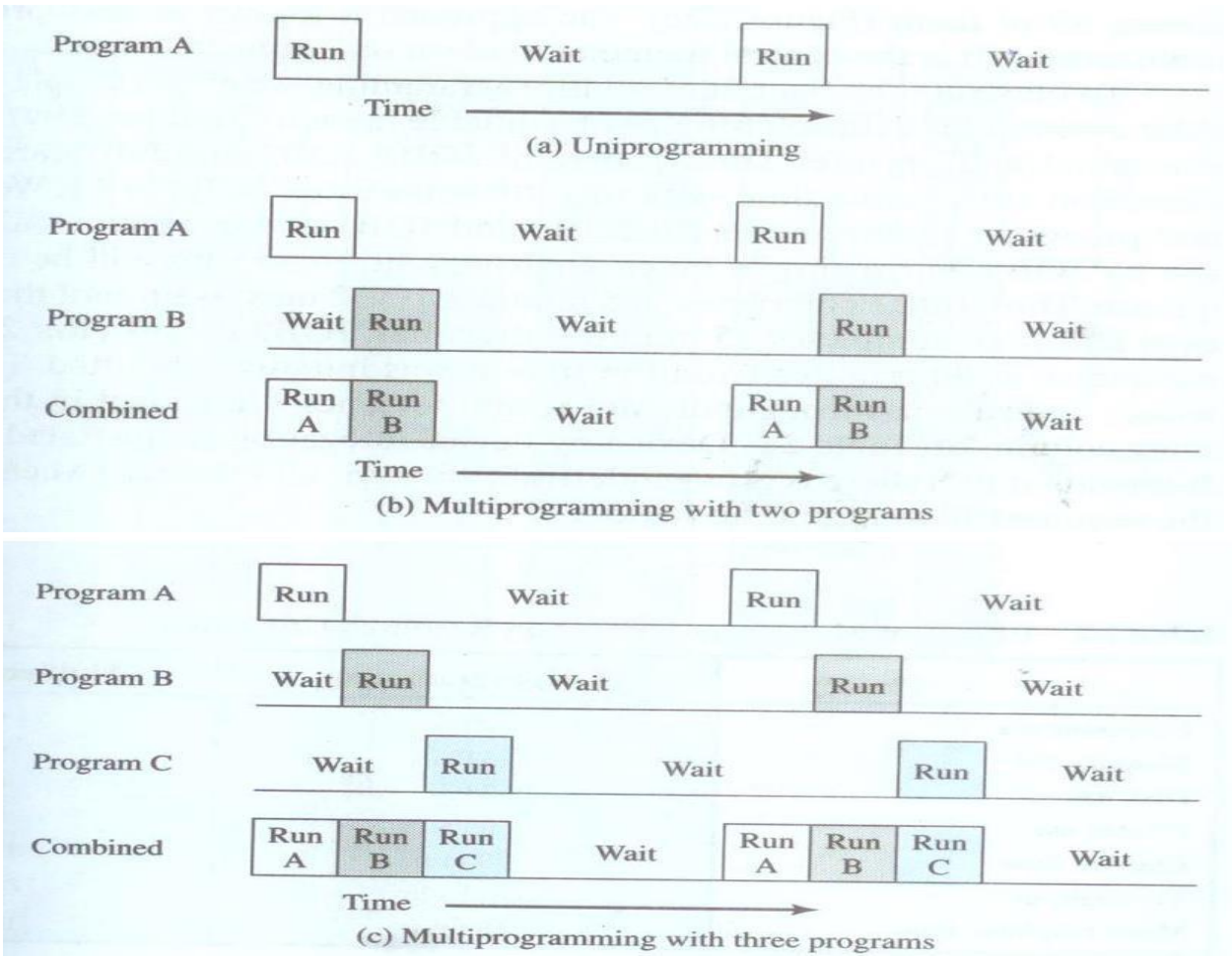


Fig 2: Multiprogramming Example

Advantages:

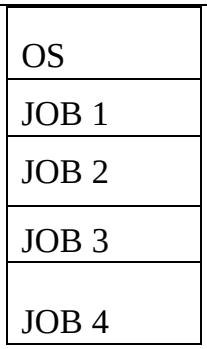
1) CPU utilization and resource utilization increases with number of programs in memory.

Dis advantage:

1) The response time of a job depends upon the previous jobs submitted. So, we can't predict response time of a JOB.
2) It can't be used for multiple interactive JOBs.

Time-Sharing Systems:

User can't predict the response time of a job in multiprogramming batch system. In a time-sharing system OS ***interleaving*** execution of each user program in a short quantum time. In time-sharing multiple users simultaneously access system through terminals. The response time of a program depends upon the number of programs submitted prior it.



Memory

Assume all jobs are submitted at the same time and time slice for each job is 4msec. The response time of a JOB 4 will be utmost 12 sec (3*4). If any prior job of JOB 4 goes for I/O, the response time of JOB 4 will reduced.

As soon as job assigned to the processor CPU starts timer. Once the timer expires system clock generates interrupt. At each clock interrupt, the OS regains control and could assign processor to another JOB. This technique is known as time slicing.

Table: Batch multiprogramming versus Time Sharing.

	Batch Multi programming	Time sharing
Objective	Maximize processor use	Minimize response time
Source of directives to operating system	Job control language	Command entered at the terminal
Interactive jobs	Not suitable	Suitable
Online Transaction Processing	Not Suitable	Suitable

Multi-Tasking: Multitasking means performing multiple tasks in parallel. Usually, CPU processes only one task at a time but the switching of CPU between the processes is so fast that it looks like CPU is executing multiple processes at a time. Example of multitasking, we listen to music and do internet browsing at the same time (they execute parallel).

Clustered Operating system:

A cluster is a set of computers harnessed together to present a single server resource to applications and to users. Cluster architectures offer significant benefits, including higher performance, higher availability, and greater scalability and lower operating costs. Cluster architectures are in place today for applications that must provide continuous, uninterrupted service.

With redundancy designed in to all subsystems — processing, storage, network, cooling, power — cluster architectures can offer various levels of **reliability**.

In addition, cluster architectures can be scaled smoothly to provide for increasing system performance. System architects can aggregate machines of different capacities in clusters of different sizes to target small, mid-size and very large system needs. Costs are reduced because clustering exploits low-cost, high-performance hardware.

Moreover, a cluster of UNIX systems provides the same standard UNIX system environment that is used for existing applications.

2. PROTECTION AND SECURITY

If a computer system has multiple users and allows the concurrent execution of multiple processes, then access to data must be regulated. For that purpose, mechanisms ensure that files, memory segments, CPU, and other resources can be operated on by only those processes that have gained proper authorization from the operating system.

Protection is a mechanism for controlling the access of processes or users to the resources defined by a computer system. This mechanism must provide means for specification of the controls to be imposed and means for enforcement. Protection can improve reliability by detecting latent errors at the interfaces between component subsystems. A protection-oriented system provides a means to distinguish between authorized and unauthorized usage. Consider a user whose authentication information is stolen. Data could be copied or deleted, even though file and memory protection are working. It is the job of **security** to defend a system from external and internal attacks. Such attacks spread across a huge range and include viruses and worms, denial-of service, identity theft, and theft of service. Protection and security require the system to be able to distinguish among all its users. Most operating systems maintain a list of user names and associated **user identifiers (user IDs).**

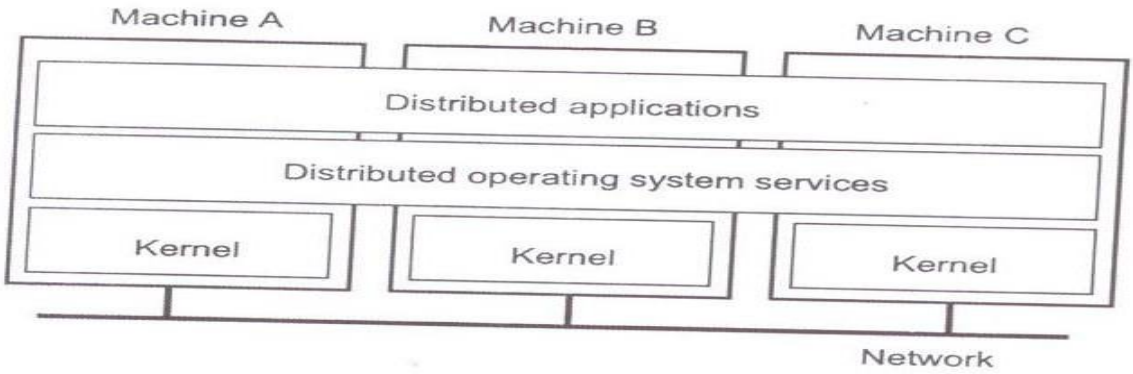
3. DISTRIBUTED SYSTEMS

A distributed system is a collection of physically separate, possibly heterogeneous computer systems that are networked to provide the users with access to the various resources that the system maintains. Access to a shared resource increases computation speed, functionality, data availability, and reliability.
A distributed system is a collection of independent computers that are connected through the network (LAN,MAN,WAN) and distributed operating system gives an impression to the user as if they are working on single computer.

Basics of Distributed Systems:

- Networked computers (loosely coupled) that provide a degree of operation transparency

- Distributed Computer System = independent processors + networking infrastructure
- Communication between processes (on the same or different computer) using message passing technologies is the basis of distributed computing



Goals of Distributed Systems:

- **Resource sharing**: easy for users to access remote resources.
- **Transparency**: to hide the fact that processes and resources are physically distributed across multiple computers.
- **Openness**: to offer services according to standard rules.
- **Scalability**: easy to expand and manage
- **Reliability** – ability to continue operating correctly for a given time
- **Fault tolerance** – resilience to partial system failure

Distributed computing examples

- rlogin or telnet (for remote access)
- network file system, network printer etc
- ATM (cash machine)
- Distributed databases
- Network computing
- Global positioning systems
- Retail point-of-sale terminals
- Air-traffic control
- WWW

Centralised Systems	**Distributed System**
System shared by users all the time	Multiple autonomous components shared by user
All resources accessible	Some resources may not be accessible
Software runs in a single process	Software can run in concurrent processes on different processors
Single physical location	Multiple physical locations

Single point of control	Multiple points of control
Single point of failure	Multiple points of failure
Global Time	No global time
Shared Memory	No shared memory

Client Server model:

The client takes the request from the user and sends the request to the server, the server will execute the request and send the reply back to the client. It is also called two- tire architecture. All the resources are under the control of server only. The client-server model firmly distinguishes the roles of the client and server.

For example, UNIX is a client- server architecture.

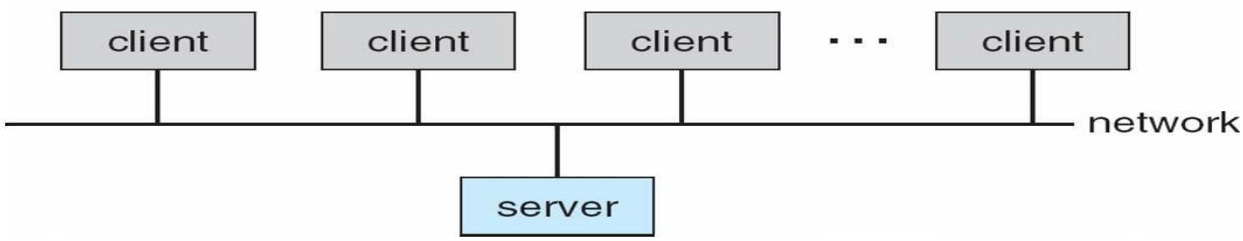


Fig : Client Server model

Peer-to-peer model:

It is another model of distributed system. A network of personal computers, all nodes in the system are considered peers and thus may act as either clients or servers - or both. A node may request a service from another peer, or the node may in fact provide such a service to other peers in the system. Peer-to-peer networks are less expensive than client/server networks but less efficient when large amounts of data need to be exchanged.

Node must join P2P network Registers its service with central lookup service on network, or Broadcast request for service and respond to requests for service via discovery protocol.

4. SPECIAL PURPOSE SYSTEMS

1) Real time embedded systems

2) Multimedia systems

3) Handheld systems.

Real Time Embedded Systems:

A real-time system is any information processing system which has to respond to externally generated input within a finite and specified period. Whenever an event is occurred action must be taken within a specified time.

- **Hard real-time systems**

1) A deadline is a given time after a triggering event, by which a response has to be completed. An overrun in response time leads to potential loss of life and/or big financial damage
2) Many of these systems are considered to be safety critical.
3) Sometimes they are "only" missioning critical, with the mission being very expensive.
E.g: Temperature control in nuclear reactor: Whenever temperature rises to certain threshold level, steam valve should be opened and fuel should be cut down. This should be done in specified time otherwise a mishap will occur.

- **Soft real-time systems**

1) Deadline overruns are tolerable, but not desired. There are no catastrophic consequences of missing one or more deadlines.
2) There is a cost associated to overrunning, but this cost may be abstract.
3) Often connected to Quality-of-Service (QoS)
E.g, ATM or Lift

- **Firm teal-time systems**

1) The computation is obsolete if the job is not finished on time.
2) Cost may be interpreted as loss of revenue.
3) Typical example is forecast systems.

- **Weakly hard real-time**

1) Systems where H out of K deadlines has to be met.
2) In most cases feedback control systems, in which the control becomes unstable with too many missed control cycles.
3) Best suited if system has to deal with other failures as well (e.g. Electro Magnetic Interference EMI).
4) Likely probabilistic guarantees sufficient.

Multimedia Systems:

Most operating systems are designed to handle conventional data such as text files, programs, word processing documents, and spread sheets. Multimedia data consist of audio and video files as well as conventional files. These data differ from conventional data in that multimedia data- such as frames of video – must be delivered (streamed) according to certain time restrictions(for example 30 frames per second). The multimedia applications are audio files such as MP3, DVD movies, video conferencing, and short video clips of movie previews. Multimedia applications may also include live web casts of speeches or sporting events. Multimedia applications often include a combination of both audio and video.

Handheld Systems:

Handheld systems include personal digital assistants (PDAs), such as Palm and packet PCs, and cellular telephones, many of which use special-purpose embedded operating systems. A handheld computing device has an operating system (OS), and can run various types of application software, known as apps. Most handheld devices can also be equipped with Wi-Fi, Bluetooth, and GPS capabilities that can allow connections to the

Internet and other Bluetooth-capable devices, such as an automobile or a microphone headset. A camera or media player feature for video or music files can also be typically found on these devices along with a stable battery power source such as a lithium battery. Some handheld devices use wireless technology, such as Bluetooth, remote access to e-mail and web browsing.

Limitations:

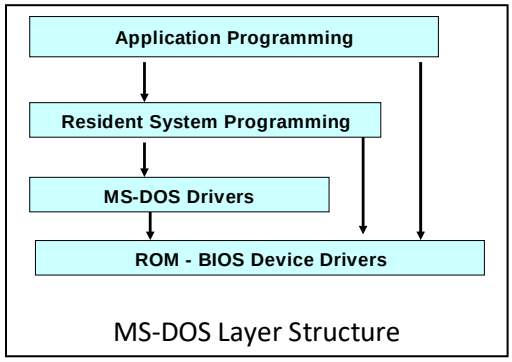
Handheld systems will have very less memory compared to desktops. As a result, the operating system and applications must manage memory efficiently. Most handheld devices use smaller, slower processors that consume less power to avoid frequent recharging of batteries. Very small size keyboards and monitors are user due to lack of space.

5. OPERATING SYSTEM STRUCTURE

Modern operating systems are large and complex. They must be engineered properly if it is to function properly and be modified easily. Common approach is to partition the task into small components rather than have one monolithic system.

Simple Structure:

Many commercial systems do not have well-defined structure. MS-DOS was originally designed and implemented by few people who had no idea that it would become so popular. It was written to provide the most functionality in the least space. So it was not divided into modules carefully.



MS-DOS Layer Structure

In MS-DOS, the interface and level of functionality are not well separated. For instance, application programs able to access the basic I/O routines to write directly to the display and disk drives. Such freedom levels MS-DOS vulnerable to errant programs, causing entire system to crashes.

UNIX consists of two separate parts: the kernel and system programs. Everything below the system call interface and above the physical hardware is kernel. The kernel provides memory management, file management, CPU scheduling and other operating system functions through system calls. The monolithic structure (kernel) was difficult to implement and maintain.

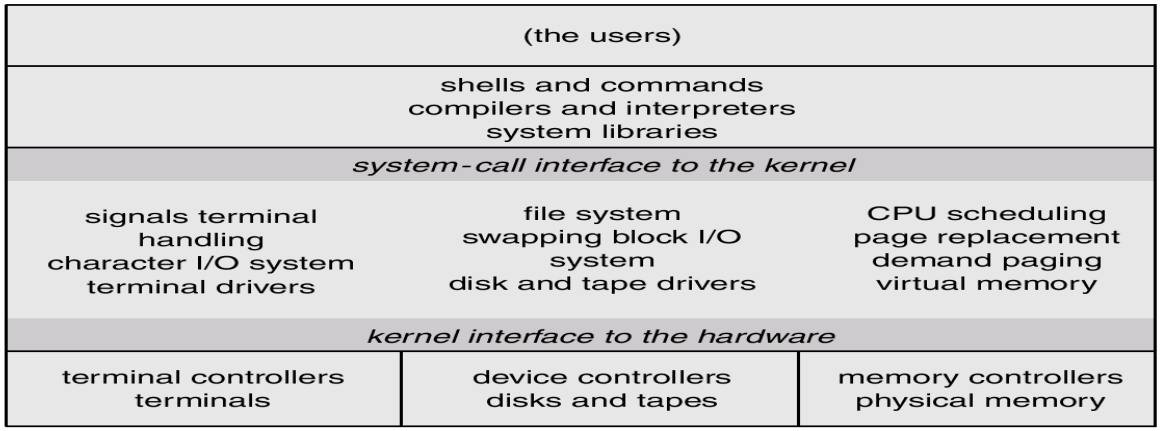


Fig : Unix system structure

Layered Approach:

Operating systems can be broken into pieces that are smaller and more appropriate than those allowed by the original MS-DOS system. The OS can then retain much greater control over the computer and over the application that make use of that computer. Implementers have more freedom in changing the inner workings of the system and in creating modular operating systems.

The operating system is divided into a number of layers (levels), each built on top of lower layers. The bottom layer (layer 0), is the hardware; the highest (layer N) is the user interface.

With modularity, layers are selected such that each uses functions (operations) and services of only lower-level layers. For example, Layer M consist of data structures and set of routines can be invoked by the layers above M, in turn, layer M can invoke services of lower layers.

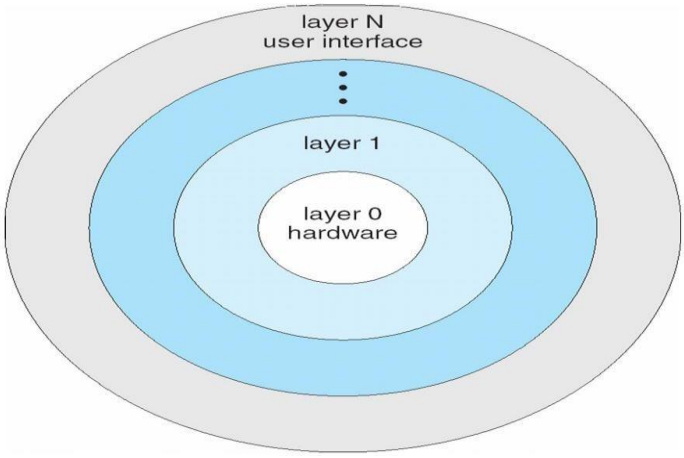


Fig: A layered operating system

Advantages of layered approach:

1) Simplicity of construction

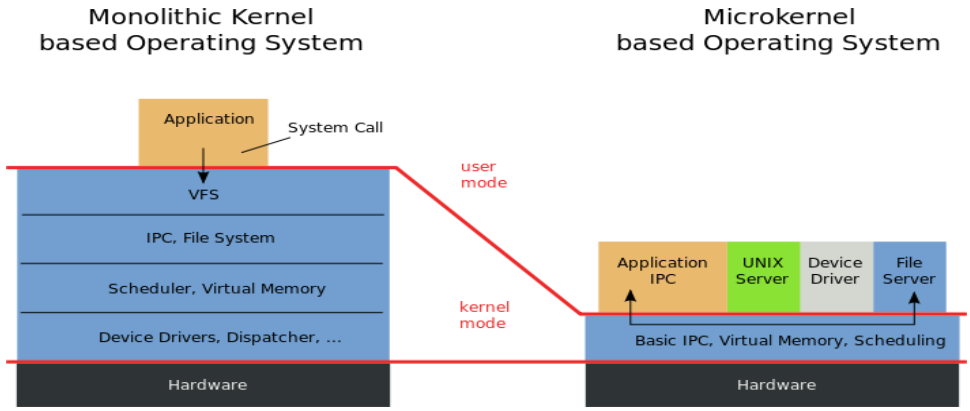
2) Easy in debugging: The first layer can be debugged without concern for the rest of the system. If an error is found in during the debugging of a particular layer, the error must be on that layer, because the layer below it is already debugged. Thus, debugging and implementation of system is simplified.
3) Each layer is implemented with only those operations provided by lower level layers. A layer does not need to know how these operations are implemented; it needs to know only what these operations do. Hence, each layer hides the existence of certain data structures, operations and hardware from higher layers.

Disadvantages:

1) Appropriately defining various layers.
2) The order of the layers.(Device layer should be below the memory management layer).
3) It is less efficient than other approaches. When a user program executes an I/O operation, it executes an I/O operation, it executes a system call that trapped to I/O layer, which calls memory management layer which in turn calls CPU-scheduling layer, which is the passed to the hardware.

Each layer adds overhead to system call; the net result is a system call takes longer than non layered system.

Microkernel's Structure:



A **microkernel** is the near-minimum amount of software that can provide the mechanisms needed to implement an operating system (OS). These mechanisms include low-level address space management, thread management, and inter-process communication (IPC). Traditional operating system functions, such as device drivers, protocol stacks and file systems, are removed from the microkernel to run in user space.

1) Moves as much from the kernel into "user" space .
2) Communication takes place between user modules using message passing .

Benefits:

Easier to extend a microkernel

Easier to port the operating system to new architectures
More reliable (less code is running in kernel mode)
Maintenance is generally easier. Patches can be tested in a separate instance, then swapped in to take over a production instance.
Rapid development time, new software can be tested without having to reboot the kernel.
More secure
Disadvantages: Performance overhead of user space to kernel space communication.
The first release of windows-NT 4.0 micro kernel version gave less performance than windows 95 monolithic version. Windows-xp architecture is more monolithic than microkernel.

Modules:

The kernel has a set of core components and dynamically links in additional services either during booting time or during run time.
Most modern operating systems implement kernel modules:

- Uses object-oriented approach
- Each core component is separate
- Each talks to the others over known interfaces
- Each is loadable as needed within the kernel
- Overall, similar to layers but with more flexible

For example, the Solaris operating system structure is organized around core kernel with seven types of loadable kernel modules as shown in the above figure. Such a design allows the kernel to provide core services yet also allows certain features to be implemented dynamically. For example, device and bus drivers for specific hardware can be added to kernel, and support for different file systems can be added as loadable modules.

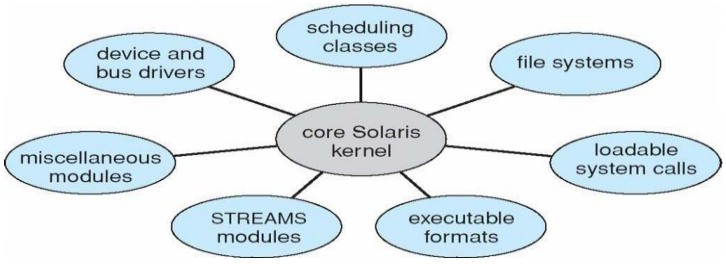


Fig: Solaris Loadable Module

This approach is like the microkernel approach in that the primary module has only core functions and knowledge of how to load and communicate with other modules; but it is more efficient, because modules do not need to invoke message passing in order to communicate.

Under what circumstances would a user be better of using a time-sharing system rather than a PC or single-user workstation?

When there are few other users, the task is large, and the hardware is fast, timesharing make sense. The full power of the system can be brought to bear on the pser's problem. The problem can be solved faster than on a personal computer. Another case occurs when lots of other users need resources at the same time.

For example, an organization generates 1000 transactions per day. A user can enter 100 transactions per day. Then we recommend for time sharing system.

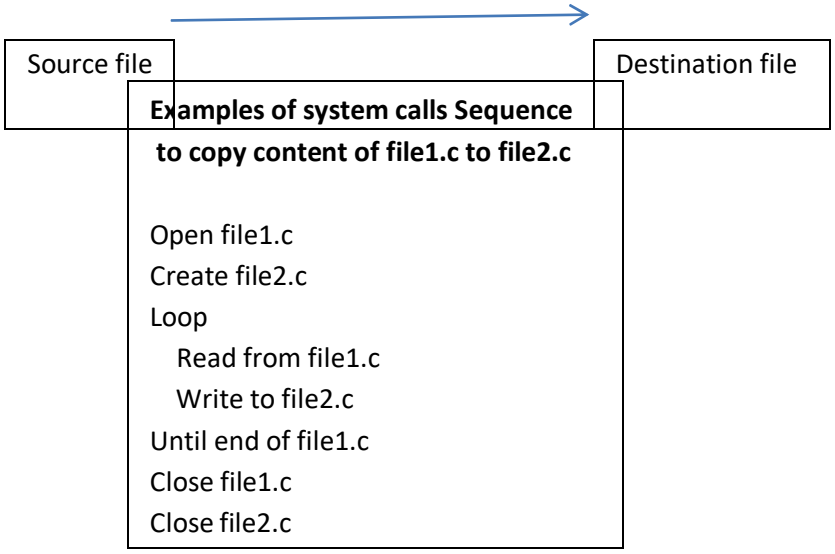
A personal computer is best when the job is small enough to be executed reasonably on it and when performance is sufficient to execute the program to the user's satisfaction.

6. SYSTEM CALLS

The interface between operating system and user programs is defined by the set of system calls that operating system provides. The system calls available in the interface vary from operating system to operating system. The system call is called by programmer whenever services are required from OS.

The process starts running in user mode. As soon as system call is called it changes from user mode to kernel mode. The mode bit specifies the process running in user mode(mode bit=1) or kernel(mode bit=0) mode.

For e.g, copy file1.c file2.c



Example how system calls are used in copy contents from file1.c to file2.c

Steps in calling system call and getting result:

1) User program is running in user mode
2) Save user program
3) Change from user mode to kernel mode
4) Load and Run the open() system call
5) Open() system call send return value to user program
6) Restore User program and start running from next statement of open() system call.

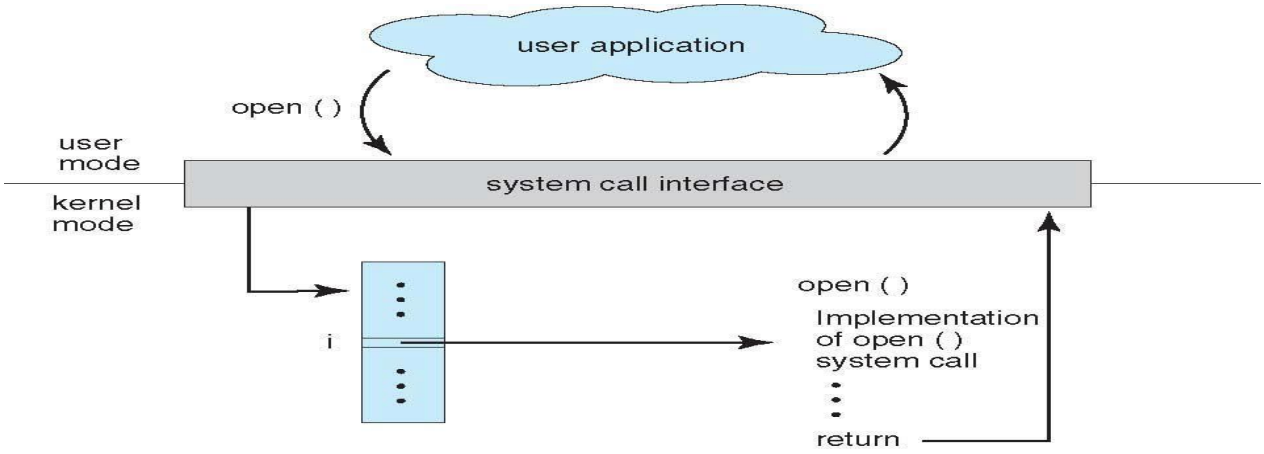
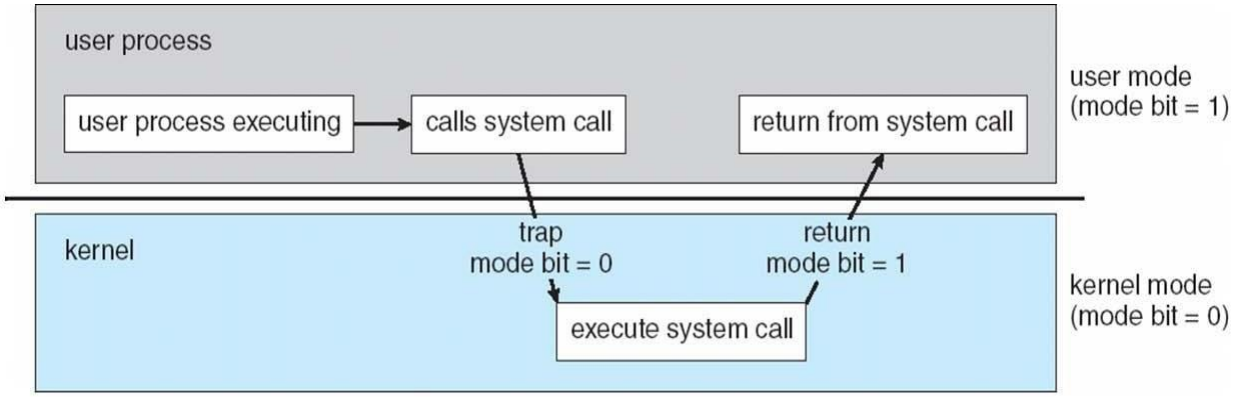


Fig : Handling of user application invoking open() system call.



Types of system Calls:

System calls can be categorized into five major categories.

Process control

- Create process(fork), terminate process(exit)
- Get process attributes(getpid), set process attributes
- Wait for event(wait)
- Wait event, signal event
- Allocate and free memory (malloc,free)

	Windows	Unix
Process Control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
File Manipulation	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
Device Manipulation	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
Information Maintenance	GetCurrentProcessID() SetTimer() Sleep()	getpid() alarm() sleep()
Communication	CreatePipe() CreateFileMapping() MapViewOfFile()	pipe() shmget() mmap()
Protection	SetFileSecurity() InitlializeSecurityDescriptor() SetSecurityDescriptorGroup()	chmod() umask() chown()

Table: These are examples of various system calls.

File Management

- Create file, delete file(create)
- Open, close (open, close)
- Read, write, reposition (read, write, seek)
- Get file attributes, set file attributes

Device management

- Request device request device, release device
- Read, write, reposition
- Get device attributes, set device attributes
- Logically attach or detach devices

Information maintenance

- Get time or date(date), set time or date
- Get system date, set system date
- Get process, file, or device attributes
- set process, file, or device attributes

Communication

- create, delete communication connection
- send, receive message
- attach or detach remote devices

Parameter passing in system calls:

Application developers design programs according to an Application Program Interface (API). The API specifies set of functions that are available to an application programmer, including the parameters that are passed to each function and their return values. The parameters are passed to system call through register, stack and memory. The most famous APIs are win32 API on windows, POSIX API on Linux, UNIX and mac OS-X.

Three general methods used to pass parameters to the OS Simplest:

1) Pass the parameters in *registers.*

2) In some cases, may be more parameters than registers Parameters stored in a *block*, or table, in memory, and address of block passed as a parameter in a register. This approach taken by Linux and Solaris.

3) Parameters placed, or *pushed*, onto the *stack* by the program and *popped* off the stack by the operating system Block and stack methods do not limit the number or length of parameters being passed

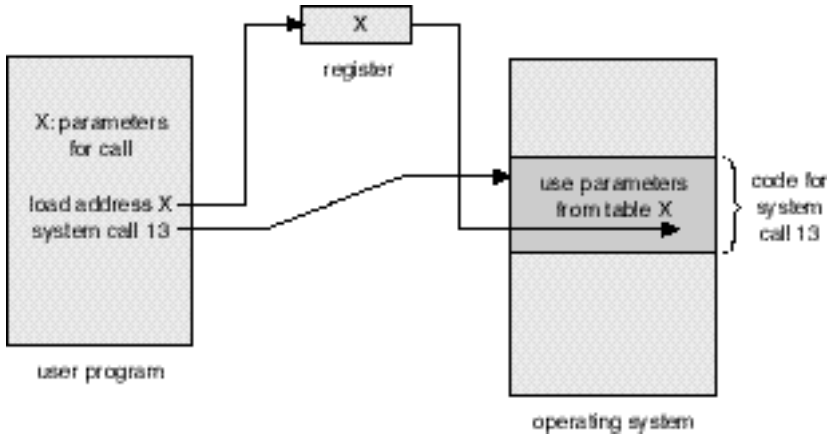


Fig: Parameter passing through registers in system calls

What is the advantage of using APIs rather than system calls?

Each operating system has its own name for each system call. The portability of application increases with the use of API rather than system call.

7. OPERATING-SYSTEM GENERATION

Operating systems are designed to run on any of a class of machines at a variety of sites with a variety of peripheral configurations. The system must then be configured or generated for each specific computer site, a process sometimes known as **system generation (SYSGEN).**

The operating system is normally distributed on disk or CD-ROM. The SYSGEN program reads from a given file, or asks the operator of the system for information concerning the specific configuration of the hardware system, or probes the hardware directly to determine what components are there. The following kinds of information must be determined.

• What CPU is to be used? What options (extended instruction sets, floating point arithmetic, and so on) are installed? For multiple CPU systems, each CPU must be described.

• How much memory is available? Some systems will determine this value themselves by referencing memory location after memory location until an "illegal address" fault is generated. This procedure defines the final legal address and hence the amount of available memory.

• What devices are available? The system will need to know how to address each device (the device number), the device interrupt number, the device's type and model, and any special device characteristics.

• What operating-system options are desired, or what parameter values are to be used? These options or values might include how many buffers of which sizes should be used, what type of CPU-scheduling algorithm is desired, what the maximum number of processes to be supported is, and so on.

UNIT-II:Process Management: Process Concept, Process Scheduling, Operations on Processes, Interprocess Communication. Threads: Overview, Multithreading Models. CPU Scheduling: Basic Concepts, Scheduling Criteria, Scheduling Algorithms

1. THE PROCESS CONCEPT

A. The Process:

A process is a program in execution. A process is more than the program code, which is sometimes known as the **text section.** It also includes the current activity, as represented by the value of the **program counter** and the contents of the processor's registers. A process generally also includes the process **stack,** which contains temporary data (such as function parameters, return addresses, and local variables), and a **data section,** which contains global variables. A process may also include a **heap,** which is memory that is dynamically allocated during process run time. The structure of a process in memory is shown in Figure.

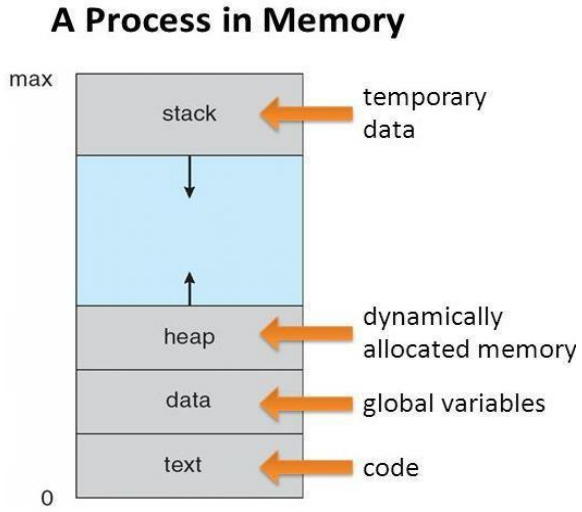


Figure: A process in memory

B. Process State:

As a process executes, it changes **state.** The state of a process is defined in part by the current activity of that process. Each process may be in one of the following states:

- **New:** The process is being created.
- **Running:** Instructions are being executed.
- **Waiting:** The process is waiting for some event to occur.
- **Ready:** The process is waiting to be assigned to a processor.
- **Terminated:** The process has finished execution.

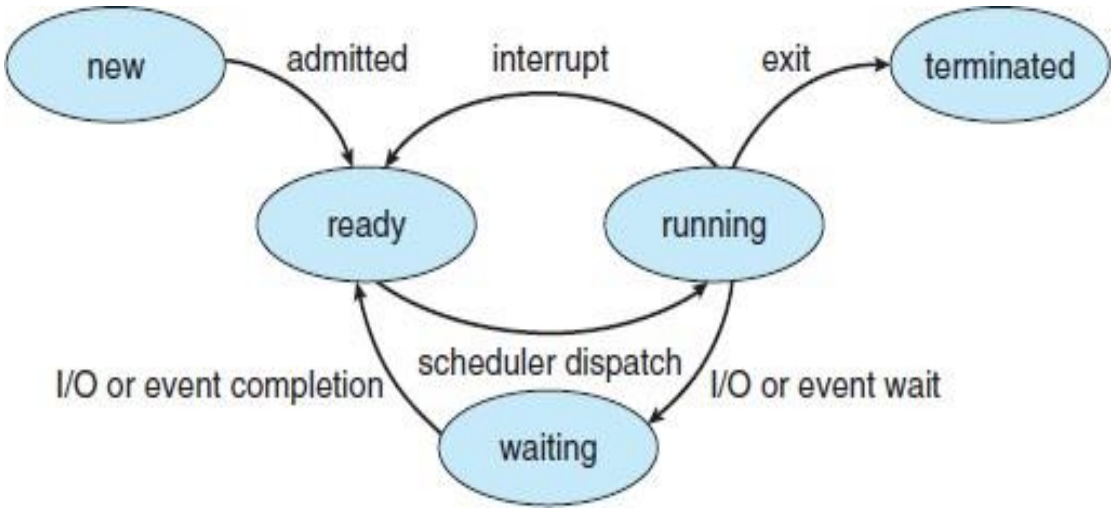


Figure: Process States

C. Process Control Block:

Each process is represented in the operating system by a **process control block (PCB)**—also called a *task control block.* A PCB is shown in Figure.

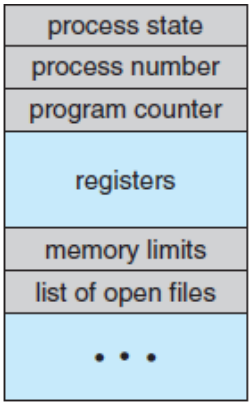


Figure Process control block (PCB).

It contains many pieces of information associated with a specific process, including these:

Process state: The state may be new, ready, running, waiting, halting, and so on.

Program counter: The counter indicates the address of the next instruction to be executed for this process.

CPU registers: The registers vary in number and type, depending on the computer architecture. They include accumulators, index registers, stack pointers, and general-purpose registers, plus any condition-code information. Along with the program counter, this state information must be saved when an interrupt occurs, to allow the process to be continued correctly afterward

CPU-scheduling information: This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters.

Memory-management information: This information may include such information as the value of the base and limit registers, the page tables, or the segment tables, depending on the memory system used by the operating system.

Accounting information: This information includes the amount of CPU and real time used, time limits, account numbers, job or process numbers, and so on.

I/O status information: This information includes the list of I/O devices allocated to the process, a list of open files, and so on.

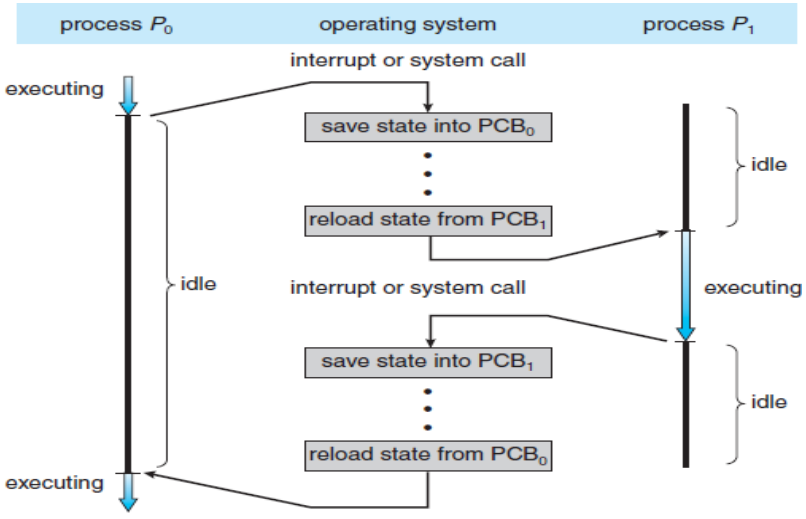


Figure Diagram showing CPU switch from process to process.

2. PROCESS SCHEDULING

A. Scheduling Queues:

A new process is initially put in the ready queue. It waits there until it is selected for execution, or is **dispatched.** Once the process is allocated the CPU and is executing, one of several events could occur:

- The process could issue an I/O request and then be placed in an I/O queue.
- The process could create a new sub process and wait for the sub process's termination.
- The process could be removed forcibly from the CPU, as a result of an interrupt, and be put back in the ready queue.

the first two cases, the process eventually switches from the waiting state to the ready state and is then put back in the ready queue. A process continues this cycle until it terminates, at which time it is removed from all queues and has its PCB and resources de allocated.

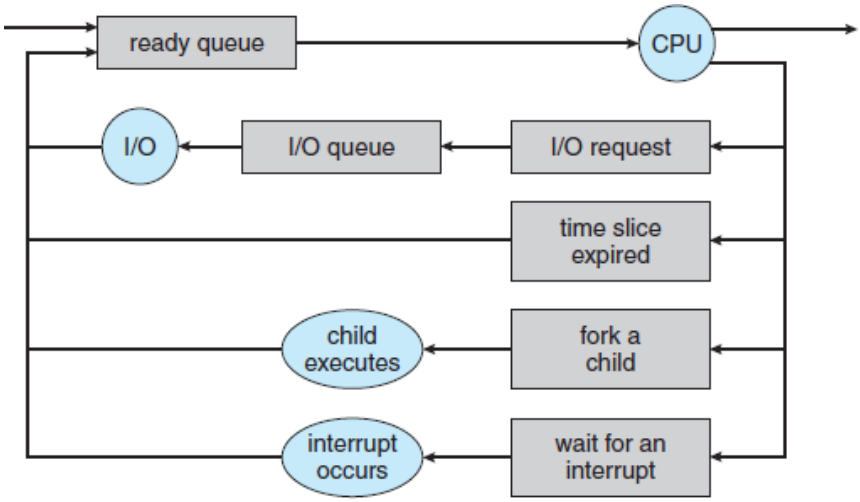


Figure: Queing-diagram representation of process scheduling.

B. Schedulers:

Schedulers are special system software's which handles process scheduling in various ways. Their main task is to select the jobs to be submitted into the system and to decide which process to run. Schedulers are of three types

- Long Term Scheduler
- Short Term Scheduler
- Medium Term Scheduler

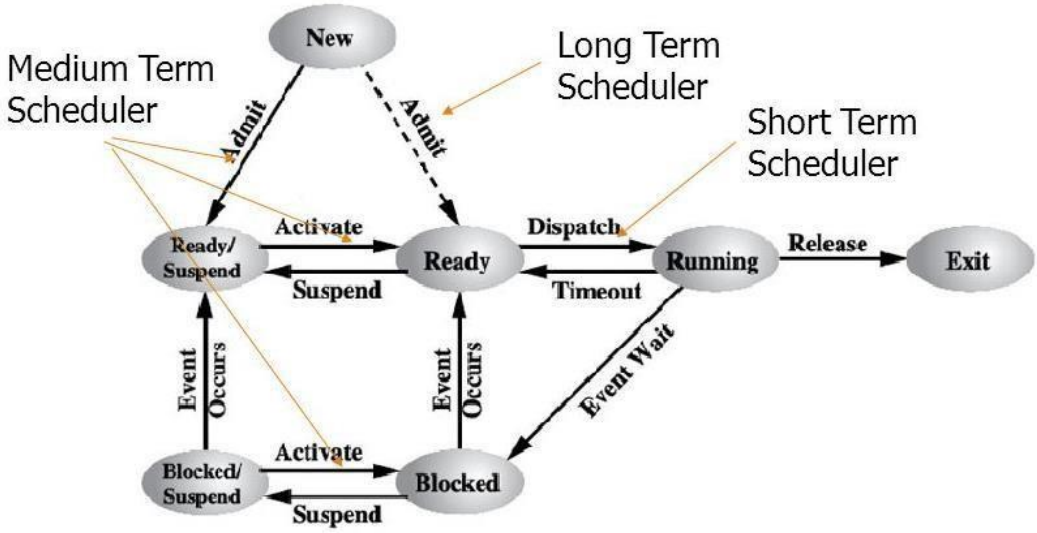


Figure: Process state transition diagram with schedulers.

i. ***Long Term Scheduler:*** It is also called job scheduler. Long term scheduler determines which programs are admitted to the system for processing. Job scheduler selects processes from the queue and loads them into memory for execution. Process loads into the memory for CPU scheduling. The primary objective of the job scheduler is to provide a balanced mix of jobs, such as I/O bound and processor bound. It also controls the degree of multiprogramming. If the degree of multiprogramming is stable, then the average rate of process creation must be equal to the average departure rate of processes leaving the system On some systems, the long term scheduler may not be available or minimal. Time-sharing operating systems have no long term scheduler. When process changes the state from new to ready, then there is use of long term scheduler.

ii. ***Short Term Scheduler:*** It is also called CPU scheduler. Main objective is increasing system performance in accordance with the chosen set of criteria. It is the change of ready state to running state of the process. CPU scheduler selects process among the processes that are ready to execute and allocates C

iii. PU to one of them.

Short term scheduler also known as dispatcher, execute most frequently and makes the fine grained decision of which process to execute next. Short term scheduler is faster than long term scheduler.

iv. ***Medium Term Scheduler:*** Medium term scheduling is part of the swapping. It removes the processes from the memory. It reduces the degree of multiprogramming. The medium term scheduler is in-charge of handling the swapped out-processes.

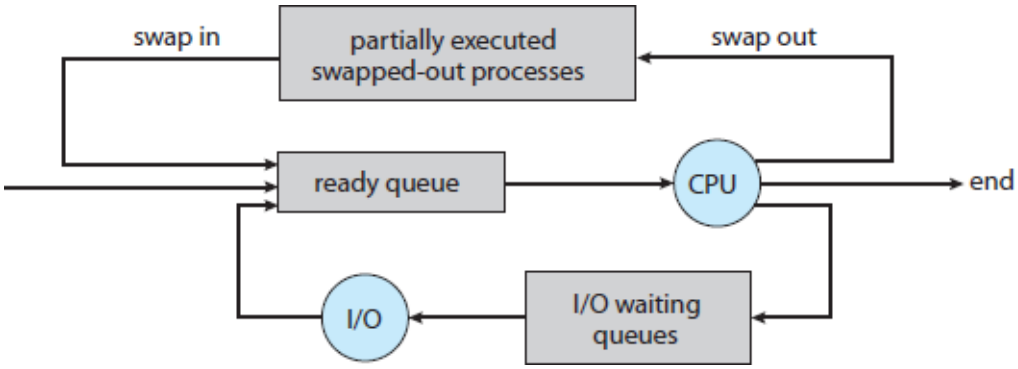


Figure: Medium term sceduler in queing diagram.

Running process may become suspended if it makes an I/O request. Suspended processes cannot make any progress towards completion. In this condition, to remove the process from memory and make space for other process, the suspended process is moved to the secondary storage. This process is called swapping, and the process is said to be swapped out or rolled out. Swapping may be necessary to improve the process mix.

Comparison between Scheduler

S.N.	Long Term Scheduler	Short Term Scheduler	Medium Term Scheduler
1	It is a job scheduler	It is a CPU scheduler	It is a process swapping scheduler.
2	Speed is lesser than short term scheduler	Speed is fastest among other two	Speed is in between both short and long term scheduler.
3	It controls the degree of multiprogramming	It provides lesser control over degree of multiprogramming	It reduces the degree of multiprogramming.
4	It is almost absent or minimal in time sharing system	It is also minimal in time sharing system	It is a part of Time sharing systems.
5	It selects processes from pool and loads them into memory for execution	It selects those processes which are ready to execute	It can re-introduce the process into memory and execution can be continued.

C. Context Switch:

Switching the CPU to another process requires performing a state save of the current process and a state restore of a different process. This task is known as a **context switch**.

The context is represented in the PCB of the process. It includes the value of the CPU registers, the process state and memory-management information. Generically, we perform a state save of the current state of the CPU, be it in kernel or user mode, and then a state restore to resume operations.

Context-switch time is pure overhead, because the system does no useful work while switching. Switching speed varies from machine to machine, depending on the memory speed, the number of registers that must be copied, and the existence of special instructions (such as a single instruction to load or store all registers).

Context-switch times are highly dependent on hardware support.

Dispatcher: Another component involved in the CPU-Scheduling function is Dispatcher. The dispatcher is the module that gives control of the CPU to the process selected by short-term scheduler. This function involves the following:

1. Switching context
2. Switching to user mode
3. Jumping to the proper location in the user program to restart that program.

Dispatch latency – time it takes for the dispatcher to stop one process and start another running.

3. OPERATIONS ON PROCESSES

A. Process Creation:

A process may create several new processes, via a create-process system call, during the course of execution. The creating process is called a parent process, and the new processes are called the children of that process. Each of these new processes may in turn create other processes, forming a tree of processes.

Most operating systems identify processes according to a unique process identifier (or Pid), which is typically an integer number.

Figure illustrates a typical process tree for the Linux operating system, showing the name of each process and its pid. (We use the term process rather loosely, as Linux prefers the term task instead.) The init process (which always has a pid of 1) serves as the root parent process for all user processes. Once the system has booted, the init process can also create various user processes, such as a web or print server, an ssh server, and the like. In Figure 3.8, we see two children of init—kthreadd and sshd. The kthreadd process is responsible for creating additional processes that perform tasks on behalf of the kernel (in this situation, khelper and pdflush). The sshd process is responsible for managing clients that connect to the system by using ssh (which is short for secure shell). The login process is responsible for managing clients that directly log onto the system. In this example, a client has logged on and is using the bash shell, which has been assigned pid 8416. Using the bash command

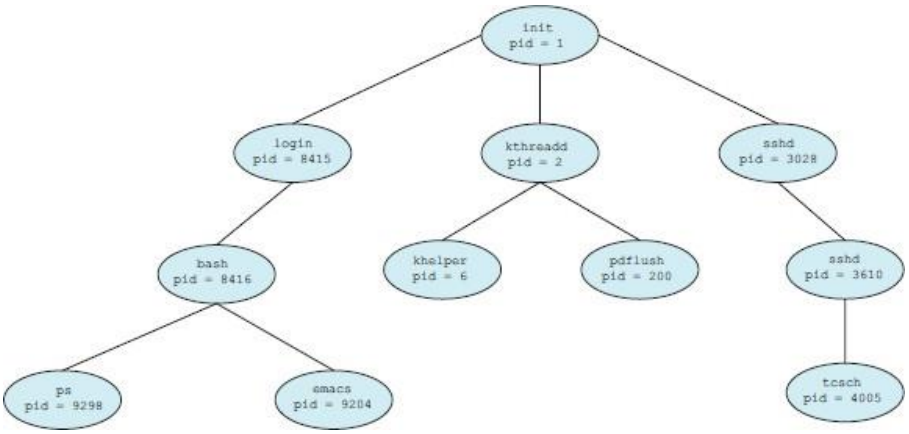


Figure: A Tree of processes on a typical Linux system

When a process creates a new process, two possibilities exist in terms of execution:

1. The parent continues to execute concurrently with its children.
2. The parent waits until some or all of its children have terminated.

There are also two possibilities in terms of the address space of the new process:

1. The child process is a duplicate of the parent process (it has the same program and data as the parent).
2. The child process has a new program loaded into it.

To illustrate these differences, let's first consider the UNIX operating system. In UNIX, as we've seen, each process is identified by its process identifier, which is a unique integer. A new process is created by the fork ( ) system call. The new process consists of a copy of the address space of the original process. This mechanism allows the parent process to communicate easily with its child process. Both processes (the parent and the child) continue execution at the instruction after the fork(), with one difference: The return code for the for() is zero for the new (child) process, whereas the (nonzero) process identifier of the child is returned to the parent.

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
int main()
{
pid-t pid;
/* fork a child process */
pid = fork();
if (pid < 0) {/* error occurred */
fprintf(stderr, "Fork Failed");
exit (-1) ;
}
else if (pid == 0} {/* child process */
execlpf"/bin/Is","Is",NULL);
}
else {/* parent process */
/* parent will wait for the child to complete */
wait(NULL);
printf("Child Complete");
exit (0) ;
}}
```

Program: Creating a separate process using the UNIX fork() system call.

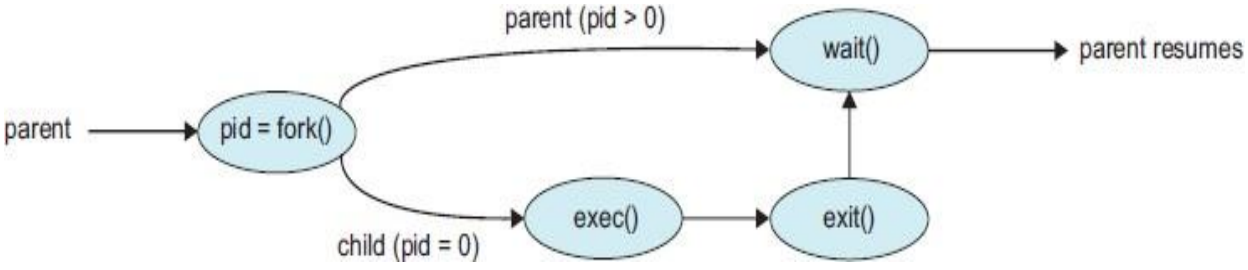


Figure: Process creation using the fork() system call.

B. Process Termination:

A process terminates when it finishes executing its final statement and asks the operating system to delete it by using the exit () system call. A parent may terminate the execution of one of its children for a variety of reasons, such as these:

1. The child has exceeded its usage of some of the resources that it has been allocated. (To determine whether this has occurred, the parent must have a mechanism to inspect the state of its children.)
2. The task assigned to the child is no longer required.
3. The parent is exiting, and the operating system does not allow a child to continue if its parent terminates.

In such systems, if a process terminates (either normally or abnormally), then all its children must also be terminated. This phenomenon, referred to as cascading termination, is normally initiated by the operating system.

4. INTER PROCESS COMMUNICATION

An independent process is one that cannot affect or be affected by the execution of another process

A cooperating process can affect or be affected by the execution of another process in the system

Advantages of process cooperation:

1. Information sharing (of the same piece of data)
2. Computation speed-up (break a task into smaller subtasks)
3. Modularity (dividing up the system functions)
4. Convenience (to do multiple tasks simultaneously)

Two fundamental models of Interprocess communication

1. Shared memory (a region of memory is shared)
2. Message passing (exchange of messages between processes)

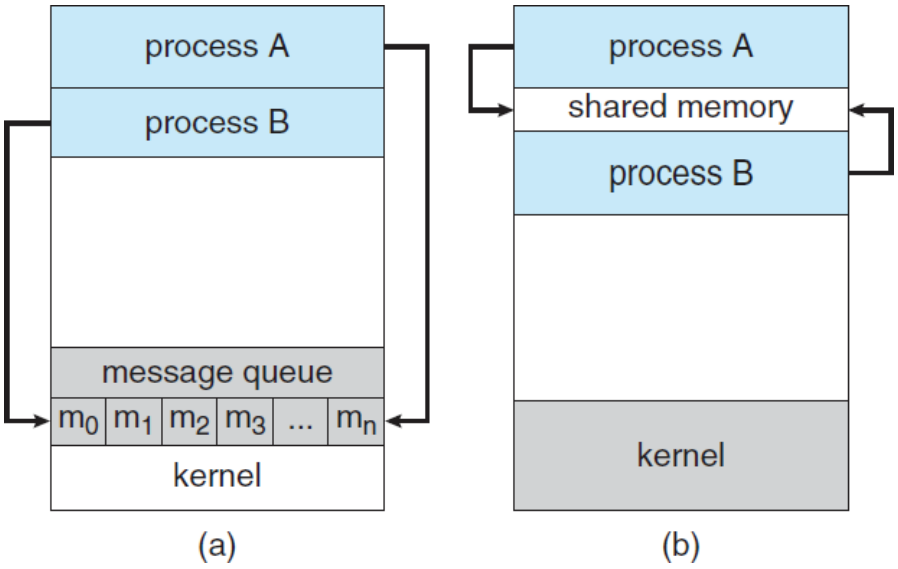


Figure Communication models: (a) Message Passing. (b) Shared Memory

A. Shared Memory Systems:

Interprocess communication using shared memory requires communicating processes to establish a region of shared memory. Typically, a shared-memory region resides in the address space of the process creating the shared-memory segment.

A producer process produces information that is consumed by a consumer process.

One solution to the producer–consumer problem uses shared memory.

Two types of buffers can be used. The unbounded buffer places no practical limit on the size of the buffer. The consumer may have to wait for new items, but the producer can always produce new items. The bounded buffer assumes a fixed buffer size. In this case, the consumer must wait if the buffer is empty, and the producer must wait if the buffer is full.

```
#define BUFFER SIZE 10
typedef struct {
. . .
}item;
item buffer[BUFFER SIZE];
int in = 0;
int out = 0;
```

Producer Process:

```
while (true) {
/* produce an item in next produced */
while (((in + 1) % BUFFER SIZE) == out)
; /* do nothing */
buffer[in] = next produced;
in = (in + 1) % BUFFER SIZE;
}
```

Consumer Process:

```
item next consumed;
while (true) {
while (in == out)
        ; /* do nothing */
next consumed = buffer[out];
out = (out + 1) % BUFFER SIZE;
/* consume the item in next consumed */
}
```

B. Message Passing Systems:

- Mechanism to allow processes to communicate and to synchronize their actions
- No address space needs to be shared; this is particularly useful in a distributed processing environment (e.g., a chat program)
- Message-passing facility provides two operations:
  - send(message) – message size can be fixed or variable
  - receive(message)
- If P and Q wish to communicate, they need to:
  - establish a communication link between them
  - exchange messages via send/receive
- Logical implementation of communication link
  - Direct or indirect communication
  - Synchronous or asynchronous communication
  - Automatic or explicit buffering

1) Direct Communication:

Processes must name each other explicitly:

send (P, message) – send a message to process P
receive(Q, message) – receive a message from process Q

Properties of communication link:

- Links are established automatically between every pair of processes that want to communicate
- A link is associated with exactly one pair of communicating processes
- Between each pair there exists exactly one link
- The link may be unidirectional, but is usually bi-directional

Disadvantages

- Limited modularity of the resulting process definitions
- Hard-coding of identifiers are less desirable than indirection techniques

2. Indirect Communication:

Messages are directed and received from mailboxes (also referred to as ports)

- Each mailbox has a unique id
- Processes can communicate only if they share a mailbox
  - send(A, message) – send message to mailbox A
  - receive(A, message) – receive message from mailbox A

Properties of communication link

- Link is established between a pair of processes only if both have a shared mailbox
- A link may be associated with more than two processes
- Between each pair of processes, there may be many different links, with each link corresponding to one mailbox

For a shared mailbox, messages are received based on the following methods:

- Allow a link to be associated with at most two processes
- Allow at most one process at a time to execute a receive() operation
- Allow the system to select arbitrarily which process will receive the message (e.g., a round robin approach)

Mechanisms provided by the operating system

- Create a new mailbox
- Send and receive messages through the mailbox
- Destroy a mailbox

Synchronization:

- Message passing may be either blocking or non-blocking
- Blocking is considered synchronous
- Blocking send has the sender block until the message is received
- Blocking receive has the receiver block until a message is available
- Non-blocking is considered asynchronous
- Non-blocking send has the sender send the message and continue
- Non-blocking receive has the receiver receive a valid message or null
- When both send() and receive() are blocking, we have a rendezvous between the sender and the receiver

Buffering:

Whether communication is direct or indirect, messages exchanged by communicating processes reside in a temporary queue

These queues can be implemented in three ways:

1. Zero capacity – the queue has a maximum length of zero - Sender must block until the recipient receives the message
2. Bounded capacity – the queue has a finite length of n - Sender must wait if queue is full
3. Unbounded capacity – the queue length is unlimited Sender never blocks

5. THREADS

A thread is a light weight process. It comprises a thread ID, a program counter, a register set, and a stack. It shares with other threads belonging to the same process its code section, and other operating-system resources such as open files and signals. For e.g, Windows-XP and Solaris kernels are multithreaded.

A. Benefits:

- **Resource Sharing:** Threads share memory and resources of the process to which they belong.
- **Economy:** Allocating memory and resources for process creation is costly. Solaris, for example, creating a process is about thirty times slower than is creating thread, and context switching is five times slower.

- **Utilization of multiprocessor architecture:** Multiple threads can run on multiple processors parallel. A single threaded process can run only on one CPU.
- **Responsive time**: Multithreading an interactive application may allow a program to continue running even if part of it is blocked, thereby increasing responsiveness.

B. Multithreading Models

Threads may be provided either at the user level, for user threads, or by the kernel, for kernel threads. User threads are supported above the kernel and are managed by without kernel support. Kernel threads are supported and managed directly by the operating system. All contemporary operating systems like Solaris, windows-XP, Linux and MAC OS x support kernel threads.

Many-to-One model:

The many-to-one model maps many user-level threads to one kernel thread. Thread management is done by thread library in user space, so it is efficient; but the entire process will block if a thread makes a blocking system call. Since, kernel is unaware of user threads, multiple threads can't be scheduled on multiple processors. GNU portable threads on UNIX Java threads, WIN32 threads and Green threads on Solaris are user thread libraries.

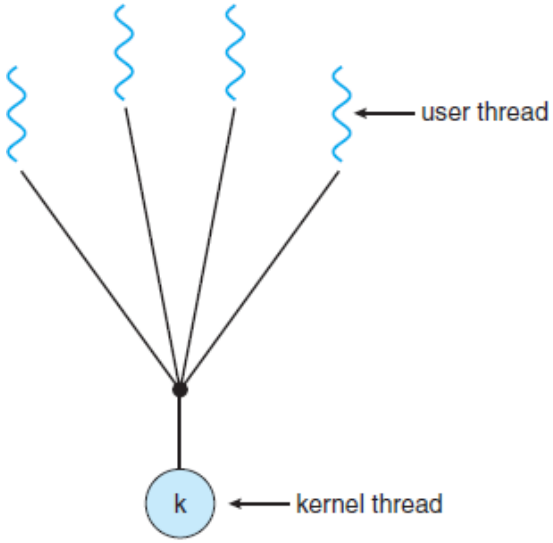


Figure Many-to-one model.

One-to-One Model:

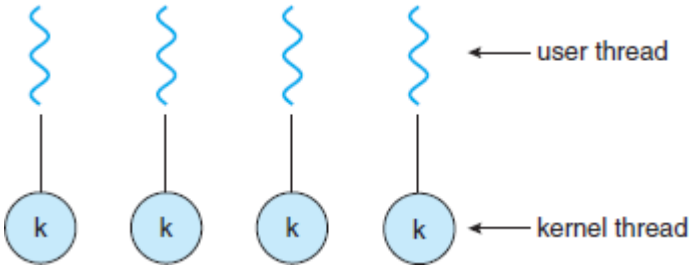


Fig: One-to-One Model

The one-to-one Model maps each user thread to a kernel thread. It provides more concurrency than the many-to-one model by allowing another thread to run when a thread makes a blocking system call; it allows multiple threads to run in parallel on multiprocessors. The only drawback is creating a kernel thread correspond to user thread. Creating a kernel thread is an overhead to the application. Solaris 9, Linux, Windows 95,98, NT, 200 and xp implement one-to-one model.

Many-to-Many Model:

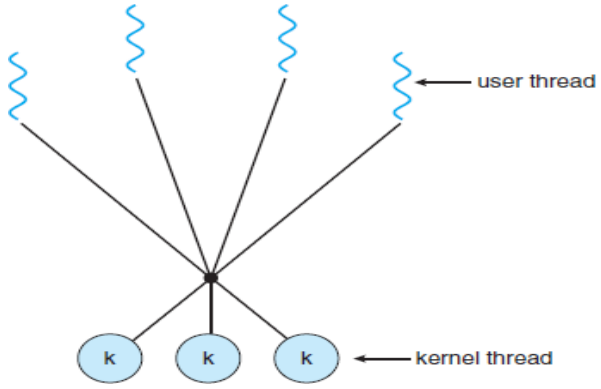


Fig: Many-to-Many Model

The many-to-many model multiplexes many user-level threads to small or equal number of kernel threads. Many-to-one model allows the developer to create as many user threads as she wishes, true concurrency is not gained because kernel can schedule only one user thread at a time. The one-to-one model allows for greater concurrency, but a developer has to be careful not to create too many threads within application. The man-to-many model does not suffers from neither of these shortcomings: Developers can create as many user threads as necessary, and corresponding kernel threads can run in parallel on a multi-processor. Also, when a thread performs a blocking system call, the kernel can schedule another thread for execution. Solaris prior to version 9, windows NT/2000 with thread fiber package supports many-to-many model.

Two-level-model:

It is a variation of many-to-many model. Two-level model still multiplexes many userlevel threads to a smaller or equal number of kernel threads but also allows a user-level thread to be bound to kernel thread. Two-level model is supported by Solaris 8 and earlier.

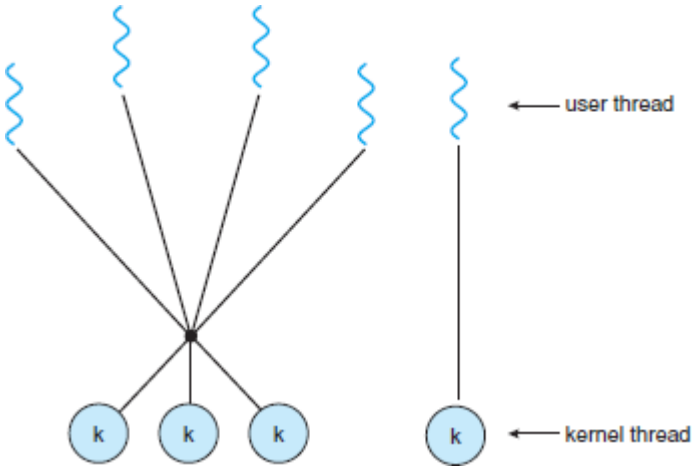


Fig: Two-level-Model

6. PROCESS SCHEDULING CRITERIA AND ALGORITHMS

Pre-emptive Scheduling:

CPU scheduling decisions may take place when a process:

1. Switches from running to waiting state
2. Switches from running to ready state
3. Switches from waiting to ready
4. Terminates

For situations 1 and 4, a new process must be selected for execution.

Non pre-emptive or cooperative scheduling (situation 1 and 4), Once a process is in the running state, it will continue until it terminates or blocks itself for I/O. This scheduling method was used by windows 3.x.

Pre-emptive:

- Currently running process may be interrupted and moved to ready state by the OS
- Pre-emption may occur when new process arrives, on an interrupt, or periodically.
- This scheduling method was used by Windows 95, 98, xp and MAC OS x.

Scheduling Criteria:

Scheduling criteria is used to compare different scheduling algorithms.

- **CPU Utilization:** We want keep the CPU as busy as possible. In real systems, CPU utilization range from 40 to 90 percent. Every scheduling algorithm tries to maximize the CPU utilization.
- **Throughput:** No of processes that are completed their execution per time unit, called throughput. Throughput depends upon scheduling policy and average length of process. Every scheduling algorithm tries to maximize the Throughput.
- **Turnaround time:** This interval from time of submission of a process to the time of completion is the turnaround time. Scheduling algorithm tries to minimize the turnaround time.

  Turnaround time = waiting time to get into memory + waiting time in ready queue + I/O time + CPU execution time.
- **Waiting time:** Waiting time is the sum of the periods spent waiting in the ready queue. Scheduling algorithm tries to minimize the waiting time.
- **Response time:** It is the time from the submission of a request until the first response produced. It is the important criteria for interactive jobs. Scheduling algorithm tries to minimize the waiting time.

Scheduling Algorithms:

CPU scheduling algorithm decides which of the processes in ready queue is to be allocated the CPU. There are many CPU scheduling algorithms.

A) First-Come, First-Served Scheduling (FCFS):

In FCFS, the process that requests the CPU first is allocated the CPU first. Each process joins the ready queue. When the CPU is free, it is allocated to the process at the head of the queue. The running process is removed from the queue. A short process may have to wait a very long time before it can execute. In FCFS average waiting time is long. FCFS scheduling algorithm is a no pre-emptive. It is not suitable for interactive jobs and time sharing systems.

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

Process	Burst Time
P1	24
P2	3
P3	3

Suppose that the processes arrive in the order: P1 , P2 , P3

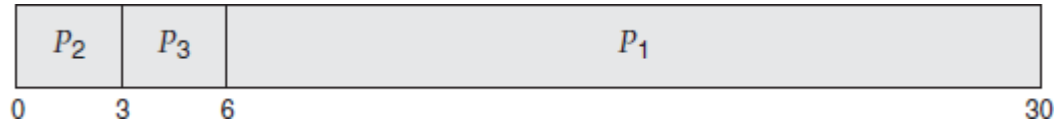
The Gantt chart for the schedule is:

Process	Burst Time	Waiting Time	Turn Around Time	Response Time
P1	24	0	24	0
P2	03	24	27	24
P3	03	27	30	27

Waiting time for P1 = 0; P2 = 24; P3 = 27

Average waiting time: (0 + 24 + 27)/3 = 17
Average turn Around Time : (24+27+30) / 3 = 27
Average Response Time : (0 + 24 + 27)/3 = 17.

Process	Burst Time
P2	3
P3	3
P1	24



Process	Burst Time	Waiting Time	Turn Around Time	Response Time
P2	03	0	3	0
P3	03	3	6	3
P1	24	6	30	6

Average waiting time: (0 + 3 + 6)/3 = 3
Average turn Around Time : (3+6+30) / 3 = 13
Average Response Time : (0 + 3 + 6)/3 = 3

Convoy effect:

Assume we have one CPU-bound process and many I/O bound processes. The CPU-bound processes will get and hold the CPU. During this time, all the other processes will finish their I/O and will move into the ready queue, waiting for the CPU. While the processes wait in ready queue, the I/O devices are idle. Eventually, the CPU bound process finishes its CPU burst and moves to an I/O device. All the I/O bound processes, which have short CPU bursts, execute quickly and move back to I/O queues. At this point, CPU is IDLE. The CPU-bound process will then move back to ready queue and be allocated the CPU. Again, all the I/O bound processes will end up waiting in ready queue until the CPU-bound processes is done. This is a CONVOY effect as all the other processes wait for one big process to get off the CPU.

B) Shortest-job-First Scheduling (SJF):

When CPU is available, it is assigned to the process that has the smallest next CPU burst. If the next CPU bursts of two processes are same, FCFS scheduling is used to break the tie. The more appropriate name for this algorithm is Shortest-next-CPU-burst algorithm. It is not suitable for time-sharing systems and interactive jobs. The longer process may get starvation.

Process	Burst Time
P1	6
P2	*8*
P3	7
P4	*3*

Process	Burst Time	Waiting Time	Turn Around Time	Response Time
P1	6	3	9	3
P2	8	16	24	16
P3	7	9	16	9
P4	3	0	3	0

Average waiting time: (0 + 3 + 9+16)/4 = 7
Average turn Around Time : (9+24+16+3) / 4 = 13
Average Response Time : (3+16+9 + 0)/4 = 7

The average waiting time is very low. The real difficult with the SJF algorithm is knowing the length of the next CPU burst. SJF algorithm is frequently used in long term scheduling and it cannot be implemented at the level of short-term scheduling

The SJF algorithm can be either pre-emptive or non-pre-emptive. The choice arises when a new process arrives at the ready queue while previous process is still executing. The next CPU burst of the newly arrived process may be shorter than what is left of the currently executing process. A pre-emptive SJF algorithm will pre-empt the currently executing process, whereas a non-pre-emptive SJF algorithm will allow the currently running process to finish its CPU burst.

Process	Burst Time(Mille Seconds)	Arrival Time
P1	8	0
P2	*4*	*1*
P3	9	2
P4	*5*	*3*

Process	Burst Time	Arrival Time	Waiting Time	Turn Around Time	Response Time
P1	8	0	9	17	0
P2	4	1	0	4	0
P3	9	2	15	24	15
P4	5	3	2	7	2

Average waiting time: (9+ 0 + 15+2)/4 = 6.5
Average turn Around Time: (9+24+16+3) / 4 = 13
Average Response Time : (3+16+9 + 0)/4 = 7

C. Priority Scheduling:

A priority is associated with each process. And the CPU is allocated to the process with highest priority. Equal priority processes are scheduled in FCFS order. (An SJF is special case of priority scheduling. The largest the CPU burst, the lower the priority, and vice versa. In this topic, we assume that low numbers represent high priority.

Process	Burst Time(Milli Seconds)	Priority
P1	10	3
P2	1	1
P3	2	4
P4	1	5
P5	5	2

Process	Burst Time	Priori	Waiting Time	Turn Around Time	Response Time
P1	10	3	6	16	6
P2	1	1	0	1	0
P3	2	4	16	18	16
P4	1	5	18	19	18
P5	5	2	1	6	1

Priorities can be defined internal or externally. Internal priorities are defined based on time limit, memory requirement, number of opened files. Priority scheduling can be either preemptive or nonpreemtive. When a process arrives at the ready queue, its priority is compared with priority of currently running process. A preemptive priority scheduling algorithm will preempt the CPU if the priority of the newly arrived process is higher than the priority of the currently running process. A nonpreemptive priority scheduling algorithm will simply put the new process at the head of the ready queue.

A major problem with priority scheduling algorithm is INDEFINITE BLOCKING or STARVATION. A priority scheduling algorithm can leave some low priority processes waiting indefinitely. A steady stream of high higher-priority processes can prevent low-priority processes waiting indefinitely. Then the low priority processes get starvation.

A solution to the problem of STARVATION of low priority processes is AGING. Aging is a technique of gradually increasing the priority of processes that wait in the system for a long time. For example, increase the priority of a low priority job by 1 for every 15 minutes.

D) Round-Robin Scheduling Algorithm (RR):

The Round-Robin (RR) Scheduling Algorithm is designed especially for time sharing systems. Each process gets a small unit of CPU time (*time quantum*), usually 10-100 milliseconds. After this time has elapsed, the process is preempted and added to the end of the ready queue.

To implement RR scheduling, we keep the ready queue as FIFO queue of processes. New processes are added to the tail of the ready queue. The CPU scheduler picks the first process from ready queue, sets timer to interrupt after 1 time quantum, and dispatches the process.

The process may have a CPU burst of less than 1 time quantum. In this case, the process itself will release the CPU voluntarily. The scheduler will then proceed to the next process in the ready queue.

If CPU burst is longer than 1 time quantum, the timer will go off and will cause an interrupt to OS. A context switch will be executed, and the running process will be put at the tail of the ready queue. The Round-Robin scheduler selects the next process in the ready queue.

The average waiting time under RR policy is often long. If time quantum increases turnaround time decreases and response time increases. In general time quantum always should be greater than context switching time. If time quantum is too long it will become FCFS scheduling algorithm.

Problem 1:

Process	Burst Time
P1	24
P2	3
P3	3

Process	Burst Time	Waiting Time	Turn Around Time	Response Time
P1	24	6	30	0
P2	3	4	7	4
P3	3	7	10	7

Assume that all jobs are submitted at the same time and time quantum is 4msec.

The average waiting time is 17/3 = 5.66 milliseconds.

Problem 2:

Process	Burst Time
P1	24
P2	3
P3	3

Assume that all jobs are submitted at the same time and time quantum is 2msec.

Process	Burst Time	Waiting Time	Turn Around Time	Response Time
P1	24	6	30	0
P2	3	6	9	2
P3	3	7	10	4

If time slice decreases, the average turnaround time increases and average response time decreases.

Problem 3:

Process	Burst Time
P1	24
P2	7
P3	7

Process	Burst Time	Waiting Time	Turn Around Time	Response Time
P1	24	13	41	0
P2	7	12	19	4
P3	10	19	29	8

Assume that all jobs are submitted at the same time and time quantum is 4msec.

UNIT-III : Process Synchronization: The Critical-Section Problem, Peterson's Solution, Synchronization Hardware, Mutex Locks, Semaphores, Classic Problems of Synchronization, Monitors.

A cooperating process is one that can affect or be affected by other process executing in the system. ***A concurrency*** is a property of system in which several computations are executing simultaneously, and potentially interacting with each other

1. BOUNDED BUFFER

Producer & Consumer Problem:

The produces elements and stores it in the better. The consumer consumes elements from the buffer.

The producer and consumer shares same butter. Both should not access the buffer at the same time

```
While(true)
{
/*produce an item in next produced */
While(counter==BUFFER_SIZE)
;/*do nothing*/
Buffer [in]=next produces;
In=[in+1]%BUFFER_SIZE;
Counter ++;
}
```

Fig: The code for the producer process

```
While(true)
{
While(counter==0)
        ;       /*do nothing*/
next consumed=buffer[out];
out=(out++)% BUFFER_SIZE;
counter--;
/*consume item in next consumed*/
}
```

Fig: The code for consumer

T0: producer execute register1=counter [register1=5]
T1: producer execute register1=counter+1 [register1=6]
T2: consumer execute register2=counter [register2=5]
T3: consumer execute register2=reg2-1 [register2=4]
T4: producer execute counter=register1 [counter=6]
T5: consumer execute counter=reg2 [counter=4]

When several process access and manipulate the data concurrently and the outcome of the execution depends on particular order in which access takes place, is called *Race Condition*.

2. THE CRITICAL SECTION PROBLEM

Consider a system consisting of 'n' process $p_0,p_1….p_{n-1}$ each process has a segment of code called a critical section, in which process may be changing common variables, updating table, writing a file and so on. The critical section problem is to design a protocol that the process can use to cooperate. Each process must request permission to enter its critical section.

```
do {
        Entry section

          Critical section

        Exit section

         Remaining section
      }while(TRUE);
```

Fig 3: General structure of a typical process p_i

The important feature of the system is that, when process is executing in its critical section no process is to be allowed to execute in its critical section.

Requirements to Critical – Section Problem

i. ***Mutual Exclusion:*** If the process p1 is executing in its critical section, then no other process can be executing in their critical sections.
ii. ***Progress:*** If no process is executing in its critical section and some process wish to enter its critical section, it should be allowed to enter its critical section and it cannot be stopped indefinitely.
iii. ***Bound waiting:*** There exist a bound, on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

3. PETERSON SOLUTION

A classic software based solution to the critical section problem known as Petersons solution. The Peterson solution is restricted to two processes that alternate execution between their critical section and remainder section. The process are numbered p_0 (p_i) and p_1 (p_j). Peterson's solution requires *two data items to be shared between the two processes.*

int turn;

boolean flag[2];

The variable turn indicates whose turn it is to enter its critical section. If flag[i] is true , p_i is ready to enter into critical section. *If flag[j] is true, p_J is ready to enter into* critical section. If turn = i, then process p_i is allowed to execute in its critical section.

If both processes try to enter at the same time, turn will be set to both i and J at roughly the same time. Only one of these assignments will be last. The eventual value of turn decides which of the two processes is allowed to enter its critical section first.

```
do {
Flag[i]=true;
Turn =j;
While(flag[j]&&turn==j);
Critical section
Flag[i]=false
Remainder section
}while(true);
```

Fig 4: The structure of p_i in Peterson solution.

To prove the solution is correct, we need to show

1. Mutual exclusion preserved
2. The progress requirement is satisfied
3. The bounded-waiting requirements is not.

To prove property – 1:

P_i enters its critical section only if either Flag[j] == false or turn == i

If both processes executing critical section at the same time, then flag[0]==flag[i]==true.

Turn can be either 0 or 1, so only one can enter critical section.

To prove properties 2 and 3:

Pi can be prevented from entering critical section only if its stuck in the while loop with the while loop with the condition flag[j]==true and turn==j;\if pj is not interested to enter critical section, then flag[j]=false, and pi can enter its critical section.

4. SYNCHRONIZATION HARDWARE

Race conditions are prevented by requiring that critical regions be protected by locks.

A process must acquire a lock before entering a critical section; it releases the lock when it exits a critical section.

```
do
{
        Acquire lock
        Critical section
        Release lock
        Remainder section
} while(True);
```

Fig 5: The structure of p_i using locks

The critical section problem could be solved simply in a uniprocessor environment by disabling interrupts while shared variable was being modified. This solution is not feasible in a multiprocessor environment. Since disabling interrupts in multiprocessor environment is a time consuming.

The message passing delays entry into each critical section, and system efficiency decreases Modern computer systems therefore provide special hardware instructions that allow us either to test and modify the content of a word or swap the contents of two words *Atomically*.

```
Boolean TestAndSet (Boolean *farget)
{
        Boolean    rv= * target;
                   *target = TRUE;
                   return    rv;
        }
```

Fig 6: The definition of two TestAndSet() instruction

```
do
        {
        while (Test and set Lock(&lock))
                                ; / * do nothing*/
            // critical section
           lock = FALSE;
             // remainder section
        } while (TRUE);
```

Fig 7: Mutual-Exclusion implementation with TestandSet()

The important characteristic of TestAndSet() instruction is that instruction is executed atomically. Thus, if two TestAndSet() instructions are executed simultaneously, they will be executed sequentially in some arbitrary order. If the machine supports the TestAndSet() instruction, then we can implement mutual exclusion by declaring Boolean variable lock, initialized to false. The structure of process P; is shown in fig 7.

The swap() instruction exchanges values of two variables. It is executed atomically. If a machine supports swap() instruction, then mutual exclusion can be implemented.

```
Void swap( Boolean *a, Boolean *b)
{
Boolean temp=*a;
        *b=*a;
        *a=temp;
}
```

Fig 8: The definition of swap() instruction

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

Fig 9: Mutual exclusion implementation with swap() instruction

A global Boolean variable lock is declared and is initialized to false. In addition, each process has a local Boolean variable key. The structure of P_i is shown in fig 9.

5. SEMAPHORES

Semaphore is a synchronous tool. A semaphore S is an integer variable. It is accessed only through two standard atomic operations wait() and signal(). The wait operation was originally termed as P(decrement) and signal() was originally called V (increment). Wait() and signal() operation are atomic.

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

Fig 10 : The definition of wait()

```
Signal(s)
{
S++;
}
```

Fig 11: The definition of signal()

Usage of semaphore:

Operating system supports two types of semaphores. The value of counting semaphore can range over an unrestricted domain. The value of binary semaphore can range only between 0 and 1. Can some systems binary semaphores are called as mutex locks. We can use binary semaphore to deal with the critical section problem for multiple processes. The n processes share a semaphore, mutex, initialized to 1. Each process P; is organized as shown below:

```
do
 {
      Wait (mutex);
      // critical section
     Signal (mutex);
      // remainder section
} while (TRUE);
```

Fig 12 : Mutual exclusion implementation with semaphore.

Counting semaphore can be used to control access to a given resource consisting of a finite number of instances. The semaphore is initialized to the number of resources available. Each process that wishes to use a resource performs wait() operation on the semaphore(decreases the count). When a process releases a resource, it performs signal operation (incrementing the count). When the count for semaphore goes to 0,all resources are being used. After that, processes that with to use a resource will block until the count becomes greater than 0. Semaphore can be used to solve various synchronization problems.

Consider two concurrent processes : P1 and P2. P1 has statements S1 and P2 has statements S2. Suppose we require that S2. Suppose we require that S2 be executed only after S1 has completed.

We can implement this scheme readily by letting P_1 and P_2 share a common semaphore synch, initialized to 0(Zero).

```
                S1
Signal(synch)          Process P1

Wait (synch);
                S2              Process P2
```

Implementation of semaphore:

The main disadvantage of semaphore definition is that it requires busy waiting when a process is in its critical section, any other process that tries to enter its critical section must loop continuously in the entry code. Busy waiting wastes CPU cycles. This type of semaphore is also called a SPINLOCK because the process "spins" while waiting for the lock.

The overcome the need for busy waiting, the definition of wait() and signal() operations can be modified. When a process executes the wait() operation and finds semaphore value is not positive, it must wait. However, rather than engaging in busy waiting, the process can BLOCK it self. The block operation places a process in to a waiting queue associated with the semaphore, and the state of the process is switched to the waiting state.

A process that is blocked, waiting on semaphore S, should be restarted when same other process executes a signal() operation. The process is then placed in the ready queue.

Implementation of semaphore:

```
typedef    struct {
                        int value;
                        Struct process *list;
                    } semaphore;
```

Each semaphore has an integer value and a list of processes list. When a process must wait on semaphore, it is added to the List of processes. A signal() operation removes one process from the list of waiting processes and awakens that process.

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

Fig 13 : The wait() semaphore operation

```
Signal(semaphore *s)
{
    s->value++;
    if(s->value <= 0)
    {
        Remove a process P from S->list;
        wake(P);
    }
}
```

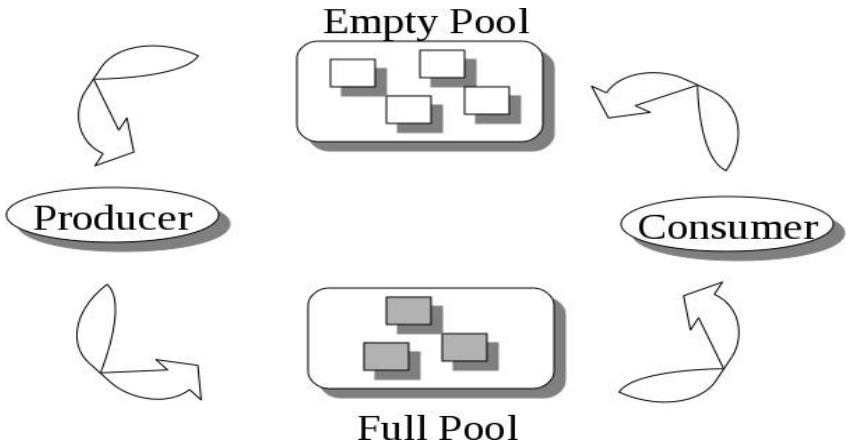
Fig 14 : The signal() semaphore operation

The block operation suspends the process that invokes it. The wakeup(P) operation resumes the execution of a blocked process P. The critical aspect of semaphore is that they be executed atomically. We must guarantee that no two processes can execute wait() and signal() on same semaphore at the same time. This is critical section problem; and in a single-processor environment, we can solve it by simply inhibiting interrupts during the time wait() and signal() operations are executing.

6. CLASSIC PROBLEMS OF SYNCHRONIZATION

The Bounded buffer Problem:

We assume that the pool consist of '*n*' buffers, each capable of holding one item. The *mutex* semaphore provides mutual exclusion for accesses to the buffer pool and is initialized to the value 1. The *empty* and *full* semaphores count the number of empty and full buffers. The semaphore *empty* is initialized to the value n; the semaphore *full* is initialized to the value 0.




```
do
{
     // produce an item nextp
           . . . . .

     wait(empty);
      wait(mutex);

      . . . . .

       // Add nextp to buffer
      . . . . .

     Signal(mutex);
     Signal(full);
  } while(TRUE);
```

Fig 17 : The structure of the producer problem

```
do
{
     wait(full);
      wait(mutex);

      . . . . .

       // Remove an item from buffer to nextc
      . . . . .

     Signal(mutex);
     Signal(empty);
  } while(TRUE);
```

Fig 18 : The structure of the consumer problem

We can interpret this code as the producer producing full buffers for the consumer or as the consumer producing empty buffers for the producer.

The Readers-Writers Problem:

A database is to be shared among several concurrent processes. Some of these processes may want only to read the database (Readers), whereas others may want to update the database (Writers). If two readers access the shared data simultaneously, no adverse effects will result. However, if a writer is accessing a database no other reader or writer is allowed to access the database. This synchronization problem is referred to as Readers-**Writers** problem. In simplest reader-writer problem, several readers can access shared data at the same time and if a writer is waiting to access the object, no new readers may start reading. The reader's processes share the following data structures:

```
Semaphore mutex, wrt;
int readcount;
```

The semaphores mutex and wrt are initialized to 1; readcount is initialized to 0. The semaphore wrt is common to both reader and writer processes. The mutex semaphore is used to ensure mutual exclusion when the variable readcount is updated. The readcount variable keeps track of how many processes are currently reading the object. The semaphore wrt functions as a mutual-exclusion semaphore for the writers. It is also used the first or last.

```
do
{
   wait(mutex);
   ……
 signal(wrt);
}while(TRUE);
```

Fig 19: The structure of a writer process

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

Fig 20: The structure of a reader process

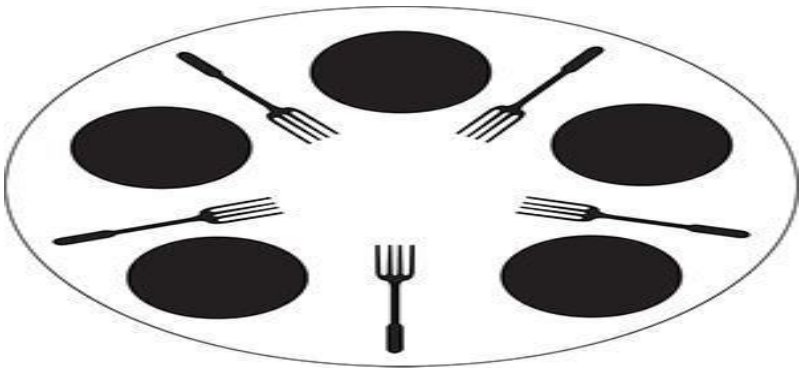
Note that, if a writer is in the critical section and *n* reader are waiting, then one reader is queued on with respect to and n-1 readers are queued on mutex.

The reader-writer locks are most useful in the following situations:

1. In applications where is easy to identify which processes only read shared data and which threads only write shared data.
2. In applications that have more readers than writers.

The Dining Philosophers Problem:

Consider five philosophers sit around a table pondering philosophical issues. A plate of food is kept in front of each philosopher, and a fork is placed in between each pair of philosophers. A philosopher wishing to eat goes to his or her assigned plate at the table and, using the two forks on either side of the plate, takes and eats some food. A philosopher can take food only if he gets both forks i.e he can't take food with one fork.



```
/*Program dining philosophers */
semaphore fork[5] = {1};
int i;
void philosopher(int i)
{
    do
    {
       wait(fork[i]);
       wait(fork[(i+1) % 5]);
          . . . . .
       // eat
      signal(fork[(i+1) % 5]);
      signal(fork[i]);

      // Think
    }while(TRUE);
}
```

Fig 22 : The structure of philosopher

After the philosopher is finished eating, the two forks are replaced on the table. This solution leads to deadlock: if all of the philosophers are hungry at the same time, they all sit down, they all pick up the fork on their left, and they all reach out for other fork, which is not there. This situation causes deadlock.

To overcome the risk of deadlock, we could place room attendant. The room attendant who only allows four philosophers at a time into dining room.

```
/*Program dining philosophers */
semaphore fork[5] = {1};
semaphore room = {4};
int i;
void philosopher(int i)
{
    do
    {
      Wait(room);
       wait(fork[i]);
       wait(fork[(i+1) % 5]);
          . . . . .
       // eat
      signal(fork[(i+1) % 5]);
      signal(fork[i]);
      signal(room);

      // Think
    }while(TRUE);
 }
```

Fig 23 : The structure of philosopher

7. MONITORS

Semaphores provide primitive yet powerful and flexible tool for implementing mutual exclusion and for coordinating processes. The difficulty is that wait() and signal() operations may be scattered throughout a program and it is not easy to see the overall effect of these operations on the semaphores they affect.

The monitor is a programming-language construct that provides equivalent functionality to that of semaphores and that is easier to control. The monitor construct has been implemented in java, modula-2, modula-3 and concurrent pascal.

A monitor is a software module consisting of one or more procedures, an initialization sequence, and local data. The chief characteristics of monitor are the following:

1. The local data variables are accessible only by the monitor's procedures and not by any external procedure.
2. A process enters the monitor by invoking one of its procedures.
3. Only one process may be executing in the monitor at a time;any other process that have invoked the monitor are blocked.

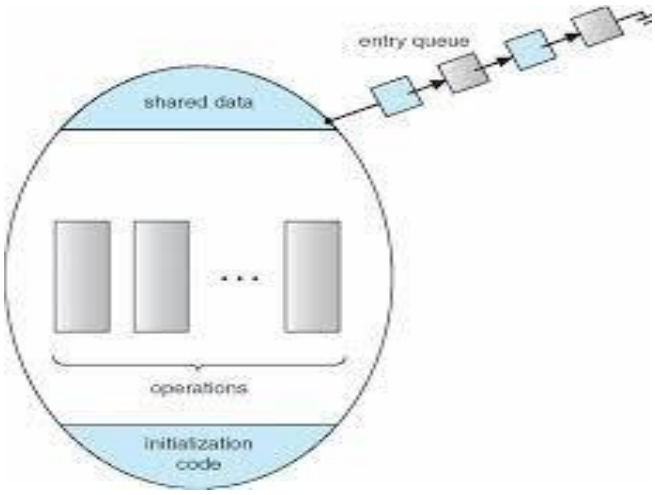
```
monitor    monitor-name
{
   // shared variable declaration

    // condition variables

     procedure  p1( .....)
     {
         . . . . .
```

```
    }
    procedure  p2(.....)
    {
        . . . . .
    }
        .
        .
    procedure  pn(.....)
    {
        . . . . .
    }
    Initialization code( ..... )
    {
        . . . . .
    }
}
```

Fig 24 : Syntax of Monitor



A monitor supports synchronization by the use of **condition variables** that are defined in the monitor and accessible within the monitor.

condition x,y;

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

x.wait();

means that the process invoking this operation is suspended until another process invokes.

x.signal();

The x.signal() operation resumes exactly one suspended process. If no process is suspended, then the signal() operation has no effect.

Now suppose that, when the x.signal() operation is invoked by a process P, there is a suspended process Q associated with the condition X. Clearly, if suspended process Q is allowed to resume its execution, the signaling process must wait. Otherwise, both P and Q would be active simultaneously within the monitor(which is not correct).

Dining-Philosophers Solution Using Monitors:

The monitor concepts present a deadlock-free solution to the dining-philosophers problem. This solution imposes the restriction that a philosopher may pick up her forks only if both of the forks are available. The philosopher may be in thinking, hungry, eating states.

enum { thinking, hungry, eating } = state[5];

Philosopher 'i' can set the variable state[i] = eating only if her two neighbors are not eating : state[(i+1) % 5] != eating) and state[(i+4) % 5] != eating)

```
monitor dp
{
    enum { THINKING , HUNGRY, EATUNG } status[5];
    condition self[5];

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

Fig : A monitor solution to the dining philosopher problem

Synchronization Examples:

- Solaris
- Windows XP
- Linux
- Pthreads

Solaris Synchronization

- Implements a variety of locks to support multitasking, multithreading (including real-time threads), and multiprocessing
- Uses **adaptive mutexes** for efficiency when protecting data from short code segments
- Uses **condition variables** and **readers-writers** locks when longer sections of code need access to data
- Uses **turnstiles** to order the list of threads waiting to acquire either an adaptive mutex or reader-writer lock

Windows XP Synchronization

- Uses interrupt masks to protect access to global resources on uniprocessor systems
- Uses **spinlocks** on multiprocessor systems
- Also provides **dispatcher objects** which may act as either mutexes and semaphores
- Dispatcher objects may also provide **events**
    - An event acts much like a condition variable

Linux Synchronization

- Linux:
    - Prior to kernel Version 2.6, disables interrupts to implement short critical sections
    - Version 2.6 and later, fully preemptive
- Linux provides:
    - semaphores
    - spin locks

Pthreads Synchronization

- Pthreads API is OS-independent
- It provides:
    - mutex locks
    - condition variables
- Non-portable extensions include:
    - read-write locks
    - spin locks