LECTURE NOTES

ON

Operating Systems

B. Tech. 4th Semester

Computer Science & Engineering

and

Computer Engineering

Prepared by

Mr. S. D. Kadam

Definition of Operating System

- An Operating System (OS) is an interface between computer user and computer hardware. An operating system is software which performs all the basic tasks like file management, memory management, process management, handling input and output, and controlling peripheral devices such as disk drives and printers.
- The operating system is designed in such a way that it can manage the overall resources and operations of the computer.
- Operating System is a fully integrated set of specialized programs that handle all the operations of the computer. It controls and monitors the execution of all other programs that reside in the computer, which also includes application programs and other system software of the computer.
- An Operating System (OS) is a collection of software that manages computer hardware resources and provides common services for computer programs. The operating system is the most important type of system software in a computer system.

Examples of Operating Systems

- **Windows** (GUI-based, PC)
- **GNU/Linux** (Personal, Workstations, ISP, File, and print server, Three-tier client/Server)
- **macOS** (Macintosh), used for Apple's personal computers and workstations (MacBook, iMac).
- **Android** (Google's Operating System for smartphones/tablets/smartwatches)
- **iOS** (Apple's OS for iPhone, iPad, and iPod Touch)

Computer System Components

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

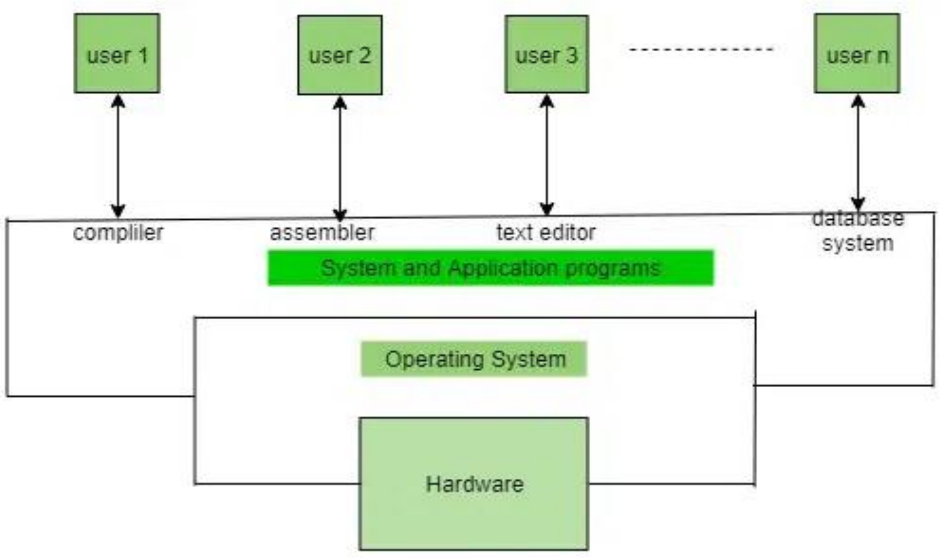


Figure Abstract view of the components of a computer system.

1. **Hardware –** provides basic computing resources (CPU, memory, I/O devices).
2. **Applications programs –** The Applications programs such as word processors/ spreadsheets/compilers, and Web browsers-define the ways in which these resources are used to solve users' computing problems.

3. **Operating system –** controls and coordinates the use of the hardware among the various application programs for the various users.
4. **Users** (people, machines, other computers).

- We can also view a computer system as consisting of hardware/ software/ and data. The operating system provides the means for proper use of these resources in the operation of the computer system.
- An operating system is similar to a government. Like a government, it performs no useful function by itself. It simply provides an environment within which other programs can do useful work.

Functions of the Operating System

The following are the Functions of the Operating System

- **Resource Management:** The operating system manages and allocates memory, CPU time, and other hardware resources among the various programs and processes running on the computer.
- **Process Management:** The operating system is responsible for starting, stopping, and managing processes and programs. It also controls the scheduling of processes and allocates resources to them.
- **Memory Management:** The operating system manages the computer's primary memory and provides mechanisms for optimizing memory usage.
- **Security:** The operating system provides a secure environment for the user, applications, and data by implementing security policies and mechanisms such as access controls and encryption.
- **Job Accounting:** It keeps track of time and resources used by various jobs or users.
- **File Management:** The operating system is responsible for organizing and managing the file system, including the creation, deletion, and manipulation of files and directories.
- **Device Management:** The operating system manages input/output devices such as printers, keyboards, mice, and displays. It provides the necessary drivers and interfaces to enable communication between the devices and the computer.
- **Networking:** The operating system provides networking capabilities such as establishing and managing network connections, handling network protocols, and sharing resources such as printers and files over a network.
- **User Interface**: The operating system provides a user interface that enables users to interact with the computer system. This can be a Graphical User Interface (GUI), a Command-Line Interface (CLI), or a combination of both.
- **Backup and Recovery:** The operating system provides mechanisms for backing up data and recovering it in case of system failures, errors, or disasters.
- **Performance Monitoring:** The operating system provides tools for monitoring and optimizing system performance, including identifying bottlenecks, optimizing resource usage, and analyzing system logs and metrics.
- **Time-Sharing:** The operating system enables multiple users to share a computer system and its resources simultaneously by providing time-sharing mechanisms that allocate resources fairly and efficiently.
- **System Calls:** The operating system provides a set of system calls that enable applications to interact with the operating system and access its resources. System calls provide a standardized interface between applications and the operating system, enabling portability and compatibility across different hardware and software platforms.
- **Error-detecting Aids:** These contain methods that include the production of dumps, traces, error messages, and other debugging and error-detecting methods.

Types of Operating system

- **Batch Operating System:**
  - This type of OS accepts more than one jobs and these jobs are batched/ grouped together according to their similar requirements. This is done by computer operator.
  - Whenever the computer becomes available, the batched jobs are sent for execution and gradually the output is sent back to the user.
  - It allowed only one program at a time. This OS is responsible for scheduling the jobs according to priority and the resource required.

- **Time-sharing Operating System:**
  - Time sharing (or multitasking) OS is a logical extension of multiprogramming, faster switching between multiple jobs to make processing faster.
  - It allows multiple users to share computer system simultaneously.
  - The users can interact with each job while it is running.
  - These systems use a concept of virtual memory for effective utilization of memory space. Hence, in this OS, no jobs are discarded. Each one is executed using virtual memory concept.
  - It uses CPU scheduling, memory management, disc management and security management. Examples: CTSS, MULTICS, CAL, UNIX etc.
  - Time-sharing Operating System is a type of operating system that allows many users to share computer resources (maximum utilization of the resources).

- **Distributed Operating System:**
  - Distributed Operating System is a type of operating system that manages a group of different computers and makes appear to be a single computer.
  - These operating systems are designed to operate on a network of computers. They allow multiple users to access shared resources and communicate with each other over the network.
  - It can be classified into two categories: Client-Server systems & Peer-to-Peer systems
  - Examples include Microsoft Windows Server and various distributions of Linux designed for servers.

- **Network Operating System:**
  - Network Operating System is a type of operating system that runs on a server and provides the capability to manage data, users, groups, security, applications, and other networking functions.
  - It create and manages user accounts on the network, Controlling access to resources on the network & provide communication services between the devices on the network etc.

- **Real-time Operating System:**
  - Real-time Operating System is a type of operating system that serves a real-time system and the time interval required to process and respond to inputs is very small.
  - These operating systems are designed to respond to events in real time.
  - They are used in applications that require quick and deterministic responses, such as scientific experiments, medical imaging systems, industrial control systems, weapon systems, robots, air traffic control systems, etc.

- **Multiprocessing Operating System:**
  - A Multiprocessor is a computer system with two or more central processing units (CPUs) share full access to a common RAM. The main objective of using a multiprocessor is to boost

the system's execution speed, with other objectives being fault tolerance and application matching.

- Multiprocessor Operating Systems are used in operating systems to boost the performance of multiple CPUs within a single computer system.
- Multiple CPUs are linked together so that a job can be divided and executed more quickly.

•Single-User Operating Systems:

- Single-User Operating Systems are designed to support a single user at a time.
- The single-user operating system allows permission to access your personal computer at a time by a single user, but sometimes it can support multiple profiles.
- It can also be used in official work and other environments as well.
- Examples include Microsoft Windows for personal computers and Apple macOS.

- **Multi-User Operating Systems:**
  - The operating system should have to meet the requirements of all its users in a balanced way so that if any problem would arise with a user, it does not affect any other user in the chain.
  - Multi-User Operating Systems are designed to support multiple users simultaneously.
  - In a multiuser operating system, multiple numbers of users can access different resources of a computer at the same time.
  - Examples include Linux and Unix.

- **Embedded Operating Systems:**
  - Embedded Operating Systems are designed to run on devices with limited resources, such as smartphones, wearable devices, and household appliances.
  - Examples include Google's Android and Apple's iOS.

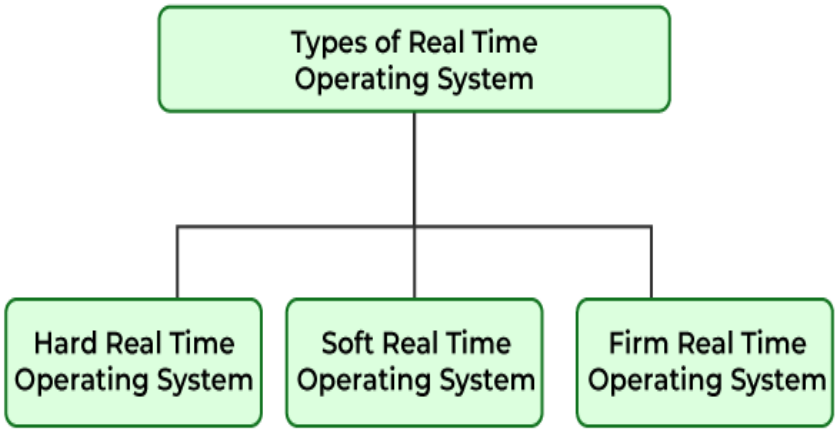
- **Cluster Operating Systems:**
  - Cluster Operating Systems are designed to run on a group of computers, or a cluster, to work together as a single system.
  - They are used for high-performance computing and for applications that require high availability and reliability.
  - Examples include Rocks Cluster Distribution and OpenMPI.

Real-Time operating system

- Real-time operating systems (RTOS) are used in environments where a large number of events, mostly external to the computer system, must be accepted and processed in a short time or within certain deadlines. Such applications are industrial control, telephone switching equipment, flight control, and real-time simulations.
- With an RTOS, the processing time is measured in tenths of seconds. This system is time-bound and has a fixed deadline.
- The processing in this type of system must occur within the specified constraints. Otherwise, otherwise the system will fail.
- A real-time system is defined as a data processing system in which the time interval required to process and respond to inputs is so small that it controls the environment. The time taken by the system to respond to an input and display of required updated information is termed as the response time.

- Real-time systems are used when there are rigid time requirements on the operation of a processor or the flow of data and real-time systems can be used as a control device in a dedicated application.

The real-time operating systems can be of 3 types –



1. **Hard Real-Time Operating System:**
   - These operating systems guarantee that critical tasks are completed within a range of time.
   - A hard real-time operating system is used when we need to complete tasks by a given deadline. If the task is not completed on time then the system is considered to be failed.
   - For example, a robot is hired to weld a car body. If the robot welds too early or too late, the car cannot be sold, so it is a hard real-time system that requires complete car welding by the robot hardly on time., scientific experiments, medical imaging systems, industrial control systems, weapon systems, robots, air traffic control systems, etc.

2. **Soft real-time operating system:**
   - This operating system provides some relaxation in the time limit.
   - A soft real-time operating system is used where few delays in time duration are acceptable. That is if the given task is taking a few seconds more than the specified time then also no critical damage takes place.
   - For example telephone switches, the sending or receiving of the call can take some time. It will not be considered a failure.

3. **Firm Real-time Operating System**:
   - A firm real-time operating system lies between the hard and soft real-time operating system.
   - A firm real-time system is one in which a few missed deadlines will not lead to total failure, but missing more than a few may lead to complete or catastrophic system failure.
   - However, unlike a hard real-time task, even if a firm real-time task is not completed within its deadline, the system doesn't fail but the late results are merely discarded.
   - For example, in Video Conferencing, when a certain frame is being played, if some preceding frame arrives at the receiver, then this frame is of no use and is discarded.

Advantages of RTOS

The advantages of real-time operating systems are as follows-

1. **Maximum consumption:** Maximum utilization of devices and systems. Thus more output from all the resources.

2. **Task Shifting:** Time assigned for shifting tasks in these systems is very less. For example, in older systems, it takes about 10 microseconds. Shifting one task to another and in the latest systems, it takes 3 microseconds.
3. **Focus On Application:** Focus on running applications and less importance to applications that are in the queue.
4. **Real-Time Operating System In Embedded System:** Since the size of programs is small, RTOS can also be embedded systems like in transport and others.
5. **Error Free:** These types of systems are error-free.
6. **Memory Allocation:** Memory allocation is best managed in these types of systems.

Disadvantages of RTOS:

The disadvantages of real-time operating systems are as follows-

1. **Limited Tasks:** Very few tasks run simultaneously, and their concentration is very less on few applications to avoid errors.
2. **Use Heavy System Resources:** Sometimes the system resources are not so good and they are expensive as well.
3. **Complex Algorithms**: The algorithms are very complex and difficult for the designer to write on.
4. **Device Driver And Interrupt signals:** It needs specific device drivers and interrupts signals to respond earliest to interrupts.
5. **Thread Priority:** It is not good to set thread priority as these systems are very less prone to switching tasks.
6. **Minimum Switching:** RTOS performs minimal task switching.

Applications of Real-time Operating System (RTOS)

Some real-life applications of RTOS are:

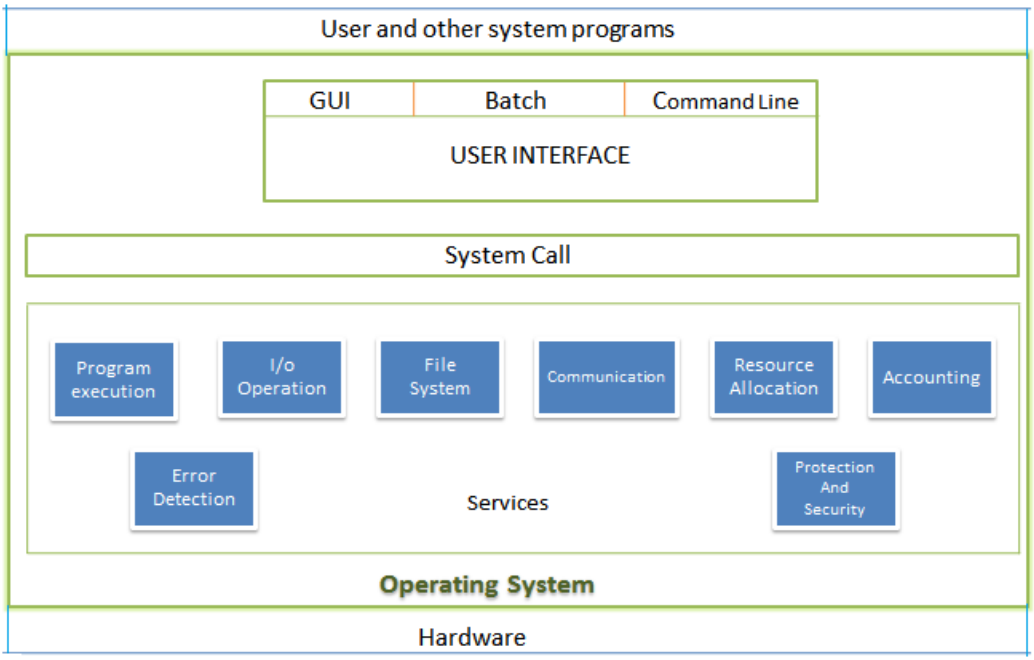
- Systems that provide immediate updating.
- Used in any system that provides up-to-date and minute information on stock prices.
- Defense application systems like RADAR.
- Airlines reservation system.
- Command Control Systems.
- Air traffic control system.
- Networked Multimedia Systems.
- Internet Telephony.
- Heart Pacemaker.
- Anti-lock Brake Systems.

System Components

- An operating system provides the environment within which programs are executed. Internally operating systems vary greatly in their makeup, since they are organized along many different lines.
- The design of a new operating system is a major task. It is important that the goals of the system be well defined before the design begins. These goals form the basis for choices among various algorithms and strategies.
- We can view an operating system from several vantage points. One view focuses on the services that the system provides; another, on the interface that it makes available to users and programmers; a third, on its components and their interconnections.

Operating System Services

- An operating system provides an environment for the execution of programs. It provides certain services to programs and to the users of those programs.
- The specific services provided, of course, differ from one operating system to another, but we can identify common classes.
- These operating-system services are provided for the convenience of the programmer, to make the programming task easier. Figure shows one view of the various operating-system services and how they interrelate.



User interface

- Almost all operating systems have a user interface (UI). This interface can take several forms.
- One is a command-line interface (CLI), which uses text commands and a method for entering them (say, a keyboard for typing in commands in a specified format with specific options).
- Another is a batch interface, in which commands and directives to control those commands are entered into files, and those files are executed. Most commonly, a graphical user interface (GUI) is used. Here, the interface is a window system with a pointing device to direct I/O, choose from menus, and make selections and a keyboard to enter text. Some systems provide two or all of these variations.

Program execution

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

I/O operations

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

File-system manipulation

- The file system is of particular interest. Obviously, programs need to read and write files and directories.

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

Communications

- There are many circumstances in which one process needs to exchange information with another process.
- Such communication may occur between processes that are executing on the same computer or between processes that are executing on different computer systems tied together by a computer network.
- Communications may be implemented via shared memory, in which two or more processes read and write to a shared section of memory, or message passing, in which packets of information in predefined formats are moved between processes by the operating system.

Error detection

- The operating system needs to be detecting and correcting errors constantly.
- Errors may occur in the CPU and memory hardware (such as a memory error or a power failure), in I/O devices (such as a parity error on disk, a connection failure on a network, or lack of power in the printer), an in the user program (such as an arithmetic overflow, an attempt to access an illegal memory location, or a too-great use of CPU time).
- For each type of error, the operating system should take the appropriate action to ensure correct and consistent computing. Sometimes, it has no choice but to halt the system. At other times, it might terminate an error-causing process or return an error code to a process for the process to detect and possibly correct.

Resource Management

- In case of multi-user or multi-tasking environment, resources such as main memory, CPU cycles and files storage are to be allocated to each user or job.
- Following are the major activities of an operating system with respect to resource management
  −The OS manages all kinds of resources using schedulers.
  −CPU scheduling algorithms are used for better utilization of CPU.

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.
- Usage statistics may be a valuable tool for researchers who wish to reconfigure the system to improve computing services.

Protection

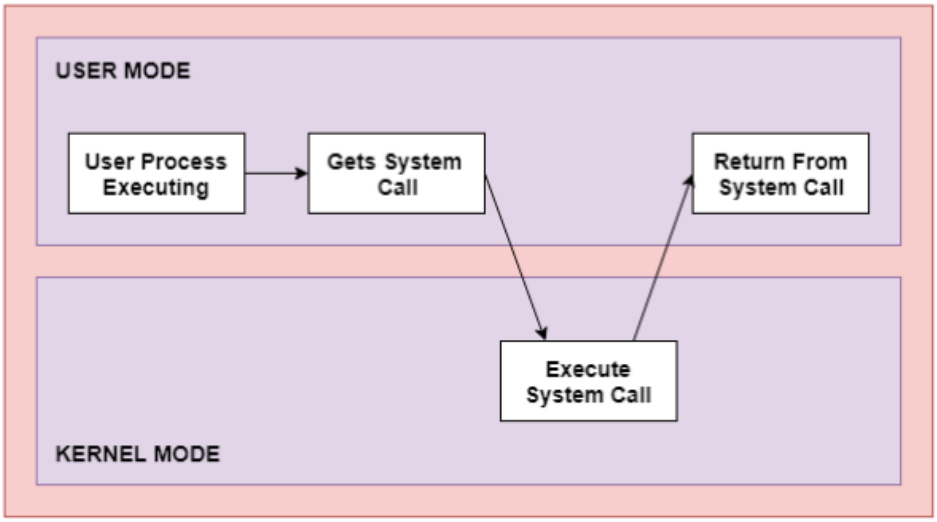
- Protection refers to a mechanism or a way to control the access of programs, processes, or users to the resources defined by a computer system.
- Following are the major activities of an operating system with respect to protection −
  - The OS ensures that all access to system resources is controlled.
  - The OS ensures that external I/O devices are protected from invalid access attempts.
  - The OS provides authentication features for each user by means of passwords.

System Calls

- System calls provide an interface to the services made available by an operating system. These calls are generally available as routines written in C and C++.
- Mostly accessed by programs via a high-level Application Program Interface (API) rather than direct system call use.
- A system call is a way for programs to interact with the operating system. A computer program makes a system call when it makes a request to the operating system's kernel.
- Three most common APIs are Win32 API for Windows, POSIX API for POSIX-based systems (including virtually all versions of UNIX, Linux, and Mac OS X), and Java API for the Java virtual machine (JVM)

How are system calls made

- The interface between a process and an operating system is provided by system calls. In general, system calls are available as assembly language instructions.
- They are also included in the manuals used by the assembly level programmers.
- System calls are usually made when a process in user mode requires access to a resource. Then it requests the kernel to provide the resource via a system call.
- A figure representing the execution of the system call is given as follows



- As can be seen from this diagram, the processes execute normally in the user mode until a system call interrupts this. Then the system call is executed on a priority basis in the kernel mode.
- After the execution of the system call, the control returns to the user mode and execution of user processes can be resumed.
- In general, system calls are required in the following situations −

  - If a file system requires the creation or deletion of files. Reading and writing from files also require a system call.
  - Creation and management of new processes.
  - Network connections also require system calls. This includes sending and receiving packets.
  - Access to a hardware devices such as a printer, scanner etc. requires a system call.

- The system-call interface that serves as the link to system calls made available by the operating system.
- The system-call interface intercepts function calls in the API and invokes the necessary system calls within the operating system.
- Typically, a number is associated with each system call, and the system-call interface maintains a table indexed according to these numbers.
- The system call interface then invokes the intended system call in the operating-system kernel and returns the status of the system call and any return values.
- The caller need know nothing about how the system call is implemented or what it does during execution. Rather it need only obey the API and understand what the operating system will do as a result of the execution of that system call.
- The relationship between an API, the system-call interface, and the operating system is shown in Figure below, which illustrates how the operating system handles a user application invoking the open() system call.

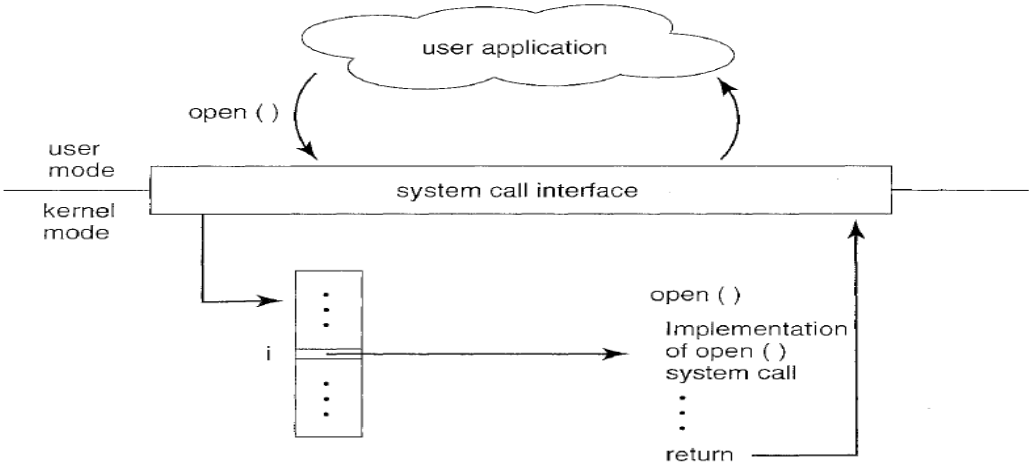


Figure The handling of a user application invoking the `open()` system call.

System Call Parameter Passing

Three general methods used to pass parameters to the operating system.

(1). Simplest: pass the parameters in registers (In some cases, may be more parameters than registers)

(2). 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)

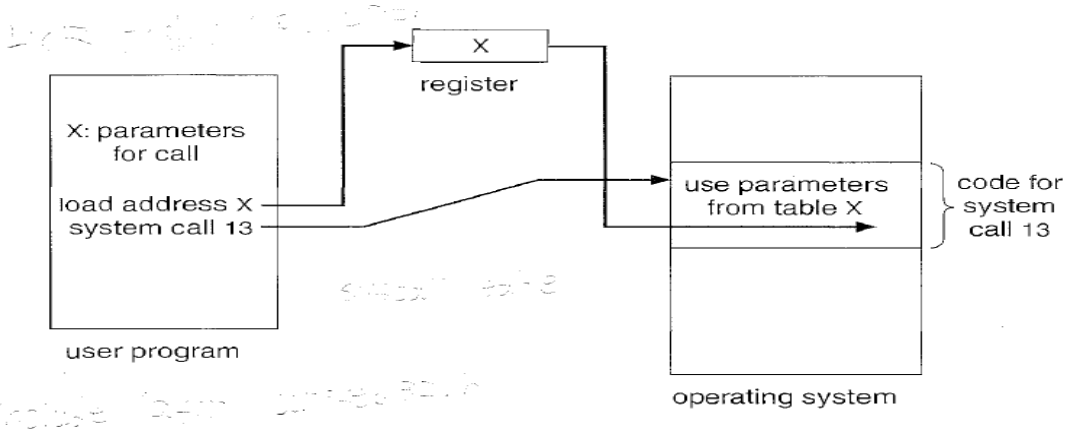


Figure Passing of parameters as a table.

Types of System Calls

- System calls can be grouped into six major categories: process control, file manipulation, device manipulation, information maintenance, communication and protection.
- Figure below summarizes the types of system calls normally provided by an operating system.

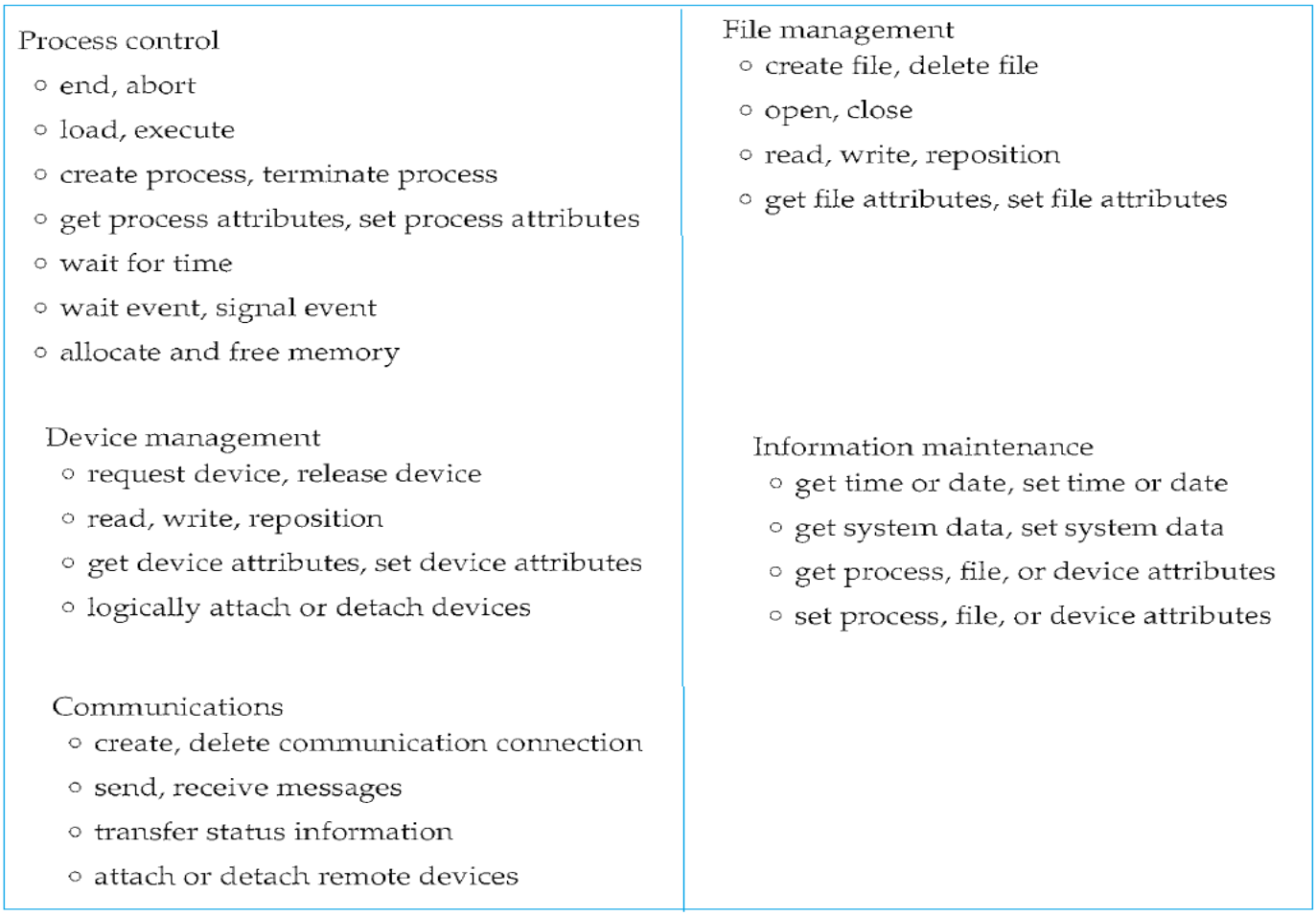


Figure Types of system calls.

Process Control

- These types of system calls deal with process creation, process termination, process allocation, deallocation etc.
- Basically manages all the process that are a part of OS.

File Management

- File management is a system call that is used to handle the files.
- These system calls are responsible for file manipulation such as creating a file, reading a file, writing into a file etc.
- We first need to be able to create and delete files. Either system call requires the name of the file and perhaps some of the file's attributes. Once the file is created, we need to open it and to use it. We may also read, write, or reposition (rewinding or skipping to the end of the file, for example). Finally, we need to close the file, indicating that we are no longer using it.
- In addition, for either files or directories, we need to be able to determine the values of various attributes and perhaps to reset them if necessary. File attributes include the file name, file type, protection codes, accounting information, and so on.

Device Management

- Device management is a system call that is used to deal with devices. Some examples of device management include read, device, write, get device attributes, release device, etc
- A process may need several resources to execute-main memory, disk drives, access to files, and so on. If the resources are available, they can be granted, and control can be returned to the user process. Otherwise, the process will have to wait until sufficient resources are available.
- Some of these devices are physical devices (for example, disk drives), while others can be thought of as abstract or virtual devices (for example, files).

Information Maintenance

- Information maintenance is a system call that is used to maintain information.
- There are some examples of information maintenance, including getting system data, set time or date, get time or date, set system data, etc.
- Many system calls exist simply for the purpose of transferring information between the user program and the operating system. For example, most systems have a system call to return the current time and date.
- Other system calls may return information about the system, such as the number of current users, the version number of the operating system, the amount of free memory or disk space, and so on.

Interprocess Communication (IPC)

- When two or more process are required to communicate, then various IPC mechanism are used by the OS which involves making numerous system calls.
- There are two common models for IPC: The message passing model and the shared-memory model.
- In message passing model, the communicating processes exchange messages with one another to transfer information.
- Messages can be exchanged between the processes either directly or indirectly through a common mailbox. Message passing is useful for exchanging smaller amounts of data, because no conflicts need be avoided.
- In the shared-memory model processes uses the shared memory to exchange messages.
- They can then exchange information by reading and writing data in the shared areas. The forms of the data are determined by the processes and are not under the operating system's control. The processes are also responsible for ensuring that they are not writing to the same location simultaneously.

Protection

- Protection provides a mechanism for controlling access to the resources provided by a computer system.
- Historically, protection was a concern only on multi-programmed computer systems with several users.
- However, with the advent of networking and the Internet, all computer systems, from servers to PDAs, must be concerned with protection.
- Typically, system calls providing protection include set permission and get permission, which manipulate the permission settings of resources such as files and disks.
- The *allow user* and *deny user* system calls specify whether particular users can-or cannot-be allowed access to certain resources.

Some of the examples of all the above types of system calls in Windows and Unix are given as follows –

Process	Windows	Unix
Process Control	CreateProcess() ExitProcess() WaitForSingleObject()	Fork() Exit() Wait()
File Manipulation	CreateFile() ReadFile() WriteFile() CloseHandle()	Open() Read() Write() Close()
Device Management	SetConsoleMode() ReadConsole() WriteConsole()	Ioctl() Read() Write()
Information Maintenance	GetCurrentProcessID() SetTimer() Sleep()	Getpid() Alarm() Sleep()
Communication	CreatePipe() CreateFileMapping() MapViewOfFile()	Pipe() Shmget() Mmap()
Protection	SetFileSecurity() InitializeSecurityDescriptor() SetSecurityDescriptorgroup()	Chmod() Umask() Chown()

System Programs

System programs, also known as system utilities, provide a convenient environment for program-development and execution. Some of them are simply user interfaces to system calls; others are considerably more complex. They can be divided into these categories:

File management

- These programs create, delete, copy, rename, print, dump, list, and generally manipulate files and directories.

Status information

- Some programs simply ask the system for the date, time, amount of available memory or disk space, number of users, or similar status information.
- Others are more complex, providing detailed performance, logging, and debugging information.
- Typically, these programs format and print the output to the terminal or other output devices or files or display it in a window of the GUI.

File modification

- Several text editors may be available to create and modify the content of files stored on disk or other storage devices.
- There may also be special commands to search contents of files or perform transformations of the text.

Programming-language support

- Compilers, assemblers, debuggers, and interpreters for common programming languages (such as C, C++, Java, Visual Basic, and PERL) are often provided to the user with the operating system.

Program loading and execution

- Once a program is assembled or compiled, it must be loaded into memory to be executed.
- The system may provide absolute loaders, relocatable loaders, linkage editors, and overlay loaders.
- Debugging systems for either higher-level languages or machine language are needed as well.

Communications

- These programs provide the mechanism for creating virtual connections among processes, users, and computer systems.
- They allow users to send messages to one another's screens, to browse Web pages, to send electronic-mail messages, to log in remotely, or to transfer files from one machine to another.

In addition to systems programs, most operating systems are supplied with programs that are useful in solving common problems or performing common operations. Such application program include web browsers, word processors, text formatter, spread sheets, database system, compilers, games etc.

Operating System structure

- The operating system structure refers to the way in which the various components of an operating system are organized and interconnected. There are several different approaches to operating system structure, each with its own advantages and disadvantages.
- The operating system can be implemented with the help of various structures. The structure of the OS depends mainly on how the various standard components of the operating system are interconnected and melded into the kernel.

Simple Structure

- It is the simplest Operating System Structure and is not well defined. It can only be used for small and limited systems.
- Many commercial operating systems do not have well-defined structures. Frequently, such systems started as small, simple, and limited systems and then grew beyond their original scope.
- MS-DOS is an example of such a system. It was originally designed and implemented by a few people who had no idea that it would become so popular. It was written to provide the most functionality in the least space, so it was not divided into modules carefully.

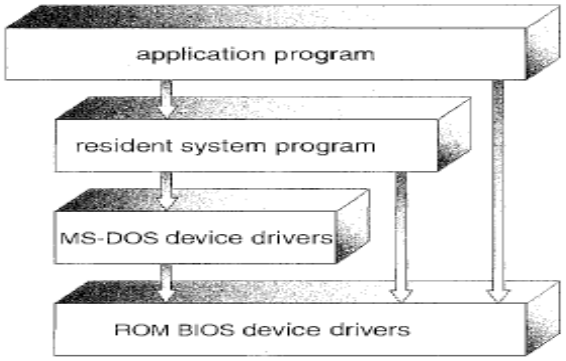


Figure MS-DOS layer structure.

- In MS-DOS, the interfaces and levels of functionality are not well separated. Application programs are able to access the basic I/O routines to write directly to the display and disk drives.
- Such freedom leaves MS-DOS exposed to malicious programs, causing entire system crashes when user programs fail.

Layered Approach

- With proper hardware support, operating systems can be broken into pieces that are smaller and more appropriate those allowed by the original MS-DOS and UNIX system.
- The bottom layer (layer 0) is the Hardware and the highest (layer N) is the user interface. This layering structure is depicted in Figure below.

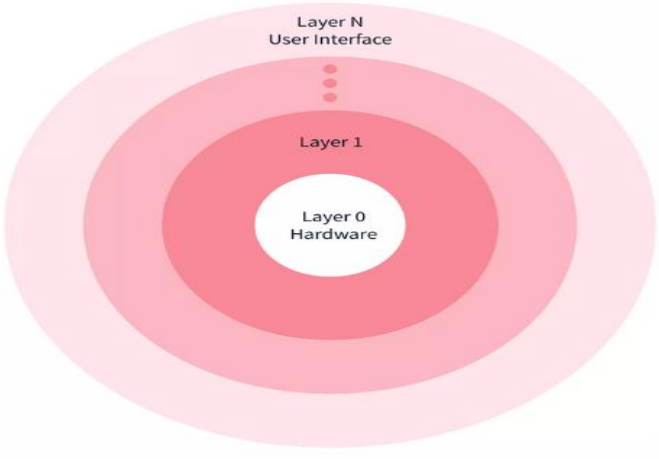


Figure A layered operating system.

- The main advantage of the layered approach is simplicity of construction and debugging. The layers are selected so that each uses functions (operations) and services of only lower-level layers.
- This approach simplifies debugging and .system verification. The first layer can be debugged without any concern for the rest of the system. Once the first layer is debugged, its correct functioning can be assumed while the second layer is debugged, and so on. If an error is found during the debugging of a particular layer, the error must be on that layer, because the layers below it are already debugged.
- 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.
- It is easier to design, maintain, and update the system if it is made in layers.
- The major difficulty with the layered approach involves appropriately defining the various layers. Because a layer can use only lower-level layers, careful planning is necessary.
- A problem with layered implementations is that they tend to be less efficient than other types.

Microkernels

- As UNIX expanded, the kernel became large and difficult to manage. In the mid-1980s, researchers at Carnegie Mellon University developed an operating system called Mach that modularized the kernel using the Microkernels approach.
- This method structures the operating system by removing all nonessential components from the kernel and implementing them as a system & user level program, the result is smaller kernel.

- The main function of the micro kernel is to provide a communication facility between the client program and the various services that are also running in user space. Communication is provided by message passing.
- For example, if the client program wishes to access a file, it must interact with the file server. The client program and service never interact directly. Rather they communicate indirectly by exchanging messages with the microkernel.
- One benefit of the microkernel approach is ease of extending the operating system. All new services are added to user space and consequently do not require modification of the kernel.
- The microkernel also provides more security and reliability, since most services are running as user rather than kernel-processes. If a service fails, the rest of the operating system remains untouched.
- Several contemporary operating systems have used the microkernel approach. *Tru64 UNIX* (formerly Digital UNIX) provides a UNIX interface to the user, but it is implemented with a Mach kernel. The Mac OS *X kernel* (also known as Darwin) is also based on the Mach micro kernel.
- Another example is QNX, a real-time operating system. The QNX microkernel provides services for message passing and process scheduling. It also handles low-level network communication and hardware interrupts.
- Unfortunately, microkernels can suffer from performance decreases due to increased system function overhead. Consider the history of Windows NT. The first release had a layered microkernel organization. Windows NT 4.0 partially redressed the performance problem by moving layers from user space to kernel space and integrating them more closely.

Modules

- The best current methodology for operating-system design involves using object-oriented programming techniques to create a modular kernel.
- The kernel has a set of core components and links in additional services either during boot time or during run time.
- Such a strategy uses dynamically loadable modules and is common in modern implementations of UNIX, such as Solaris, Linux, and Mac OS X. For example, the Solaris operating system structure, shown in Figure below, is organized around a core kernel with seven types of loadable kernel modules:

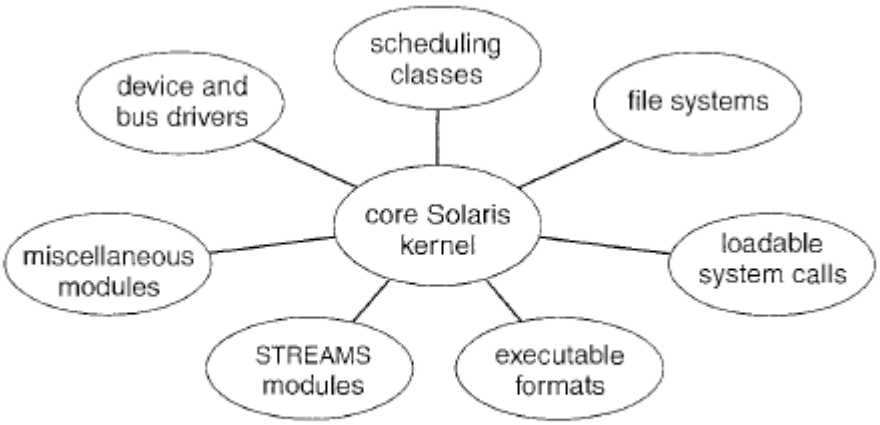


Figure Solaris loadable modules.

- Module design of Solaris OS 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 the kernel, and support for different file systems can be added as loadable modules.
- It is more flexible than a layered system in that any module can call any other module.

Advantages of Modular Structure

- A modular structure is highly modular, meaning that each module is independent of the others. This makes it easier to understand, develop, and maintain the operating system.
- A modular structure is very flexible. New modules can be added easily, and existing modules can be modified or removed without affecting the rest of the operating system.

Disadvantages of Modular Structure

- There can be some performance overhead associated with the communication between modules. This is because modules must communicate with each other through well-defined interfaces.
- A modular structure can be more complex than other types of operating system structures. This is because the modules must be carefully designed to ensure that they interact correctly.

Virtual Machines

- Virtual Machines (VMs) are a form of virtualization technology that allows multiple operating systems to run on a single physical machine simultaneously.
- Each virtual machine acts as an independent, isolated system with its own OS and applications.
- The fundamental idea behind a virtual machine is to abstract the hardware of a single computer (the CPU, memory, disk drives, network interface cards, and so forth) into several different execution environments, thereby creating the illusion that each separate execution environment is running its own private computer.

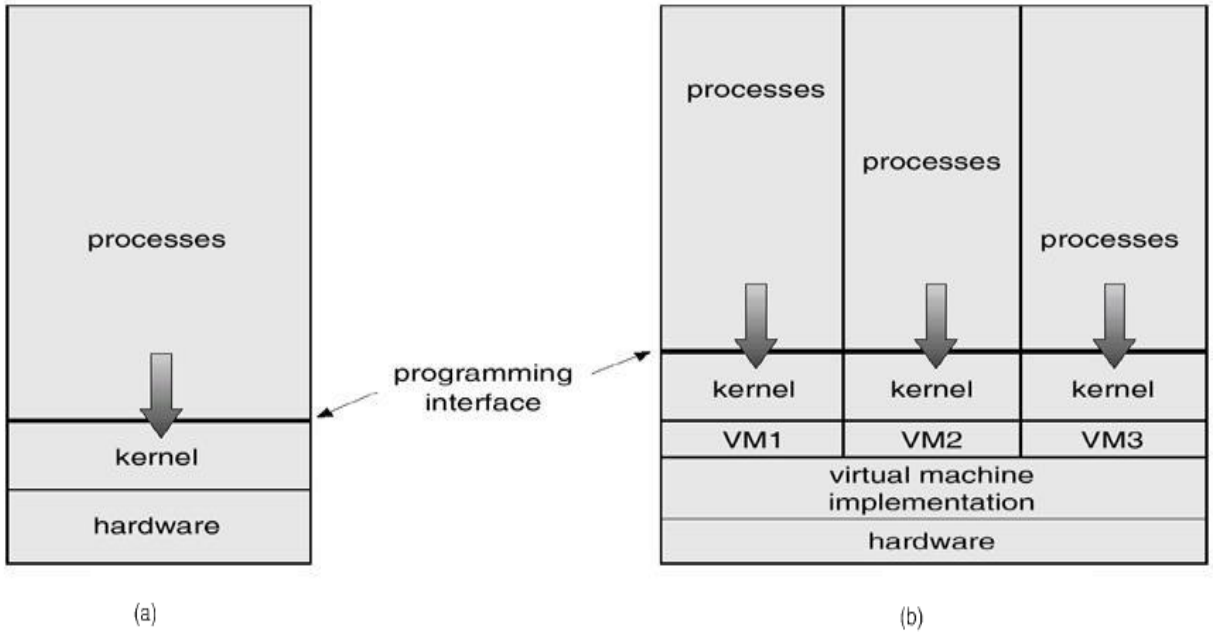


Figure System models. (a) Nonvirtual machine. (b) Virtual machine.

- The single physical machine can run multiple operating systems concurrently, each in its own virtual machine.
- Virtual machines first appeared commercially on IBM mainframes via the VM operating system in 1972

Advantages of Virtual Machines

- Virtual machine able to share the same hardware and runs several different execution environments (that is, different operating systems) concurrently.
- VMs provide a high degree of isolation between guest operating systems. This makes it difficult for malware or other problems in one guest to affect other guests or the host system.
- VMs can be used to create secure environments for running untrusted code. For example, a VM can be used to run a web browser without risking the entire system to malware infection.
- VMs can be easily moved from one physical machine to another. This makes it easy to deploy and manage applications across multiple servers.
- Another advantage of virtual machines for developers is that multiple operating systems can be running on the developer's workstation concurrently. This virtualized workstation allows for rapid porting and testing of programs in varying environments.

Disadvantages of Virtual Machines

- VMs typically have some performance overhead compared to running software directly on the hardware. This is because the VM must emulate the hardware for the guest operating system.
- VMs can be complex to manage. This is because they require additional software to be installed and configured.
- VMs can consume a significant amount of system resources. This is because they must run a guest operating system in addition to the host operating system.

Examples of Virtual Machine

A. VMware

- It is a popular commercial application that abstracts Intel X86 and compatible hardware into isolated virtual machines.
- VMware Workstation runs as an application on a host operating system such as Windows or Linux and allows this host system to concurrently run several different guest operating systems as independent virtual machines.
- The architecture of such a system is shown in Figure below. In this scenario, Linux is running as the host operating system; and FreeBSD, Windows NT, and Windows XP are running as guest operating systems.

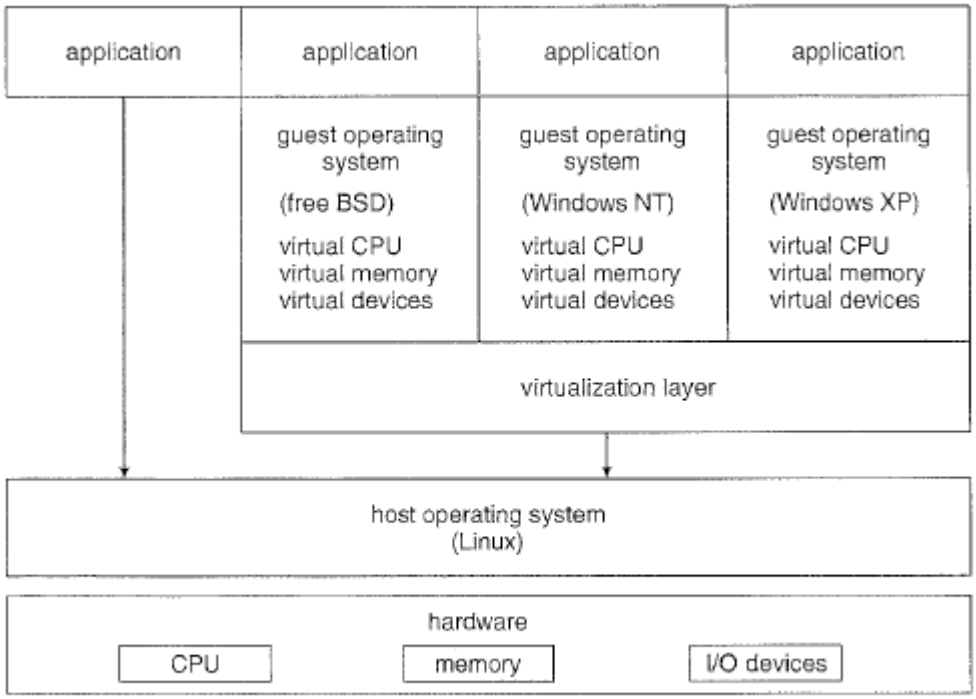


Figure VMware architecture.

- The virtualization layer is the heart of VMware, as it abstracts the physical hardware into isolated virtual machines running as guest operating systems. Each virtual machine has its own virtual CPU, memory, disk drives, network interfaces, and so forth.

B. The Java Virtual Machine

- Java is a popular object-oriented programming language introduced by Sun Microsystems in 1995. In addition to a language specification and a large API library, Java also provides a specification for a Java virtual machine-or JVM.
- For each Java class, the compiler produces an architecture-neutral bytecode output (.class) file that will run on any implementation of the JVM.
- The JVM is a specification for an abstract computer. It consists of a class loader and a Java interpreter that executes the architecture-neutral bytecodes, as diagrammed in Figure below.

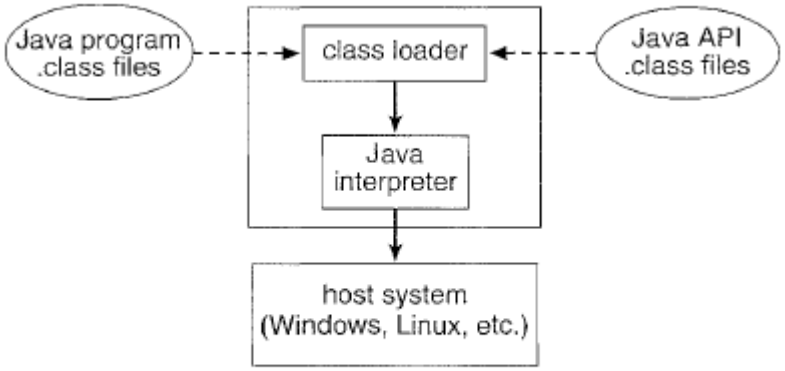


Figure The Java virtual machine.

- The JVM is responsible for loading, verifying, and executing Java bytecode. Bytecode is the intermediate code that is generated by the Java compiler. It is a platform-independent format that can be executed by any JVM.
- The class loader loads the compiled *.class* files from both the Java program and the Java API for execution by the Java interpreter. After a class is loaded, the verifier checks that the *.class* file is valid Java bytecode and does not overflow or underflow the stack.
- It also ensures that the bytecode does not perform pointer arithmetic, which could provide illegal memory access. If the class passes verification, it is run by the Java interpreter. The Java interpreter converts the code into machine code.
- The JVM may be implemented in software on top of a host operating system, such as Windows, Linux, or Mac OS X, or as part of a Web browser.

System Design and Implementation

- The operating system is needed to design and implement because without proper design and implementation any system cannot work properly.
- For any development a proper design and implementation should be necessary so that it can work in good manner and we can easily debug if any failures occur.
- There are different types of approaches that have proved successful to design and implement the operating system given below.

Design goals

- The first problem in designing a system is to define goals and specifications. At the highest level, the design of the system will be affected by *the choice of hardware and the type of system*: batch, time shared, single user, multiuser, distributed, real time, or general purpose.

- Defining the requirements for an operating system divided into two basic groups: user goals and system goals.
- **User Goals**: The operating system should be convenient, easy to use, reliable, safe and fast according to the users. However, these specifications are not very useful as there is no set method to achieve these goals.
- **System Goals:** The operating system should be easy to design, implement and maintain. These are specifications required by those who create, maintain and operate the operating system. But there is not specific method to achieve these goals as well.

Mechanisms and Policies

- One important principle is the separation of policy from mechanism. Mechanism determines how to do something; policies determine what will be done.
- For example we can use the timer to prevent a user program from running too long, the timer construct is a mechanism for ensuring CPU protection, but deciding how long the timer is to be set for a particular user is a policy decision.
- The separation of policy and mechanism is important for flexibility. Policies are likely to change across places or over time. In worst case, each change in policy would require a change in the underlying mechanism.
- For example - If the mechanism and policy are independent, then few changes are required in mechanism if policy changes. If a policy favours I/O intensive processes over CPU intensive processes, then a policy change to preference of CPU intensive processes will not change the mechanism.
- Policy decisions are important for all resource allocation. Whenever it is necessary to decide whether or not to allocate a resource, a policy decision must be made. Whenever the question is how rather than what, it is a mechanism that must be determined.

Implementation

- Once an operating system is designed, it must be implemented.
- Traditionally, operating systems have been written in assembly language. Now, however, they are most commonly written in higher-level languages such as C or C++.
- The Linux and Windows XP operating systems are written mostly in C, although there are some small sections of assembly code for device drivers and for saving and restoring the state of registers.
- The advantages of using a higher-level language for implementing operating systems are the code can be written faster, is more compact, and is easier to understand and debug.
- Operating system is easier to port-to move to some other hardware if it is written in a higher-level language.
- The only possible disadvantages of implementing an operating system in a higher-level language are reduced speed and increased storage requirements.

System Generations

- 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, on CD-ROM or DVD-ROM, or as an "ISO" image. SYSGEN program obtains information concerning the specific configuration of the hardware system.

- 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 will 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 will the boot disk be formatted? How many sections, or "partitions," will it be separated into, and what will go into each partition?
    - 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.

- Once this information is determined, it can be used in several ways. At one extreme, a system administrator can use it to modify a copy of the source code of the operating system.
- The operating system then is completely compiled. Data declarations, initializations, and constants, along with conditional compilation, produce an output-object version of the operating system.

Booting –

The procedure of starting a computer by loading the kernel is known as booting the system. Most computer systems have a small piece of code, stored in ROM, known as the bootstrap program or bootstrap loader. This code is able to locate the kernel, load it into main memory, and start its execution. Some computer systems, such as PCs, use a two-step process in which a simple bootstrap loader fetches a more complex boot program from disk, which in turn loads the kernel.

Process Concept

- A process can be thought of as a program in execution. A process will need certain resources- such as CPU time, memory, files, and 1/0 devices to accomplish its task. These resources are allocated to the process either when it is created or while it is executing.
- A process is the unit of work in most systems. Systems consist of a collection of processes: *Operating-system processes* execute system code, and *user processes* execute user code. All these processes may execute concurrently.
- 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. In addition, a process generally includes the process stack, which contains temporary data (such as method parameters, return addresses, and local variables), and a data section, which contains global variables.
- Although two processes may be associated with the same program, they are nevertheless considered two separate execution sequences. For instance, several users may be running different copies of the mail program, or the same user may invoke many copies of the Web browser program. Each of these is a separate process; and although the text sections are equivalent, the data, heap, and stack sections vary.

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 (*such as an I/0 completion or reception of a signal).*
  - **Ready**- The process is waiting to be assigned to a processor.
  - **Terminated**- The process has finished execution.
- The state diagram corresponding to these states is presented in Figure

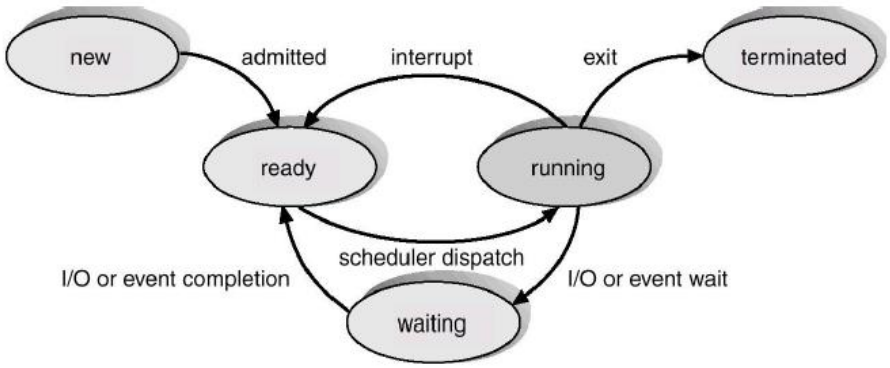


Figure Diagram of Process State

- It is important to realize that only one process can be running on any processor at any instant. Many processes may be ready and waiting state.

Process Control Block (PCB)

- Each process is represented in the operating system by a process control block also called a task control block.
- The PCB shown in figure below:

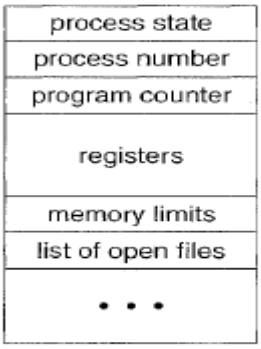


Figure Process control block (PCB).

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

- **Process state**-The state may be new, ready running, waiting, halted, 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 shown in figure below.
- **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.

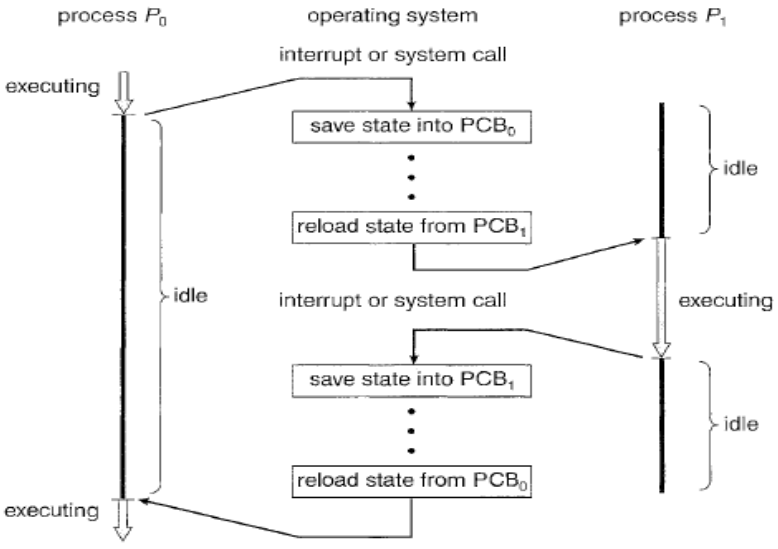


Figure Diagram showing CPU switch from process to process.

Process Scheduling

- The objective of multiprogramming is to have some process running at all times, to maximize CPU utilization. The objective of time sharing is to switch the CPU among processes so frequently that users can interact with each program while it is running.
- To meet these objectives, the process scheduler selects an available process (possibly from a set of several available processes) for program execution on the CPU.

Scheduling Queues

The Operating system includes following queues:

- **Job Queue:** As processes enter the system, they are put into a job queue, which consists of all processes in the system**.** This queue consists of all processes in the system; those processes are entered to the system as new processes.

- **Ready Queue:** This queue consists of the processes that are residing in main memory and are ready and waiting to execute by CPU. This queue is generally stored as a linked list. A ready-queue header contains pointers to the first and final PCBs in the list. Each PCB includes a pointer field that points to the next PCB in the ready queue.

- **Device Queue:** This queue consists of the processes that are waiting for a particular I/O device. Each device has its own device queue.

A common representation of process scheduling is a queueing diagram, such as that in shown in Figure below:

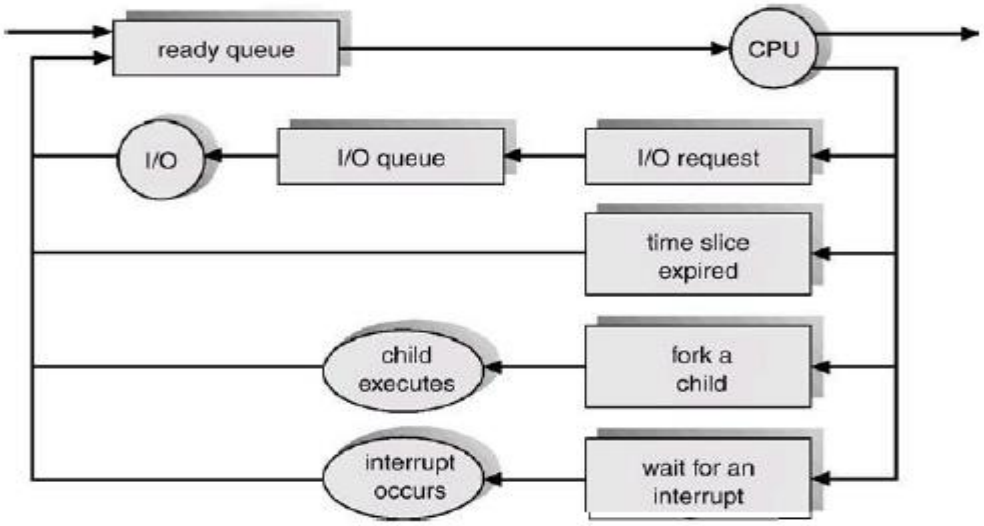


Figure Queueing-diagram representation of process scheduling.

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/0 request and then be placed in an I/0 queue.
- The process could create a new subprocess and wait for the subprocess'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.

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

Schedulers

- A process migrates among the various scheduling queues throughout its lifetime. The operating system must select, for scheduling purposes, processes from these queues in some fashion. The selection process is carried out by the appropriate scheduler.
- A scheduler is a decision maker that selects the processes from one scheduling queue to another or allocates CPU for execution.

The Operating System has three types of scheduler:

1. Long-term scheduler or Job scheduler

- In a batch system, more processes are submitted than can be executed immediately. These processes are spooled to a mass-storage device (typically a disk), where they are kept for later execution. The long-term scheduler, or job scheduler, selects processes from this pool and loads them into memory for execution.
- The long-term scheduler or job scheduler selects processes from discs and loads them into main memory for execution. It executes much less frequently.
- The long-term scheduler controls the degree of multiprogramming (the number of processes in memory). If the degree of multiprogramming is stable, then the average rate of process creation must be equal to the average departure rate of processes leaving the system.
- Because of the longer interval between executions, the long-term scheduler can afford to take more time to select a process for execution.

2. Short-term scheduler or CPU scheduler

- The short-term scheduler or CPU scheduler selects a process from among the processes that are ready to execute and allocates the CPU.
- The short-term scheduler must select a new process for the CPU frequently. A process may execute for only a few milliseconds before waiting for an I/O request.

3. Medium-term scheduler

- Some operating systems, such as time-sharing systems, may introduce an additional, intermediate level of scheduling. This medium-term scheduler is diagrammed in Figure below.
- The key idea behind a medium-term scheduler is that sometimes it can be advantageous to remove processes from memory (and from active contention for the CPU) and thus reduce the degree of multiprogramming. Later, the process can be reintroduced into memory, and its execution can be continued where it left off. This scheme is called swapping.
- The process is swapped out, and is later swapped in, by the medium-term scheduler. Swapping may be necessary to improve the process mix or because a change in memory requirements has overcommitted available memory, requiring memory to be freed.

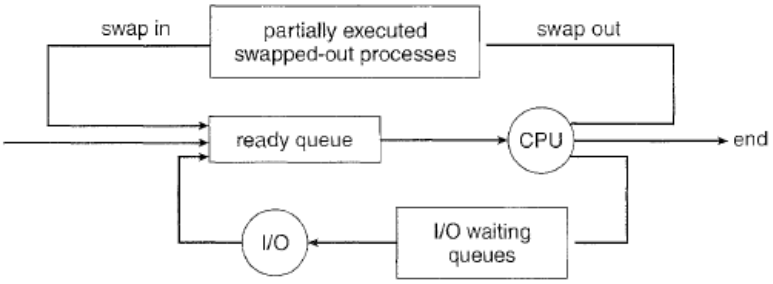


Figure Addition of medium-term scheduling to the queueing diagram.

Context Switch

- Interrupts cause the operating system to change a CPU from its current task and to run a kernel routine.

- When an interrupt occurs, the system needs to save the current context of the process running on the CPU so that it can restore that context when its processing is done, essentially suspending the process and then resuming it.
- 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.
- Context-switch time is pure overhead, because the system does no useful work while switching. Its 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.
- Context-switch times are highly dependent on hardware support.

Operation on process

The processes in most systems can execute concurrently, and they may be created and deleted dynamically. Thus, these systems must provide a mechanism for process creation and termination.

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 (including UNIX and the Windows family of operating systems) identify processes according to a unique process identifier (or pid), which is typically an integer number.
- In general, a process will need certain resources (CPU time, memory, files, I/0 devices) to accomplish its task. When a process creates a subprocess, that subprocess may be able to obtain its resources directly from the operating system, or it may be constrained to a subset of the resources of the parent process.
- The parent may have to partition its resources among its children, or it may be able to share some resources (such as memory or files) among several of its children.
- In addition to the various physical and logical resources that a process obtains when it is created, initialization data (input) may be passed along by the parent process to the child process.

- When a process creates a new process, two possibilities exist in terms of execution:
  - The parent continues to execute concurrently with its children.
  - 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:
  - The child process is a duplicate of the parent process (it has the same program and data as the parent).
  - The child process has a new program loaded into it.

- In UNIX new process is created by the *fork()* system call. The new process consists of a copy of the address space of the original process.
- Figure below illustrates Process creation using *fork()* system call in the UNIX

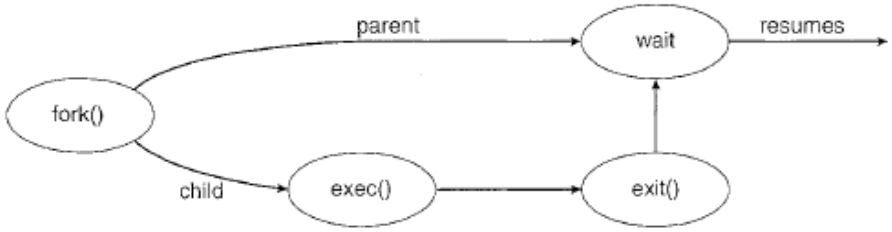


Figure Process creation using `fork()` system call.

- Both processes (*the parent and the child*) continue execution at the instruction after the *fork ()* with one difference: the return code for the *fork()* is zero for the new (child) process, whereas the (nonzero) process identifier of the child is returned to the parent.
- The *exec()* system call is used after a *fork()* system call by one of the two processes to replace the process's memory space with a new program.
- The *exec()* system call loads a binary file into memory (destroying the memory image of the program containing the *exec()* system call) and starts its execution.
- The parent can then create more children; or, if it has nothing else to do while the child runs, it can issue *await()* system call to move itself off the ready queue until the termination of the child.

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.
- At that point, the process may return a status value (typically an integer) to its parent process (via the *wait()* system call). All the resources of the process-including physical and virtual memory, open files, and I/0 buffers are deallocated by the operating system.
- Termination can occur in other circumstances as well. A process can cause the termination of another process via an appropriate system call (for example, *TerminateProcess ()* in Win32). Usually, such a system call can be invoked only by the parent of the process that is to be terminated. Otherwise, users could arbitrarily kill each other's jobs.

- A parent may terminate the execution of one of its children for a variety of reasons, such as these:
    - The child has exceeded its usage of some of the resources that it has been allocated. (To determine whether this has occurred, the parent must have a mechanism to inspect the state of its children.)
    - The task assigned to the child is no longer required.
    - The parent is exiting, and the operating system does not allow a child to continue if its parent terminates.

- Some systems, including VMS, do not allow a child to exist if its parent has terminated. 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.

Inter-process Communication

- Processes executing concurrently in the operating system may be either independent processes or cooperating processes.
- A process is independent if it cannot affect or be affected by the other processes executing in the system. Any process that does not share data with any other process is independent.
- A process is cooperating if it can affect or be affected by the other processes executing in the system. Clearly, any process that shares data with other processes is a cooperating process.

There are several reasons for providing an environment that allows process cooperation:

Information sharing

- Since several users may be interested in the same piece of information (for instance, a shared file), we must provide an environment to allow concurrent access to such information.

Computation speedup

- If we want a particular task to run faster, we must break it into subtasks, each of which will be executing in parallel with the others.

Modularity

- We may want to construct the system in a modular fashion, dividing the system functions into separate processes or threads.

Convenience

- Even an individual user may work on many tasks at the same time. For instance, a user may be editing, printing, and compiling in parallel.

- Cooperating processes require an interprocess communication (IPC) mechanism that will allow them to exchange data and information.
- There are two fundamental models of interprocess communication: Shared memory and Message passing.

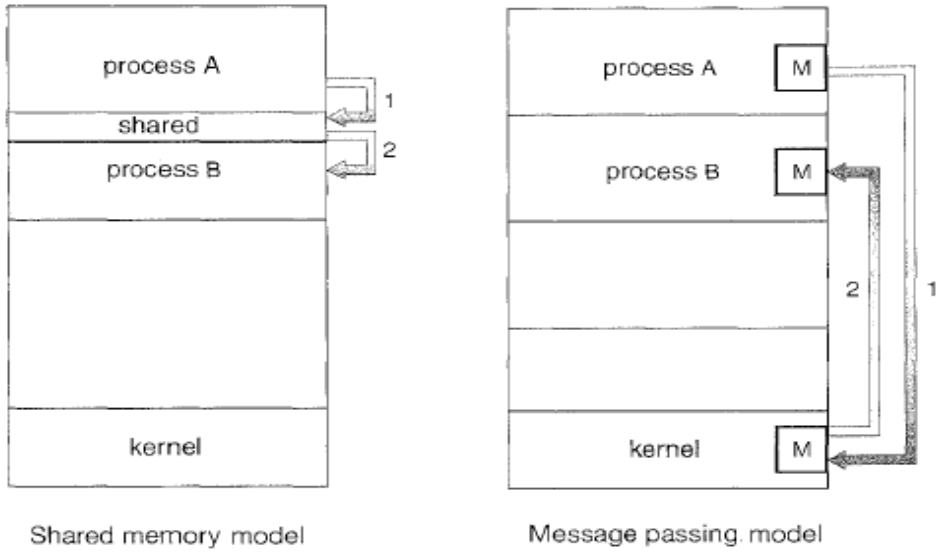


Figure Communications models.

Shared memory model

- In the shared-memory model, a region of memory that is shared by cooperating processes is established. Processes can then exchange information by reading and writing data to the shared region.
- Shared memory allows maximum speed and convenience of communication. Shared memory is faster than message passing.
- In shared memory systems, system calls are required only to establish shared-memory regions. Once shared memory is established, all accesses are treated as routine memory accesses, and no assistance from the kernel is required.
- Typically, a shared-memory region resides in the address space of the process creating the shared memory segment. Other processes that wish to communicate using this shared memory segment must attach it to their address space.
- The operating system tries to prevent one process from accessing another process's memory. Shared memory requires that two or more processes agree to remove this restriction. They can then exchange information by reading and writing data in the shared areas.
- The processes are also responsible for ensuring that they are not writing to the same location simultaneously.

Message passing model

- In the message passing model, communication takes place by means of messages exchanged between the cooperating processes.

- Message passing is useful for exchanging smaller amounts of data. Message passing is also easier to implement than is shared memory for intercomputer communication.
- As message passing systems are typically implemented using system calls and thus require the more time-consuming task of kernel intervention.
- Message passing provides a mechanism to allow processes to communicate and to synchronize their actions without sharing the same address space and is particularly useful in a distributed environment, where the communicating processes may reside on different computers connected by a network.
- A message-passing facility provides at least two operations: *send(message*) and *receive(message).* Messages sent by a process can be of either fixed or variable size.
- If only fixed-sized messages can be sent, the system-level implementation is straightforward. This restriction, however, makes the task of programming more difficult. Conversely, variable-sized messages require a more complex system-level implementation, but the programming task becomes simpler.
- If processes P and Q want to communicate, they must send messages to and receive messages from each other; a communication link must exist between them. Here are several methods for logically implementing a link and the *send 0/receive()* operations:
    - Direct or indirect communication
    - Synchronous or asynchronous communication
    - Automatic or explicit buffering
- Under *direct communication,* each process that wants to communicate must explicitly name the recipient or sender of the communication. In this scheme, the *send() and receive()* primitives are defined as:
    - *send(P, message)* -Send a message to process P.
    - *receive (Q, message)*-Receive a message from process Q.
- With *indirect communication,* the messages are sent to and received from mailboxes, or ports. A mailbox can be viewed abstractly as an object into which messages can be placed by processes and from which messages can be removed. Each mailbox has unique identification.
    - *send (A, message)* -Send a message to mailbox A.
    - *receive (A, message)*-Receive a message from mailbox A.
- *Synchronization:* Communication between processes takes place through calls to send() and receive () primitives. There are different design options for implementing each primitive. Message passing may be either blocking or nonblocking also known as synchronous and asynchronous.
    - Blocking send: The sending process is blocked until the message is received by the receiving process or by the mailbox.
    - Nonblocking send: The sending process sends the message and resumes operation.
    - Blocking receive: The receiver blocks until a message is available.
    - Nonblocking receive: The receiver retrieves either a valid message or a null.
- *Buffering:* Whether communication is direct or indirect, messages exchanged by communicating processes reside in a temporary queue.

Threads

- A thread is a basic unit of CPU utilization; 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, data section, and other operating-system resources, such as open files and signals.
- A traditional (or heavy weight) process has a single thread of control. If a process has multiple threads of control, it can perform more than one task at a time. Figure below illustrates the difference between a traditional process and a process.

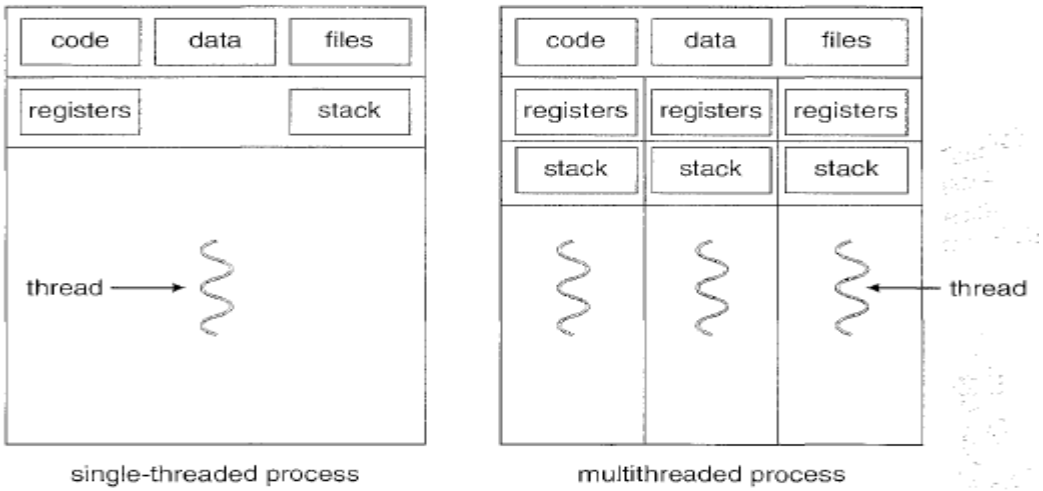


Figure Single-threaded and multithreaded processes.

- Many software packages that run on modern desktop PCs are multithreaded. An application typically is implemented as a separate process with several threads of control. For example a Web browser might have one thread display images or text while another thread retrieves data from the network.
- In certain situations, a single application may be required to perform several similar tasks. For example, a Web server accepts client requests for Web pages, images, sound, and so forth. A busy Web server may have several clients concurrently accessing it. If the Web server ran as a traditional single threaded process, it would be able to service only one client at a time, and a client might have to wait a very long time for its request to be serviced.
- If the Web-server process is multithreaded, the server will create a separate thread that listens for client requests. When a request is made, rather than creating another process, the server will create a new thread to service the request and resume listening for additional requests.

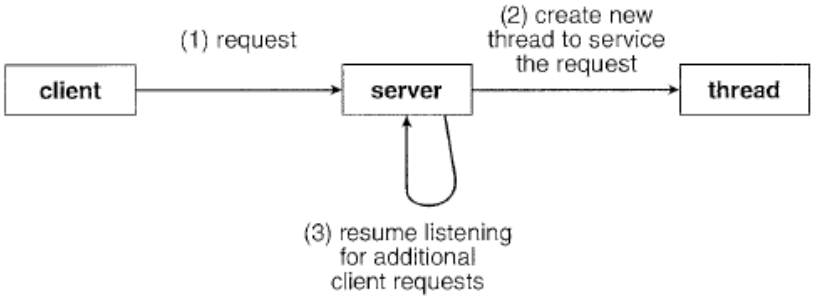


Figure Multithreaded server architecture.

- Most operating systems kernels are now multithreaded; several threads operate in the kernel, and each thread performs a specific task, such as managing devices or interrupt handling.

Benefits of multithreaded programming

The benefits of multithreaded programming can be broken down into four major categories:

Responsiveness

- Multithreading an interactive application may allow a program to continue running even if part of it is blocked or is performing a lengthy operation, thereby increasing responsiveness to the user.
- For instance a multithreaded Web browser could allow user interaction in one thread while an image was being loaded in another thread.

Resource sharing

- Processes may only share resources through techniques such as shared memory or message passing.

- Threads share the memory and the resources of the process to which they belong by default. The benefit of sharing code and data is that it allows an application to have several different threads of activity within the same address space.

Economy

- Allocating memory and resources for process creation is costly. Because threads share the resources of the process to which they belong, it is more economical to create and context-switch threads.

Scalability

- The benefits of multithreading can be greatly increased in a multiprocessor architecture, where threads may be running in parallel on different processors.
- A single-threaded process can only run on one processor, regardless how many are available. Multithreading on a multi-CPU machine increases parallelism.

Multicore Programming

- A recent trend in system design has been to place multiple computing cores on a single chip, where each core appears as a separate processor to the operating system.
- Multithreaded programming provides a mechanism for more efficient use of multiple cores and improved concurrency.
- Consider an application with four threads. On a system with a single computing core, concurrency merely means that the execution of the threads will be interleaved over time (Figure A), as the processing core is capable of executing only one thread at a time.

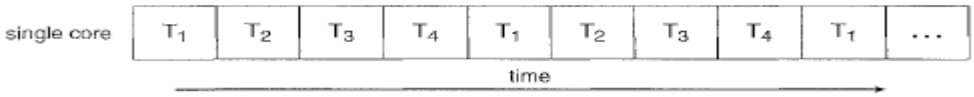


Figure A Concurrent execution on a single-core system.

- On a system with multiple cores, however, concurrency means that the threads can run in parallel, as the system can assign a separate thread to each core (Figure B).

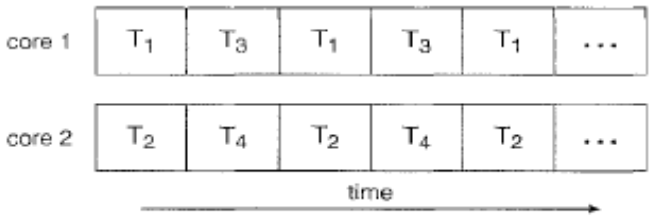


Figure B. Parallel execution on a multicore system.

- Designers of operating systems must write scheduling algorithms that use multiple processing cores to allow the parallel execution.

Multithreading model

- Multithreading is a process of multiple threads executes at same time. Support for 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 without kernel support. Operating system does not directly support user threads. Instead, threads are managed by a user-level thread library, which is part of the application
- Whereas kernel threads are supported and managed directly by the operating system.
- Many operating systems support kernel thread and user thread in a combined way. Example of such system is Solaris. Multi-threading model are of following types.

Many to One Model

- The many-to-one model maps many user-level threads to one kernel thread.

- Thread management is done by the thread library in user space, so it is efficient; but the entire process will block if a thread makes a blocking system call.
- Also, because only one thread can access the kernel at a time, multiple threads are unable to rum in parallel on multiprocessors

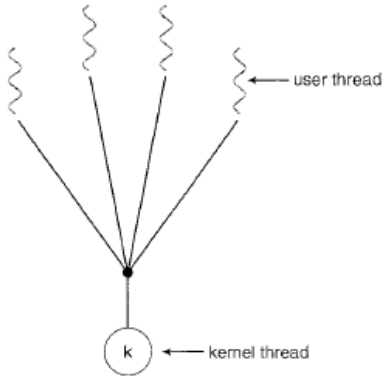


Figure Many-to-one model.

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 also allows multiple threads to run in parallel on multiprocessors.
- The only drawback to this model is that creating a user thread requires creating the corresponding kernel thread.

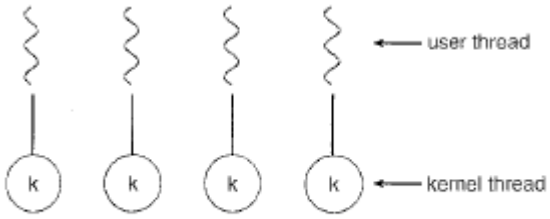


Figure One-to-one model.

Many to Many Model

- The many-to-many model multiplexes many user-level threads to a smaller or equal number of kernel threads.
- The number of kernel threads may be specific to either a particular application or a particular machine.
- Advantage of this model is if a user thread is blocked we can schedule others user thread to other kernel thread. Thus, System doesn't block if a particular thread is blocked.

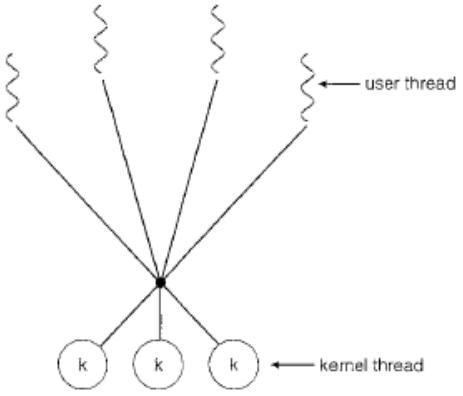


Figure Many-to-many model.

- One popular variation on the many-to-many model still multiplexes many user-level threads to a smaller or equal number of kernel threads but also allows a user-level thread to be bound to a kernel thread.
- This variation, sometimes referred to as the two-level model, is supported by operating systems such as IRlX, HP-UX, and Tru64 UNIX.

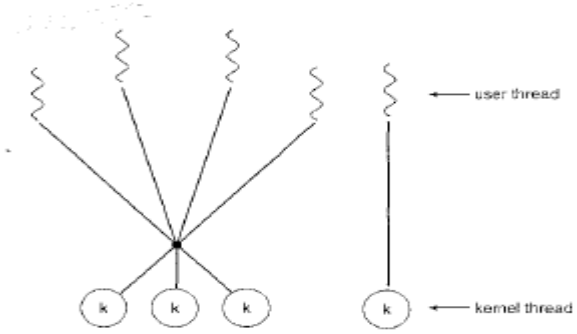


Figure Two-level model.

Scheduling

Basic concept

- The objective of multiprogramming is to have some process rum1ing at all times, to maximize CPU utilization.
- CPU scheduling is the basis of multiprogrammed operating systems. By switching the CPU among processes, the operating system can make the computer more productive.
- A process is executed until it must wait, typically for the completion of some I/O request. In a simple computer system, the CPU then just sits idle. All this waiting time is wasted; no useful work is accomplished. With multiprogramming, we try to use this time productively.
- Several processes are kept in memory at one time. When one process has to wait, the operating system takes the CPU away from that process and gives the CPU to another process.
- Scheduling of this kind is a fundamental operating-system function. Almost all computer resources are scheduled before use. The CPU is, of course, one of the primary computer resources. Thus, its scheduling is central to operating-system design.
- Whenever the CPU becomes idle, the operating system must select one of the processes in the ready queue to be executed. The selection process is carried out by the short-term scheduler (or CPU scheduler). The scheduler selects a process from the processes in memory that are ready to execute and allocates the CPU to that process.

Preemptive Scheduling

- CPU-scheduling decisions may take place under the following four circumstances:
    1) When a process switches from the running state to the waiting state (for example, as the result of an I/0 request or an invocation of wait for the termination of one of the child processes)
    2) When a process switches from the running state to the ready state (for example, when an interrupt occurs)
    3) When a process switches from the waiting state to the ready state (for example, at completion of I/0)
    4) When a process terminates

- When scheduling takes place only under circumstances 1 and 4, we say that the scheduling scheme is nonpreemptive or cooperative; otherwise, it is preemptive.
- Under nonpreemptive scheduling, once the CPU has been allocated to a process, the process keeps the CPU until it releases the CPU either by terminating or by switching to the waiting state.

- *Preemptive Scheduling*: Preemptive scheduling is used when a process switches from running state to ready state or from the waiting state to the ready state.
- *Non-Preemptive Scheduling*: Non-Preemptive scheduling is used when a process terminates , or when a process switches from running state to waiting state.

Scheduling criteria

- Different CPU-scheduling algorithms have different properties. In choosing which algorithm to use in a particular situation, we must consider the properties of the various algorithms.
- Many criteria have been suggested for comparing CPU-scheduling algorithms.The criteria include the following:

CPU utilization.

- We want to keep the CPU as busy as possible. Conceptually, CPU utilization can range from 0 to 100 percent.
- In a real system, it should range from 40 percent (for a lightly loaded system) to 90 percent (for a heavily used system).

Throughput

- If the CPU is busy executing processes, then work is being done. One measure of work is the number of processes that are completed per time unit, called throughput.
- For long processes, this rate may be one process per hour; for short transactions, it may be ten processes per second.

Turnaround time

- From the point of view of a particular process, the important criterion is how long it takes to execute that process. The interval from the time of submission of a process to the time of completion is the turnaround time.
- Turnaround time is the sum of the periods spent waiting to get into memory, waiting in the ready queue, executing on the CPU, and doing I/O.

Waiting time

- The CPU-scheduling algorithm does not affect the amount of time during which a process executes or does I/O; it affects only the amount of time that a process spends waiting in the ready queue.
- Waiting time is the sum of the periods spent waiting in the ready queue.

Response time

- In an interactive system, turnaround time may not be the best criterion. Often, a process can produce some output fairly early and can continue computing new results while previous results are being output to the user.
- Thus, another measure is the time from the submission of a request until the first response is produced. This measure, called response time, is the timee it takes to start responding, not the time it takes to output the response.

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

Scheduling Algorithms

- CPU scheduling deals with the problem of deciding which of the processes in the ready queue is to be allocated the CPU. There are many different CPU-scheduling algorithms, several of them given below:

First-Come, First-Served Scheduling

- The simplest CPU-scheduling algorithm is the first-come, first-served (FCFS) scheduling algorithm.
- FCFS considered to be the simplest of all operating system scheduling algorithms. First come first serve scheduling algorithm states that the process that requests the CPU first is allocated the CPU first and is implemented by using FIFO queue.
- When a process enters the ready queue, its PCB is linked onto the tail of the queue. When the CPU is free, it is allocated to the process at the head of the queue. The running process is then removed from the queue. The code for FCFS scheduling is simple to write and understand.
- On the negative side, the average waiting time under the FCFS policy is often quite long. Consider the following set of processes that arrive at time 0, with the length of the CPU burst given in milliseconds:

Process	Burst Time
P_1	24
P_2	3
P_3	3

- If the processes arrive in the order P1, P2, P3, and are served in FCFS order, we get the result shown in the following Gantt chart, which is a bar chart that illustrates a particular schedule, including the start and finish times of each of the participating processes:

- The waiting time is 0 milliseconds for process P1, 24 milliseconds for process P2 , and 27 milliseconds for process P3 . Thus, the average waiting time is (0 + 24 + 27)/3 = 17 milliseconds.
- If the processes arrive in the order P2, P3 , P1, however, the results will be as shown in the following Gantt chart:

- The average waiting time is now (6 + 0 + 3)/3 = 3 milliseconds. This reduction is substantial. Thus, the average waiting time under an FCFS policy is generally not minimal and may vary substantially if the processes CPU burst times vary greatly.
- The FCFS scheduling algorithm is nonpreemptive. Once the CPU has been allocated to a process, that process keeps the CPU until it releases the CPU, either by terminating or by requesting I/0. The FCFS algorithm is thus particularly troublesome for time-sharing systems, where it is important that each user get a share of the CPU at regular intervals

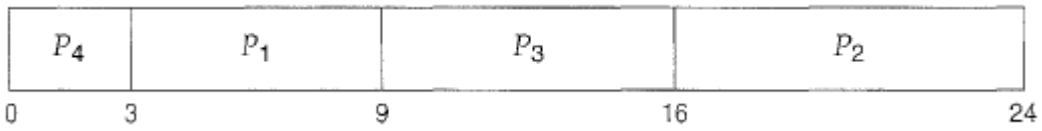
Shortest-Job-First Scheduling

- **Shortest job first (SJF)** is a scheduling process that selects the waiting process with the smallest execution time to execute next.
- This algorithm associates with each process the length of the process's next CPU burst. When the 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 the same, FCFS scheduling is used to break the tie.

- The more appropriate term for this scheduling method would be the shortest-next-CPU-burst algorithm, because scheduling depends on the length of the next CPU burst of a process, rather than its total length.
- As an example of SJF scheduling, consider the following set of processes, with the length of the CPU burst given in milliseconds:

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

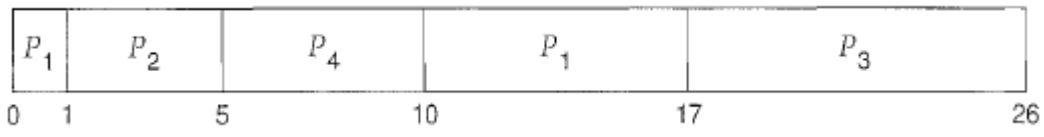
- Using SJF scheduling, we would schedule these processes according to the following Gantt chart:



- The waiting time is 3 milliseconds for process P1, 16 milliseconds for process P2, 9 milliseconds for process P3, and 0 milliseconds for process P4. Thus, the average waiting time is (3 + 16 + 9 + 0) I 4 = 7 milliseconds. By comparison, if we were using the FCFS scheduling scheme, the average waiting time would be 10.25 milliseconds.
- The SJF algorithm is optimal, it cannot be implemented at the level of short-term CPU scheduling. With short-term scheduling, there is no way to know the length of the next CPU burst.
- We may not know the length of the next CPU burst, but we may be able to predict its value. By computing an approximation of the length of the next CPU burst, we can pick the process with the shortest predicted CPU burst. The next CPU burst is generally predicted as an exponential average of the measured lengths of previous CPU bursts.
- The SJF algorithm can be either preemptive or nonpreemptive. The choice arises when a new process arrives at the ready queue while a 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 preemptive SJF algorithm will preempt the currently executing process, whereas a nonpreemptive SJF algorithm will allow the currently running process to finish its CPU burst. Preemptive SJF scheduling is sometimes called *shortest-remaining-time-first scheduling*.
- As an example, consider the following four processes, with the length of the CPU burst given in milliseconds:

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

- If the processes arrive at the ready queue at the times shown and need the indicated burst times, then the resulting preemptive SJF schedule is as depicted in the following Gantt chart:



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

process P2 (4 milliseconds), so process P1 is preempted, and process P2 is scheduled. The average waiting time for this example is

[(10- 1) + (1 - 1) + (17- 2) +(5-3)]/ 4 = 26/4 = 6.5 milliseconds.

Q: Find average waiting time using premptive SJF

Process	Burst Time	Arrival Time
P1	6	2
P2	2	5
P3	8	1
P4	3	0
P5	4	4

Gantt Chart: Premptive SJF

P4	P1	P5	P2	P5	P1	P3
0	3	4	5	7	10	15	23

Waiting Time

P1	(3-2)+(10-4) = 7
P2	(5-5) = 0
P3	(15-1) = 14
P4	0
P5	(4-4)+(7-5) = 2

Average Waiting Time

= (7+0+14+0+2)/5
= 4.6

Priority Scheduling

- In the priority scheduling a priority is associated with each process, and the CPU is allocated to the process with the highest priority. Equal-priority processes are scheduled in FCFS order.
- Priorities are generally indicated by some fixed range of numbers, such as 0 to 7 or 0 to 4,095. Some systems use low numbers to represent low priority; others use low numbers for high priority. We assume that low numbers represent high priority.
- As an example, consider the following set of processes, assumed to have arrived at time 0 in the order P1, P2, · · ·, Ps, with the length of the CPU burst given in milliseconds:

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

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

- The average waiting time for this example

[(6-0) + (0-0) + (16- 0) +(18-0) +(1-0)]/ 5 = 41/5 = 8.2 milliseconds.

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

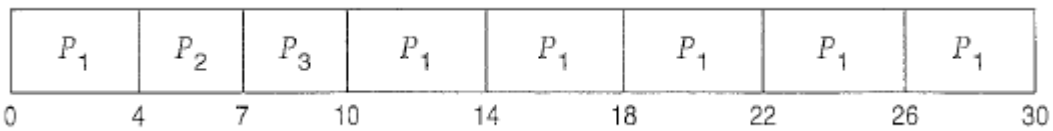
- A preemptive priority scheduling algorithm will preempt the CPU if the priority of the newly arrived process is higher than the priority of the currently running process. A nonpreemptive priority scheduling algorithm will simply put the new process at the head of the ready queue.
- A major problem with priority scheduling algorithms is indefinite blocking, or starvation. A process that is ready to run but waiting for the CPU can be considered blocked. A priority scheduling algorithm can leave some low priority processes waiting indefinitely. In a heavily loaded computer system, a steady stream of higher-priority processes can prevent a low-priority process from ever getting the CPU.
- A solution to the problem of indefinite blockage 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, if priorities range from 127 (low) to 0 (high), we could increase the priority of a waiting process by 1 every 15 minutes.

Round-Robin Scheduling

- The round-robin (RR) scheduling algorithm is designed especially for timesharing systems. It is similar to FCFS scheduling, but preemption is added to enable the system to switch between processes.
- A small unit of time, called a time quantum (or time slice), is defined. A time quantum is generally from 10 to 100 milliseconds in length.
- The ready queue is treated as a circular queue. The CPU scheduler goes around the ready queue, allocating the CPU to each process for a time interval of up to 1 time quantum.
- To implement RR scheduling, we keep the ready queue as a FIFO queue o£ processes. New processes are added to the tail of the ready queue. The CPU scheduler picks the first process from the ready queue, sets a timer to interrupt after 1 time quantum, and dispatches the process.
- One of two things will then happen. First 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.
- Otherwise, if the CPU burst of the currently running process is longer than 1 time quantum, the timer will go off and will cause an interrupt to the operating system. A context switch will be executed, and the process will be put at the tail o£ the ready queue. The CPU scheduler will then select the next process in the ready queue.
- The average waiting time under the RR policy is often long. Consider the following set of processes that arrive at time 0, with the length of the CPU burst given in milliseconds:

Process	Burst Time
P_1	24
P_2	3
P_3	3

- If we use a time quantum of 4 milliseconds, then process P1 gets the first 4 milliseconds. Since it requires another 20 milliseconds, it is preempted after the first time quantum, and the CPU is given to the next process in the queue, process P2 . Process P2 does not need 4 milliseconds, so it quits before its time quantum expires. The CPU is then given to the next process, process P3. Once each process has received 1 time quantum, the CPU is returned to process P1 for an additional time quantum. The resulting RR schedule is as follows:



- Let's calculate the average waiting time for the above schedule.

P1 waits for 6 millisconds (10- 4), P2 waits for 4 millisconds, and P3 waits for 7 millisconds. *Thus, the average waiting time is 17/3 = 5.66 milliseconds.*

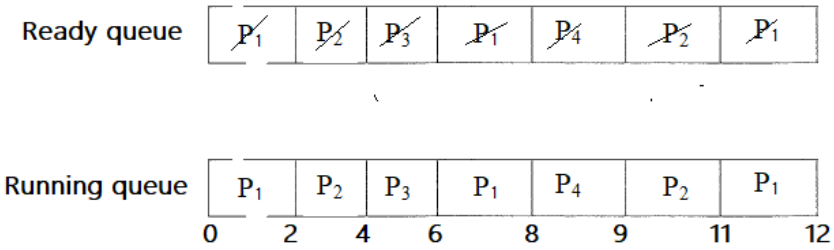
- If there are n. processes in the ready queue and the time quantum is q, then each process gets 1 In of the CPU time in chunks of at most q time units. Each process must wait no longer than (11 - 1) x q time units until its next time quantum. For example, with five processes and a time quantum of 20 milliseconds, each process will get up to 20 milliseconds every 100 milliseconds.

Example: Consider the following data with burst time given in milliseconds

Processes	AT	BT
P_1	0	5
P_2	1	4
P_3	2	2
P_4	4	1

1) Draw Gantt charts for the execution of these processes using RR (quantum=2) scheduling.
2) What is turnaround time and waiting time of each process.

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



Turnaround Time=Completion Time(CT) – AT
Waiting Time(WT)= Turnaround Time – BT
Response Time (RT)= CPU first time-AT

Processes	AT	BT	CT	Turnaround Time	WT	RT
P_1	0	5	12	12	7	0
P_2	1	4	11	10	6	1
P_3	2	2	6	4	2	2
P_4	4	1	9	5	4	4

Multilevel Queue Scheduling

- Another class of scheduling algorithms has been created for situations in which processes are easily classified into different groups. For example, a common division is made between foreground (interactive) processes and background (batch) processes.
- These two types of processes have different response-time requirements and so may have different scheduling needs. In addition, foreground processes may have priority (externally defined) over background processes.

- A multilevel queue scheduling algorithm partitions the ready queue into several separate queues.
- The processes are permanently assigned to one queue, generally based on some property of the process, such as memory size, process priority, or process type.
- Each queue has its own scheduling algorithm. For example, separate queues might be used for foreground and background processes. The foreground queue might be scheduled by an RR algorithm, while the background queue is scheduled by an FCFS algorithm.

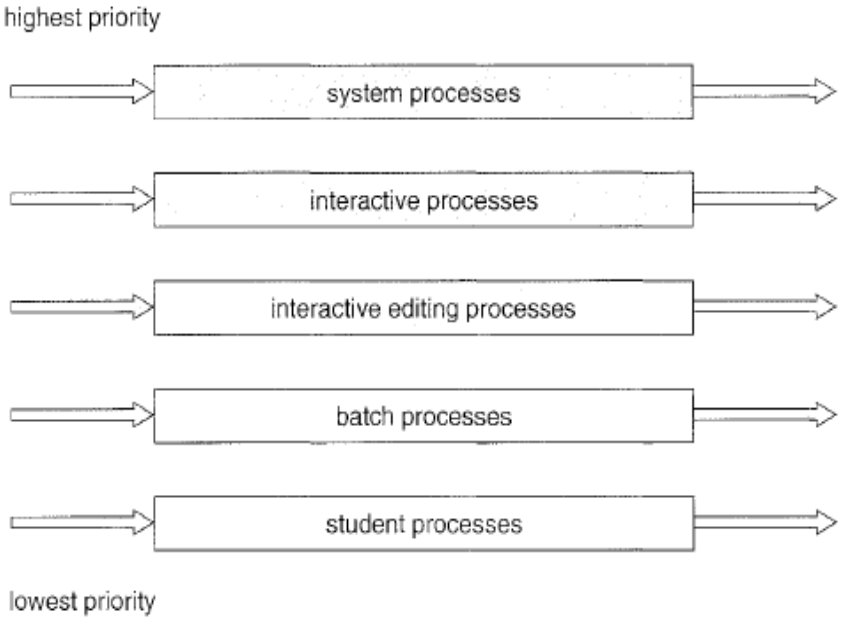


Figure Multilevel queue scheduling.

- In addition, there must be scheduling among the queues, which is commonly implemented as fixed-priority preemptive scheduling. For example, the foreground queue may have absolute priority over the background queue.
- Each queue has absolute priority over lower-priority queues. No process in the batch queue, for example, could run unless the queues for system processes, interactive processes, and interactive editing processes were all empty. If an interactive editing process entered the ready queue while a batch process was running, the batch process would be preempted.

Multilevel Feedback Queue Scheduling

- Normally, when the multilevel queue scheduling algorithm is used, processes are permanently assigned to a queue when they enter the system. If there are separate queues for foreground and background processes, for example, processes do not move from one queue to the other, since processes do not change their foreground or background nature. This setup has the advantage of low scheduling overhead, but it is inflexible.
- The multilevel feedback queue scheduling algorithm, in contrast, allows a process to move between queues. The idea is to separate processes according to the characteristics of their CPU bursts. If a process uses too much CPU time, it will be moved to a lower-priority queue. This scheme leaves I/O-bound and interactive processes in the higher-priority queues. In addition, a process that waits too long in a lower-priority queue may be moved to a higher-priority queue. This form of aging prevents starvation.
- For example, consider a multilevel feedback queue scheduler with three queues, numbered from 0 to 2 shown in Figure below.

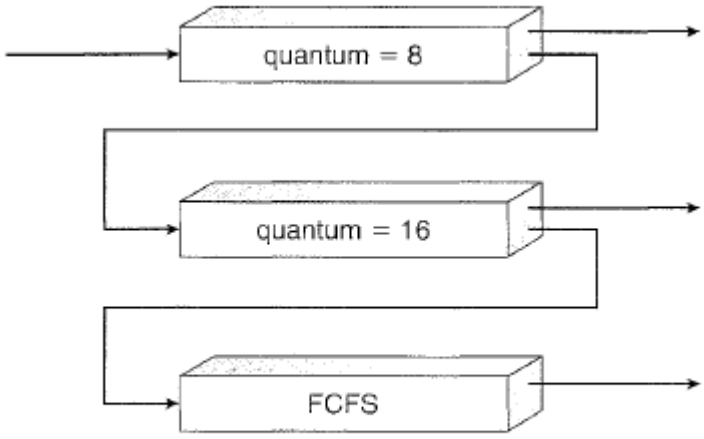


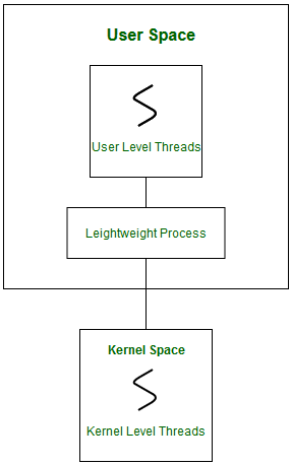
Figure Multilevel feedback queues.

- The scheduler first executes all processes in queue 0. Only when queue 0 is empty will it execute processes in queue 1. Similarly, processes in queue 2 will only be executed if queues 0 and 1 are empty. A process that arrives for queue 1 will preempt a process in queue 2. A process in queue 1 will in turn be preempted by a process arriving for queue 0.
- A process entering the ready queue is put in queue 0. A process in queue 0 is given a time quantum of 8 milliseconds. If it does not filcish within this time, it is moved to the tail of queue 1. If queue 0 is empty, the process at the head of queue 1 is given a quantum of 16 milliseconds. If it does not complete, it is preempted and is put into queue 2. Processes in queue 2 are run on an FCFS basis but are run only when queues 0 and 1 are empty.
- This scheduling algorithm gives highest priority to any process with a CPU burst of 8 milliseconds or less. Such a process will quickly get the CPU, finish its CPU burst, and go off to its next I/0 burst. Processes that need more than 8 but less than 24 milliseconds are also served quickly, although with lower priority than shorter processes. Long processes automatically sink to queue 2 and are served in FCFS order with any CPU cycles left over from queues 0 and 1.
- In general, a multilevel feedback queue scheduler is defined by the following parameters:
    - The number of queues
    - The scheduling algorithm for each queue
    - The method used to determine when to upgrade a process to a higher priority queue
    - The method used to determine when to demote a process to a lower priority queue
    - The method used to determine which queue a process will enter when that process needs service.

Thread Scheduling

- A thread is a single sequential flow of execution of tasks of a process so it is also known as thread of execution or thread of control. There is a way of thread execution inside the process of any operating system. Apart from this, there can be more than one thread inside a process. Each thread of the same process makes use of a separate program counter and a stack of activation records and control blocks. Thread is often referred to as a lightweight process.
- The process can be split down into so many threads. For example, in a browser, many tabs can be viewed as threads. Threads can share the common data, they do not need to use Inter- Process communication. Context switching is faster when working with threads. It takes less time to terminate a thread than a process.
- There are two types of threads-*user-level* and *kernel-level* threads. On operating systems that support them, it is kernel-level threads-not processes-that are being scheduled by the operating system. User-level threads are managed by a thread library, and the kernel is unaware of them.
- To run on a CPU, user-level threads must ultimately be mapped to an associated kernel-level thread, although this mapping may be indirect and may use a lightweight process (LWP).

- Light-weight process is threads in the user space that acts as an interface for the user-level thread to access the physical CPU resources. Thread library schedules which thread of a process to run on which LWP and how long. The number of LWPs created by the thread library depends on the type of application.



- We explore scheduling issues involving user-level and kernel-level threads and offer specific examples of scheduling for Pthreads.

Contention Scope

- The word contention here refers to the competition or fight among the User level threads to access the kernel resources.

Process-contention scope (PCS)

- On systems implementing the many-to-one and many-to-many models, the thread library schedules user-level threads to run on an available LWP, a scheme known as *process-contention scope (PCS)*, since competition for the CPU takes place among threads belonging to the same process.
- The contention takes place among threads within a same process. The thread library schedules the high-prioritized PCS thread to access the resources via available LWPs.
- PCS is done according to priority-the scheduler selects the runnable thread with the highest priority to run.
- User-level thread priorities are set by the programmer and are not adjusted by the thread library, although some thread libraries may allow the programmer to change the priority of a thread.

System Contention Scope (SCS)

- When we say the thread library schedules user threads onto available LWPs, we do not mean that the thread is actually running on a CPU; this would require the operating system to schedule the kernel thread onto a physical CPU. To decide which kernel thread to schedule onto a CPU, the kernel uses system-contention scope (SCS).
- The contention takes place among all threads in the system. In this case, every SCS thread is associated to each LWP by the thread library and are scheduled by the system scheduler to access the kernel resources.
- Systems using the one-to-one model , such as Windows XP, Solaris, and Linux, schedule threads using only SCS.

Pthread Scheduling

- In LINUX and UNIX operating systems, the POSIX Pthread library provides a function *Pthread_attr_setscope* to define the type of contention scope for a thread during its creation.
- The POSIX Pthread API that allows specifying either PCS or SCS during thread creation.

```
int Pthread_attr_setscope(pthread_attr_t *attr, int scope)
```

- The first parameter denotes to which thread within the process the scope is defined.
  The second parameter defines the scope of contention for the thread pointed.

- Pthreads identifies the following contention scope values:

PTHREAD_SCOPE_PROCESS	schedules threads using PCS scheduling.
PTHREAD_SCOPE_SYSTEM	schedules threads using SCS scheduling.

Multiple-Processor Scheduling

- In multiple-processor scheduling **multiple CPU's** are available and hence **Load Sharing** becomes possible. However multiple processor scheduling is more **complex** as compared to single processor scheduling.
- In multiple processor scheduling there are cases when the processors are identical i.e. HOMOGENEOUS, in terms of their functionality; we can use any processor available to run any process in the queue.
- Multiple-processor scheduling is important because it enables a computer system to perform multiple tasks simultaneously, which can greatly improve overall system performance and efficiency.

Approaches to Multiple-Processor Scheduling

- One approach is when all the scheduling decisions and I/O processing are handled by a single processor which is called the **Master Server** and the other processors executes only the **user code**. This is simple and reduces the need of data sharing. This entire scenario is called **Asymmetric Multiprocessing**.
- A second approach uses **symmetric multiprocessing** (SMP), where each processor is self-scheduling. All processes may be in a common ready queue, or each processor may have its own private queue of ready processes. The scheduling proceeds further by having the scheduler for each processor examine the ready queue and select a process to execute.

Processor Affinity

- Processor Affinity means a processes has an **affinity** for the processor on which it is currently running.
- When a process runs on a specific processor there are certain effects on the cache memory. The data most recently accessed by the process populate the cache for the processor and as a result successive memory access by the process are often satisfied in the cache memory.
- Now if the process migrates to another processor, the contents of the cache memory must be invalidated for the first processor and the cache for the second processor must be repopulated.
- Because of the high cost of invalidating and repopulating caches, most of the SMP(symmetric multiprocessing) systems try to avoid migration of processes from one processor to another and try to keep a process running on the same processor. This is known as **PROCESSOR AFFINITY**.
- There are two types of processor affinity:
  1. Soft Affinity – When an operating system has a policy of attempting to keep a process running on the same processor but not guaranteeing it will do so, this situation is called soft affinity.
  2. Hard Affinity – Some systems -such as Linux -also provide system calls that support hard affinity, thereby allowing a process to specify that it is not to migrate to other processors.

Load Balancing

- Load Balancing is the **phenomena** which keeps the **workload** evenly **distributed** across all processors in an symmetric multiprocessing system.
- Load balancing is necessary only on systems where each processor has its own private queue of process which are eligible to execute.

- On SMP(symmetric multiprocessing), it is important to keep the workload balanced among all processors to fully utilize the benefits of having more than one processor else one or more processor will sit idle while other processors have high workloads along with lists of processors awaiting the CPU.
- There are two general approaches to load balancing :
    - **Push Migration –** In push migration a task routinely checks the load on each processor and if it finds an imbalance then it evenly distributes load on each processors by moving the processes from overloaded to idle or less busy processors.
    - **Pull Migration –** Pull Migration occurs when an idle processor pulls a waiting task from a busy processor for its execution.

Multicore Processors –

- In multicore processors **multiple processor** cores are places on the same physical chip. Each core has a register set to maintain its architectural state and thus appears to the operating system as a separate physical processor.
- SMP systems that use multicore processors are faster and consume less power than systems in which each processor has its own physical chip.
- However multicore processors may complicate the scheduling problems. When processor accesses memory then it spends a significant amount of time waiting for the data to become available. This situation is called MEMORY STALL. It occurs for various reasons such as cache miss, which is accessing the data that is not in the cache memory.
- In such cases the processor can spend up to fifty percent of its time waiting for data to become available from the memory. To solve this problem recent hardware designs have implemented multithreaded processor cores in which two or more hardware threads are assigned to each core. Therefore if one thread stalls while waiting for the memory, core can switch to another thread.
- There are two ways to multithread a processor :

  1. Coarse-Grained Multithreading **–** In coarse grained multithreading a thread executes on a processor until a long latency event such as a memory stall occurs, because of the delay caused by the long latency event, the processor must switch to another thread to begin execution. The cost of switching between threads is high as the instruction pipeline must be terminated before the other thread can begin execution on the processor core. Once this new thread begins execution it begins filling the pipeline with its instructions.

  2. Fine-Grained Multithreading **–** This multithreading switches between threads at a much finer level mainly at the boundary of an instruction cycle. The architectural design of fine grained systems include logic for thread switching and as a result the cost of switching between threads is small.

Virtualization and Scheduling

- A system with virtualization, even a single-CPU system, frequently acts like a multiprocessor system.
- The virtualization software presents one or more virtual CPUs to each of the virtual machines rum1.ing on the system and then schedules the use of the physical CPUs among the virtual machines.
- In general, though, most virtualized environments have one host operating system and many guest operating systems. The host operating system creates and manages the virtual machines, and each virtual machine has a guest operating system installed and applications running within that guest. Each guest operating system may be fine-tuned for specific use cases, applications, and users, including time sharing or even real-time operation.

Scheduling Algorithms evaluation

- There are many scheduling algorithms, each with its own parameters. As a result, selecting an algorithm can be difficult. The first problem is defining the criteria to be used in selecting an algorithm.
- Criteria are often defined in terms of CPU utilization, response time, or throughput. To select an algorithm, we must first define the relative importance of these elements. Our criteria may include several measures, such as:
    - Maximizing CPU utilization under the constraint that the maximum response time is 1 second
    - Maximizing throughput such that turnaround time is (on average) linearly proportional to total execution time
- Once the selection criteria have been defined, we next describe the various evaluation methods we can use.

Deterministic Modeling

- One major class of evaluation methods is analytic evaluation. Analytic evaluation uses the given algorithm and the system workload to produce a formula or number that evaluates the performance of the algorithm for that workload.
- Deterministic modeling is one type of analytic evaluation. This method takes a particular predetermined workload and defines the performance of each algorithm for that workload.
- For example, assume that we have the workload shown below. All five processes arrive at time 0, in the order given, with the length of the CPU burst given in milliseconds:

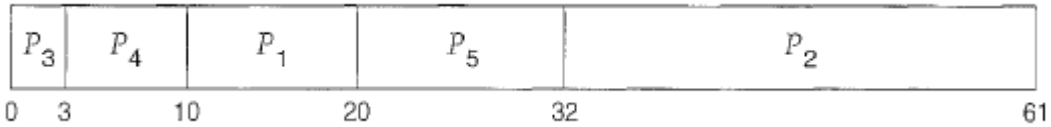
Process	Burst Time
P_1	10
P_2	29
P_3	3
P_4	7
P_5	12

Consider the FCFS, SJF, and RR (quantum = 10 milliseconds) scheduling algorithms for this set of processes. Which algorithm would give the minimum average waiting time? For the **FCFS algorithm**, we would execute the processes as

The waiting time is 0 milliseconds for process P1, 10 milliseconds for process P2 , 39 milliseconds for process P3, 42 milliseconds for process P4, and 49 milliseconds for process P5 . Thus, the average waiting time is

(0 + 10 + 39 + 42 + 49)/5 = ***28 milliseconds***.

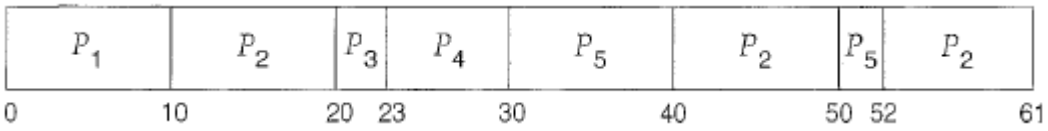
With **nonpreemptive SJF scheduling**, we execute the processes as:



The waiting time is 10 milliseconds for process P11 32 milliseconds for process P2, 0 milliseconds for process P3, 3 milliseconds for process P4 , and 20 milliseconds for process P5 . Thus, the average waiting time is

(10 + 32 + 0+ 3 + 20) I 5 = ***13 milliseconds.***

With the **RR algorithm,** we execute the processes as



The waiting time is 0 milliseconds for process P1, 32 milliseconds for process P2, 20 milliseconds for process P3, 23 milliseconds for process P4 , and 40 milliseconds for process P5 . Thus, the average waiting time is

(0 + 32 + 20+ 23 + 40)/5 = ***23 milliseconds.***

- We see that, in this case, the average waiting time obtained with the SJF policy is less than half that obtained with FCFS scheduling; the RR algorithm gives us an intermediate value.
- Deterministic modeling is simple and fast. It gives us exact numbers, allowing us to compare the algorithms.

Queueing Models

- Queueing analysis can be useful in comparing scheduling algorithms.
- The computer system is described as a network of servers. Each server has a queue of waiting processes. The CPU is a server with its ready queue, as is the I/0 system with its device queues. Knowing arrival rates and service rates, we can compute utilization, average queue length, average wait time, and so on. This area of study is called queueing-network analysis.
- As an example, let n be the average queue length (excluding the process being serviced), let W be the average waiting time in the queue, and let ƛ be the average arrival rate for new processes in the queue (such as three processes per second). We expect that during the time W that a process waits, ƛ x W new processes will arrive in the queue. If the system is in a steady state, then the number of processes leaving the queue must be equal to the number of processes that arrive. Thus,

  n = ƛ x W.

  This equation, known as Little's formula, is particularly useful because it is valid for any scheduling algorithm and arrival distribution.

Simulations

- To get a more accurate evaluation of scheduling algorithms, we can use simulations.
- Running simulations involves programming a model of the computer system. Software data structures represent the major components of the system. The simulator has a variable representing a clock; as this variable's value is increased, the simulator modifies the system state to reflect the activities of the devices, the processes, and the scheduler. As the simulation executes, statistics that indicate algorithm performance are gathered and printed.
- The data to drive the simulation can be generated in several ways. The most common method uses a random-number generator that is programmed to generate processes, CPU burst times, arrivals, departures, and so on, according to probability distributions.
- Simulations can be expensive, often requiring hours of computer time. A more detailed simulation provides more accurate results, but it also takes more computer time.

Implementation

- Even a simulation is of limited accuracy. The only completely accurate way to evaluate a scheduling algorithm is to code it up, put it in the operating system, and see how it works. This approach puts the actual algorithm in the real system for evaluation under real operating conditions.
- The major difficulty with this approach is the high cost. The expense is incurred not only in coding the algorithm and modifying the operating system to support it (along with its required data structures) but also in the reaction of the users to a constantly changing operating system.