Operating System

CS604

Delivered by

Dr. Syed Mansoor Sarwar

Virtual University of Pakistan

Knowledge beyond the boundaries

TABLE OF CONTENTS

Operating Systems
Lecture No. 1

Reading Material

- Operating Systems Concepts, Chapter 1
- PowerPoint Slides for Lecture 1

Summary

- Introduction and purpose of the course
- Organization of a computer system
- Purpose of a computer system
- Requirements for achieving the purpose – Setting the stage for OS concepts and principles
- Outline of topics to be discussed
- What is an Operating System?

Organization of a Computer System

As shown in Figure 1.1, the major high-level components of a computer system are:

1. **Hardware**, which provides basic computing resources (CPU, memory, I/O devices).
2. **Operating system**, which manages the use of the hardware among the various application programs for the various users and provides the user a relatively simple machine to use.
3. **Applications programs** that define the ways in which system resources are used to solve the computing problems of the users (compilers, database systems, video games, business programs).
4. **Users,** which include people, machines, other computers.

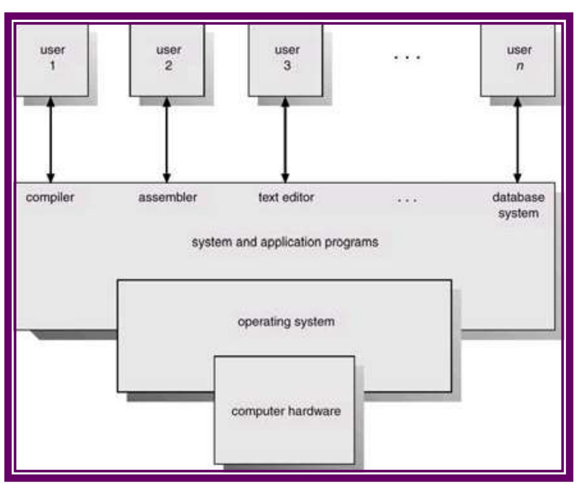


Figure 1.1. High-level components of a computer system

Purpose of a Computer—Setting the Stage for OS Concepts and Principles

Computer systems consist of software and hardware that are combined to provide a tool to implement solutions for specific problems in an efficient manner and to execute programs. Figure 1.2 shows the general organization of a contemporary computer system and how various system components are interconnected.

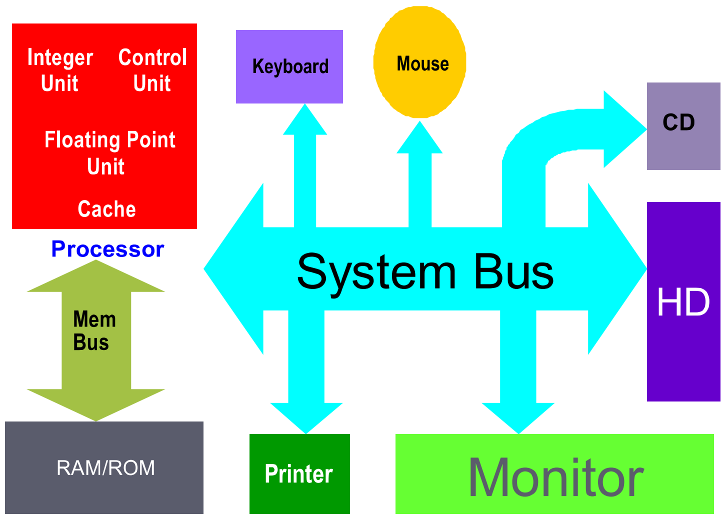


Figure 1.2. Organization of a Computer System

Viewing things closely will reveal that the primary purpose of a computer system is to generate executable programs and execute them. The following are some of the main issues involved in performing these tasks.

1. Storing an executable on a secondary storage device such as hard disk
2. Loading executable from disk into the main memory
3. Setting the CPU state appropriately so that program execution could begin
4. Creating multiple cooperating processes, synchronizing their access to shared data, and allowing them to communicate with each other

The above issues require the operating system to provide the following services and much more:

- Manage secondary storage devices
    - Allocate appropriate amount of disk space when files are created
    - Deallocate space when files are removing
    - Insure that a new file does not overwrite an existing file
    - Schedule disk requests
- Manage primary storage
    - Allocate appropriate amount of memory space when programs are to be loaded into the memory for executing
    - Deallocate space when processes terminate
    - Insure that a new process is not loaded on top of an existing process
    - Insure that a process does not access memory space that does not belong to it
    - Minimize the amount of unused memory space
    - Allow execution of programs larger in size than the available main memory
- Manage processes

- Allow simultaneous execution of processes by scheduling the CPU(s)
- Prevent deadlocks between processes
- Insure integrity of shared data
- Synchronize executions of cooperating processes
- Allow a user to manage his/her files and directories properly
  - User view of directory structure
  - Provide a mechanism that allows users to protect their files and directories

In this course, we will discuss in detail these operating system services (and more), with a particular emphasis on the UNIX and Linux operating systems. See the course outline for details of topics and lecture schedule.

What is an Operating System?

There are two views about this. The top-down view is that it is a program that acts as an intermediary between a user of a computer and the computer hardware, and makes the computer system convenient to use. It is because of the operating system that users of a computer system don't have to deal with computer's hardware to get their work done. Users can use simple commands to perform various tasks and let the operating system do the difficult work of interacting with computer hardware. Thus, you can use a command like `copy file1 file2` to copy 'file1' to 'file2' and let the operating system communicate with the controller(s) of the disk that contain(s) the two files.

A computer system has many hardware and software resources that may be required to solve a problem: CPU time, memory space, file storage space, I/O devices etc. The operating system acts as the manager of these resources, facing numerous and possibly conflicting requests for resources, the operating system must decide how (and when) to allocate (and deallocate) them to specific programs and users so that it can operate the computer system efficiently, fairly, and securely. So, the bottom-up view is that operating system is a resource manager who manages the hardware and software resources in the computer system.

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

Operating Systems
Lecture No. 2

Reading Material
- Operating Systems Concepts, Chapter 1
- PowerPoint Slides for Lecture 2

Summary
- Single-user systems
- Batch systems
- Multi programmed systems
- Time-sharing systems
- Real time systems
- Interrupts, traps and software interrupts (UNIX signals)
- Hardware protection

Single-user systems
A computer system that allows only one user to use the computer at a given time is known as a **single-user system**. The goals of such systems are maximizing user convenience and responsiveness, instead of maximizing the utilization of the CPU and peripheral devices. Single-user systems use I/O devices such as keyboards, mice, display screens, scanners, and small printers. They can adopt technology developed for larger operating systems. Often individuals have sole use of computer and do not need advanced CPU utilization and hardware protection features. They may run different types of operating systems, including DOS, Windows, and MacOS. Linux and UNIX operating systems can also be run in single-user mode.

Batch Systems
Early computers were large machines run from a console with card readers and tape drives as input devices and line printers, tape drives, and card punches as output devices. The user did not interact directly with the system; instead the user prepared a job, (which consisted of the program, data, and some control information about the nature of the job in the form of control cards) and submitted this to the computer operator. The job was in the form of punch cards, and at some later time the output was generated by the system—user didn't get to interact with his/her job. The output consisted of the result of the program, as well as a dump of the final memory and register contents for debugging.

To speed up processing, operators batched together jobs with similar needs, and ran them through the computer as a group. For example, all FORTRAN programs were complied one after the other. The major task of such an operating system was to transfer control automatically from one job to the next. In this execution environment, the CPU is often idle because the speeds of the mechanical I/O devices such as a tape drive are slower than that of electronic devices. Such systems in which the user does not get to interact with his/her jobs and jobs with similar needs are executed in a "batch", one after the other, are known as **batch systems**. Digital Equipment Corporation's VMS is an example of a batch operating system.

Figure 2.1 shows the memory layout of a typical computer system, with the **system space** containing operating system code and data currently in use and the **user space** containing user programs (processes). In case of a batch system, the user space contains one process at a time because only one process is executing at a given time.

Figure 2.1 Memory partitioned into user and system spaces

Multi-programmed Systems

Multi-programming increases CPU utilization by organizing jobs so that the CPU always has one to execute. The operating system keeps several jobs in memory simultaneously, as shown in Figure 2.2. This set of jobs is a subset of the jobs on the disk which are ready to run but cannot be loaded into memory due to lack of space. Since the number of jobs that can be kept simultaneously in memory is usually much smaller than the number of jobs that can be in the job pool; the operating system picks and executes one of the jobs in the memory. Eventually the job has to wait for some task such as an I/O operation to complete. In a non multi-programmed system, the CPU would sit idle. In a multi-programmed system, the operating system simply switches to, and executes another job. When that job needs to wait, the CPU simply switches to another job and so on.

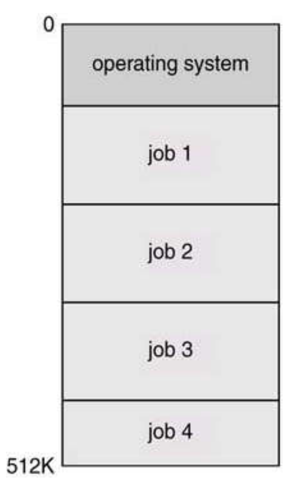


Figure 2.2 Memory layout for a multi-programmed batch system

Figure 2.3 illustrates the concept of multiprogramming by using an example system with two processes, P1 and P2. The CPU is switched from P1 to P2 when P1 finishes its CPU burst and needs to wait for an event, and vice versa when P2 finishes it CPU burst and has to wait for an event. This means that when one process is using the CPU, the other is waiting for an event (such as I/O to complete). This increases the utilization of the CPU and I/O devices as well as throughput of the system. In our example below, P1 and P2 would finish their execution in 10 time units if no multiprogramming is used and in six time units if multiprogramming is used.

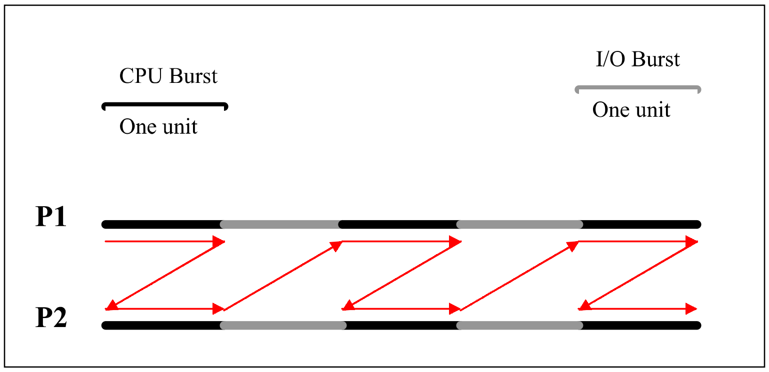


Figure 2.3 Illustration of the multiprogramming concept

All jobs that enter the system are kept in the **job pool**. This pool consists of all processes residing on disk awaiting allocation of main memory. If several jobs are ready to be brought into memory, and there is not enough room for all of them, then the system must choose among them. This decision is called *job scheduling*. In addition if several jobs are ready to run at the same time, the system must choose among them. We will discuss CPU scheduling in Chapter 6.

Time-sharing systems

A **time-sharing system** is multi-user, multi-process, and interactive system. This means that it allows multiple users to use the computer simultaneously. A user can run one or more processes at the same time and interact with his/her processes. A time-shared system uses multiprogramming and CPU scheduling to provide each user with a small portion of a time-shared computer. Each user has at least one separate program in memory. To obtain a reasonable response time, jobs may have to be swapped in and out of main memory. UNIX, Linux, Widows NT server, and Windows 2000 server are time-sharing systems. We will discuss various elements of time-sharing systems throughout the course.

Real time systems

Real time systems are used when rigid time requirements are placed on the operation of a processor or the flow of data; thus it is often used as a control device in a dedicated application. Examples are systems that control scientific experiments, medical imaging systems, industrial control systems and certain display systems.

A **real time system** has well defined, fixed time constraints, and if the system does not produce output for an input within the time constraints, the system will fail. For instance, it would not do for a robot arm to be instructed to halt after it had smashed into the car it was building.

Real time systems come in two flavors: hard and soft. A **hard real time system** guarantees that critical tasks be completed on time. This goal requires that all delays in the system be completed on time. This goal requires that all delays in the system be bounded, from the retrieval of stored data to the time it takes the operating system to finish any request made of it. Secondary storage of any sort is usually limited or missing, with data instead being stored in short-term memory or in read only memory. Most advanced operating system features are absent too, since they tend to separate the user from the hardware, and that separation results in uncertainty about the amount of time an operation will take.

A less restrictive type of real time system is a **soft real time system**, where a critical real-time task gets priority over other tasks, and retains that priority until it completes. As in hard real time systems, the operating system kernel delays need to be bounded. Soft real time is an achievable goal that can be mixed with other types of systems, whereas hard real time systems conflict with the operation of other systems such as time-sharing systems, and the two cannot be mixed.

Interrupts, traps and software interrupts

An **interrupt** is a signal generated by a hardware device (usually an I/O device) to get CPU's attention. Interrupt transfers control to the **interrupt service routine** (ISR), generally through the **interrupt vector table**, which contains the addresses of all the service routines. The interrupt service routine executes; on completion the CPU resumes the interrupted computation. Interrupt architecture must save the address of the interrupted instruction. Incoming interrupts are disabled while another interrupt is being processed to prevent a *lost interrupt*. An operating system is an interrupt driven software.

A **trap** (or an *exception)* is a software-generated interrupt caused either by an error (division by zero or invalid memory access) or by a user request for an operating system service.

A **signal** is an event generated to get attention of a process. An example of a signal is the event that is generated when you run a program and then press `<Ctrl-C>`. The signal generated in this case is called SIGINT (Interrupt signal). Three actions are possible on a signal:

1. Kernel-defined default action—which usually results in process termination and, in some cases, generation of a 'core' file that can be used the programmer/user to know the state of the process at the time of its termination.
2. Process can intercept the signal and ignore it.
3. Process can intercept the signal and take a programmer-defined action.

We will discuss signals in detail in some of the subsequent lectures.

Hardware Protection

Multi-programming put several programs in memory at the same time; while this increased system utilization it also increased problems. With sharing, many processes could be adversely affected by a bug in one program. One erroneous program could also modify the program or data of another program or even the resident part of the operating system. A file may overwrite another file or folder on disk. A process may get the CPU and never relinquish it. So the issues of hardware protection are: I/O protection, memory protection, and CPU protection. We will discuss them one by one, but first we talk about the dual-mode operation of a CPU.

a) Dual Mode Operation

To ensure proper operation, we must protect the operating system and all other programs and their data from any malfunctioning program. Protection is needed for any shared resources. Instruction set of a modern CPU has two kinds of instructions, privileged instructions and non-privileged instructions. Privileged instructions can be used to perform hardware operations that a normal user process should not be able to perform, such as communicating with I/O devices. If a user process tries to execute a privileged instruction, a trap should be generated and process should be terminated prematurely. At the same time, a piece of operating system code should be allowed to execute privileged instructions. In order for the CPU to be able to differentiate between a user process and an operating system code, we need two separate modes of operation: user mode and monitor mode (also called supervisor mode, system mode, or privileged mode). A bit, called the **mode bit**, is added to the hardware of the computer to indicate the current mode: **monitor mode** (0) or **user mode** (1). With the mode bit we are able to distinguish between a task that is executed on behalf of the operating system and one that is executed on behalf of the user.

The concept of privileged instructions also provides us with the means for the user to interact with the operating system by asking it to perform some designated tasks that only the operating system should do. A user process can request the operating system to perform such tasks for it by executing a **system call**. Whenever a system call is made or an interrupt, trap, or signal is generated, CPU mode is switched to system mode before the relevant kernel code executes. The CPU mode is switched back to user mode before the control is transferred back to the user process. This is illustrated by the diagram in Figure 2.4.

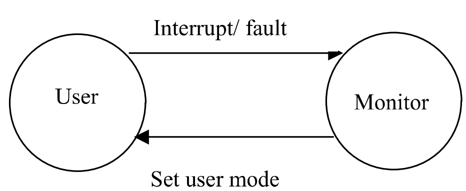


Figure 2.4 The dual-mode operation of the CPU

b) I/O Protection

A user process may disrupt the normal operation of the system by issuing illegal I/O instructions, by accessing memory locations within the operating system itself, or by refusing to relinquish the CPU. We can use various mechanisms to ensure that such disruptions cannot take place in the system.

To prevent users from performing illegal I/O, we define all I/O instructions to be privileged instructions. Thus users cannot issue I/O instructions directly; they must do it through the operating system. For I/O protection to be complete, we must be sure that a user program can never gain control of the computer in monitor mode. If it could, I/O protection could be compromised.

Consider a computer executing in user mode. It will switch to monitor mode whenever an interrupt or trap occurs, jumping to the address determined from the interrupt from the interrupt vector. If a user program, as part of its execution, stores a new address in the interrupt vector, this new address could overwrite the previous address with an address in the user program. Then, when a corresponding trap or interrupt occurred, the hardware would switch to monitor mode and transfer control through the modified interrupt vector table to a user program, causing it to gain control of the computer in monitor mode. Hence we need all I/O instructions and instructions for changing the contents of the system space in memory to be protected. A user process could request a privileged operation by executing a system call such as read (for reading a file).

Operating Systems
Lecture No. 3

Reading Material
- Computer System Structures, Chapter 2
- Operating Systems Structures, Chapter 3
- PowerPoint Slides for Lecture 3

Summary
- Memory and CPU protection
- Operating system components and services
- System calls
- Operating system structures

Memory Protection
The region in the memory that a process is allowed to access is known as **process address space**. To ensure correct operation of a computer system, we need to ensure that a process cannot access memory outside its address space. If we don't do this then a process may, accidentally or deliberately, overwrite the address space of another process or memory space belonging to the operating system (e.g., for the interrupt vector table).

Using two CPU registers, specifically designed for this purpose, can provide memory protection. These registered are:
- Base register – it holds the smallest legal physical memory address for a process
- Limit register – it contains the size of the process

When a process is loaded into memory, the base register is initialized with the starting address of the process and the limit register is initialized with its size. Memory outside the defined range is protected because the CPU checks that every address generated by the process falls within the memory range defined by the values stored in the base and limit registers, as shown in Figure 3.1.

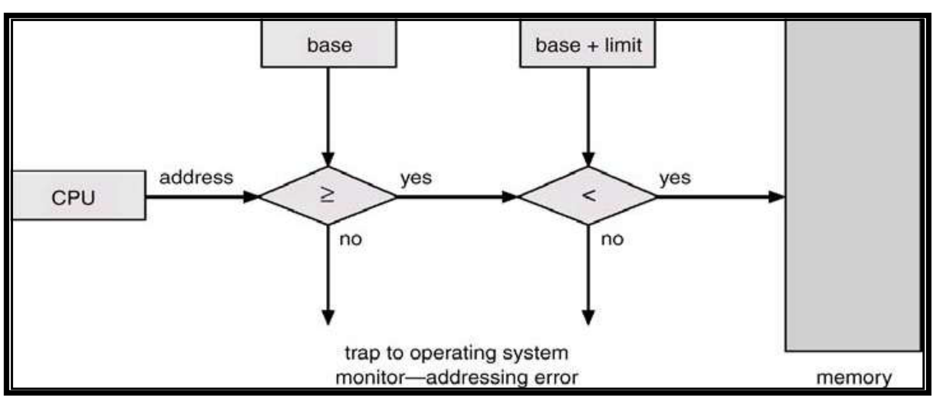


Figure 3.1 Hardware address protection with base and limit registers

In Figure 3.2, we use an example to illustrate how the concept outlined above works. The base and limit registers are initialized to define the address space of a process. The process starts at memory location 300040 and its size is 120900 bytes (assuming that memory is byte addressable). During the execution of this process, the CPU insures (by using the logic outlined in Figure 3.1) that all the addresses generated by this process are greater than or equal to 300040 and less than (300040+120900), thereby preventing this process to access any memory area outside its address space. Loading the base and limit registers are privileged instructions.

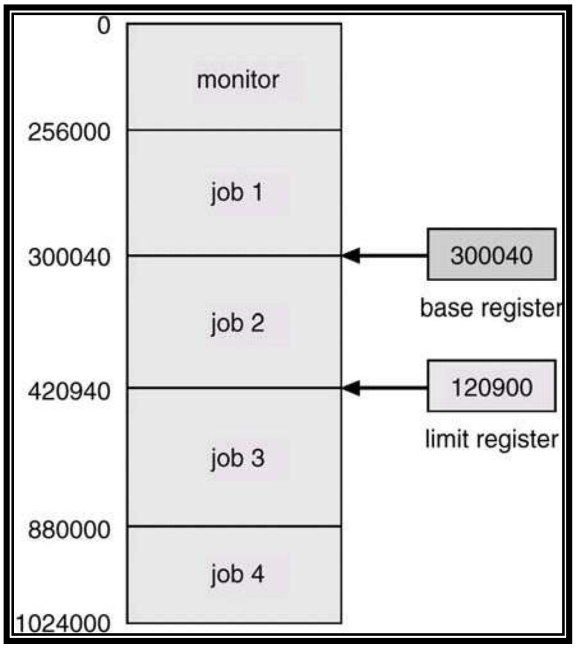


Figure 3.2 Use of Base and Limit Register

CPU Protection

In addition to protecting I/O and memory, we must ensure that the operating system maintains control. We must prevent the user program from getting stuck in an infinite loop or not calling system services and never returning control to the CPU. To accomplish this we can use a **timer**, which interrupts the CPU after specified period to ensure that the operating system maintains control. The timer period may be variable or fixed. A *fixed-rate clock* and a *counter* are used to implement a variable timer. The OS initializes the counter with a positive value. The counter is decremented every clock tick by the clock interrupt service routine. When the counter reaches the value 0, a timer interrupt is generated that transfers control from the current process to the next scheduled process. Thus we can use the timer to prevent a program from running too long. In the most straight forward case, the timer could be set to interrupt every N milliseconds, where N is the **time slice** that each process is allowed to execute before the next process gets control of the CPU. The OS is invoked at the end of each time slice to perform various housekeeping tasks. This issue is discussed in detail under CPU scheduling in Chapter 7.

Another use of the timer is to compute the current time. A timer interrupt signals the passage of some period, allowing the OS to compute the current time in reference to some initial time. Load-timer is a privileged instruction.

OS Components

An operating system has many components that manage all the resources in a computer system, insuring proper execution of programs. We briefly describe these components in this section.

■ Process management

A process can be thought of as a program in execution. It needs certain resources, including CPU time, memory, files and I/O devices to accomplish its tasks. The operating system is responsible for:

- Creating and terminating both user and system processes
- Suspending and resuming processes
- Providing mechanisms for process synchronization
- Providing mechanisms for process communication
- Providing mechanisms for deadlock handling

■ Main memory management

Main memory is a large array of words or bytes (called memory locations), ranging in size from hundreds of thousands to billions. Every word or byte has its own address. Main memory is a repository of quickly accessible data shared by the CPU and I/O devices. It contains the code, data, stack, and other parts of a process. The central processor reads instructions of a process from main memory during the machine cycle—fetch-decode-execute.

The OS is responsible for the following activities in connection with memory management:

- Keeping track of free memory space
- Keeping track of which parts of memory are currently being used and by whom
- Deciding which processes are to be loaded into memory when memory space becomes available
- Deciding how much memory is to be allocated to a process
- Allocating and deallocating memory space as needed
- Insuring that a process is not overwritten on top of another

■ Secondary storage management

The main purpose of a computer system is to execute programs. The programs, along with the data they access, must be in the main memory or **primary storage** during their execution. Since main memory is too small to accommodate all data and programs, and because the data it holds are lost when the power is lost, the computer system must provide **secondary storage** to backup main memory. Most programs are stored on a disk until loaded into the memory and then use disk as both the source and destination of their processing. Like all other resources in a computer system, proper management of disk storage is important.

The operating system is responsible for the following activities in connection with disk management:

- Free-space management

- Storage allocation and deallocation
- Disk scheduling

■ I/O system management

The I/O subsystem consists of:

- A memory management component that includes buffering, caching and spooling
- A general device-driver interface
- Drivers for specific hardware devices

■ File management

Computers can store information on several types of physical media, e.g. magnetic tape, magnetic disk and optical disk. The OS maps files onto physical media and accesses these media via the storage devices.

The OS is responsible for the following activities with respect to file management:

- Creating and deleting files
- Creating and deleting directories
- Supporting primitives (operations) for manipulating files and directories
- Mapping files onto the secondary storage
- Backing up files on stable (nonvolatile) storage media

■ Protection system

If a computer system has multiple users and allows concurrent execution of multiple processes then the various processes must be protected from each other's activities.

Protection is any mechanism for controlling the access of programs, processes or users to the resources defined by a computer system.

■ Networking

A **distributed system** is a collection of processors that do not share memory, peripheral devices or a clock. Instead, each processor has it own local memory and clock, and the processors communicate with each other through various communication lines, such as high- speed buses or networks.

The processors in a communication system are connected through a **communication network**. The communication network design must consider message routing and connection strategies and the problems of contention and security.

A distributed system collects physically separate, possibly heterogeneous, systems into a single coherent system, providing the user with access to the various resources that the system maintains.

■ Command-line interpreter (shells)

One of the most important system programs for an operating system is the **command interpreter**, which is the interface between the user and operating system. Its purpose is to read user commands and try to execute them. Some operating systems include the command interpreter in the kernel. Other operating systems (e.g. UNIX, Linux, and DOS) treat it as a special program that runs when a job is initiated or when a user first logs on (on time sharing systems). This program is sometimes called the **command-line interpreter** and is often known as the **shell**. Its function is simple: to get the next command statement and execute it. Some of the famous shells for UNIX and Linux are

Bourne shell (sh), C shell (csh), Bourne Again shell (bash), TC shell (tcsh), and Korn shell (ksh). You can use any of these shells by running the corresponding command, listed in parentheses for each shell. So, you can run the Bourne Again shell by running the `bash` or `/usr/bin/bash` command.

Operating System Services

An operating system provides the environment within which programs are executed. It provides certain services to programs and users of those programs, which vary from operating system to operating system. Some of the common ones are:

- **Program execution:** The system must be able to load a program into memory and to run that programs. The program must be able to end its execution.
- **I/O Operations:** A running program may require I/O, which may involve a file or an I/O device. For efficiency and protection user usually cannot control I/O devices directly. The OS provides a means to do I/O.
- **File System Manipulation:** Programs need to read, write files. Also they should be able to create and delete files by name.
- **Communications:** There are cases in which one program needs to exchange information with another process. This can occur between processes that are executing on the same computer or between processes that are executing on different computer systems tied together by a computer network. Communication may be implemented via shared **memory** or **message passing**.
- **Error detection:** The OS constantly needs to be aware of possible errors. Error may occur in the CPU and memory hardware, in I/O devices and in the user program. For each type of error, the OS should take appropriate action to ensure correct and consistent computing.

In order to assist the efficient operation of the system itself, the system provides the following functions:

- **Resource allocation:** When multiple users are logged on the system or multiple jobs are running at the same time, resources must be allocated to each of them. There are various routines to schedule jobs, allocate plotters, modems and other peripheral devices.
- **Accounting:** We want to keep track of which users use how many and which kinds of computer resources. This record keeping may be used for accounting or simply for accumulating usage statistics.
- **Protection:** The owners of information stored in a multi user computer system may want to control use of that information. When several disjointed processes execute concurrently it should not b possible for one process to interfere with the others or with the operating system itself. Protection involves ensuring that all access to system resources is controlled.

Entry Points into Kernel

As shown in Figure 3.3, there are four events that cause execution of a piece of code in the kernel. These events are: interrupt, trap, system call, and signal. In case of all of these events, some kernel code is executed to service the corresponding event. You have discussed interrupts and traps in the computer organization or computer architecture course. We will discuss system calls execution in this lecture and signals subsequent lectures. We will talk about many UNIX and Linux system calls and signals throughout the course.

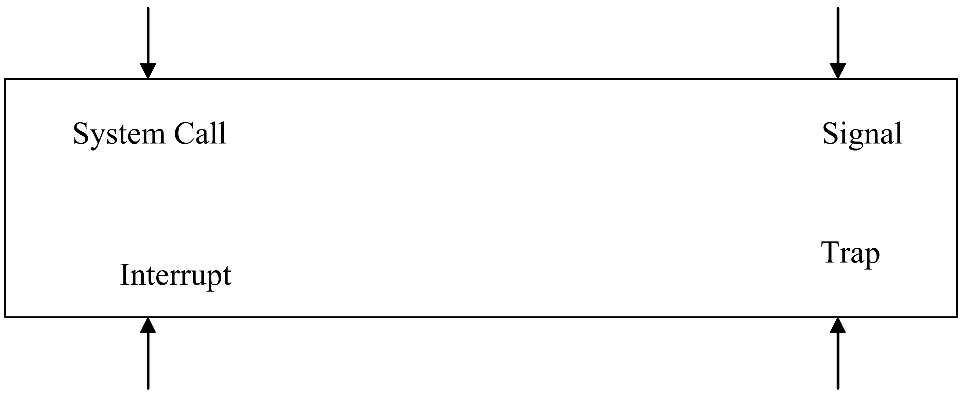


Figure 3.3 Entry points into the operating system kernel

System Calls

System calls provide the interface between a process and the OS. These calls are generally available as assembly language instructions. The system call interface layer contains entry point in the kernel code; because all system resources are managed by the kernel any user or application request that involves access to any system resource must be handled by the kernel code, but user process must not be given open access to the kernel code for security reasons. So that user processes can invoke the execution of kernel code, several openings into the kernel code, also called *system calls*, are provided. System calls allow processes and users to manipulate system resources such as files and processes.

System calls can be categorized into the following groups:

- Process Control
- File Management
- Device Management
- Information maintenance
- Communications

Semantics of System Call Execution

The following sequence of events takes place when a process invokes a system call:

- The user process makes a call to a library function
- The library routine puts appropriate parameters at a well-known place, like a register or on the stack. These parameters include arguments for the system call, return address, and call number. Three general methods are used to pass parameters between a running program and the operating system.
  – Pass parameters in *registers*.
  – Store the parameters in a table in the main memory and the table address is passed as a parameter in a register.
  – *Push* (store) the parameters onto the *stack* by the program, and *pop* off the stack by operating system.

- A `trap` instruction is executed to change mode from user to kernel and give control to operating system.
- The operating system then determines which system call is to be carried out by examining one of the parameters (the call number) passed to it by library routine.
- The kernel uses call number to index a kernel table (the **dispatch table)** which contains pointers to service routines for all system calls.
- The service routine is executed and control given back to user program via return from trap instruction; the instruction also changes mode from system to user.
- The library function executes the instruction following trap; interprets the return values from the kernel and returns to the user process.

Figure 3.4 gives a pictorial view of the above steps.

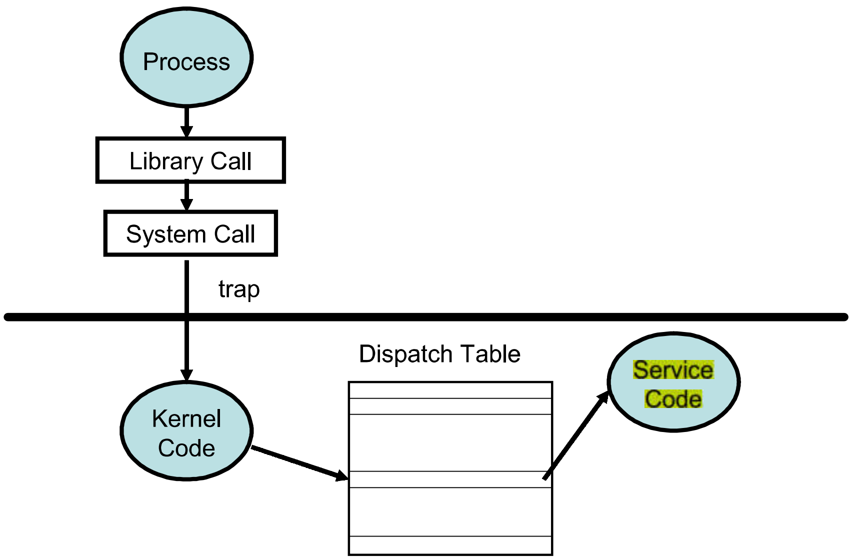


Figure 3.4 Pictorial view of the steps needed for execution of a system call

Operating Systems Structures

Just like any other software, the operating system code can be structured in different ways. The following are some of the commonly used structures.

- Simple/Monolithic Structure

In this case, the operating system code has not structure. It is written for functionality and efficiency (in terms of time and space). DOS and UNIX are examples of such systems, as shown in Figures 3.5 and 3.6. UNIX consists of two separable parts, the kernel and the system programs. The kernel is further separated into a series of interfaces and devices drivers, which were added and expanded over the years. Every thing below the system call interface and above the physical hardware is the kernel, which provides the file system, CPU scheduling, memory management and other OS functions through system calls. Since this is an enormous amount of functionality combined in one level, UNIX is difficult to enhance as changes in one section could adversely affect other areas. We will discuss the various components of the UNIX kernel throughout the course.

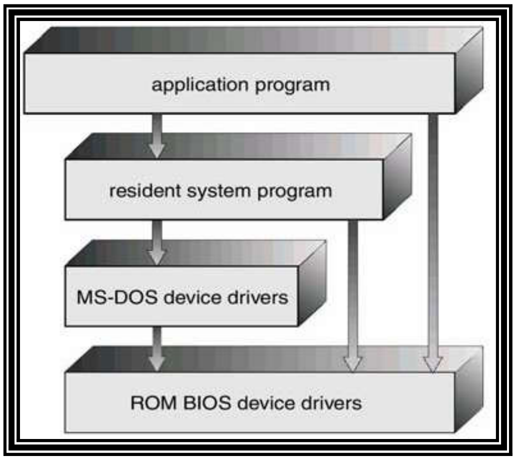


Figure 3.5 Logical structure of DOS

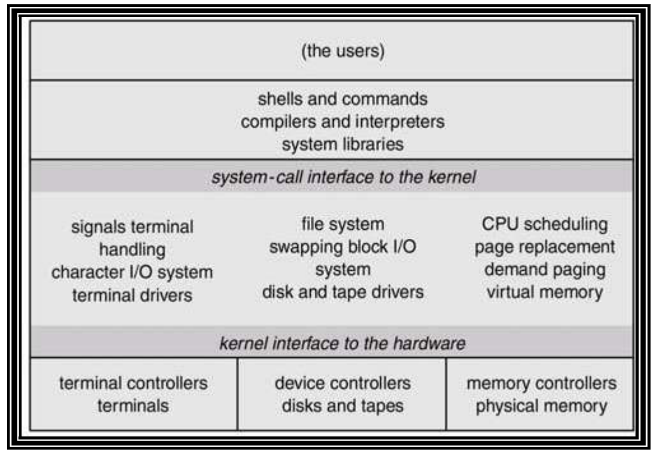


Figure 3.6 Logical structure of UNIX

Operating Systems
Lecture No. 4

Reading Material
- Operating Systems Structures, Chapter 3
- PowerPoint Slides for Lecture 3

Summary
- Operating system structures
- Operating system design and implementation
- UNIX/Linux directory structure
- Browsing UNIX/Linux directory structure

Operating Systems Structures (continued)

- Layered Approach

The modularization of a system can be done in many ways. As shown in Figure 4.1, in the layered approach the OS is broken up into a number of layers or levels each built on top of lower layer. The bottom layer is the hardware; the highest layer (layer N) is the user interface. A typical OS layer (layer-M) consists of data structures and a set of routines that can be invoked by higher-level layers. Layer M in turn can invoke operations on lower level layers.

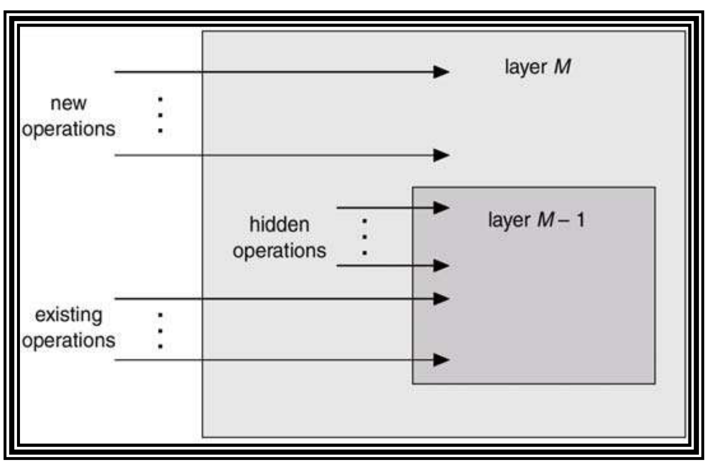


Figure 4.1 The layered structure

The main advantage of the layered approach is modularity. The layers are selected such that each uses functions and services of only lower layers. This approach simplifies debugging and system verification.

The major difficulty with layered approach is careful definition of layers, because a layer can only use the layers below it. Also it tends to be less efficient than other approaches. Each layer adds overhead to a system call (which is trapped when the program executes a I/O operation, for instance). This results in a system call that takes longer than does one on a non-layered system. THE operating system by Dijkstra and IBM's OS/2 are examples of layered operating systems.

■ Micro kernels

This method structures the operating system by removing all non-essential components from the kernel and implementing as system and user level programs. The result is a smaller kernel. Micro kernels typically provide minimum process and memory management in addition to a communication facility. The main function of the micro kernel is to provide a communication facility between the client program and the various services that are also running in the user space.

The benefits of the micro kernel approach include the ease of extending the OS. All new services are added to user space and consequently do not require modification of the kernel. When the kernel does have to be modified, the changes tend to be fewer because the micro kernel is a smaller kernel. The resulting OS is easier to port from one hard ware design to another. It also provides more security and reliability since most services are running as user rather than kernel processes. Mach, MacOS X Server, QNX, OS/2, and Windows NT are examples of microkernel based operating systems. As shown in Figure 4.2, various types of services can be run on top of the Windows NT microkernel, thereby allowing applications developed for different platforms to run under Windows NT.

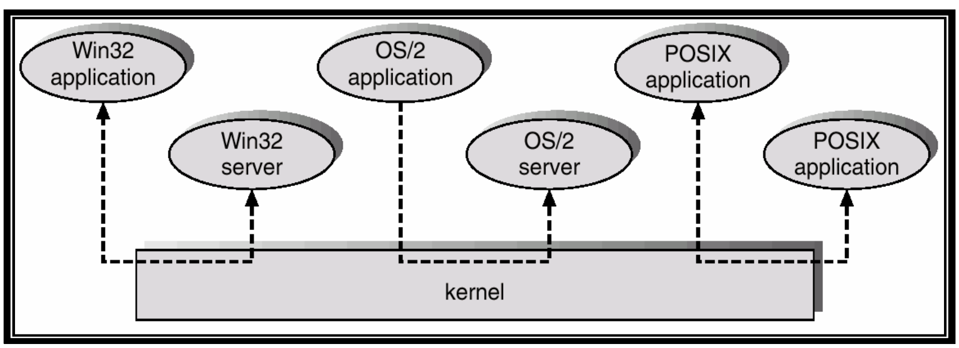
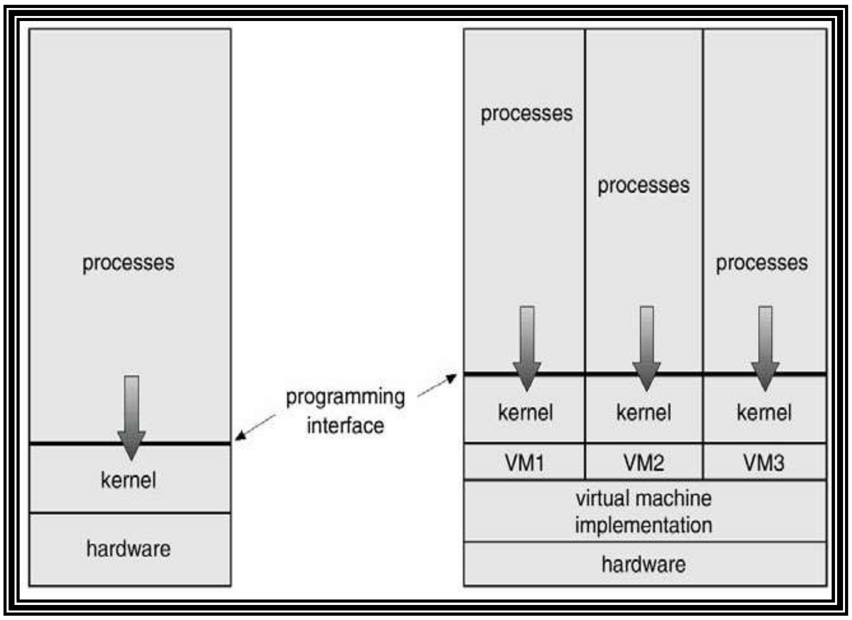


Figure 4.2 Windows NT client-server structure

■ Virtual Machines

Conceptually a computer system is made up of layers. The hardware is the lowest level in all such systems. The kernel running at the next level uses the hardware instructions to create a set of system call for use by outer layers. The system programs above the kernel are therefore able to use either system calls or hardware instructions and in some ways these programs do not differentiate between these two. System programs in turn treat the hardware and the system calls as though they were both at the same level. In some systems the application programs can call the system programs. The application programs view everything under them in the hierarchy as though the latter were part of the machine itself. This layered approach is taken to its logical conclusion in the concept of a **virtual machine** (VM). The VM operating system for IBM systems is the best example of VM concept.

By using CPU scheduling and virtual memory techniques an operating system can create the illusion that a process has its own memory with its own (virtual) memory. The virtual machine approach on the other hand does not provide any additional functionality but rather provides an interface that is identical to the underlying bare hardware. Each process is provided with a virtual copy of the underlying computer. The physical computer shares resources to create the virtual machines. Figure 4.3 illustrates the concepts of virtual machines by a diagram.



Non Virtual Machine Virtual Machine

Figure 4.3 Illustration of virtual and non-virtual machines

Although the virtual machine concept is useful it is difficult to implement. There are two primary advantages to using virtual machines: first by completely protecting system resources the virtual machine provides a robust level of security. Second the virtual machine allows system development to be done without disrupting normal system operation.

Java Virtual Machine (JVM) loads, verifies, and executes programs that have been translated into Java Bytecode, as shown in Figure 4.4. **VMWare** can be run on a Windows platform to create a virtual machine on which you can install an operating of your choice, such as Linux. We have shown a couple of snapshots of VMWare on a Windows platform in the lecture slides. **Virtual PC** software works in a similar fashion.

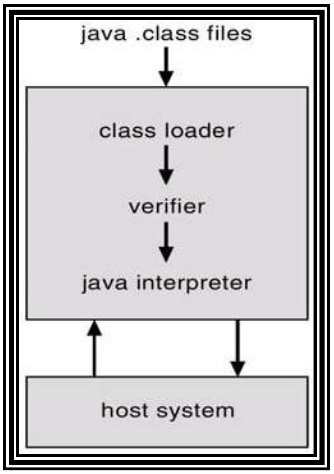


Figure 4.4 Java Virtual Machine

System Design and Implementation

■ Design Goals

At the highest level, the deign of the system will be affected by the choice of hardware and type of system: batch , time shared, single user, multi user, distributed , real time or general purpose. Beyond this highest level, the requirements may be much harder to specify. The requirements can be divided into much two basic groups: *user goal* and *system goals.* Users desire a system that is easy to use, reliable, safe and fast. People who design, implement and operate the system, require a system that is easy to design, implement and maintain. An important design goal is separation of mechanisms and policies.

- **Mechanism:** they determine how to do something. A general mechanism is more desirable. Example: CPU protection.
- **Policy:** determine what will be done. Example: Initial value in the counter used for CPU protection.

The separation of policy and mechanism is important for flexibility, as policies are likely to change across places or over time. For example, the system administrator can set the initial value in counter before booting a system.

■ Implementation

Once an operating system is designed, it must be implemented. Traditionally operating systems have been written in assembly language. Now however they are written in higher-level languages such as C/ C++ since these allow the code to be written faster, more compact, easier to understand and easier to port.

UNIX/LINUX Directory Structure

Dennis Ritchie and Ken Thomsom wrote UNIX at the Bell Labs in 1969. It was initially written in assembly language and a high-level language called Bit was later converted from B to C language. Linus Torvalds, an undergraduate student at the University of

Helsinki, Finland, wrote Linux in 1991. It is one of the most popular operating systems, certainly for PCs.

UNIX has a **hierarchical file system structure** consisting of a **root directory** (denoted as /) with other directories and files hanging under it. Unix uses a directory hierarchy that is commonly represented as folders. However, instead of using graphical folders typed commands (in a command line user interface) are used to navigate the system. Particular files are then represented by paths and filenames much like they are in html addresses. A pathname is the list of directories separated by slashes (/). If a pathname starts with a /, it refers to the root directory. The last component of a path may be a file or a directory. A pathname may simply be a file or directory name. For example, /usr/include/sys/param.h, ~/courses/cs604, and prog1.c are pathnames.

When you log in, the system places you in a directory called your **home directory** (also called **login directory**). You can refer to your home directory by using the ~ or $PATH in Bash, Bourne shell, and Korn shells and by using $path in the C and TC shells.

Shells also understand both **relative** and **absolute pathnames**. An absolute pathname starts with the root directory (/) and a relative pathname starts with your home directory, your current directory, or the parent of your **current directory** (the directory that you are currently in). For example, /usr/include/sys/param.h is an absolute pathname and ~/courses/cs604 and prog1.c are relative pathnames.

You can refer to your current directory by using `.` (pronounced dot) and the parent of your current directory by using `..` (pronounced dotdot). For example, if nadeem is currently in the courses directory, he can refer to his home directory by using .. and his personal directory by using ../personal. Similarly, he can refer to the directory for this course by using cs604.

Figures 4.5 and 4.6 show sample directory structures in a UNIX/Linux system. The user nadeem has a subdirectory under his home directory, called courses. This directory contains subdirectories for the courses that you have taken, including one for this course.

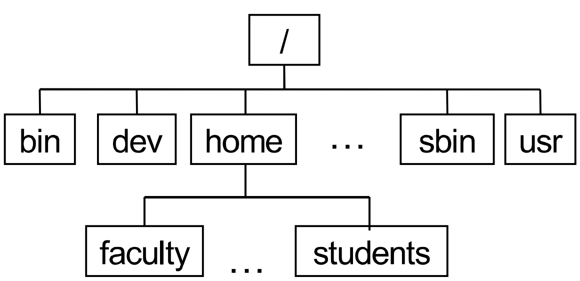


Figure 4.5 UNIX/Linux directory hierarchy

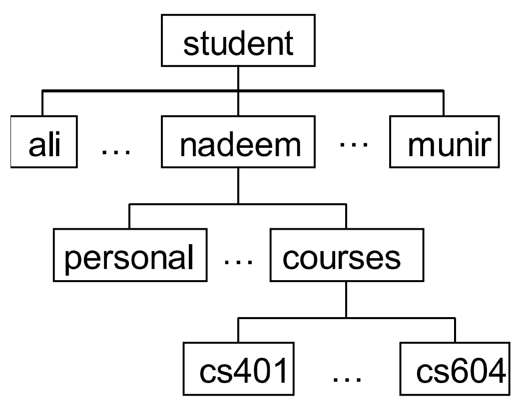


Figure 4.6 Home directories of students

Directory Structure

Some of the more important and commonly used directories in the Linux directory hierarchy are listed in Table 4.1. Many of the directories listed in the table are also found in a UNIX file system.

Table 4.1 Important directories in the Linux operating system and their purpose

/	**The root directory** (not to be concerned with the root account) is similar to a drive letter in Windows (C:\, D:\, etc.) except that in the Linux directory structure there is only one root directory and everything falls under it (including other file systems and partitions). The root directory is the directory that contains all other directories. When a directory structure is displayed as a tree, the root directory is at the top. Typically no files or programs are stored directly under root.
/bin	This directory holds binary executable files that are essential for correct operation of the system (exactly which binaries are in this directory is often dependent upon the distribution). These binaries are usually available for use by all users. **/usr/bin** can also be used for this purpose as well.
/boot	This directory includes essential system boot files including the kernel image .
/dev	This directory contains the devices available to Linux. Remember that Linux treats devices like files and you can read and write to them as if they were. Everything from floppy drives to printers to your mouse is contained in this directory. Included in this directory is the notorious /dev/null, which is most useful for deleting outputs of various, functions and programs.
/etc	Linux uses this directory to store system configuration files. Most files in this directory are text and can be edited with your favorite text editor. This is one of Linux's greatest advantages because there is never a hidden check box and just about all your configurations are in one place. **/etc/inittab** is a text file that details what processes are started at system boot up and during regular operation. **/etc/fstab** identifies file systems and their mount points (like floppy, CD-ROM, and hard disk drives). **/etc/passwd** is where users are defined.
/home	This is where every user on a Linux system will have a personal directory. If your username is "chris" then your home directory will be "/home/chris". A quick way to return to your home directory is by entering the "cd" command. Your current working directory will be changed to your home directory. Usually, the permissions on user directories are set so that only root and the user the directory belongs to can access or store information inside of it. When partitioning a Linux file system this directory will typically need the most space.
/lib	Shared libraries and kernel modules are stored in this directory. The

libraries can be dynamically linked which makes them very similar to DLL files in the Windows environment.

/lost+found	This is the directory where Linux keeps files that are restored after a crash or when a partition hasn't been unmounted properly before a shutdown.
/mnt	Used for mounting temporary filesystems. Filesystems can be mounted anywhere but the /mnt directory provides a convenient place in the Linux directory structure to mount temporary file systems.
/opt	Often used for storage of large applications packages
/proc	This is a special, "virtual" directory where system processes are stored. This directory doesn't physically exist but you can often view (or read) the entries in this directory.
/root	The home directory for the superuser (root). Not to be confused with the root (/) directory of the Linux file system.
/sbin	Utilities used for system administration (halt, ifconfig, fdisk, etc.) are stored in this directory. /usr/sbin, and /usr/local/sbin are other directories that are used for this purpose as well. **/sbin/init.d** are scripts used by /sbin/init to start the system.
/tmp	Used for storing temporary files. Similar to C:\Windows\Temp.
/usr	Typically a shareable, read-only directory. Contains user applications and supporting files for those applications. **/usr/X11R6** is used by the X Window System. **/usr/bin** contains user accessible commands. **/usr/doc** holds documentation for /usr applications. **/usr/include** this directory contains header files for the C compiler. **/usr/include/g++** contains header files for the C++ compiler. **/usr/lib** libraries, binaries, and object files that aren't usually executed directly by users. **/usr/local** used for installing software locally that needs to be safe from being overwritten when system software updates occur. **/usr/man** is where the manual pages are kept. **/usr/share** is for read-only independent data files. **/usr/src** is used for storing source code of applications installed and kernel sources and headers.
/var	This directory contains variable data files such as logs (**/var/log**), mail (**/var/mail**), and spools (**/var/spool**) among other things.

(Source: http://www.chrisshort.net/archives/2005/01/linux-directory-structure.php)

Operating Systems
Lecture No. 5

Reading Material

- Operating Systems Structures, Chapter 4
- PowerPoint Slides for Lecture 3

Summary

- Browsing UNIX/Linux directory structure
- Useful UNIX/Linux commands
- Process concept
- Process scheduling concepts
- Process creation and termination

Browsing UNIX/Linux directory structure

We discussed in detail the UNIX/Linux directory structure in lecture 4. We will continue that discussion and learn how to browse the UNIX/Linux directory structure. In Figure 5.1, we have repeated for our reference the home directory structure for students. In the rest of this section, we discuss commands for creating directories, removing directories, and browsing the UNIX/Linux directory structure.

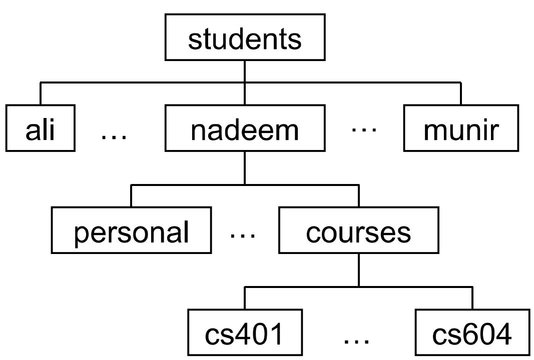


Figure 5.1 Home directories for students

Displaying Directory Contents

You can display the contents (names of files and directories) of a directory with the `ls` command. Without an argument, it assumes your current working directory. So, if you run the `ls` command right after you login, it displays names of files and directories in your home directory. It does not list those files whose names start with a dot (.). Files that start with a dot are known as **hidden files** (also called dot files). You should not modify these files unless you are quite familiar with the purpose of these files and why you want to modify them. You can display all the files in a directory by using `ls -a` command. Your can display the long listing for the contents of a directory by using the `ls -l` command. The following session shows sample runs of these commands.

```
$ ls
books         courses         LinuxKernel     chatClient.c  chatServer.c
$ ls -a
.             .bash_history   courses         .login        .profile
..            .bash_profile   .cshrc          books
chatClient.c  chatServer.c    LinuxKernel
$ ls -l
drwxr-xr-x   3 msarwar  faculty      512 Oct 28 10:28 books
-rw-r--r--   1 msarwar  faculty     9076 Nov  4 10:14 chatClient.c
-rw-r--r--   1 msarwar  faculty     8440 Nov  4 10:16 chatServer.c
drwxr-xr-x   2 msarwar  faculty      512 Feb 27 17:21 courses
drwxr-xr-x   2 msarwar  faculty      512 Oct 21 14:55 LinuxKernel
$
```

The output of the `ls -l` command gives you the following information about a file:

- 1st character: type of a file
- Rest of letters in the 1st field: access privileges on the file
- 2nd field: number of hard links to the file
- 3rd field: owner of the file
- 4th field: Group of the owner
- 5th field: File size in bytes
- 6th and 7th fields: Date last updated
- 8th field: Time last updated
- 9th field: File name

We will talk about file types and hard links later in the course.

Creating Directories

You can use the `mkdir` command to create a directory. In the following session, the first command creates the courses directory in your current directory. If we assume that your current directory is your home directory, this command creates the courses directory under your home directory. The second command creates the cs604 directory under the ~/courses directory (i.e., the under the courses directory under your home directory). The third command creates the programs directory under your ~/courses/cs604 directory.

```
$ mkdir courses
$ mkdir ~/courses/cs604
$ mkdir ~/courses/cs604/programs
$
```

You could have created all of the above directories with the `mkdir -p ~/courses/cs604/programs` command.

Removing (Deleting) Directories
You can remove (delete) an empty directory with the `mkdir` command. The command in the following session is used to remove the ~/courses/cs604/programs directory if it is empty.

```
$ rmdir courses
$
```

Changing Directory
You can jump from one directory to another (i.e., change your working directory) with the `cd` command. You can use the `cd` ~/courses/cs604/programs command to make ~/courses/cs604/programs directory your working directory. The `cd` or `cd $HOME` command can be used to make your home directory your working directory.

Display Absolute Pathname of Your Working Directory
You can display the absolute pathname of your working directory with the `pwd` command, as shown below.

```
$ pwd
/home/students/nadeem/courses/cs604/programs
$
```

Copying, Moving, and Removing Files

We now discuss the commands to copy, move (or rename), and remove files.

Copying Files
You can use the `cp` command for copying files. You can use the `cp file1 file2` command to copy file1 to file2. The following command can be used to copy file1 in your home directory to the ~/memos directory as file2.

```
$ cp ~/file1 ~/memos/file2
$
```

Moving Files
You can use the `mv` command for moving files. You can use the `mv file1 file2` command to move file1 to file2. The following command can be used to move file1 in your home directory to the ~/memos directory as file2.

```
$ mv ~/file1 ~/memos/file2
$
```

Removing Files
You can use the `rm` command to remove files. You can use the `rm file1` command to remove file1. You can use the first command the following command

to remove the test.c file in the ~/courses/cs604/programs directory and the second command to remove all the files with .o extension (i.e., all object files) in your working directory.

```
$ rm ~/courses/cs604/programs/test.c
$ rm *.o
$
```

Compiling and Running C Programs

You can compile your program with the `gcc` command. The output of the compiler command, i.e., the executable program is stored in the a.out file by default. To compile a source file titled program.c, type:

```
$ gcc program.c
$
```

You can run the executable program generated by this command by typing./a.out and hitting the <Enter> key, as shown in the following session.

```
$ ./a.out
[ ... program output ... ]
$
```

You can store the executable program in a specific file by using the `-o` option. For example, in the following session, the executable program is stored in the assignment file.

```
$ gcc program.c -o assignment
$
```

The `gcc` compiler does not link many libraries automatically. You can link a library explicitly by using the `-l` option. In the following session, we are asking the compiler to link the math library with our object file as it creates the executable file.

```
$ gcc program.c -o assignment -lm
$ assignment
[ ... program output ... ]
$
```

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 accompany 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. Such a system consists 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.

A batch system executes jobs (background processes), whereas a time-shared system has user programs, or tasks. Even on a single user system, a user may be able to run several programs at one time: a word processor, web browser etc.

A process is more than 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 register. In addition, a process generally includes the process stack, which contains temporary data (such as method parameters, the process stack, which contains temporary data), and a data section, which contains global variables.

A program by itself is not a process: a program is a passive entity, such as contents of a file stored on disk, whereas a process is an active entity, with a program counter specifying the next instruction to execute and a set of associated resources. Although two processes may be associated with the same program, they are considered two separate sequences of execution. E.g. several users may be running different instances of the mail program, of which the text sections are equivalent but the data sections vary.

Processes may be of two types:

- **IO bound processes:** spend more time doing IO than computations, have many short CPU bursts. Word processors and text editors are good examples of such processes.
- **CPU bound processes:** spend more time doing computations, few very long CPU bursts.

Process States

As a process executes, it changes states. The state of a process is defined in part by the current activity of that process. Each process may be in either of the following states, as shown in Figure 5.2:

- **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.

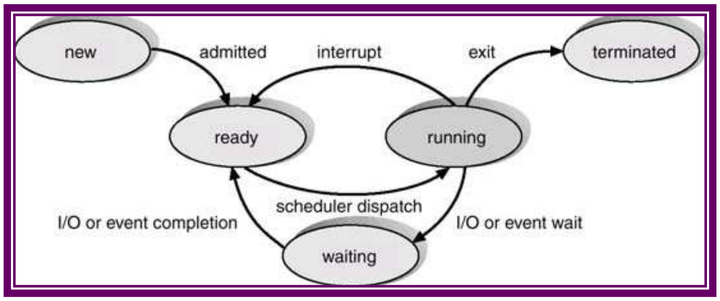


Figure 5.2 Process state diagram

Process Control Block

Each process is represented in the operating system by a **process control block** (PCB) – also called a task control block, as shown in Figure 5.3. A PCB 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 afterwards.
- **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 such 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:** The information includes the list of I/O devices allocated to the process, a list of open files, and so on.

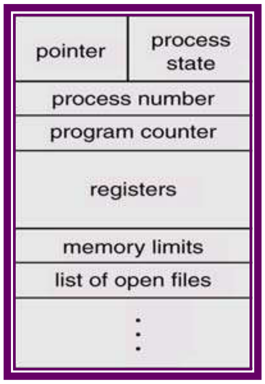


Figure 5.3 Process control block (PCB)

Process Scheduling

The objective of multiprogramming is to have some process running all the time so as to maximize CPU utilization. The objective of time-sharing is to switch the CPU among processors so frequently that users can interact with each program while it is running. A uniprocessor system can have only one running process at a given time. If more processes exist, the rest must wait until the CPU is free and can be rescheduled. Switching the CPU from one process to another requires saving of the context of the current process and loading the state of the new process, as shown in Figure 5.4. This is called **context switching**.

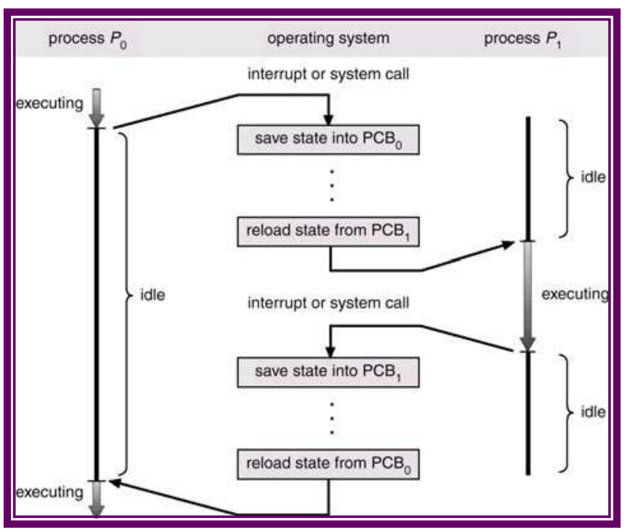


Figure 5.4 Context switching

Scheduling Queues

As shown in Figure 5.5, a contemporary computer system maintains many scheduling queues. Here is a brief description of some of these queues:

- **Job Queue:** As processes enter the system, they are put into a job queue. This queue consists of all processes in the system.
- **Ready Queue:** 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 is extended to include a pointer field that points to the next PCB in the ready queue.
- **Device Queue:** When a process is allocated the CPU, it executes for a while, and eventually quits, is interrupted or waits for a particular event, such as completion of an I/O request. In the case of an I/O request, the device may be busy with the I/O request of some other process, hence the list of processes waiting for a particular I/O device is called a device queue. Each device has its own device queue.

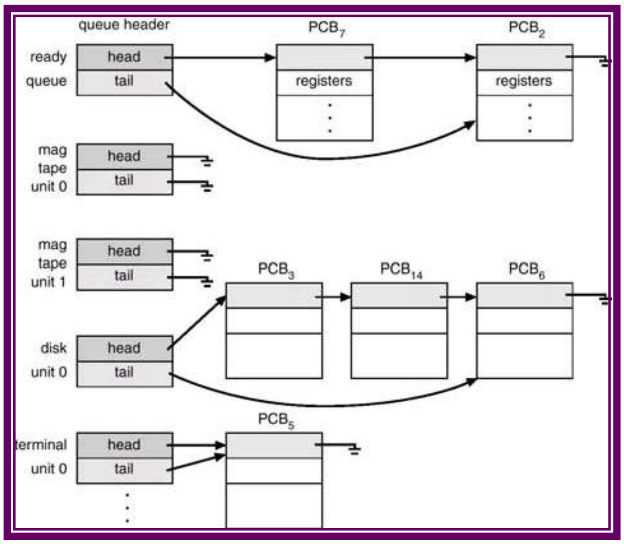


Figure 5.5 Scheduling queue

In the queuing diagram shown in Figure 5.6 below, each rectangle box represents a queue, and two such queues are present, the ready queue and an I/O queue. A new process is initially put in the ready queue, until it is dispatched. Once the process is executing, one of the several events could occur:

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

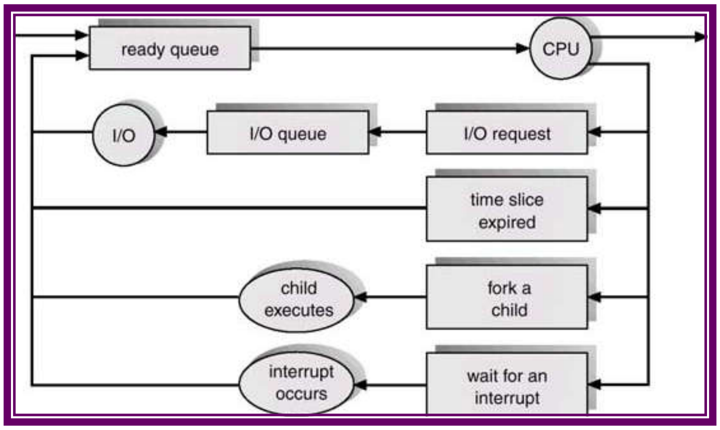


Figure 5.6 Queuing diagram of a computer system

Schedulers

A process migrates between the various scheduling queues throughout its lifetime. The operating system must select, for scheduling purposes, processes from these queues in some fashion. The appropriate scheduler carries out this selection process. The **Long-term scheduler** (or job scheduler) selects which processes should be brought into the ready queue, from the job pool that is the list of all jobs in the system. The **Short-term scheduler** (or CPU scheduler) selects which process should be executed next and allocates CPU.

The primary distinction between the two schedulers is the 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. Often the short-term scheduler executes at least once every 100 milliseconds. Because of the brief time between executions, the short-term scheduler must be fast. If it takes 10 milliseconds to decide to execute a process for 100 milliseconds, then 10/(100+10)=9 % of the CPU is being used for scheduling only. The long-term scheduler, on the other hand executes much less frequently. There may be minutes between the creations of new processes in the system. The long-term scheduler controls the degree of multiprogramming – the number of processes in memory. If the degree of multiprogramming is stable, then the average rate of process creation must be equal to the average department rate of processes leaving the system. Because of the longer interval between execution s, the long-term scheduler can afford to take more time to select a process for execution.

The long-term scheduler must select a good mix of I/O bound and CPU bound jobs. The reason why the long-term scheduler must select a good mix of I/O bound and CPU bound jobs is that if the processes are I/O bound, the ready queue will be mostly empty and the short-term scheduler will have little work. On the other hand, if the processes are mostly CPU bound, then the devices will go unused and the system will be unbalanced.

Some operating systems such as time-sharing systems may introduce a **medium-term scheduler**, which removes processes from memory (and from active contention for the CPU) and thus reduces the degree of multiprogramming. At some later time the process can be reintroduced at some later stage, 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 job mix, or because a change is memory requirements has over committed available memory, requiring memory to be freed up. As shown in Figure 5.7, the work carried out by the swapper to move a process from the main memory to disk is known as swap out and moving it back into the main memory is called swap in. The area on the disk where swapped out processes are stored is called the **swap space**.

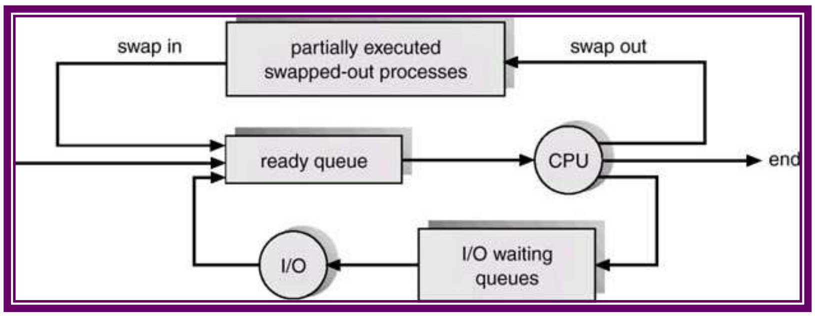


Figure 5.7 Computer system queues, servers, and swapping

Operating Systems
Lecture No. 6

Reading Material
- Operating Systems Concepts, Chapter 4
- UNIX/Linux manual pages for the `fork()` system call

Summary
- Process creation and termination
- Process management in UNIX/Linux— system calls: `fork`, `exec`, `wait`, `exit`
- Sample codes

Operations on Processes
The processes in the system execute concurrently and they must be created and deleted dynamically thus the operating system must provide the mechanism for the creation and deletion of processes.

Process Creation
A process may create several new processes via a create-process system call during the course of its execution. The creating process is called a **parent process** while 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. Figure 6.1 shows partially the process tree in a UNIX/Linux system.

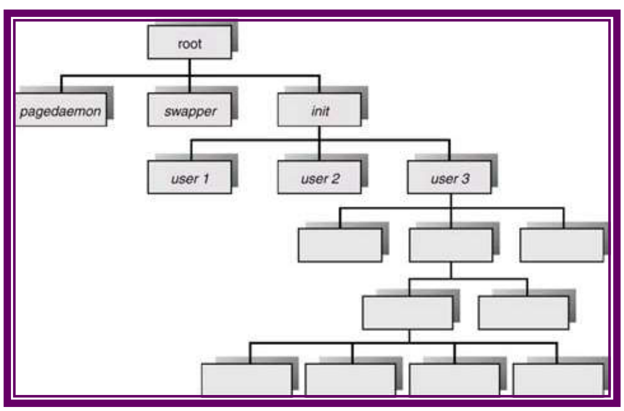


Figure 6.1 Process tree in UNIX/Linux

In general, a process will need certain resources (such as CPU time, memory files, I/O devices) to accomplish its task. When a process creates a sub process, also known as a child, that sub process may be able to obtain its resources directly from the operating system or may be constrained to a subset of the resources of the parent process. The parent may have to partition its resources among several of its children. Restricting a process to a subset of the parent's resources prevents a process from overloading the system by creating too many sub processes.

When a process is created it obtains in addition to various physical and logical resources, initialization data that may be passed along from the parent process to the child process. 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.
2. The child process has a program loaded into it.

In order to consider these different implementations let us consider the UNIX operating system. In UNIX its process identifier identifies a process**,** which is a unique integer. A new process is created by the `fork` system call. The new process consists of a copy of the address space of the parent. This mechanism allows the parent process to communicate easily with the child process. Both processes continue execution at the instruction after the `fork` call, with one difference, the return code for the `fork` system call is zero for the child process, while the process identifier of the child is returned to the parent process.

Typically the `execlp` system call is used after a `fork` system call by one of the two processes to replace the process' memory space with a new program. The `execlp` system call loads a binary file in memory –destroying the memory image of the program containing the `execlp` 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. The parent waits for the child process to terminate, and then it resumes from the call to wait where it completes using the `exit` system call.

Process termination

A process terminates when it finishes executing its final statement and asks the operating system to delete it by calling the exit system call. At that point, the process may return data to its parent process (via the wait system call). All the resources of the process including physical and virtual memory, open the files and I/O buffers – are de allocated by the operating system.

Termination occurs under additional circumstances. A process can cause the termination of another via an appropriate system call (such as abort). Usually only the parent of the process that is to be terminated can invoke this system call. Therefore parents need to know the identities of its children, and thus when one process creates another 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:

- The child has exceeded its usage of some of the resources that it has been allocated. This requires the parent to have a mechanism to inspect the state of its children.
- The task assigned to the child is no longer required.