Unit-1

**Introduction: C**omputer System organization & Architecture, Operating System Structure &Operations, Process, memory and Storage Managements, Protection and Security, Distributed and Special-Purpose Systems, Computing Environments.

**System Structures**: Operating-System Services, User Operating System Interface, System calls, Types of System Calls, System Programs, Operating-System Structure, Virtual Machines, Operating – System Generation, System Boot.

**Process Concept:** Overview, Process Scheduling, Operations on Processes, Inter-process communication,

Examples of IPC Systems, Communication in Client/Server Systems.

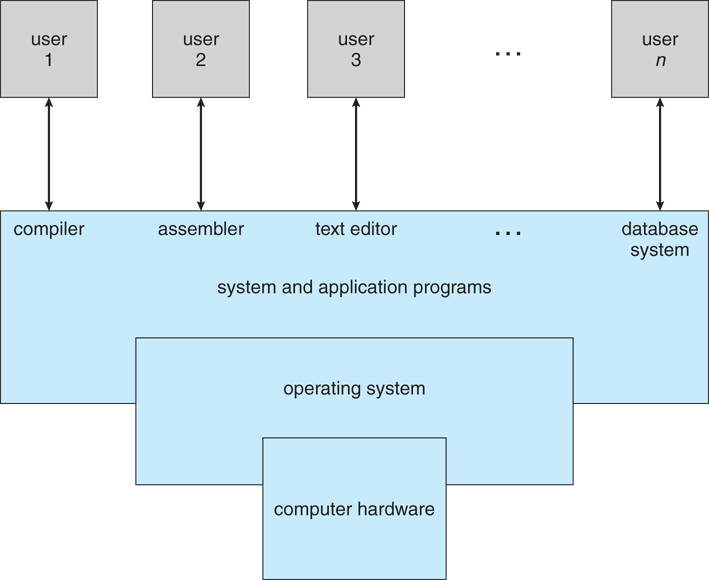
**Multithreaded Programming**: Overview, Multithreading Models, Thread Libraries, Threading Issues, Operating-System Examples.

**Chapter 1**

**Introduction**

Operating System: An operating system is a program that manages the computer hardware. It also provides a basis for application programs and acts as an intermediary between the computer user and the computer hardware. Mainframe operating systems are designed primarily to optimize utilization of hardware. Personal computer (PC) operating systems support complex games, business applications, and everything in between. Operating systems for handheld computers are designed to provide an environment in which a user can easily interface with the computer to execute programs.

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



The hardware—the central processing unit (CPU), the memory, and the input/output (I/O) devices—provides the basic computing resources for the system. The application programs—such as word processors, spreadsheets, compilers, and web browsers—define the ways in which these resources are used to solve users' computing problems. The operating system controls and coordinates the use of the hardware among the various application programs for the various users. The operating system provides the means for proper use of these resources in the operation of the computer system.

Operating system role in two view points: that of the user and that of the system.

*The user's view* of the computer varies according to the interface being used. Most computer users sit in front of a PC, consisting of a monitor, keyboard, mouse, and system unit. Such a system is designed for one user to monopolize its resources. The goal is to maximize the work (or play) that the user is performing. In this case, the operating system is designed mostly for ease of use, with some attention paid to performance and none paid to resource utilization—how various hardware and software resources are shared.

*In System view* the operating system is the program most intimately involved with the hardware. Operating system is a resource allocator. A computer system has many resources that may be required to solve a problem: CPU time, memory space, file-storage space, I/O devices, and so on. The operating system acts as the manager of these resources. Facing numerous and possibly conflicting requests for resources, the operating system must decide how to allocate them to specific programs and users so that it can operate the computer system efficiently and fairly. As we have seen, resource allocation is especially important where many users access the same mainframe or minicomputer.

A slightly different view of an operating system emphasizes the need to control the various I/O devices and user programs. An operating system is a control program. A control program manages the execution of user programs to prevent errors and improper use of the computer. It is especially concerned with the operation and control of I/O devices.

Operating systems exist because they offer a reasonable way to solve the problem of creating a usable computing system. The fundamental goal of computer systems is to execute user programs and to make solving user problems easier. Toward this goal, computer hardware is constructed. The common functions of controlling and allocating resources are then brought together into one piece of software: the operating system.

**Operating system components**

1.process management

2.memory management

3.secondary storage management

4.file management

5.I/O system

6.networking

7.command interpreter system

8.protection system

An operating system provides the environment within which programs are executed. To construct such an environment, the system is partitioned into small modules with a well defined interface. The design of a new operating system is a major task. It is very important that the goals of the system be will defined before the design begins. The type of system desired is the foundation for choices between various algorithms and strategies that will be necessary. A system as large and complex as an operating system can only be created by partitioning it into smaller pieces. Each of these pieces should be a well defined portion of the system with carefully defined inputs, outputs, and function. Obviously, not all systems have the same structure.

**Process Management** The CPU executes a large number of programs. While its main concern is the execution of user programs, the CPU is also needed for other system activities. These activities are called processes. A process is a program in execution. Typically, a batch job is a process. A timeshared user program is a process. A system task, such as spooling, is also a process. For now, a process may be considered as a job or a time-shared program, but the concept is actually more general. In general, a process will need certain resources such as CPU time, memory, files, I/O devices, etc., to accomplish its task. These resources are given to the process when it is created. In addition to the various physical and logical resources that a process obtains when its is created, some initialization data (input) may be passed along.

For example, a process whose function is to display on the screen of a terminal the status of a file, say F1, will get as an input the name of the file F1 and execute the appropriate program to obtain the desired information. We emphasize that a program by itself is not a process; a program is a passive entity, while a process is an active entity. It is known that two processes may be associated with the same program, they are nevertheless considered two separate execution sequences. A process is the unit of work in a system. Such a system consists of a collection of processes, some of which are operating system processes, those that execute system code, and the rest being user processes, those that execute user code. All of those processes can potentially execute concurrently. The operating system is responsible for the following activities in connection with processes managed.

o The creation and deletion of both user and system processes o The suspension are resumption of processes.

o The provision of mechanisms for process synchronization o The provision of mechanisms for deadlock handling.

**Memory Management** Memory is central to the operation of a modern computer system. Memory is a large array of words or bytes, each with its own address. Interaction is achieved through a sequence of reads or writes of specific memory address. The CPU fetches from and stores in memory. In order for a program to be executed it must be mapped to absolute addresses and loaded in to memory. As the program executes, it accesses program instructions and data from memory by generating these absolute is declared available, and the next program may be loaded and executed. In order to improve both the utilization of CPU and the speed of the computer's response to its users, several processes must be kept in memory. There are many different algorithms depends on the particular situation. Selection of a memory management scheme for a specific system depends upon many factor, but especially upon the hardware design of the system. Each algorithm requires its own hardware support. The operating system is responsible for the following activities in connection with memory management.

o Keep track of which parts of memory are currently being used and by whom.

o Decide which processes are to be loaded into memory when memory space becomes available.

Allocate and de-allocate memory space as needed.

**Storage Management** The main purpose of a computer system is to execute programs. These programs, together with the data they access, must be in main memory during execution. Since the main memory is too small to permanently accommodate all data and program, the computer system must provide secondary storage to backup main memory. Most modem computer systems use disks as the primary on-line storage of information, of both programs and data. Most programs, like compilers, assemblers, sort routines, editors, formatters, and so on, are stored on the disk until loaded into memory, and then use the disk as both the source and destination of their processing. Hence the proper management of disk storage is of central importance to a computer system. There are few alternatives. Magnetic tape systems are generally too slow. In addition, they are limited to sequential access. Thus tapes are more suited for storing infrequently used files, where speed is not a primary concern. The operating system is responsible for the following activities in connection with disk management o Free space management o Storage allocation o Disk scheduling.

**I/O System** One of the purposes of an operating system is to hide the peculiarities of specific hardware devices from the user. For example, in Unix, the peculiarities of I/O devices are hidden from the bulk of the operating system itself by the I/O system. The I/O system consists of:

o A buffer caching system o A general device driver code

o Drivers for specific hardware devices.

Only the device driver knows the peculiarities of a specific device.

**File Management** File management is one of the most visible services of an operating system. Computers can store information in several different physical forms; magnetic tape, disk, and drum are the most common forms. Each of these devices has it own characteristics and physical organization. For convenient use of the computer system, the operating system provides a uniform logical view of information storage. The operating system abstracts from the physical properties of its storage devices to define a logical storage unit, the file. Files are mapped, by the operating system, onto physical devices. A file is a collection of related information defined by its creator. Commonly, files represent programs (both source and object forms) and data. Data files may be numeric, alphabetic or alphanumeric. Files may be free-form, such as text files, or may be rigidly formatted.

In general a files is a sequence of bits, bytes, lines or records whose meaning is defined by its creator and user. It is a very general concept. The operating system implements the abstract concept of the file by managing mass storage device, such as types and disks. Also files are normally organized into directories to ease their use. Finally, when multiple users have access to files, it may be desirable to control by whom and in what ways files may be accessed. The operating system is responsible for the following activities in connection with file management: o The creation and deletion of files o The creation and deletion of directory o The support of primitives for manipulating files and directories o The mapping of files onto disk storage. o Backup of files on stable (non volatile) storage.

**Protection System** The various processes in an operating system must be protected from each other’s activities. For that purpose, various mechanisms which can be used to ensure that the files, memory segment, cpu and other resources can be operated on only by those processes that have gained proper authorization from the operating system. For example, memory addressing hardware ensure that a process can only execute within its own address space. The timer ensure that no process can gain control of the CPU without relinquishing it. Finally, no process is allowed to do it’s own I/O, to protect the integrity of the various peripheral devices. Protection refers to a mechanism for controlling the access of programs, processes, or users to the resources defined by a computer controls to be imposed, together with some means of enforcement. Protection can improve reliability by detecting latent errors at the interfaces between component subsystems. Early detection of interface errors can often prevent contamination of a healthy subsystem by a subsystem that is malfunctioning. An unprotected resource cannot defend against use (or misuse) by an unauthorized or incompetent user.

**Networking** A distributed system is a collection of processors that do not share memory or a clock. Instead, each processor has its own local memory, and the processors communicate with each other through various communication lines, such as high speed buses or telephone lines. Distributed systems vary in size and function. They may involve microprocessors, workstations, minicomputers, and large general purpose computer systems. The processors in the system are connected through a communication network, which can be configured in the number of different ways. The network may be fully or partially connected. The communication network design must consider routing and connection strategies, and the problems of connection and security. A distributed system provides the user with access to the various resources the system maintains. Access to a shared resource allows computation speed-up, data availability, and reliability.

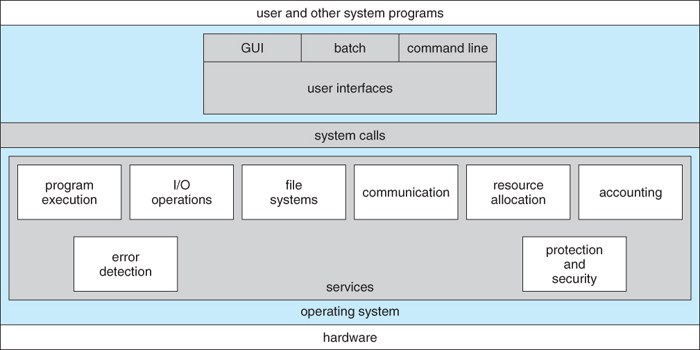
**Command Interpreter System** One of the most important component of an operating system is its command interpreter. The command interpreter is the primary interface between the user and the rest of the system. Many commands are given to the operating system by control statements. When a new job is started in a batch system or when a user logs-in to a time-shared system, a program which reads and interprets control statements is automatically executed.

This program is variously called (1) the control card interpreter, (2) the command line interpreter, (3) the shell (in Unix), and so on. Its function is quite simple: get the next command statement, and execute it. The command statement themselves deal with process management, I/O handling, secondary storage management, main memory management, file system access, protection, and networking.

Chapter 2

**Operating-System Structures**

**Operating-System Services**

  
**Figure - A view of operating system services**

OS provide environments in which programs run, and services for the users of the system, including:

* **User Interfaces** - Means by which users can issue commands to the system. Depending on the system these may be a command-line interface ( e.g. sh, csh, ksh, tcsh, etc. ), a GUI interface ( e.g. Windows, X-Windows, KDE, Gnome, etc. ), or a batch command systems. The latter are generally older systems using punch cards of job-control language, JCL, but may still be used today for specialty systems designed for a single purpose.
* **Program Execution** - The OS must be able to load a program into RAM, run the program, and terminate the program, either normally or abnormally.
* **I/O Operations** - The OS is responsible for transferring data to and from I/O devices, including keyboards, terminals, printers, and storage devices.
* **File-System Manipulation** - In addition to raw data storage, the OS is also responsible for maintaining directory and subdirectory structures, mapping file names to specific blocks of data storage, and providing tools for navigating and utilizing the file system.
* **Communications** - Inter-process communications, IPC, either between processes running on the same processor, or between processes running on separate processors or separate machines. May be implemented as either shared memory or message passing, ( or some systems may offer both. )
* **Error Detection** - Both hardware and software errors must be detected and handled appropriately, with a minimum of harmful repercussions. Some systems may include complex error avoidance or recovery systems, including backups, RAID drives, and other redundant systems. Debugging and diagnostic tools aid users and administrators in tracing down the cause of problems.

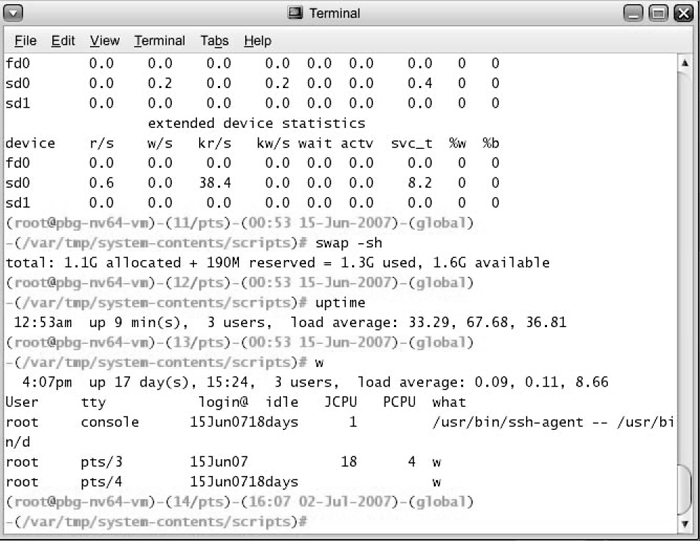
Other systems aid in the efficient operation of the OS itself:

* **Resource Allocation** - E.g. CPU cycles, main memory, storage space, and peripheral devices. Some resources are managed with generic systems and others with very carefully designed and specially tuned systems, customized for a particular resource and operating environment.
* **Accounting** - Keeping track of system activity and resource usage, either for billing purposes or for statistical record keeping that can be used to optimize future performance.
* **Protection and Security** - Preventing harm to the system and to resources, either through wayward internal processes or malicious outsiders. Authentication, ownership, and restricted access are obvious parts of this system. Highly secure systems may log all process activity down to excruciating detail, and security regulation dictate the storage of those records on permanent non-erasable medium for extended times in secure ( off-site ) facilities.

**User Operating-System Interface**

**Command Interpreter**

* Gets and processes the next user request, and launches the requested programs.
* In some systems the CI may be incorporated directly into the kernel.
* More commonly the CI is a separate program that launches once the user logs in or otherwise accesses the system.
* UNIX, for example, provides the user with a choice of different shells, which may either be configured to launch automatically at login, or which may be changed on the fly. ( Each of these shells uses a different configuration file of initial settings and commands that are executed upon startup. )
* Different shells provide different functionality, in terms of certain commands that are implemented directly by the shell without launching any external programs. Most provide at least a rudimentary command interpretation structure for use in shell script programming ( loops, decision constructs, variables, etc. )
* An interesting distinction is the processing of wild card file naming and I/O re-direction. On UNIX systems those details are handled by the shell, and the program which is launched sees only a list of filenames generated by the shell from the wild cards. On a DOS system, the wild cards are passed along to the programs, which can interpret the wild cards as the program sees fit.

  
**Figure - The Bourne shell command interpreter in Solaris 10**

**Graphical User Interface, GUI**

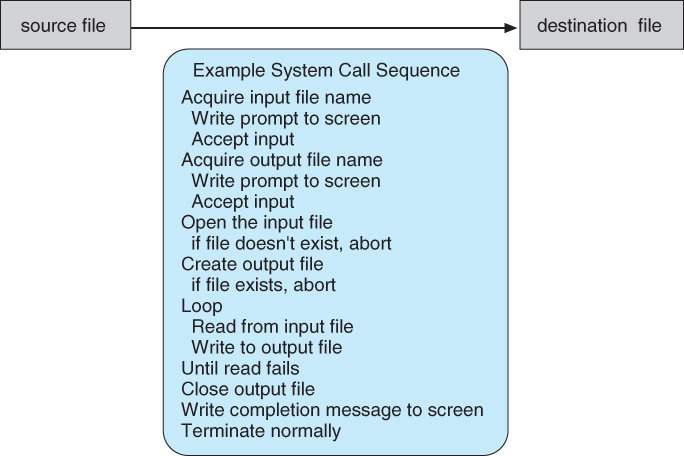
* Generally implemented as a desktop metaphor, with file folders, trash cans, and resource icons.
* Icons represent some item on the system, and respond accordingly when the icon is activated.
* First developed in the early 1970's at Xerox PARC research facility.
* In some systems the GUI is just a front end for activating a traditional command line interpreter running in the background. In others the GUI is a true graphical shell in its own right.
* Mac has traditionally provided ONLY the GUI interface. With the advent of OSX ( based partially on UNIX ), a command line interface has also become available.
* Because mice and keyboards are impractical for small mobile devices, these normally use a touch-screen interface today, that responds to various patterns of swipes or "gestures". When these first came out they often had a physical keyboard and/or a trackball of some kind built in, but today a virtual keyboard is more commonly implemented on the touch screen.

**Choice of interface**

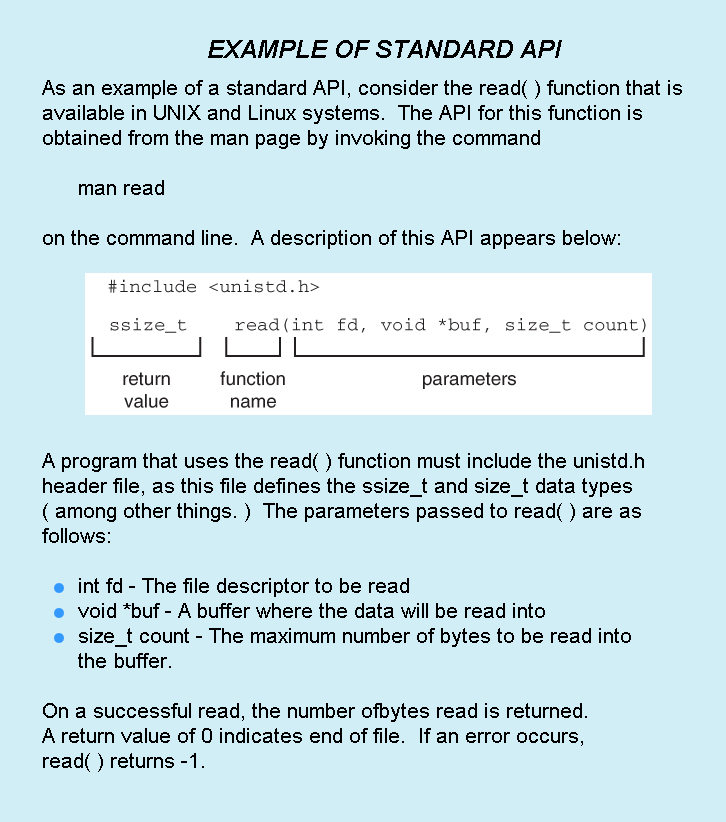
* Most modern systems allow individual users to select their desired interface, and to customize its operation, as well as the ability to switch between different interfaces as needed. System administrators generally determine which interface a user starts with when they first log in.
* GUI interfaces usually provide an option for a terminal emulator window for entering command-line commands.
* Command-line commands can also be entered into **shell scripts**, which can then be run like any other programs.

**2.3 System Calls**

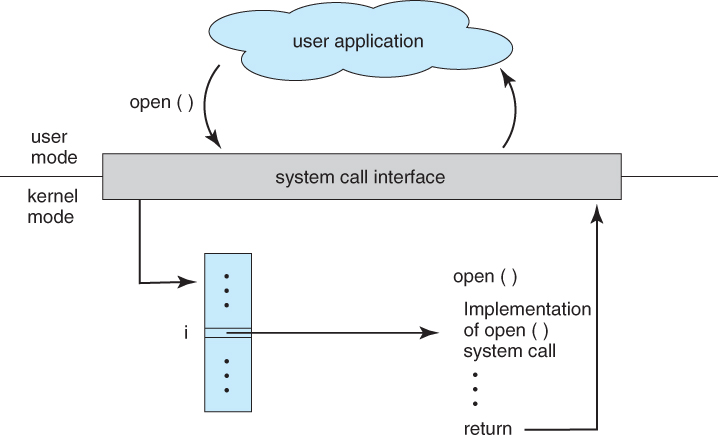
* System calls provide a means for user or application programs to call upon the services of the operating system.
* Generally written in C or C++, although some are written in assembly for optimal performance.

  
**Figure 2.5 - Example of how system calls are used.**

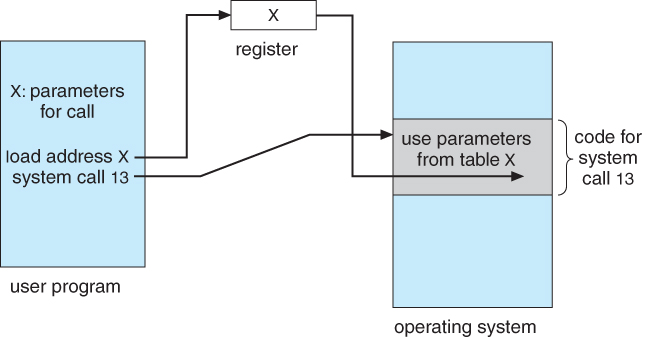
* You can use "strace" to see more examples of the large number of system calls invoked by a single simple command. Read the man page for strace, and try some simple examples. ( strace mkdir temp, strace cd temp, strace date > t.t, strace cp t.t t.2, etc. )
* Most programmers do not use the low-level system calls directly, but instead use an "Application Programming Interface", API. The following sidebar shows the read( ) call available in the API on UNIX based systems::



The use of APIs instead of direct system calls provides for greater program portability between different systems. The API then makes the appropriate system calls through the **system call interface**, using a table lookup to access specific numbered system calls, as shown in Figure below:

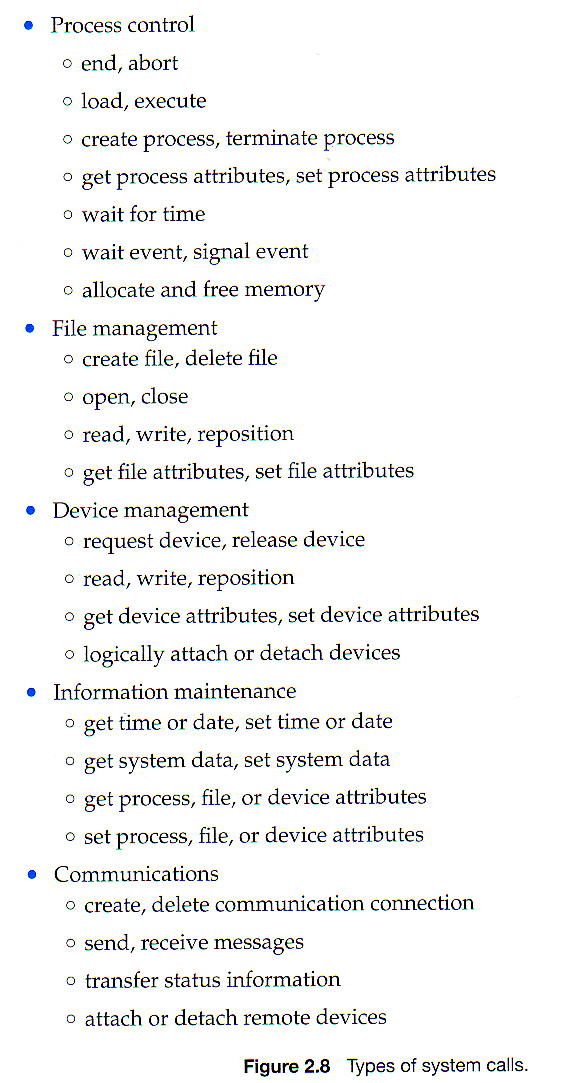
  
**Figure - The handling of a user application invoking the open( ) system call**

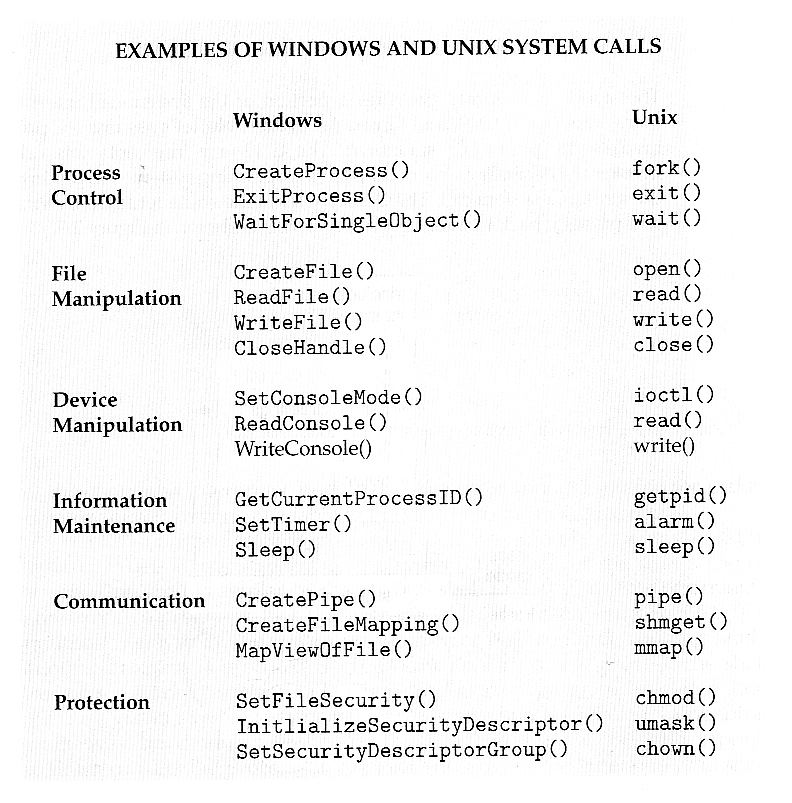
* Parameters are generally passed to system calls via registers, or less commonly, by values pushed onto the stack. Large blocks of data are generally accessed indirectly, through a memory address passed in a register or on the stack, as shown in Figure below:

  
**Figure - Passing of parameters as a table**

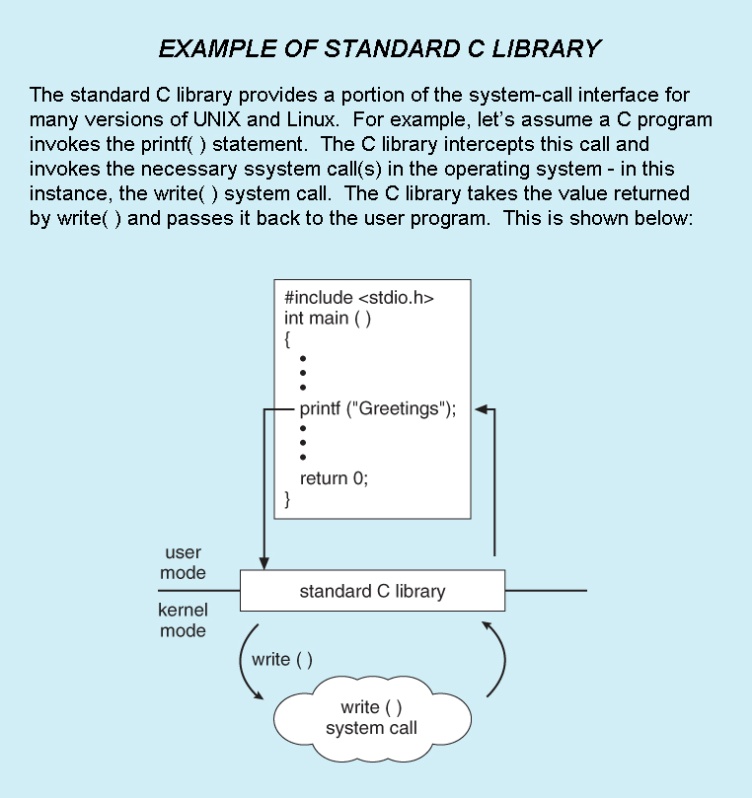
**2.4 Types of System Calls**

Six major categories, as outlined in Figure and the following six subsections:

  
**( Sixth type, protection, not shown here but described below. )**

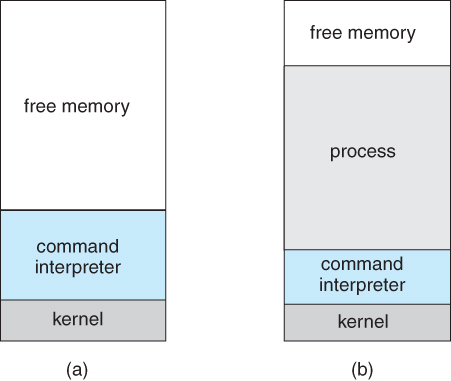


* Standard library calls may also generate system calls, as shown here:

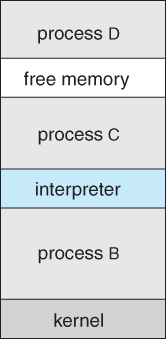
****

**Process Control**

* Process control system calls include end, abort, load, execute, create process, terminate process, get/set process attributes, wait for time or event, signal event, and allocate and free memory.
* Processes must be created, launched, monitored, paused, resumed and eventually stopped.
* When one process pauses or stops, then another must be launched or resumed
* When processes stop abnormally it may be necessary to provide core dumps and/or other diagnostic or recovery tools.
* Compare DOS ( a single-tasking system ) with UNIX ( a multi-tasking system ).
  + When a process is launched in DOS, the command interpreter first unloads as much of itself as it can to free up memory, then loads the process and transfers control to it. The interpreter does not resume until the process has completed, as shown in Figure:

  
**Figure - MS-DOS execution. (a) At system startup. (b) Running a program.**

* + Because UNIX is a multi-tasking system, the command interpreter remains completely resident when executing a process, as shown in Figure below.
    - The user can switch back to the command interpreter at any time, and can place the running process in the background even if it was not originally launched as a background process.
    - In order to do this, the command interpreter first executes a "fork" system call, which creates a second process which is an exact duplicate ( clone ) of the original command interpreter. The original process is known as the parent, and the cloned process is known as the child, with its own unique process ID and parent ID.
    - The child process then executes an "exec" system call, which replaces its code with that of the desired process.
    - The parent (command interpreter) normally waits for the child to complete before issuing a new command prompt, but in some cases it can also issue a new prompt right away, without waiting for the child process to complete. (The child is then said to be running "in the background", or "as a background process".)

  
**Figure - FreeBSD running multiple programs**

**File Management**

* File management system calls include create file, delete file, open, close, read, write, reposition, get file attributes, and set file attributes.
* These operations may also be supported for directories as well as ordinary files.

**Device Management**

* Device management system calls include request device, release device, read, write, reposition, get/set device attributes, and logically attach or detach devices.
* Devices may be physical ( e.g. disk drives ), or virtual / abstract ( e.g. files, partitions, and RAM disks ).
* Some systems represent devices as special files in the file system, so that accessing the "file" calls upon the appropriate device drivers in the OS. See for example the /dev directory on any UNIX system.

**Information Maintenance**

* Information maintenance system calls include calls to get/set the time, date, system data, and process, file, or device attributes.
* Systems may also provide the ability to dump memory at any time, single step programs pausing execution after each instruction, and tracing the operation of programs, all of which can help to debug programs.

**Communication**

* Communication system calls create/delete communication connection, send/receive messages, transfer status information, and attach/detach remote devices.
* The **message passing** model must support calls to:
  + Identify a remote process and/or host with which to communicate.
  + Establish a connection between the two processes.
  + Open and close the connection as needed.
  + Transmit messages along the connection.
  + Wait for incoming messages, in either a blocking or non-blocking state.
  + Delete the connection when no longer needed.
* The **shared memory** model must support calls to:
  + Create and access memory that is shared amongst processes ( and threads. )
  + Provide locking mechanisms restricting simultaneous access.
  + Free up shared memory and/or dynamically allocate it as needed.
* Message passing is simpler and easier, ( particularly for inter-computer communications ), and is generally appropriate for small amounts of data.
* Shared memory is faster, and is generally the better approach where large amounts of data are to be shared, ( particularly when most processes are reading the data rather than writing it, or at least when only one or a small number of processes need to change any given data item. )

**Protection**

* Protection provides mechanisms for controlling which users / processes have access to which system resources.
* System calls allow the access mechanisms to be adjusted as needed, and for non-priveleged users to be granted elevated access permissions under carefully controlled temporary circumstances.
* Once only of concern on multi-user systems, protection is now important on all systems, in the age of ubiquitous network connectivity.

**System Programs**

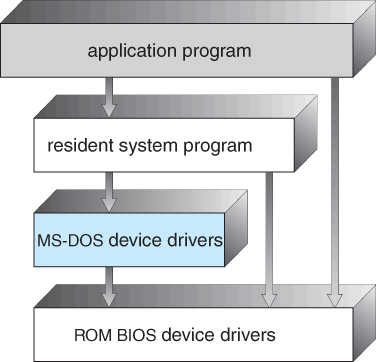
* System programs provide OS functionality through separate applications, which are not part of the kernel or command interpreters. They are also known as system utilities or system applications.
* Most systems also ship with useful applications such as calculators and simple editors, ( e.g. Notepad ). Some debate arises as to the border between system and non-system applications.
* System programs may be divided into these categories:
  + **File management** - programs to create, delete, copy, rename, print, list, and generally manipulate files and directories.
  + **Status information** - Utilities to check on the date, time, number of users, processes running, data logging, etc. System **registries** are used to store and recall configuration information for particular applications.
  + **File modification** - e.g. text editors and other tools which can change file contents.
  + **Programming-language support** - E.g. Compilers, linkers, debuggers, profilers, assemblers, library archive management, interpreters for common languages, and support for make.
  + **Program loading and execution** - loaders, dynamic loaders, overlay loaders, etc., as well as interactive debuggers.
  + **Communications** - Programs for providing connectivity between processes and users, including mail, web browsers, remote logins, file transfers, and remote command execution.
  + **Background services** - System daemons are commonly started when the system is booted, and run for as long as the system is running, handling necessary services. Examples include network daemons, print servers, process schedulers, and system error monitoring services.
* Most operating systems today also come complete with a set of **application programs** to provide additional services, such as copying files or checking the time and date.
* Most users' views of the system is determined by their command interpreter and the application programs. Most never make system calls, even through the API, (with the exception of simple ( file ) I/O in user-written programs. )

**Operating-System Structure**

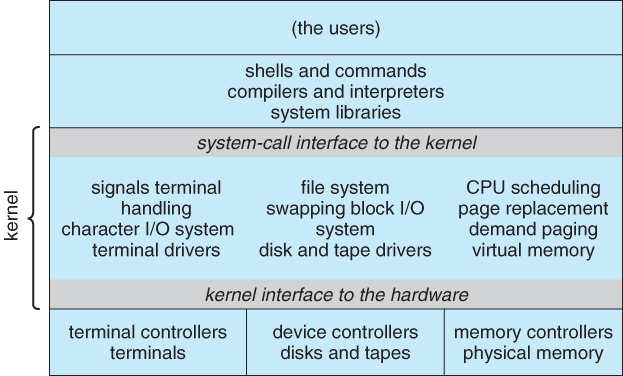
For efficient performance and implementation an OS should be partitioned into separate subsystems, each with carefully defined tasks, inputs, outputs, and performance characteristics. These subsystems can then be arranged in various architectural configurations:

**Simple Structure**

When DOS was originally written its developers had no idea how big and important it would eventually become. It was written by a few programmers in a relatively short amount of time, without the benefit of modern software engineering techniques, and then gradually grew over time to exceed its original expectations. It does not break the system into subsystems, and has no distinction between user and kernel modes, allowing all programs direct access to the underlying hardware. ( Note that user versus kernel mode was not supported by the 8088 chip set anyway, so that really wasn't an option back then. )

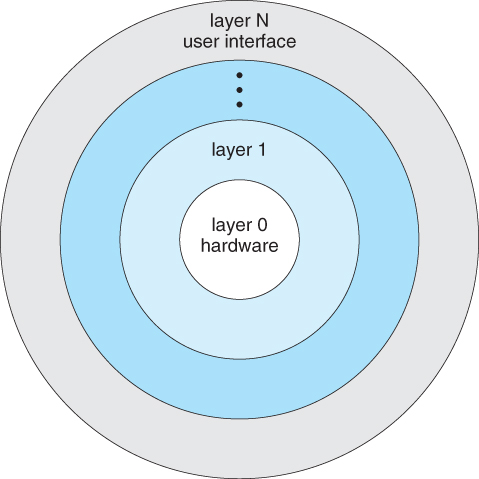
  
**Figure - MS-DOS layer structure**

The original UNIX OS used a simple layered approach, but almost all the OS was in one big layer, not really breaking the OS down into layered subsystems:

**  
- Traditional UNIX system structure**

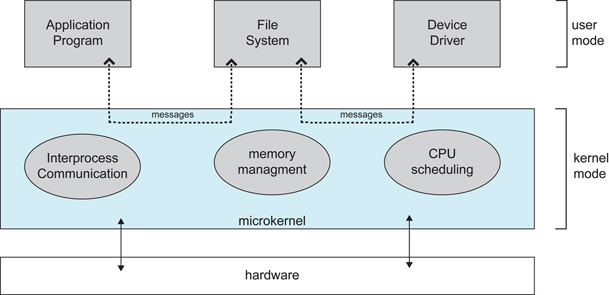
**Layered Approach**

* Another approach is to break the OS into a number of smaller layers, each of which rests on the layer below it, and relies solely on the services provided by the next lower layer.
* This approach allows each layer to be developed and debugged independently, with the assumption that all lower layers have already been debugged and are trusted to deliver proper services.
* The problem is deciding what order in which to place the layers, as no layer can call upon the services of any higher layer, and so many chicken-and-egg situations may arise.
* Layered approaches can also be less efficient, as a request for service from a higher layer has to filter through all lower layers before it reaches the HW, possibly with significant processing at each step.

  
**- A layered operating system**

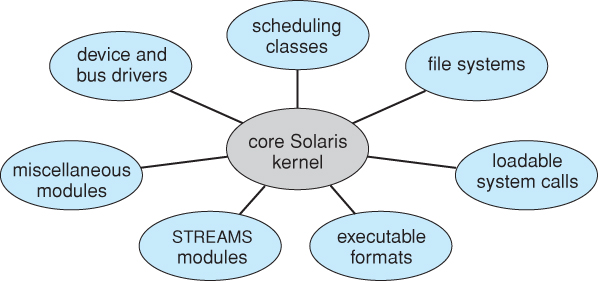
**Microkernels**

* The basic idea behind micro kernels is to remove all non-essential services from the kernel, and implement them as system applications instead, thereby making the kernel as small and efficient as possible.
* Most microkernels provide basic process and memory management, and message passing between other services, and not much more.
* Security and protection can be enhanced, as most services are performed in user mode, not kernel mode.
* System expansion can also be easier, because it only involves adding more system applications, not rebuilding a new kernel.
* Mach was the first and most widely known microkernel, and now forms a major component of Mac OSX.
* Windows NT was originally microkernel, but suffered from performance problems relative to Windows 95. NT 4.0 improved performance by moving more services into the kernel, and now XP is back to being more monolithic.
* Another microkernel example is QNX, a real-time OS for embedded systems.

  
**- Architecture of a typical microkernel**

**Modules**

* Modern OS development is object-oriented, with a relatively small core kernel and a set of ***modules*** which can be linked in dynamically. See for example the Solaris structure, as shown in Figure 2.13 below.
* Modules are similar to layers in that each subsystem has clearly defined tasks and interfaces, but any module is free to contact any other module, eliminating the problems of going through multiple intermediary layers, as well as the chicken-and-egg problems.
* The kernel is relatively small in this architecture, similar to microkernels, but the kernel does not have to implement message passing since modules are free to contact each other directly.

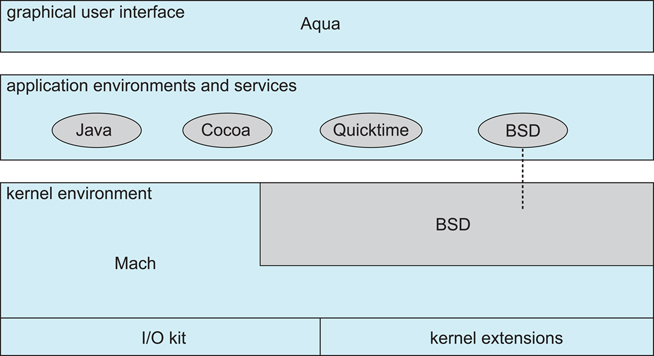
  
**- Solaris loadable modules**

**Hybrid Systems**

* Most OS’s today do not strictly adhere to one architecture, but are hybrids of several.

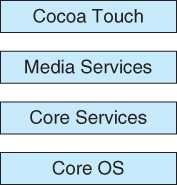
**Mac OS X**

* The Max OSX architecture relies on the Mach microkernel for basic system management services, and the BSD kernel for additional services. Application services and dynamically loadable modules ( kernel extensions ) provide the rest of the OS functionality:

  
**- The Mac OS X structure**

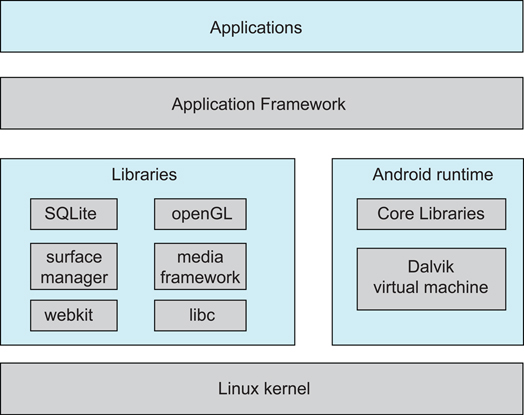
**iOS**

* The **iOS** operating system was developed by Apple for iPhones and iPads. It runs with less memory and computing power needs than Max OS X, and supports touchscreen interface and graphics for small screens:

  
**- Architecture of Apple's iOS.**

**Android**

* The Android OS was developed for Android smartphones and tablets by the Open Handset Alliance, primarily Google.
* Android is an open-source OS, as opposed to iOS, which has lead to its popularity.
* Android includes versions of Linux and a Java virtual machine both optimized for small platforms.
* Android apps are developed using a special Java-for-Android development environment.

  
 **- Architecture of Google's Android**

**Virtual Machine**

A virtual machine takes the layered approach to its logical conclusion. It treats hardware and the operating system kernel as though they were all hardware!

■  A virtual machine provides an interface identical to the underlying bare hardware!

■  Virtualization gives an illusion of multiple machines (complete hardware & OS). !

■  The operating system creates the illusion of multiple processes, each executing on its own processor with its own (virtual) memory

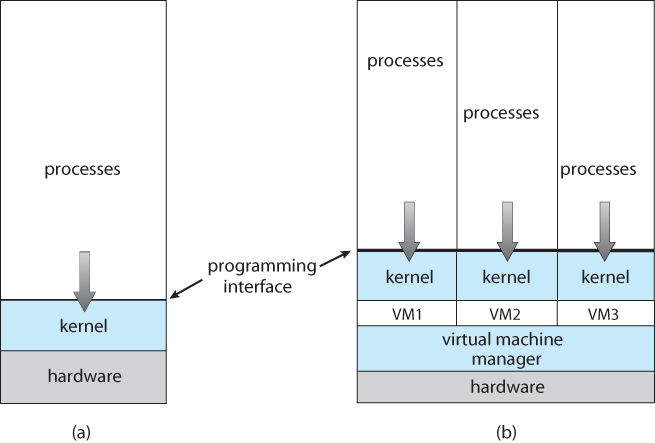
The resources of the physical computer are shared to create the virtual machines!

●  CPU scheduling can create the appearance that users have their own processor!

●  Spooling and a ﬁle system can provide virtual card readers and virtual line printers!

●  A normal user time-sharing terminal serves as the virtual machine operator’s console!

●  A key enabler is VMM (Virtural Machine Monitor)!



The virtual-machine concept provides complete protection of system resources since each virtual machine is isolated from all other virtual machines. This isolation, however, permits no direct sharing of resources!

■  A virtual-machine system is a perfect vehicle for operating-systems research and development. System development is done on the virtual machine, instead of on a physical machine and so does not disrupt normal system operation!

■  The virtual machine concept is difficult to implement due to the effort required to provide an exact duplicate to the underlying machine.

**Virtualization Implementation**

„Difficult to implement –must provide an exact duplicate of underlying machines Typically runs in user mode, creates virtual user mode and virtual kernel mode. „Timing can be an issue –slower than real machine. „Hardware support needed More support-> better virtualization i.e. AMD provides “host”and “guest”modes

**Benefits**

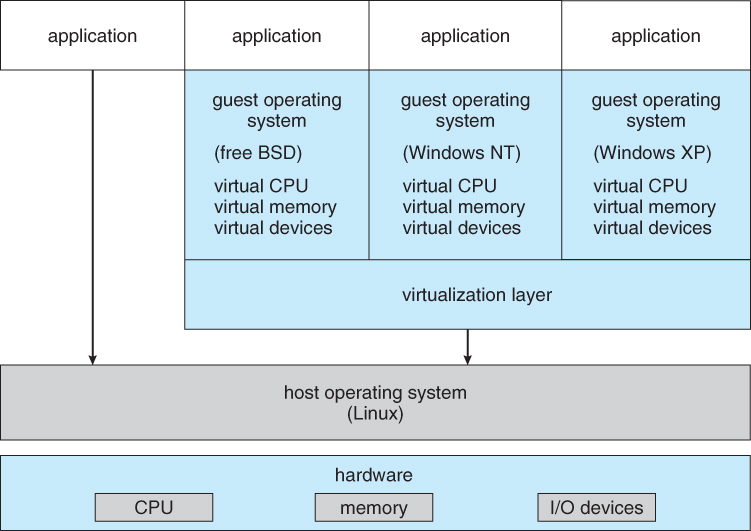
The virtual-machine concept has several advantages. Notice that, in this environment, there is complete protection of the various system resources. Each virtual machine is completely isolated from all other virtual machines, so there are no protection problems. At the same time, however, there is no direct sharing of resources. Two approaches to provide sharing have been implemented. First, it is possible to share a minidisk and thus to share files. This scheme is modeled after a physical shared disk but is implemented by software. Second, it is possible to define a network of virtual machines, each of which can send information over the virtual communications network. Again, the network is modeled after physical communication networks but is implemented in software.

Such a virtual-machine system is a perfect vehicle for operating-systems research and development. Normally, changing an operating system is a difficult task. Operating systems are large and complex programs, and it is difficult to be sure that a change in one part will not cause obscure bugs in some other part. The power of the operating system makes changing it particularly dangerous. Because the operating system executes in kernel mode, a wrong change in a pointer could cause an error that would destroy the entire file system. Thus, it is necessary to test all changes to the operating system carefully.

The operating system, however, runs on and controls the entire machine. Therefore, the current system must be stopped and taken out of use while changes are made and tested. This period is commonly called system development time. Since it makes the system unavailable to users, system development time is often scheduled late at night or on weekends, when system load is low.

A virtual-machine system can eliminate much of this problem. System programmers are given their own virtual machine, and system development is done on the virtual machine instead of on a physical machine. Normal system operation seldom needs to be disrupted for system development.

**VMWare**:



VMware 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.

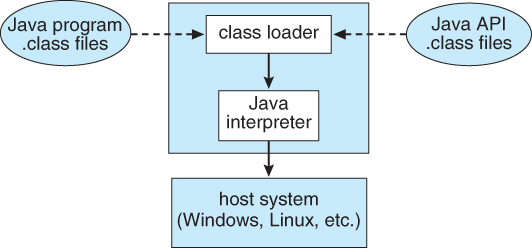
Consider the following scenario: A developer has designed an application and would like to test it on Linux, FreeBSD, Windows NT, and Windows XP. One option is for her to obtain four different computers, each running a copy of one of these operating systems. Another alternative is for her first to install Linux on a computer system and test the application, then to install FreeBSD and test the application, and so forth. This option allows her to use the same physical computer but is time-consuming, since she must install a new operating system for each test. Such testing could be accomplished concurrently on the same physical computer using VMware. In this case, the programmer could test the application on a host operating system and on three guest operating systems with each system running as a separate virtual machine.

The architecture of such a system is shown in figure. In this scenario, Linux is running as the host operating system; FreeBSD, Windows NT, and Windows XP are running as guest operating systems. 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.

**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. Java objects are specified with the class construct; a Java program consists of one or more classes. 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 . 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 JVM also automatically manages memory by performing garbage collection—the practice of reclaiming memory from objects no longer in use and returning it to the system. Much research focuses on garbage collection algorithms for increasing the performance of Java programs in the virtual machine.



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. Alternatively, the JVM may be implemented in hardware on a chip specifically designed to run Java programs. If the JVM is implemented in software, the Java interpreter interprets the bytecode operations one at a time. A faster software technique is to use a just-in-time (JIT) compiler. Here, the first time a Java method is invoked, the bytecodes for the method are turned into native machine language for the host system. These operations are then cached so that subsequent invocations of a method are performed using the native machine instructions and the bytecode operations need not be interpreted all over again. A technique that is potentially even faster is to run the JVM in hardware on a special Java chip that executes the Java bytecode operations as native code, thus bypassing the need for either a software interpreter or a just-in-time compiler.

**Operating-System Generation**

* OSes may be designed and built for a specific HW configuration at a specific site, but more commonly they are designed with a number of variable parameters and components, which are then configured for a particular operating environment.
* Systems sometimes need to be re-configured after the initial installation, to add additional resources, capabilities, or to tune performance, logging, or security.
* Information that is needed to configure an OS include:
  + What CPU(s) are installed on the system, and what optional characteristics does each have?
  + How much RAM is installed? (This may be determined automatically, either at install or boot time.)
  + What devices are present? The OS needs to determine which device drivers to include, as well as some device-specific characteristics and parameters.
  + What OS options are desired, and what values to set for particular OS parameters. The latter may include the size of the open file table, the number of buffers to use, process scheduling ( priority ) parameters, disk scheduling algorithms, number of slots in the process table, etc.
* At one extreme the OS source code can be edited, re-compiled, and linked into a new kernel.
* More commonly configuration tables determine which modules to link into the new kernel, and what values to set for some key important parameters. This approach may require the configuration of complicated make files, which can be done either automatically or through interactive configuration programs; Then make is used to actually generate the new kernel specified by the new parameters.
* At the other extreme a system configuration may be entirely defined by table data, in which case the "rebuilding" of the system merely requires editing data tables.
* Once a system has been regenerated, it is usually required to reboot the system to activate the new kernel. Because there are possibilities for errors, most systems provide some mechanism for booting to older or alternate kernels.

**System Boot**

The general approach when most computers boot up goes something like this:

* When the system powers up, an interrupt is generated which loads a memory address into the program counter, and the system begins executing instructions found at that address. This address points to the "bootstrap" program located in ROM chips ( or EPROM chips ) on the motherboard.
* The ROM bootstrap program first runs hardware checks, determining what physical resources are present and doing power-on self tests ( POST ) of all HW for which this is applicable. Some devices, such as controller cards may have their own on-board diagnostics, which are called by the ROM bootstrap program.
* The user generally has the option of pressing a special key during the POST process, which will launch the ROM BIOS configuration utility if pressed. This utility allows the user to specify and configure certain hardware parameters as where to look for an OS and whether or not to restrict access to the utility with a password.
  + Some hardware may also provide access to additional configuration setup programs, such as for a RAID disk controller or some special graphics or networking cards.
* Assuming the utility has not been invoked, the bootstrap program then looks for a non-volatile storage device containing an OS. Depending on configuration, it may look for a floppy drive, CD ROM drive, or primary or secondary hard drives, in the order specified by the HW configuration utility.
* Assuming it goes to a hard drive, it will find the first sector on the hard drive and load up the fdisk table, which contains information about how the physical hard drive is divided up into logical partitions, where each partition starts and ends, and which partition is the "active" partition used for booting the system.
* There is also a very small amount of system code in the portion of the first disk block not occupied by the fdisk table. This bootstrap code is the first step that is not built into the hardware, i.e. the first part which might be in any way OS-specific. Generally this code knows just enough to access the hard drive, and to load and execute a ( slightly ) larger boot program.
* For a single-boot system, the boot program loaded off of the hard disk will then proceed to locate the kernel on the hard drive, load the kernel into memory, and then transfer control over to the kernel. There may be some opportunity to specify a particular kernel to be loaded at this stage, which may be useful if a new kernel has just been generated and doesn't work, or if the system has multiple kernels available with different configurations for different purposes. ( Some systems may boot different configurations automatically, depending on what hardware has been found in earlier steps. )
* For dual-boot or multiple-boot systems, the boot program will give the user an opportunity to specify a particular OS to load, with a default choice if the user does not pick a particular OS within a given time frame. The boot program then finds the boot loader for the chosen single-boot OS, and runs that program as described in the previous bullet point.
* Once the kernel is running, it may give the user the opportunity to enter into single-user mode, also known as maintenance mode. This mode launches very few if any system services, and does not enable any logins other than the primary log in on the console. This mode is used primarily for system maintenance and diagnostics.
* When the system enters full multi-user multi-tasking mode, it examines configuration files to determine which system services are to be started, and launches each of them in turn. It then spawns login programs ( gettys ) on each of the login devices which have been configured to enable user logins.
  + ( The getty program initializes terminal I/O, issues the login prompt, accepts login names and passwords, and authenticates the user. If the user's password is authenticated, then the getty looks in system files to determine what shell is assigned to the user, and then "execs" ( becomes ) the user's shell. The shell program will look in system and user configuration files to initialize itself, and then issue prompts for user commands. Whenever the shell dies, either through logout or other means, then the system will issue a new getty for that terminal device. )

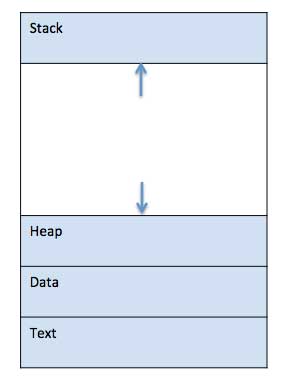
**Chapter-3**

**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 I/O 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. Although traditionally a process contained only a single thread of control as it ran, most modern operating systems now support processes that have multiple threads. The operating system is responsible for the following activities in connection with process and thread management: the creation and deletion of both user and system processes; the scheduling of processes; and the provision of mechanisms for synchronization, communication, and deadlock handling for processes.

A batch system executes jobs, whereas a time-shared system has user programs, or tasks. Even on a single-user system such as Microsoft Windows, a user may be able to run several programs at one time: a word processor, a web browser, and an e-mail package. Even if the user can execute only one program at a time, the operating system may need to support its own internal programmed activities, such as memory management. In many respects, all these activities are similar, so we call all of them processes. The terms job and process are used almost interchangeably in this text. Although we personally prefer the term process, much of operating-system theory and terminology was developed during a time when the major activity of operating systems was job processing. It would be misleading to avoid the use of commonly accepted terms that include the word job (such as job scheduling) simply because process has superseded job.

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



A program is a passive entity, such as a file containing a list of instructions stored on disk (often called an executable file), whereas a process is an active entity, with a program counter specifying the next instruction to execute and a set of associated resources. A program becomes a process when an executable file is loaded into memory. Two common techniques for loading executable files are double-clicking an icon representing the executable file and entering the name of the executable file on the command line (as in prog. exe or a. out.)

**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.

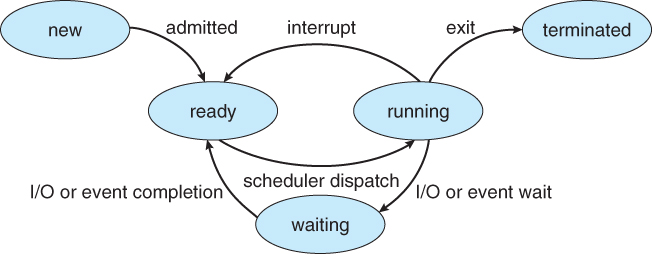
• Running. Instructions are being executed.

• Waiting. The process is waiting for some event to occur (such as an I/O completion or reception of a signal).

• Ready. The process is waiting to be assigned to a processor.

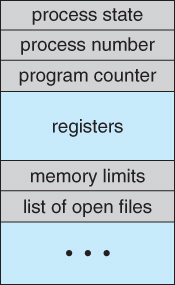
• Terminated. The process has finished execution.

These names are arbitrary, and they vary across operating systems. The states that they represent are fotind on all systems, however. Certain operating systems also more finely delineate process states. It is important to realize that only one process can be running on any processor at any instant. Many processes may be ready and limiting, however. The state diagram corresponding to these states is presented in Figure.



**Process Control Block**

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



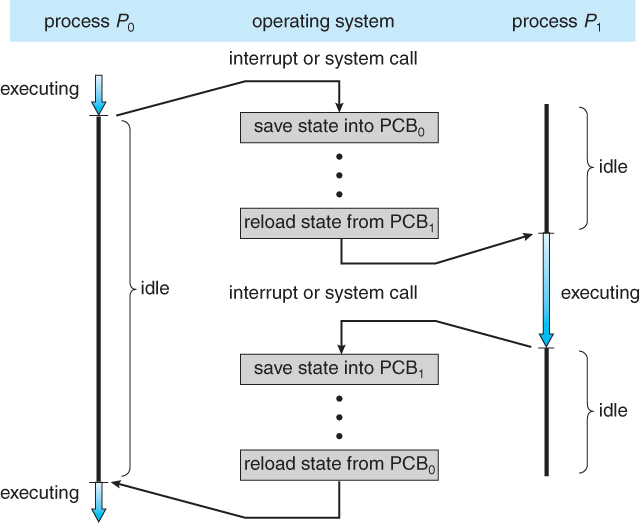
It contains many pieces of information associated with a specific process, including these:

• 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 (Figure).

• 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.

PCB simply serves as the repository for any information that may vary from process to process.

**Threads**

The process model discussed so far has implied that a process is a program that performs a single thread of execution. For example, when a process is running a word-processor program, a single thread of instructions is being executed. This single thread of control allows the process to perform only one task at one time. The user cannot simultaneously type in characters and run the spell checker within the same process, for example. Many modern operating systems have extended the process concept to allow a process to have multiple threads of execution and thus to perform more than one task at a time.

**Process Scheduling**:

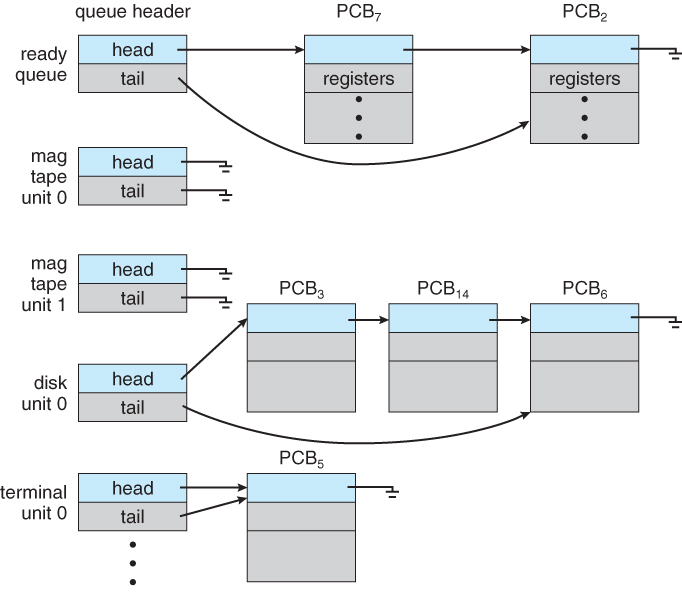
The objective of multiprogramming is to have some process running at all times, to maximize CPU utilization. The objective of time sharing is to switch the CPU among processes so frequently that users can interact with each program while it is running. To meet these objectives, the process scheduler selects an available process (possibly from a set of several available processes) for program execution on the CPU. For a single-processor system, there will never be more than one running process. If there are more processes, the rest will have to wait until the CPU is free and can be rescheduled.

**Scheduling Queues**

As processes enter the system, they are put into a job queue, which consists of all processes in the system. The processes that are residing in main memory and are ready and waiting to execute are kept on a list called the ready queue. 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.

The system also includes other queues. When a process is allocated the CPU, it executes for a while and eventually quits, is interrupted, or waits for the occurrence of a particular event, such as the completion of an I/O request.

Suppose the process makes an I/O request to a shared device, such as a disk. Since there are many processes in the system, the disk may be busy with the I/O request of some other process. The process therefore may have to wait for the disk. The list of processes waiting for a particular I/O device is called a device queue. Each device has its own device queue (Figure)

.

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

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

• The process could create a new 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**

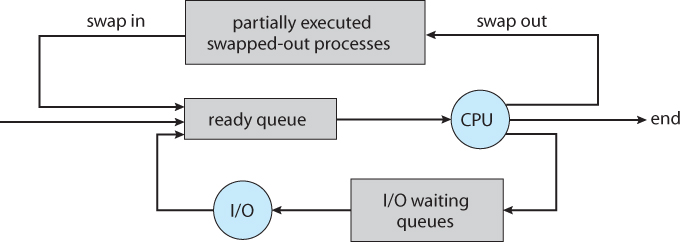
A process migrates among the various scheduling queues throughout its lifetime. The operating system must select, for scheduling purposes, processes from these queues in some fashion. The selection process is carried out by the appropriate scheduler.

Often, 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 short-term scheduler, or CPU scheduler, selects from among the processes that are ready to execute and allocates the CPU to one of them.

The primary distinction between these two schedulers lies in frequency of execution. 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.

It is important that the long-term scheduler make a careful selection. In general, most processes can be described as either L/O bound or CPU bound. An I/O-bound process is one that spends more of its time doing I/O than it spends doing computations. A CPU-bound process, in contrast, generates I/O requests infrequently, using more of its time doing computations. It is important that the long-term scheduler select a good process mix of I/O-bound and CPU-bound processes. If all processes are I/O bound, the ready queue will almost always be empty, and the short-term scheduler will have little to do. If all processes are CPU bound, the I/O waiting queue will almost always be empty, devices will go unused, and again the system will be unbalanced. The system with the best performance will thus have a combination of CPU-bound and I/O-bound processes.

Operating systems, such as time-sharing systems, may introduce an additional, intermediate level of scheduling. This medium-term scheduler is diagrammed in Figure.

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 up.

**Context Switch**

When an interrupt occurs, the system needs to save the current context of the process currently running on the CPU so that it can restore that context when its processing is done, essentially suspending the process and then resuming it. The context is represented in the PCB of the process; it includes the value of the CPU registers, the process state, and memory-management information. Generically, we perform a state save of the current state of the CPU, be it in kernel or user mode, and then a state restore to resume operations.

Switching the CPU to another process requires performing a stat^ save of the current process and a state restore of a different process. This task is known as a context switch. When a context switch occurs, the kernel saves the context of the old process in its PCB and loads the saved context of the new process scheduled to run. 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.

**Operations 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.

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

1. The parent continues to execute concurrently with its children.

2. The parent waits until some or all of its children have terminated.

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

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

2. The child process has a new program loaded into it.

let's first consider the UNIX operating system. In UNIX, as we've seen, each process is identified by its process identifier which is a unique integer. A new process is created by the forkO system call. The new process consists of a copy of the address space of the original process. This mechanism allows the parent process to communicate easily with its child process. Both processes (the parent and the child) continue execution at the instruction after the f ork(), with one difference: The return code for the forkO is zero for the new (child) process, whereas the (nonzero) process identifier of the child is returned to the parent. Typically, the execO system call is used after a forkO 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 execO system call) and starts its execution. In this manner, the two processes are able to communicate and then go their separate ways. The parent can then create more children; or, if it has nothing else to do while the child runs, it can issue a wait () 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/O 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. 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. Note that a parent needs to know the identities of its children. Thus, when one process creates a new process, the identity of the newly created process is passed to the parent. 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.

Consider that, in UNIX, we can terminate a process by using the exit() system call; its parent process may wait for the termination of a child process by using the waitO system call. The wait () system call returns the process identifier of a terminated child so that the parent can tell which of its possibly many children has terminated. If the parent terminates, however, all its children have assigned as their new parent the init process. Thus, the children still have a parent to collect their status and execution statistics.

**Interprocess Communication**

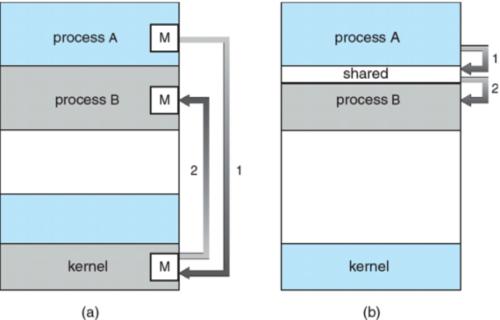
Processes executing concurrently in the operating system may be either independent processes or cooperating processes. A process is independent if it cannot affect or be affected by the other processes executing in the system. Any process that does not share data with any other process is independent. 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. Notice that such a speedup can be achieved only if the computer has multiple processing elements (such as CPUs or I/O channels).

• 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: (1) shared memory and (2) message passing. 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. In the messagepassing model, communication takes place by means of messages exchanged between the cooperating processes. The two communications models are contrasted in Figure.

Message passing is useful for exchanging smaller amounts of data, because no conflicts need be avoided. Message passing is also easier to implement than is shared memory for intercomputer communication. Shared memory allows maximum speed and convenience of communication, as it can be done at memory speeds when within a computer. Shared memory is faster than message passing, as message-passing systems are typically implemented using system calls and thus require the more time consuming task of kernel intervention. In contrast, 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.

**Shared-Memory Systems**

Interprocess communication using shared memory requires communicating processes to establish a region of shared memory. Typically, a shared-memory region resides in the address space of the process creating the shared-memory segment. Other processes that wish to communicate using this shared-memory segment must attach it to their address space. Recall that, normally, 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 form of the data and the location are determined by these 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.

Let's consider the producer-consumer problem, which is a common paradigm for cooperating processes. A producer process produces information that is consumed by a consumer process. For example, a compiler may produce assembly code, which is consumed by an assembler. The assembler, in turn, may produce object modules, which are consumed by the loader. The producer-consumer problem also provides a useful metaphor for the client-server paradigm. We generally think of a server as a producer and a client as a consumer. For example, a web server produces (that is, provides) HTML files and images, which are consumed (that is, read) by the client web browser requesting the resource.

One solution to the producer-consumer problem uses shared memory. To allow producer and consumer processes to run concurrently, we must have available a buffer of items that can be filled by the producer and emptied by the consumer. This buffer will reside in a region of memory that is shared by the producer and consumer processes. A producer can produce one item while the consumer is consuming another item. The producer and consumer must be synchronized, so that the consumer does not try to consume an item that has not yet been produced. Two types of buffers can be used. The unbounded buffer places no practical limit on the size of the buffer. The consumer may have to wait for new items, but the producer can always produce new items. The bounded buffer assumes a fixed buffer size. In this case, the consumer must wait if the buffer is empty, and the producer must wait if the buffer is full.

**Message-Passing Systems**

The scheme requires that these processes share a region of memory and that the code for accessing and manipulating the shared memory be written explicitly by the application programmer. Another way to achieve the same effect is for the operating system to provide the means for cooperating processes to communicate with each other via a message-passing facility. 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. For example, a chat program used on the World Wide Web could be designed so that chat participants communicate with one another by exchanging messages.

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. This is a common kind of tradeoff seen throughout operating system design. 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. This link can be implemented in a variety of ways. We are concerned here not with the link's physical implementation but rather with its logical implementation. Here are several methods for logically implementing a link and the send()/receive () operations:

• Direct or indirect communication

• Synchronous or asynchronous communication

• Automatic or explicit buffering

**Naming** '

Processes that want to communicate must have a way to refer to each other. They can use either direct or indirect communication. 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.

A communication link in this scheme has the following properties:

• A link is established automatically between every pair of processes that want to communicate. The processes need to know only each other's identity to communicate.

• A link is associated with exactly two processes.

• Between each pair of processes, there exists exactly one link.

This scheme exhibits symmetry in addressing; that is, both the sender process and the receiver process must name the other to communicate. A variant of this scheme employs asymmetry in addressing. Here, only the sender names the recipient; the recipient is not required to name the sender. In this scheme, the send() and receive () primitives are defined as follows:

• send(P, message)—Send a message to process P.

• receive(id, message)—-Receive a message from any process; the variable id is set to the name of the process with which communication has taken place.

The disadvantage in both of these schemes (symmetric and asymmetric) is the limited modularity of the resulting process definitions. Changing the identifier of a process may necessitate examining all other process definitions. All references to the old identifier must be found, so that they can be modified to the new identifier. In general, any such hard-coding techniques, where identifiers must be explicitly stated, are less desirable than techniques involving indirection, as described next. 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 a unique identification. For example, POSIX message queues use an integer value to identify a mailbox. In this scheme, a process can communicate with some other process via a number of different mailboxes. Two processes can communicate only if the processes have a shared mailbox, however. The send() and receive () primitives are defined as follows:

• send(A, message)—Send a message to mailbox A.

• receive(A, message)—Receive a message from mailbox A.

In this scheme, a communication link has the following properties:

• A link is established between a pair of processes only if both members of the pair have a shared mailbox.

• A link may be associated with more than two processes.

• Between each pair of communicating processes, there may be a number of different links, with each link corresponding to one mailbox.

Now suppose that processes P1, P2, and P3 all share mailbox A Process P1 sends a message to A, while both P2 and P3 execute a receive() from A Which process will receive the message sent by P1? The answer depends on which of the following methods we choose:

• Allow a link to be associated with two processes at most.

• Allow at most one process at a time to execute a receive () operation.

• Allow the system to select arbitrarily which process will receive the message (that is, either P2 or P3, but not both, will receive the message). The system also may define an algorithm for selecting which process will receive the message (that is, round robin where processes take turns receiving messages). The system may identify the receiver to the sender.

A mailbox may be owned either by a process or by the operating system. If the mailbox is owned by a process (that is, the mailbox is part of the address space of the process), then we distinguish between the owner (who can only receive messages through this mailbox) and the user (who can only send messages to the mailbox). Since each mailbox has a unique owner, there can be no confusion about who should receive a message sent to this mailbox. When a process that owns a mailbox terminates, the mailbox disappears. Any process that subsequently sends a message to this mailbox must be notified that the mailbox no longer exists. In contrast, a mailbox that is owned by the operating system has an existence of its own. It is independent and is not attached to any particular process. The operating system then must provide a mechanism that allows a process to do the following:

• Create a new mailbox.

• Send and receive messages through the mailbox.

• Delete a mailbox.

The process that creates a new mailbox is that mailbox's owner by default. Initially, the owner is the only process that can receive messages through this mailbox. However, the ownership and receiving privilege may be passed to other processes through appropriate system calls. Of course, this provision could result in multiple receivers for each mailbox.

***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 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.

Different combinations of send() and receive () are possible. When both sendQ and receive() are blocking, we have a rendezvous between the sender and the receiver. The solution to the producer-consumer problem becomes trivial when we use blocking send() and receive0 statements. The producer merely invokes the blocking send() call and waits until the message is delivered to either the receiver or the mailbox. Likewise, when the consumer invokes receive (), it blocks until a message is available. Note that the concepts of synchronous and asynchronous occur frequently in operating-system I/O algorithms,

***Buffering***

Whether communication is direct or indirect, messages exchanged by communicating processes reside in a temporary queue. Basically, such queues can be implemented in three ways:

• Zero capacity. The queue has a maximum length of zero; thus, the link cannot have any messages waiting in it. In this case, the sender must block until the recipient receives the message.

• Bounded capacity. The queue has finite length n; thus, at most n messages can reside in it. If the queue is not full when a new message is sent, the message is placed in the queue (either the message is copied or a pointer to the message is kept), and the sender can continue execution without waiting. The links capacity is finite, however. If the link is full, the sender must block until space is available in the queue.

• Unbounded capacity. The queues length is potentially infinite; thus, any number of messages can wait in it. The sender never blocks.

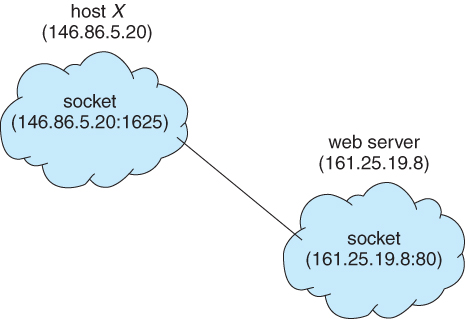
The zero-capacity case is sometimes referred to as a message system with no buffering; the other cases are referred to as systems with automatic buffering.

**Communication in Client-Server Systems**

In this section, we explore three other strategies for communication in client-server systems: sockets, remote procedure calls (RPCs), and Java's remote method invocation (RMI).

Sockets

A socket is defined as an endpoint for communication. A pair of processes communicating over a network employ a pair of sockets—one for each process. A socket is identified by an IP address concatenated with a port number. In general, sockets use a client-server architecture. The server waits for incoming client requests by listening to a specified port. Once a request is received, the server accepts a connection from the client socket to complete the connection. Servers implementing specific services (such as telnet, ftp, and http) listen to well-known ports (a telnet server listens to port 23, an ftp server listens to port 21, and a web, or http, server listens to port 80). All ports below 1024 are considered ivell known; we can use them to implement standard services. When a client process initiates a request for a connection, it is assigned a port by the host computer. This port is some arbitrary number greater than 1024. For example, if a client on host X with IP address 146.86.5.20 wishes to establish a connection with a web server (which is listening on port 80) at address 161.25.19.8, host X may be assigned port 1625. The connection will consist of a pair of sockets: (146.86.5.20:1625) on host X and (161.25.19.8:80) on the web server. This situation is illustrated in Figure . The packets traveling between the hosts are delivered to the appropriate process based on the destination port number. All connections must be unique. Therefore, if another process also on host X wished to establish another connection with the same web server, it would be assigned a port number greater than 1024 and not equal to 1625. This ensures that all connections consist of a unique pair of sockets.



Java provides three different types of sockets. Connection-oriented (TCP) sockets are implemented with the Socket class. Connectionless (UDP) sockets use the DatagramSocket class. Finally, the Mult icastSocket class is a subclass of the DatagramSocket class. A multicast socket allows data to be sent to multiple recipients.

Communication using sockets—although common and efficient—is considered a low-level form of communication between distributed processes. One reason is that sockets allow only an unstructured stream of bytes to be exchanged between the communicating threads. It is the responsibility of the client or server application to impose a structure on the data. In the next two subsections, we look at two higher-level methods of communication: remote procedure calls (RPCs) and remote method invocation (RMI).

**Remote Procedure Calls**

The RPC was designed as a way to abstract the procedure-call mechanism for use between systems with network connections. It is similar in many respects to the IPC mechanism described in Section 3.4, and it is usually built on top of such a system. Here, however, because we are dealing with an environment in which the processes are executing on separate systems, we must use a message-based communication scheme to provide remote service.

In contrast to the IPC facility, the messages exchanged in RPC communication are well structured and are thus no longer just packets of data. Each message is addressed to an RPC daemon listening to a port on the remote system, and each contains an identifier of the function to execute and the parameters to pass to that function. The function is then executed as requested, and any output is sent back to the requester in a separate message. A port is simply a number included at the start of a message packet. Whereas a system normally has one network address, it can have many ports within that address to differentiate the many network services it supports. If a remote process needs a service, it addresses a message to the proper port.

For instance, if a system wished to allow other systems to be able to list its current users, it would have a daemon supporting such an RPC attached to a port—say, port 3027. Any remote system could obtain the needed information (that is, the list of current users) by sending an RPC message to port 3027 on the server; the data would be received in a reply message.

The semantics of RPCs allow a client to invoke a procedure on a remote host as it would invoke a procedure locally. The RPC system hides the details that allow communication to take place by providing a stub on the client side. Typically, a separate stub exists for each separate remote procedure. When the client invokes a remote procedure, the RPC system calls the appropriate stub, passing it the parameters provided to the remote procedure. This stub locates the port on the server and marshals the parameters. Parameter marshalling involves packaging the parameters into a form that can be transmitted over a network. The stub then transmits a message to the server using message passing. A similar stub on the server side receives this message and invokes the procedure on the server. If necessary, return values are passed back to the client using the same technique.

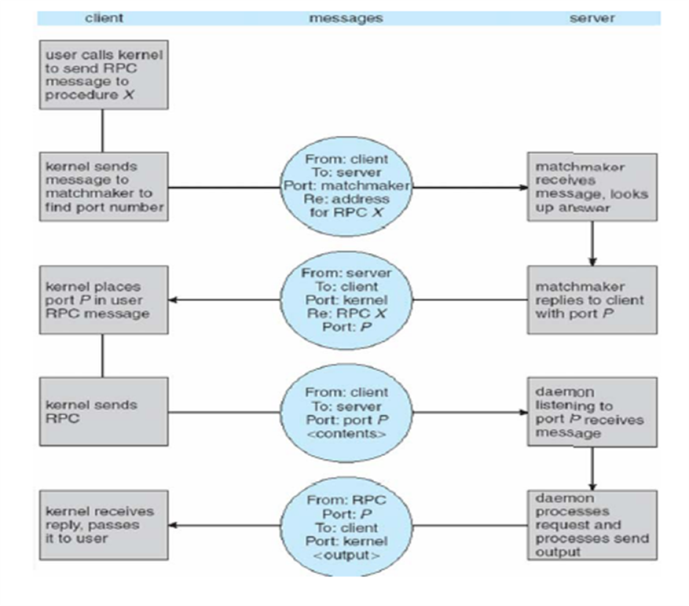
One issue that must be dealt with concerns differences in data representation on the client and server machines. Consider the representation of 32-bit integers. Some systems (known as big-endian) use the high memory address to store the most significant byte, while other systems (known as little-endian) store the least significant byte at the high memory address. To resolve differences like this, many RPC systems define a machine-independent representation of data. One such representation is known as external data representation (XDR). On the client side, parameter marshalling involves converting the machine dependent data into XDR before they are sent to the server. On the server side, the XDR data are unmarshalled and converted to the machine-dependent representation for the server. Another important issue involves the semantics of a call. Whereas local procedure calls fail only under extreme circumstances, RPCs can fail, or be duplicated and executed more than once, as a result of common network errors. One way to address this problem is for the operating system to ensure that messages are acted on exactly once, rather than at most once. Most local procedure calls have the "exactly once" functionality, but it is more difficult to implement.

First, consider "at most once". This semantic can be assured by attaching a timestamp to each message. The server must keep a history of all the timestamps of messages it has already processed or a history large enough to ensure that repeated messages are detected. Incoming messages that have a timestamp already in the history are ignored. The client can then send a message one or more times and be assured that it only executes once.

For "exactly once," we need to remove the risk that the server never receives the request. To accomplish this, the server must implement the "at most once" protocol described above but must also acknowledge to the client that the RPC call was received and executed. These ACK messages are common throughout networking. The client must resend each RPC call periodically until it receives the ACK for that call.

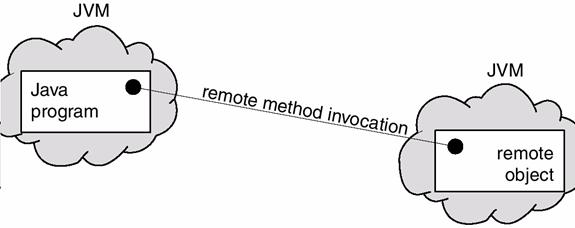
Another important issue concerns the communication between a server and a client. With standard procedure calls, some form of binding takes place during link, load, or execution time so that a procedure call's name is replaced by the memory address of the procedure call. The RPC scheme requires a similar binding of the client and the server port, but how does a client know the port numbers on the server? Neither system has full information about the other because they do not share memory.

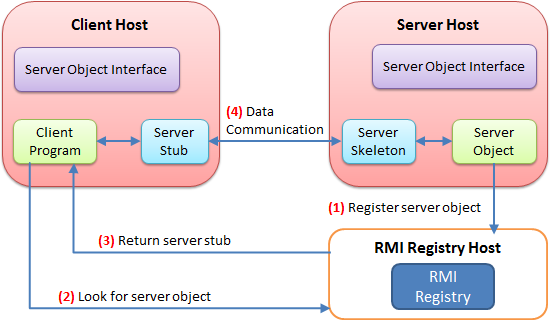
Two approaches are common. First, the binding information may be predetermined, in the form of fixed port addresses. At compile time, an RPC call has a fixed port number associated with it. Once a program is compiled, the server cannot change the port number of the requested service. Second, binding can be done dynamically by a rendezvous mechanism. Typically, an operating system provides a rendezvous (also called a matchmaker) daemon on a fixed RPC port. A client then sends a message containing the name of the RPC to the rendezvous daemon requesting the port address of the RPC it needs to execute. The port number is returned, and the RPC calls can be sent to that port until the process terminates (or the server crashes). This method requires the extra overhead of the initial request but is more flexible than the first approach. Figure, shows a sample interaction.

The RPC scheme is useful in implementing a distributed file system. Such a system can be implemented as a set of RPC daemons and clients. The messages are addressed to the distributed file system port on a server on which a file operation is to take place. The message contains the disk operation to be performed. The disk operation might be read, write, rename, delete, or status, corresponding to the usual file-related system calls. The return message contains any data resulting from that call, which is executed by the DFS daemon on behalf of the client. For instance, a message might contain a request to transfer a whole file to a client or be limited to a simple block request. In the latter case, several such requests may be needed if a whole file is to be transferred.

**Remote Method Invocation**

Remote method invocation (RMI) is a Java feature similar to RPCs. RMI allows a thread to invoke a method on a remote object. Objects are considered remote if they reside in a different Java virtual machine (JVM). Therefore, the remote object may be in a different JVM on the same computer or on a remote host connected by a network. This situation is illustrated in Figure.

RMI and RPCs differ in two fundamental ways. First, RPCs support procedural programming, whereby only remote procedures or functions can be called. In contrast, RMI is object-based: It supports invocation of methods on remote objects. Second, the parameters to remote procedures are ordinary data structures in RPC; with RMI, it is possible to pass objects as parameters to remote methods. By allowing a Java program to invoke methods on remote objects, RMI makes it possible for users to develop Java applications that are distributed across a network.

To make remote methods transparent to both the client and the server, RMI implements the remote object using stubs and skeletons. A stub is a proxy for the remote object; it resides with the client. When a client invokes a remote method, the stub for the remote object is called. This client-side stub is responsible for creating a parcel consisting of the name of the method to be invoked on the server and the marshalled parameters for the method. The stub then sends this parcel to the server, where the skeleton for the remote object receives it. The skeleton is responsible for unmarshalling the parameters and invoking the desired method on the server. The skeleton then marshals the return value (or exception, if any) into a parcel and returns this parcel to the client. The stub unmarshals the return value and passes it to the client.

Few rules about the behavior of parameter passing.

• If the marshalled parameters are local (or nonremote) objects, they are passed by copy using a technique known as object serialization. However, if the parameters are also remote objects, they are passed by reference. In our example, if A is a local object and B a remote object, A is serialized and passed by copy, and B is passed by reference. This in turn allows the server to invoke methods on B remotely.

• If local objects are to be passed as parameters to remote objects, they must implement the interface java.io.Serializable. Many objects in the core Java API implement Serializable, allowing them to be used with RMI. Object serialization allows the state of an object to be written to a byte stream.

**Brief of RPC and RMI**

**Definition of RPC**

Remote Procedure Call (RPC) is a programming language feature devised for the distributed computing and based on semantics of local procedure calls. It is the most common forms of remote service and was designed as a way to abstract the procedure call mechanism to use between systems connected through a network. It is similar to IPC mechanism where the operating system allows the processes to manage shared data and deal with an environment where different processes are executing on separate systems and necessarily require message-based communication.

Let’s understand how RPC is implemented through the given steps:

* The client process calls the client stub with parameters, and its execution is suspended until the call is completed.
* The parameters are then translated into machine-independent form by marshalling through client stub. Then the message is prepared which contain the representation of the parameters.
* To find the identity of the site the client stub intercommunicate with name server at which remote procedure exists.
* Using blocking protocol the client stub sends the message to the site where remote procedure call exists. This step halt the client stub until it gets a reply.RPC mechanism
* The server site receives the message sent from the client side and converts it into machine specific format.
* Now server stub executes a call on the server procedure along with the parameters, and the server stub is discontinued till the procedure gets completed.
* The server procedure returns the generated results to the server stub, and the results get converted into machine-independent format at server stub and create a message containing the results.
* The result message is sent to the client stub which is converted back into machine specific format suitable for the client stub.
* At last client, stub returns the results to the client process.

**Definition of RMI**

Remote Method Invocation (RMI) is similar to RPC but is language specific and a feature of java. A thread is permitted to call the method on a remote object. To maintain the transparency on the client and server side, it implements remote object using stubs and skeletons. The stub resides with the client and for the remote object it behaves as a proxy.

When a client calls a remote method, the stub for the remote method is called. The client stub is accountable for creating and sending the parcel containing the name of a method and the marshalled parameters, and the skeleton is responsible for receiving the parcel.RMI mechanismThe skeleton unmarshals parameters and invokes the desired method on the server. The skeleton marshals the given value (or exceptions) with the parcel and sends it to client stub. The stub reassembles the return parcel and sends it to the client.

In Java, the parameters are passed to methods and returned in the form of reference. This could be troublesome for RMI service since not all objects are possibly remote methods. So, it must determine which could be passed as reference and which could not.

Java uses process named as serialisation where the objects are passed as value. The remote object is localised by pass by value. It can also pass an object by reference through passing a remote reference to the object along with the URL of the stub class. Pass by reference restricts a stub for the remote object.

**Key Differences Between RPC and RMI**

* RPC supports procedural programming paradigms thus is C based, while RMI supports object-oriented programming paradigms and is java based.
* The parameters passed to remote procedures in RPC are the ordinary data structures. On the contrary, RMI transits objects as a parameter to the remote method.
* RPC can be considered as the older version of RMI, and it is used in the programming languages that support procedural programming, and it can only use pass by value method. As against, RMI facility is devised based on modern programming approach, which could use pass by value or reference. Another advantage of RMI is that the parameters passed by reference can be changed.
* RPC protocol generates more overheads than RMI.
* The parameters passed in RPC must be “in-out” which means that the value passed to the procedure and the output value must have the same datatypes. In contrast, there is no compulsion of passing “in-out” parameters in RMI.
* In RPC, references could not be probable because the two processes have the distinct address space, but it is possible in case of RMI.

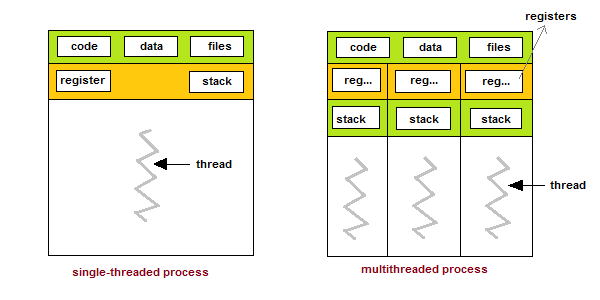
**Chapter-4**

**Multithread Programming**

### What are Threads?

Thread is an execution unit which consists of its own program counter, a stack, and a set of registers. Threads are also known as Lightweight processes. Threads are popular way to improve application through parallelism. The CPU switches rapidly back and forth among the threads giving illusion that the threads are running in parallel.

As each thread has its own independent resource for process execution, multpile processes can be executed parallely by increasing number of threads.



#### Types of Thread

There are two types of threads :

* User Threads
* Kernel Threads

**User threads**, are above the kernel and without kernel support. These are the threads that application programmers use in their programs.

**Kernel threads** are supported within the kernel of the OS itself. All modern OSs support kernel level threads, allowing the kernel to perform multiple simultaneous tasks and/or to service multiple kernel system calls simultaneously.

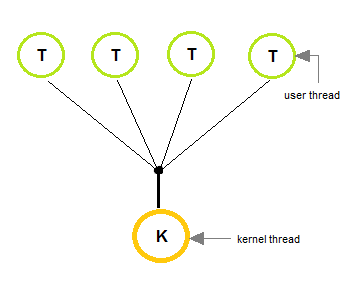
#### Multithreading Models

The user threads must be mapped to kernel threads, by one of the following strategies.

* Many-To-One Model
* One-To-One Model
* Many-To-Many Model

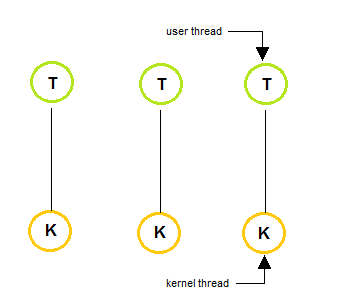
#### Many-To-One Model

* In the many-to-one model, many user-level threads are all mapped onto a single kernel thread.
* Thread management is handled by the thread library in user space, which is efficient in nature.



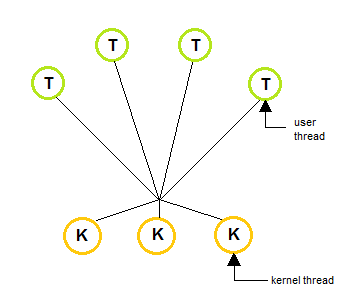
#### One-To-One Model

* The one-to-one model creates a separate kernel thread to handle each and every user thread.
* Most implementations of this model place a limit on how many threads can be created.
* Linux and Windows from 95 to XP implement the one-to-one model for threads.



#### Many-To-Many Model

* The many-to-many model multiplexes any number of user threads onto an equal or smaller number of kernel threads, combining the best features of the one-to-one and many-to-one models.
* Users can create any number of the threads.
* Blocking the kernel system calls does not block the entire process.
* Processes can be split across multiple processors.



#### Thread Libraries

Thread libraries provides programmers with API for creating and managing of threads.

Thread libraries may be implemented either in user space or in kernel space. The user space involves API functions implemented solely within user space, with no kernel support. The kernel space involves system calls, and requires a kernel with thread library support.

**There are three types of thread :**

* POSIX Pitheads, may be provided as either a user or kernel library, as an extension to the POSIX standard.
* Win32 threads, are provided as a kernel-level library on Windows systems.
* Java threads - Since Java generally runs on a Java Virtual Machine, the implementation of threads is based upon whatever OS and hardware the JVM is running on, i.e. either Pitheads or Win32 threads depending on the system

#### Benefits of Multithreading

1. Responsiveness
2. Resource sharing, hence allowing better utilization of resources.
3. Economy. Creating and managing threads becomes easier.
4. Scalability. One thread runs on one CPU. In Multithreaded processes, threads can be distributed over a series of processors to scale.
5. Context Switching is smooth. Context switching refers to the procedure followed by CPU to change from one task to another.

#### Multithreading Issues

1. **Thread Cancellation**.

Thread cancellation means terminating a thread before it has finished working. There can be two approaches for this, one is **Asynchronous cancellation**, which terminates the target thread immediately. The other is **Deferred cancellation** allows the target thread to periodically check if it should be cancelled.

1. **Signal Handling**.

Signals are used in UNIX systems to notify a process that a particular event has occurred. Now in when a Multithreaded process receives a signal, to which thread it must be delivered? It can be delivered to all, or a single thread.

1. **fork() System Call**.

fork() is a system call executed in the kernel through which a process creates a copy of itself. Now the problem in Multithreaded process is, if one thread forks, will the entire process be copied or not?

1. **Security Issues** because of extensive sharing of resources between multiple threads.

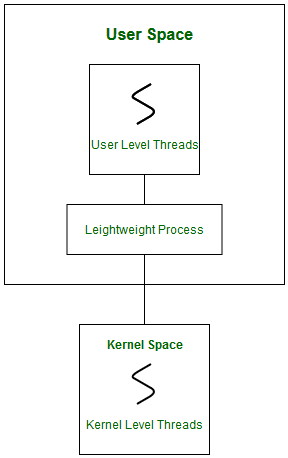
There are many other issues that you might face in a multithreaded process, but there are appropriate solutions available for them. Pointing out some issues here was just to study both sides of the coin.

Scheduling of [threads](https://www.geeksforgeeks.org/threads-and-its-types-in-operating-system/) involves two boundary scheduling,

* Scheduling of user level threads (ULT) to kernel level threads (KLT) via leightweight process (LWP) by the application developer.
* Scheduling of kernel level threads by the system scheduler to perform different unique os functions.

**Leightweight Process (LWP) :**  
Light-weight process are threads in the user space that acts as an interface for the ULT 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 LWP created by the thread library depends on the type of application. In the case of an I/O bound application, the number of LWP depends on the number of user-level threads. This is because when an LWP is blocked on an I/O operation, then to invoke the other ULT the thread library needs to create and schedule another LWP. Thus, in an I/O bound application, the number of LWP is equal to the number of the ULT. In the case of a CPU bound application, it depends only on the application. Each LWP is attached to a separate kernel-level thread.

Attention reader! Don’t stop learning now. Get hold of all the important CS Theory concepts for SDE interviews with the [**CS Theory Course**](https://practice.geeksforgeeks.org/courses/SDE-theory?vC=1) at a student-friendly price and become industry ready.



In real-time, the first boundary of thread scheduling is beyond specifying the scheduling policy and the priority. It requires two controls to be specified for the User level threads: Contention scope, and Allocation domain. These are explained as following below.

**1. Contention Scope :**  
The word contention here refers to the competition or fight among the User level threads to access the kernel resources. Thus, this control defines the extent to which contention takes place. It is defined by the application developer using the thread library. Depending upon the extent of contention it is classified as **Process Contention Scope** and **System Contention Scope**.

1. **Process Contention Scope (PCS) –**  
   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 (priority as specified by the application developer during thread creation).
2. **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.

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.

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. It takes two values.

PTHREAD\_SCOPE\_SYSTEM

PTHREAD\_SCOPE\_PROCESS

If the scope value specified is not supported by the system, then the function returns *ENOTSUP*.

**2. Allocation Domain :**  
The allocation domain is **a set of one or more resources** for which a thread is competing. In a multicore system, there may be one or more allocation domains where each consists of one or more cores. One ULT can be a part of one or more allocation domain. Due to this high complexity in dealing with hardware and software architectural interfaces, this control is not specified. But by default, the multicore system will have an interface that affects the allocation domain of a thread.