Contents

1. Introduction to Operating System

2. Operating System Structure

3. Process Management

4. Deadlocks

Introduction to Operating System

1.1 Introduction and history (Generation of OS)

1.1.1 Introduction

In a computer system, we find four main components: the hardware, the operating system, the application software and the users. In a computer system the hardware provides the basic computing resources. The applications programs define the way in which these resources are used to solve the computing problems of the users. The operating system controls and coordinates the use of the hardware among the various systems programs and application programs for the various users.

We can view an operating system as a resource allocator. A computer system has many resources (hardware and software) that may be required to solve a problem: CPU time, memory space, files storage space, input/output devices etc. The operating system acts as the manager of these resources and allocates them to specific programs and users as necessary for their tasks. Since there may be many, possibly conflicting, requests for resources, the operating system must decide which requests are allocated resources to operate the computer system fairly and efficiently.

An operating system is a control program. This program controls the execution of user programs to prevent errors and improper use of the computer. Operating systems exist because: they are a reasonable way to solve the problem of creating a usable computing system. The fundamental goal of a computer system is to execute user programs and solve user problems.

While there is no universally agreed upon definition of the concept of an operating system, the following is a reasonable starting point:

A computer's operating system is a group of programs designed to serve two basic purposes:

- To control the allocation and use of the computing system's resources among the various users and tasks, and
- To provide an interface between the computer hardware and the programmer that simplifies and makes feasible the creation, coding, debugging, and maintenance of application programs.

An effective operating system should accomplish the following functions:

- Should act as a command interpreter by providing a user friendly environment.
- Should facilitate communication with other users.
- Facilitate the directory/file creation along with the security option.
- Provide routines that handle the intricate details of I/O programming.
- Provide access to compilers to translate programs from high-level languages to machine language
- Provide a loader program to move the compiled program code to the computer's memory for execution.
- Assure that when there are several active processes in the computer, each will get fair and non-interfering access to the central processing unit for execution.
- Take care of storage and device allocation.
- Provide for long term storage of user information in the form of files.
- Permit system resources to be shared among users when appropriate, and be protected from unauthorized or mischievous intervention as necessary.

Though systems programs such as editors and translators and the various utility programs (such as sort and file transfer program) are not usually considered part of the operating system, the operating system is responsible for providing access to these system resources.

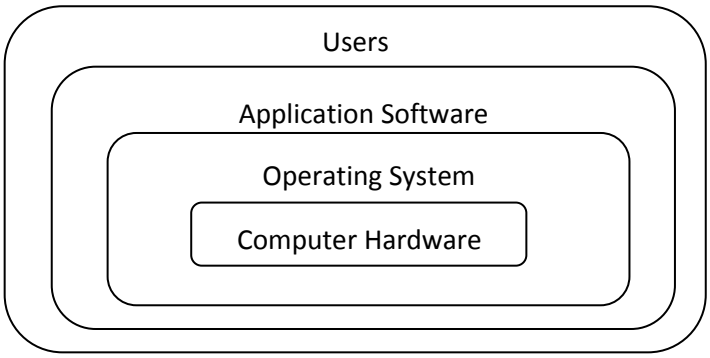


Fig: Layers of Operating systems

1.1.2 History of Operating system:

Operating systems have been evolving over the years. We will briefly look at this development of the operating systems with respect to the evolution of the hardware / architecture of the computer systems in this section. Since operating systems have historically been closely tied with the architecture of the computers on which they run, we will look at successive generations of computers to see what their operating systems were like. We may not exactly map the operating systems generations to the generations of the computer, but roughly it provides the idea behind them.

We can roughly divide them into five distinct generations that are characterized by hardware component technology, software development, and mode of delivery of computer services.

1.1.2.1 The First Generation (1945-55) Vacuum Tubes and Plug boards

After Babbage's unsuccessful efforts, little progress was made in constructing digital computers until World War II. Around the mid-1940s, Howard Aiken at Harvard, John von Neumann at the Institute for Advanced Study in Princeton, J. Presper Eckert and William Mauchley at the University of Pennsylvania, and Konrad Zuse in Germany, among others, all succeeded in building calculating engines. The first ones used mechanical relays but were very slow, with cycle times measured in seconds. Relays were later replaced by vacuum tubes. These machines were enormous, filling up entire rooms with tens of thousands of vacuum tubes, but they were still millions of times slower than even the cheapest personal computers available today.

In these early days, a single group of people designed, built, programmed, operated, and maintained each machine. All programming was done in absolute machine language, often by wiring up plug boards to control the machine's basic functions. Programming languages were unknown (even assembly language was unknown). Operating systems were unheard of. The usual made of operation was for the programmer to sign up for a block of time on the signup sheet on the wall, then come down to the machine room, insert his or her plug board into the computer, and spend the next few hours hoping that none of the 20,000 or so vacuum tubes would burn out during the run. Virtually all the problems were straightforward numerical calculations, such as grinding out tables of sines, cosines, and logarithms.

By the early 1950s, the routine had improved somewhat with the introduction of punched cards. It was now possible to write programs on cards and read them in instead of using plug boards; otherwise, the procedure was the same.

1.1.2.2 The Second Generation (1955-65) Transistors and Batch Systems

The introduction of the transistor in the mid-1950s changed the picture radically. Computers became reliable enough that they could be manufactured and sold to paying customers with the expectation that they would continue to function long enough to get some useful work done. For the first time, there was a clear separation between designers, builders, operators, programmers, and maintenance personnel.

These machines, now called **mainframes,** were locked away in specially air conditioned computer rooms, with staffs of professional operators to run them. Only big corporations or major government agencies or universities could afford the multimillion dollar price tag. To run a **job** (i.e., a program or set of programs), a programmer would first write the program on paper (in FORTRAN or assembler), then punch it on cards. He would then bring the card deck down to the input room and hand it to one of the operators and go drink coffee until the output was ready.

When the computer finished whatever job it was currently running, an operator would go over to the printer and tear off the output and carry it over to the output room, so that the

programmer could collect it later. Then he would take one of the card decks that had been brought from the input room and read it in. If the FORTRAN compiler was needed, the operator would have to get it from a file cabinet and read it in. Much computer time was wasted while operators were walking around the machine room.

The solution generally adopted was the **batch system.** The idea behind it was to collect a tray full of jobs in the input room and then read them onto a magnetic tape using a small (relatively) inexpensive computer, such as the IBM 1401, which was very good at reading cards, copying tapes, and printing output, but not at all good at numerical calculations. Other, much more expensive machines, such as the IBM 7094, were used for the real computing.

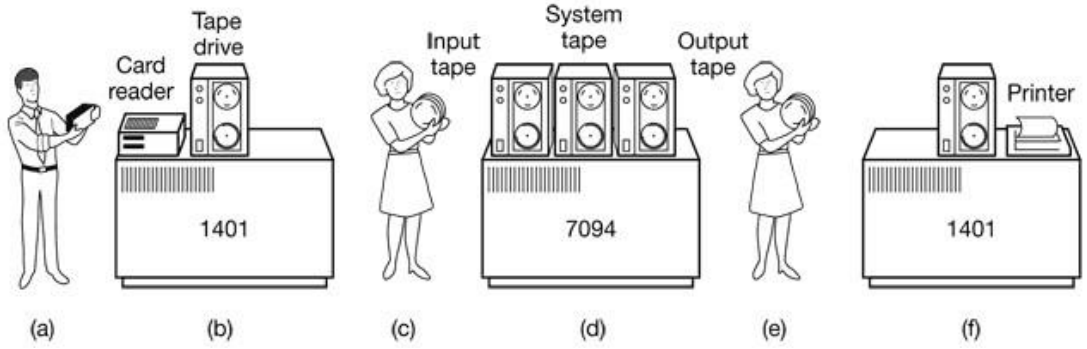


Fig: (a) Programmers bring cards to 1401
(b) 1401 reads batch of jobs onto tape
(c) Operator carries input tape to 7094
(d) 7094 does computing
(e) Operator carries output tape to 1401
(f) 1401 prints output

1.1.2.3 The Third Generation (1965-1980) ICs and Multiprogramming

By the early 1960s, most computer manufacturers had two distinct, and totally incompatible, product lines. On the one hand there were the word-oriented, large-scale scientific computers, such as the 7094, which were used for numerical calculations in science and engineering. On the other hand, there were the character-oriented, commercial computers, such as the 1401, which were widely used for tape sorting and printing by banks and insurance companies.

IBM attempted to solve both of these problems at a single stroke by introducing the System/360. The 360 was a series of software-compatible machines ranging from 1401-sized to much more powerful than the 7094. The machines differed only in price and performance (maximum memory, processor speed, number of I/O devices permitted, and so forth). Since all the machines had the same architecture and instruction set, programs written for one machine could run on all the others, at least in theory. Furthermore, the 360 was designed to handle both scientific (i.e., numerical) and commercial computing. Thus a single family of machines could satisfy the needs of all customers. In subsequent years, IBM has come out with compatible successors to the 360 line, using more modern technology, known as the 370, 4300, 3080, and 3090 series.

The greatest strength of the "one family" idea was simultaneously its greatest weakness. The intention was that all software, including the operating system, OS/360 had to work on all models. It had to run on small systems, which often just replaced 1401s for copying cards to tape, and on very large systems, which often replaced 7094s for doing weather forecasting and other heavy computing. It had to be good on systems with few peripherals and on systems with many peripherals. It had to work in commercial environments and in scientific environments. Above all, it had to be efficient for all of these different uses.

Despite its enormous size and problems, OS/360 and the similar third-generation operating systems produced by other computer manufacturers actually satisfied most of their customers reasonably well. They also popularized several key techniques absent in second-generation operating systems. Probably the most important of these was **multiprogramming.** On the 7094, when the current job paused to wait for a tape or other I/O operation to complete, the CPU simply sat idle until the I/O finished. With heavily CPU-bound scientific calculations, I/O is infrequent, so this wasted time is not significant. With commercial data processing, the I/O wait time can often be 80 or 90 percent of the total time, so something had to be done to avoid having the (expensive) CPU be idle so much.

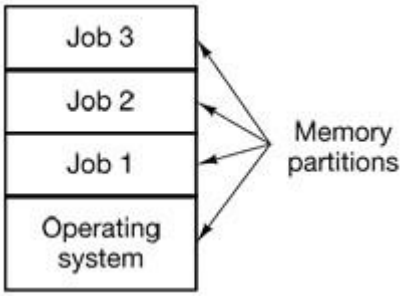


Fig: A multiprogramming system with three jobs in memory.

1.1.2.4 The Fourth Generation (1980-Present) Personal Computers

With the development of LSI (Large Scale Integration) circuits, chips containing thousands of transistors on a square centimeter of silicon, the age of the personal computer dawned. In terms of architecture, personal computers (initially called **microcomputers)** were not all that different from minicomputers of the PDP-11 class, but in terms of price they certainly were different. Where the minicomputer made it possible for a department in a company or university to have its own computer, the microprocessor chip made it possible for a single individual to have his or her own personal computer.

In 1974, when Intel came out with the 8080, the first general-purpose 8-bit CPU, it wanted an operating system for the 8080, in part to be able to test it. Intel asked one of its consultants, Gary Kildall, to write one. Kildall and a friend first built a controller for the newly-released Shugart Associates 8-inch floppy disk and hooked the floppy disk up to the 8080, thus producing the first microcomputer with a disk. Kildall then wrote a disk-based operating system called **CP/M (Control Program for Microcomputers)** for it.

In the early 1980s, IBM designed the IBM PC and looked around for software to run on it. People from IBM contacted Bill Gates to license his BASIC interpreter. They also asked him if he knew of an operating system to run on the PC, Gates suggested that IBM contact Digital Research, then the world's dominant operating systems company. Making what was surely the worst business decision in recorded history, Kildall refused to meet with IBM, sending a subordinate instead. To make matters worse, his lawyer even refused to sign IBM's nondisclosure agreement covering the not-yet-announced PC. Consequently, IBM went back to Gates asking if he could provide them with an operating system.

When IBM came back, Gates realized that a local computer manufacturer, Seattle Computer Products, had a suitable operating system. **DOS (Disk Operating System).** He approached them and asked to buy it (allegedly for $50,000), which they readily accepted. Gates then offered IBM a DOS/BASIC package which IBM accepted. IBM wanted certain modifications, so Gates hired the person who wrote DOS, Tim Paterson, as an employee of Gates' fledgling company, Microsoft, to make them. The revised system was renamed **MSDOS (MicroSoft Disk Operating System)** and quickly came to dominate the IBM PC market.

CP/M, MS-DOS, and other operating systems for early microcomputers were all based on users typing in commands from the keyboard. That eventually changed due to research done by Doug Engelbart at Stanford Research Institute in the 1960s. Engelbart invented the **GUI (Graphical User Interface),** pronounced "gooey," complete with windows, icons, menus, and mouse. These ideas were adopted by researchers at Xerox PARC and incorporated into machines they built.

One day, Steve Jobs, who co-invented the Apple computer in his garage, visited PARC, saw a GUI, and instantly realized its potential value; something Xerox management famously did not (Smith and Alexander, 1988). Jobs then embarked on building an Apple with a GUI. This project led to the Lisa, which was too expensive and failed commercially. Jobs' second attempt, the Apple Macintosh, was a huge success, not only because it was much cheaper than the Lisa, but also because it was **user friendly,** meaning that it was intended for users who not only knew nothing about computers but furthermore had absolutely no intention whatsoever of learning.

Another Microsoft operating system is Windows NT (NT stands for New Technology), which is compatible with Windows 95 at a certain level, but a complete rewrite from scratch internally. It is a full 32-bit system.

An interesting development that began taking place during the mid-1980s is the growth of networks of personal computers running **network operating systems** and **distributed operating** systems. In a network operating system, the users are aware of the existence of multiple computers and can log in to remote machines and copy files from one machine to another. Each machine runs its own local operating system and has its own local user (or users).

1.2 Objectives (Resource manager and extended machine)

There are two primary objectives of operating system:

1.2.1 Operating system as an extended machine:

The architecture (instruction set, memory organization, I/O, and bus structure) of most computers at the machine language level is primitive and awkward to program, especially for input/output. The program that hides the truth about the hardware from the programmer and presents a nice, simple view of named files that can be read and written is operating system. Just as the operating system shields the programmer from the disk hardware and presents a simple file-oriented interface, it also conceals a lot of unpleasant business concerning interrupts, timers, memory management, and other low-level features. In each case, the abstraction offered by the operating system is simpler and easier to use than that offered by the underlying hardware.

In this view, the function of the operating system is to present the user with the equivalent of an extended machine or virtual machine that is easier to program than the underlying hardware.

1.2.2. The Operating System as a Resource Manager

The main important objective of an operating system is to manage the various resources of the computer system. This involves performing such tasks as keeping track of who is using which resources, granting resource requests, accounting for resource usage, and mediating conflicting requests from different programs and users.

Executing a job on a computer system often requires several of its resource such as CPU time, memory space, file storage space, I/O devices and so on. The operating system acts as the manager of the various resources of a computer system and allocates them to specific programs and users to execute their jobs successfully. When a computer system is used to simultaneously handle several applications, there may be many, possibly conflicting, requests for resources. In such a situation, the operating system must decide which requests are allocated resources to operate the computer system efficiently and fairly (providing due attention to all users). The efficient and fair sharing of resources among users and/or programs is a key goal of most operating system.

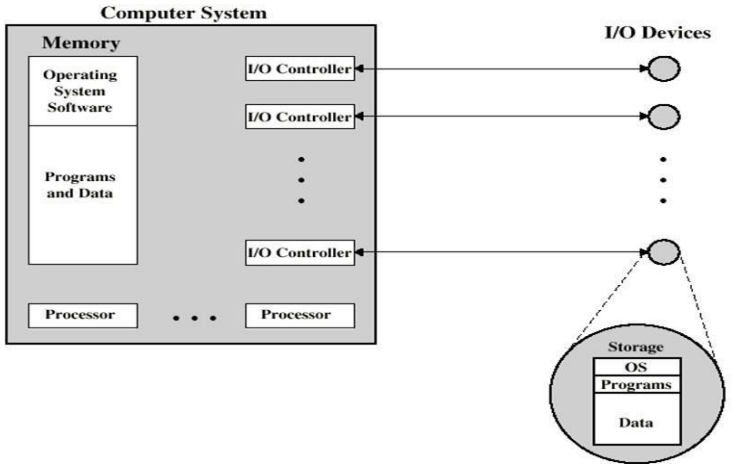


Fig: Operating System as a resource manager

1.3 Types of Operating system

1.3.1 Batch Processing Operating System

In a batch processing operating system environment users submit jobs to a central place where these jobs are collected into a batch, and subsequently placed on an input queue at the computer where they will be run. In this case, the user has no interaction with the job during its processing, and the computer's response time is the turnaround time-the time from submission of the job until execution is complete, and the results are ready for return to the person who submitted the job.

1.3.2 Time Sharing

Another mode for delivering computing services is provided by time sharing operating systems. In this environment a computer provides computing services to several or many users concurrently on-line. Here, the various users are sharing the central processor, the memory, and other resources of the computer system in a manner facilitated, controlled, and monitored by the operating system. The user, in this environment, has nearly full interaction with the program during its execution, and the computer's response time may be expected to be no more than a few second.

1.3.3 Real Time Operating System (RTOS)

The third class is the real time operating systems, which are designed to service those applications where response time is of the essence in order to prevent error, misrepresentation or even disaster. Examples of real time operating systems are those which handle airlines reservations, machine tool control, and monitoring of a nuclear power station. The systems, in this case, are designed to be interrupted by external signals that require the immediate attention of the computer system.

These real time operating systems are used to control machinery, scientific instruments and industrial systems. An RTOS typically has very little user-interface capability, and no end user utilities. A very important part of an RTOS is managing the resources of the computer so that a particular operation executes in precisely the same amount of time every time it occurs. In a complex machine, having a part move more quickly just because system resources are available may be just as catastrophic as having it not move at all because the system is busy.

1.3.4 Multiprogramming Operating System

A multiprogramming operating system is a system that allows more than one active user program (or part of user program) to be stored in main memory simultaneously. Thus, it is evident that a time-sharing system is a multiprogramming system, but note that a multiprogramming system is not necessarily a time-sharing system. A batch or real time operating system could, and indeed usually does, have more than one active user program simultaneously in main storage. Another important, and all too similar, term is "multiprocessing".

1.3.5 Multiprocessing System

A multiprocessing system is a computer hardware configuration that includes more than one independent processing unit. The term multiprocessing is generally used to refer to large computer hardware complexes found in major scientific or commercial applications.

1.3.6 Networking Operating System

A networked computing system is a collection of physical interconnected computers. The operating system of each of the interconnected computers must contain, in addition to its own stand-alone functionality, provisions for handing communication and transfer of program and data among the other computers with which it is connected.

Network operating systems are not fundamentally different from single processor operating systems. They obviously need a network interface controller and some low-level software to drive it, as well as programs to achieve remote login and remote files access, but these additions do not change the essential structure of the operating systems.

1.3.7 Distributed Operating System

A distributed computing system consists of a number of computers that are connected and managed so that they automatically share the job processing load among the constituent computers, or separate the job load as appropriate particularly configured processors. Such a system requires an operating system which, in addition to the typical stand-alone functionality, provides coordination of the operations and information flow among the

component computers. The networked and distributed computing environments and their respective operating systems are designed with more complex functional capabilities. In a network operating system, the users are aware of the existence of multiple computers, and can log in to remote machines and copy files from one machine to another. Each machine runs its own local operating system and has its own user (or users).

A distributed operating system, in contrast, is one that appears to its users as a traditional uniprocessor system, even though it is actually composed of multiple processors. In a true distributed system, users should not be aware of where their programs are being run or where their files are located; that should all be handled automatically and efficiently by the operating system.

True distributed operating systems require more than just adding a little code to a uniprocessor operating system, because distributed and centralized systems differ in critical ways. Distributed systems, for example, often allow program to run on several processors at the same time, thus requiring more complex processor scheduling algorithms in order to optimize the amount of parallelism achieved.

1.3.8 Operating Systems for Embedded Devices

As embedded systems (PDAs, cell phones, point-of-sale devices, VCR's, industrial robot control, or even your toaster) become more complex hardware-wise with every generation, and more features are put into them day-by-day, applications they run require more and more to run on actual operating system code in order to keep the development time reasonable. Some of the popular OS are:

- Nexus's Conix - an embedded operating system for ARM processors.
- Sun's Java OS - a standalone virtual machine not running on top of any other OS; mainly targeted at embedded systems.
- Palm Computing's Palm OS - Currently the leader OS for PDAs, has many applications and supporting companies.
- Microsoft's Windows CE and Windows NT Embedded OS.

1.3.9 Mainframe Operating System

The operating systems for mainframes are heavily oriented toward processing many jobs at once, most of which need prodigious amounts of I/O. They typically offer three kinds of services: batch, transaction processing, and timesharing. A batch system is one that processes routine jobs without any interactive user present. Claims processing in an insurance company or sales reporting for a chain of stores are typically done in batch mode. Transaction processing systems handle large numbers of small requests; for example, check processing at a bank or airline reservations. Each unit of work is small, but the system must handle hundreds or thousands per second. Timesharing systems allow multiple remote users to run jobs on the

computer at once, such as querying a big database. These functions are closely related: mainframe operating systems often perform all of them. An example mainframe operating system is OS/390, a descendant of OS/360.

1.3.10 Personal Computer (PC) Operating System

The next category is the personal computer operating system. Their job is to provide a good interface to a single user. They are widely used for word processing, spreadsheets, and Internet access. Common examples are Windows 98, Windows 2000, the Macintosh operating system, and Linux. Personal computer operating systems are so widely known that probably little introduction is needed. In fact, many people are not even aware that other kinds exist.

1.3.11 Smart Card Operating System

The smallest operating systems run on smart cards, which are credit card-sized devices containing a CPU chip. They have very severe processing power and memory constraints. Some of them can handle only a single function, such as electronic payments, but others can handle multiple functions on the same smart card. Often these are proprietary systems.

Some smart cards are Java oriented. What this means is that the ROM on the smart card holds an interpreter for the Java Virtual Machine (JVM). Java applets (small programs) are downloaded to the card and are interpreted by the JVM interpreter. Some of these cards can handle multiple Java applets at the same time, leading to multiprogramming and the need to schedule them. Resource management and protection also become an issue when two or more applets are present at the same time. These issues must be handled by the (usually extremely primitive) operating system present on the card.

1.4 Function of Operating system

The main functions of an operating system are as follows:

- Process Management
- Memory Management
- Secondary Storage Management
- I/O Management
- File Management
- Protection
- Networking Management
- Command Interpretation.

1.4.1 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. The operating system is responsible for the following activities in connection with processes management:

- The creation and deletion of both user and system processes
- The suspension and resumption of processes.
- The provision of mechanisms for process synchronization
- The provision of mechanisms for deadlock handling.

1.4.2 Memory Management

Memory is the most expensive part in the 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.

There are various algorithms that depend on the particular situation to manage the memory. Selection of a memory management scheme for a specific system depends upon many factors, 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.

- Keep track of which parts of memory are currently being used and by whom.
- Decide which processes are to be loaded into memory when memory space becomes available.
- Allocate and deallocate memory space as needed.

1.4.3 Secondary 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:

- Free space management
- Storage allocation
- Disk scheduling.

1.4.4 I/O Management

One of the purposes of an operating system is to hide the peculiarities or 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 operating system is responsible for the following activities in connection to I/O management:

- A buffer caching system
- To activate a general device driver code
- To run the driver software for specific hardware devices as and when required.

1.4.5 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 file 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 to the file management:

- The creation and deletion of files.
- The creation and deletion of directory.
- The support of primitives for manipulating files and directories.
- The mapping of files onto disk storage.
- Backup of files on stable (non volatile) storage.
- Protection and security of the files.

1.4.6 Protection

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 ensures that a process can only execute within its own address space. The timer ensures that no process can gain control of the CPU without relinquishing it. Finally, no process is allowed to do its 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.

1.4.7 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.

1.4.8 Command Interpretation

One of the most important components 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 statements themselves deal with process management, I/O handling, secondary storage management, main memory management, file system access, protection, and networking.

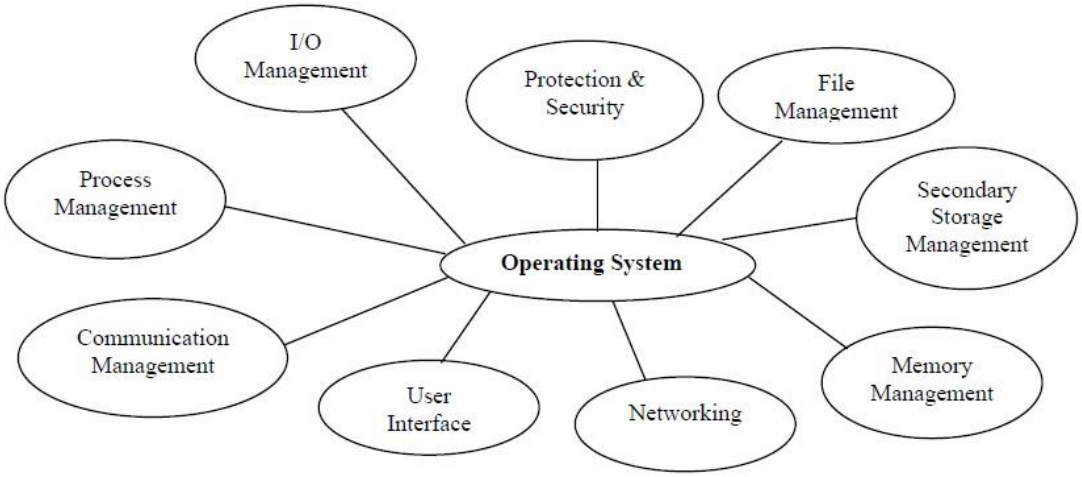


Figure: Functions Coordinated by the Operating System

1.5 Exercises:

1. What is OS? List all types of operating system and explain any three.
2. What is an OS? Describe its features.
3. Define Operating System. Why Operating System is considered as a resource manager and extended machine? Explain.

Operating System Structure

2.1 Introduction

Now that we have seen what operating systems look like on the outside (i.e., the programmer's interface), it is time to take a look inside. In the following sections, we will examine five different structures that have been tried, in order to get some idea of the spectrum of possibilities. These are by no means exhaustive, but they give an idea of some designs that have been tried in practice. The five designs are monolithic systems, layered systems, virtual machines, exokernels, and client-server systems.

2.2 Monolithic Systems

The operating system is written as a collection of procedures, each of which can call any of the other ones whenever it needs to. When this technique is used, each procedure in the system has a well-defined interface in terms of parameters and results, and each one is free to call any other one, if the latter provides some useful computation that the former needs.

To construct the actual object program of the operating system when this approach is used, one first compiles all the individual procedures, or files containing the procedures, and then binds them all together into a single object file using the system linker. In terms of information hiding, there is essentially none—every procedure is visible to every other procedure.

This organization suggests a basic structure for the operating system:

1. A main program that invokes the requested service procedure.
2. A set of service procedures that carry out the system calls.
3. A set of utility procedures that help the service procedures.

In this model, for each system call there is one service procedure that takes care of it. The utility procedures do things that are needed by several service procedures, such as fetching data from user programs.

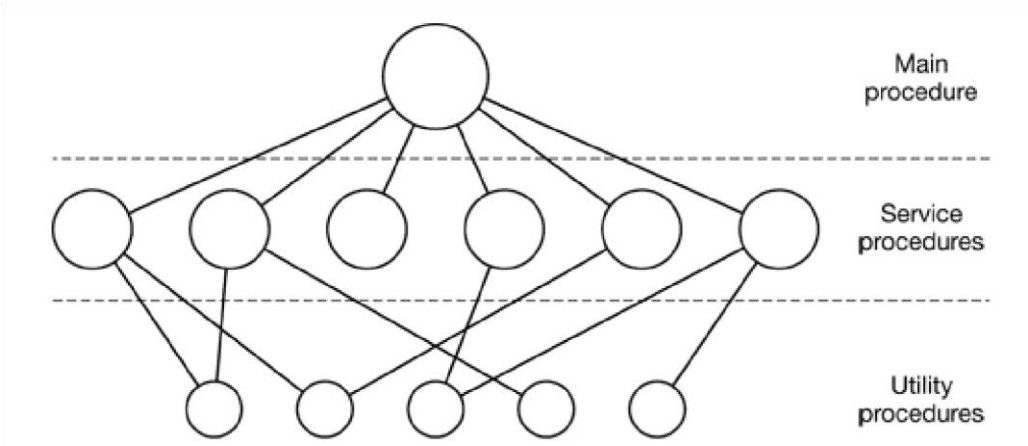


Fig: A simple structuring model for a monolithic system.

2.3 Layered System

The system had 6 layers, as shown in Fig. Layer 0 dealt with allocation of the processor, switching between processes when interrupts occurred or timers expired. Above layer 0, the system consisted of sequential processes, each of which could tie programmed without having to worry about the fact that multiple processes were running on a single processor. In other words, layer 0 provided the basic multiprogramming of the CPU.

Layer	Function
5	The operator
4	User programs
3	Input/output management
2	Operator-process communication
1	Memory and drum management
0	Processor allocation and multiprogramming

Fig: Structure of the operating system.

Layer 1 did the memory management. It allocated space for processes in main memory and on a 512K word drum used for holding parts of processes (pages) for which there was no room in main memory. Above layer 1, processes did not have to worry about whether they were in memory or on the drum; the layer 1 software took care of making sure pages were brought into memory whenever they were needed.

Layer 2 handled communication between each process and the operator console. Above this layer each process effectively had its own operator console. Layer 3 took care of managing the I/O devices and buffering the information streams to and from them. Above layer 3 each process could deal with abstract I/O devices with nice properties, instead of real devices with many peculiarities. Layer 4 was where the user programs were found. They did not have to worry about process, memory, console, or I/O management. The system operator process was located in layer 5.

2.4 Kernel

The kernel is the central module of an operating system (OS). It is the part of the operating system that loads first, and it remains in main memory. Because it stays in memory, it is important for the kernel to be as small as possible while still providing all the essential services required by other parts of the operating system and applications. The kernel code is usually loaded into a protected area of memory to prevent it from being overwritten by programs or other parts of the operating system.

Typically, the kernel is responsible for memory management, process and task management, and disk management. The kernel connects the system hardware to the application software. Every operating system has a kernel. For example the Linux kernel is used numerous operating systems including Linux, FreeBSD, Android and others.

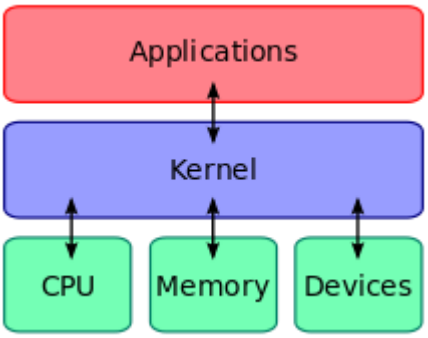


Fig: Kernel

2.4.1 Types of kernel

2.4.1.1. Monolithic Kernels:

In a modern day approach to monolithic architecture, the kernel consists of different modules which can be dynamically loaded and un-loaded. This modular approach allows easy extension of OS's capabilities. With this approach, maintainability of kernel became very easy as only the concerned module needs to be loaded and unloaded every time there is a change or bug fix in a particular module. So, there is no need to bring down and recompile the whole kernel for a smallest bit of change. Also, stripping of kernel for various platforms (say for embedded devices etc) became very easy as we can easily unload the module that we do not want.

Linux follows the monolithic modular approach.

2.4.1.2. Micro Kernels

This architecture majorly caters to the problem of ever growing size of kernel code which we could not control in the monolithic approach. This architecture allows some basic services like device driver management, protocol stack, file system etc to run in user space. This reduces the kernel code size and also increases the security and stability of OS as we have the bare minimum code running in kernel. So, if suppose a basic service like network service crashes

due to buffer overflow, then only the networking service's memory would be corrupted, leaving the rest of the system still functional.

In this architecture, all the basic OS services which are made part of user space are made to run as servers which are used by other programs in the system through inter process communication (IPC). eg: we have servers for device drivers, network protocol stacks, file systems, graphics, etc. Microkernel servers are essentially daemon programs like any others, except that the kernel grants some of them privileges to interact with parts of physical memory that are otherwise off limits to most programs. This allows some servers, particularly device drivers, to interact directly with hardware. These servers are started at the system start-up.

2.5 Client-server model

A trend in modern operating systems is to take the idea of moving code up into higher layers even further and remove as much as possible from kernel mode, leaving a minimal **microkernel.** The usual approach is to implement most of the operating system in user processes. To request a service, such as reading a block of a file, a user process (now known as the **client process)** sends the request to a **server process,** which then does the work and sends back the answer.

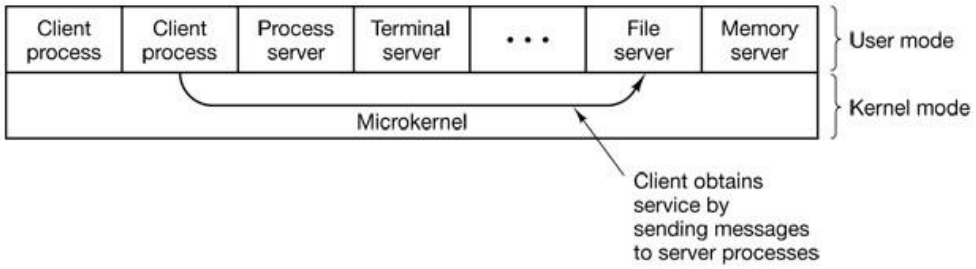


Fig: The client-server model.

In this model, shown in Fig., all the kernel does is handle the communication between clients and servers. By splitting the operating system up into parts, each of which only handles one facet of the system, such as file service, process service, terminal service, or memory service, each part becomes small and manageable. Furthermore, because all the servers run as user mode processes, and not in kernel mode, they do not have direct access to the hardware. As a consequence, if a bug in the file server is triggered, the file service may crash, but this will not usually bring the whole machine down.

Another advantage of the client-server model is its adaptability to use in distributed. If a client communicates with a server by sending it messages, the client need not know whether the message is handled locally in its own machine, or whether it was sent across a network to a server on a remote machine. As far as the client is concerned, the same thing happens in both cases: a request was sent and a reply came back.

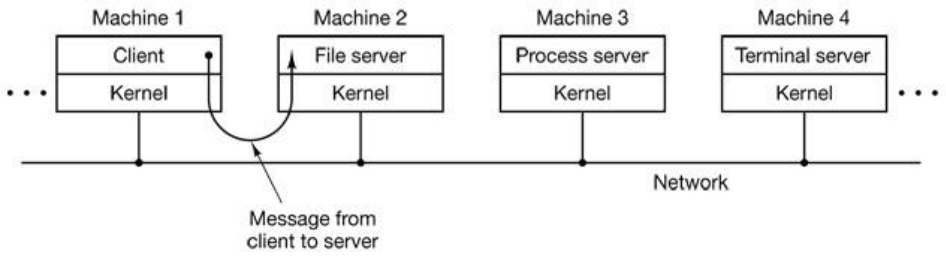


Fig: The client-server model in a distributed system.

2.6 Virtual Machine

This system, originally called CP/CMS and later renamed VM/370, was based on an astute observation: a timesharing system provides (1) multiprogramming and (2) an extended machine with a more convenient interface than the bare hardware. The essence of VM/370 is to completely separate these two functions.

The heart of the system, known as the **virtual machine monitor,** runs on the bare hardware and does the multiprogramming, providing not one, but several virtual machines to the next layer up, as shown in Fig. However, unlike all other operating systems, these virtual machines are not extended machines, with files and other nice features. Instead, they are *exact* copies of the bare hardware, including kernel/user mode, I/O, interrupts, and everything else the real machine has.

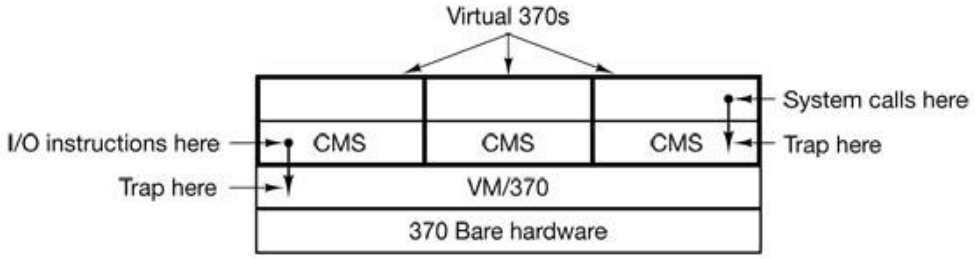


Fig: The structure of VM/370 with CMS.

Because each virtual machine is identical to the true hardware, each one can run any operating system that will run directly on the bare hardware. Different virtual machines can, and frequently do, run different operating systems Some run one of the descendants of OS/360 for batch or transaction processing, while other ones run a single-user, interactive system called **CMS (Conversational Monitor System)** for interactive timesharing users.

The idea of a virtual machine is heavily used nowadays in a different context: running old MS-DOS programs on a Pentium (or other 32-bit Intel CPU). When designing the Pentium and its software, both Intel and Microsoft realized that there would be a big demand for running old software on new hardware. For this reason, Intel provided a virtual 8086 mode on

the Pentium. In this mode, the machine acts like an 8086 (which is identical to an 8088 from a software point of view), including 16-bit addressing with a 1-MB limit.

This mode is used by Windows and other operating systems for running MS-DOS programs. These programs are started up in virtual 8086 mode. As long as they execute normal instructions, they run on the bare hardware. However, when a program tries to trap to the operating system to make a system call, or tries to do protected I/O directly, a trap to the virtual machine monitor occurs.

2.7 Shell

The operating system is the code that carries out the system calls. Editors, compilers, assemblers, linkers, and command interpreters definitely are not part of the operating system, even though they are important and useful. At the risk of confusing things somewhat, in this section we will look briefly at the UNIX command interpreter, called the **shell.** Although it is not part of the operating system, it makes heavy use of many operating system features and thus serves as a good example of how the system calls can be used. It is also the primary interface between a user sitting at his terminal and the operating system, unless the user is using a graphical user interface. Many shells exist, including *sh, csh, ksh,* and *bash.*

When any user logs in, a shell is started up. The shell has the terminal as standard input and standard output. It starts out by typing the **prompt,** a character such as a dollar sign, which tells the user that the shell is waiting to accept a command. If the user now types

date

for example, the shell creates a child process and runs the *date* program as the child. While the child process is running, the shell waits for it to terminate. When the child finishes, the shell types the prompt again and tries to read the next input line.
The user can specify that standard output be redirected to a file, for example,

date >file

Similarly, standard input can be redirected, as in *sort*
<file1 >file2

which invokes the sort program with input taken from *file1* and output sent to *file2.*
The output of one program can be used as the input for another program by connecting them with a pipe. Thus

cat file1 file2 file3 | sort >/dev/lp

invokes the *cat* program to concatenate three files and send the output to *sort* to arrange all the lines in alphabetical order. The output of *sort* is redirected to the file */dev/lp* , typical1y the printer.

2.8 Exercise

1. What do you mean by the kernel of Operating System? Explain about the monolithic kernel structure.
2. What is a Kernel? What role does it plays in an operating system? Differentiate between monolithic and micro kernel.

Process Management

3.1 Process Concepts

All modern computers can do several things at the same time. While running a user program, a computer can also be reading from a disk and outputting text to a screen or printer. In a multiprogramming system, the CPU also switches from program to program, running each for tens or hundreds of milliseconds. While, strictly speaking, at any instant of time, the CPU is running only one program, in the course of 1 second, it may work on several programs, thus giving the users the illusion of parallelism. Sometimes people speak of **pseudoparallelism** in this context, to contrast it with the true hardware parallelism of **multiprocessor** systems (which have two or more CPUs sharing the same physical memory). Keeping track of multiple, parallel activities is hard for people to do. Therefore, operating system designers over the years have evolved a conceptual model (sequential processes) that makes parallelism easier to deal with.

3.1.1 Definitions of process

The term "process" was first used by the operating system designers of the MULTICS system way back in 1960s. There are different definitions to explain the concept of process. Some of these are, a process is:

- An instance of a program in execution
- An asynchronous activity
- The "animated spirit" of a procedure
- The "locus of control" of a procedure in execution
- The "dispatchable" unit
- Unit of work individually schedulable by an operating system.

Formally, we can define a **process** is an executing program, including the current values of the program counter, registers, and variables. The subtle difference between a process and a program is that the program is a group of instructions whereas the process is the activity.

In multiprogramming systems, processes are performed in a pseudo-parallelism as if each process has its own processor. In fact, there is only one processor but it switches back and forth from process to process. Henceforth, by saying *execution* of a process, we mean the processor's operations on the process like changing its variables, etc. and *I/O work* means the interaction of the process with the I/O operations like reading something or writing to somewhere. They may also be named as "*processor (CPU) burst"* and *"I/O burst"* respectively. According to these definitions, we classify programs as:

- **Processor- bound program**: A program having long processor bursts (execution instants)
- **I/O- bound program**: A program having short processor bursts.

The operating system works as the computer system software that assists hardware in performing process management functions. Operating system keeps track of all the active processes and allocates system resources to them according to policies devised to meet design performance objectives. To meet process requirements OS must maintain many data structures efficiently. The process abstraction is fundamental means for the OS to manage concurrent program execution. OS must interleave the execution of a number of processes to maximize processor use while providing reasonable response time. It must allocate resources to processes in conformance with a specific policy. In general, a process will need certain resources such as CPU time, memory, files, I/O devices etc. to accomplish its tasks. These resources are allocated to the process when it is created. A single processor may be shared among several processes with some scheduling algorithm being used to determine when to stop work on one process and provide service to a different one which we will discuss later in this unit.

Operating systems must provide some way to create all the processes needed. In simple systems, it may be possible to have all the processes that will ever be needed be present when the system comes up. In almost all systems however, some way is needed to create and destroy processes as needed during operations. In UNIX, for instant, processes are created by the *fork* system call, which makes an identical copy of the calling process. In other systems, system calls exist to create a process, load its memory, and start it running. In general, processes need a way to create other processes. Each process has one parent process, but zero, one, two, or more children processes.

For an OS, the process management functions include:

- Process creation
- Termination of the process
- Controlling the progress of the process
- Process Scheduling
- Dispatching
- Interrupt handling / Exceptional handling
- Switching between the processes
- Process synchronization
- Interprocess communication support
- Management of Process Control Blocks.

3.1.2 The process model

In this model, all the runnable software on the computer, sometimes including the operating system, is organized into a number of **sequential processes**, or just processes for short. A

process is just an executing program, including the current values of the program counter, registers, and variables. Conceptually, each process has its own virtual CPU. In reality, of course, the real CPU switches back and forth from process to process, but to understand the system, it is much easier to think about a collection of processes running in (pseudo) parallel, than to try to keep track of how the CPU switches from program to program. This rapid switching back and forth is called **multiprogramming.**

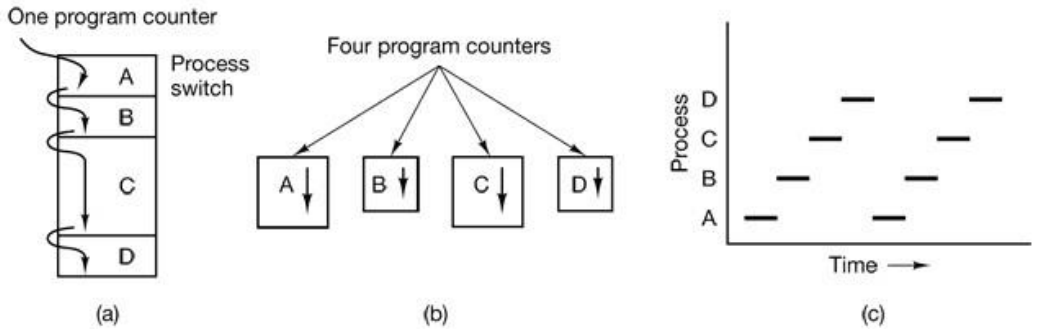


Fig: (a) Multiprogramming of four programs. (b) Conceptual model of four independent, sequential processes. (c) Only one program is active at once.

With the CPU switching back and forth among the processes, the rate at which a process performs its computation will not be uniform and probably not even reproducible if the same processes are run again. Thus, processes must not be programmed with built-in assumptions about timing. The difference between a process and a program is subtle, but crucial. An analogy makes help here.

Consider a culinary-minded computer scientist who is baking a birthday cake for his daughter. He has a birthday cake recipe and a kitchen well stocked with all the input: flour, eggs, sugar, extract of vanilla, and so on. In this analogy, the recipe is the program (i.e., an algorithm expressed in some suitable notation), the computer scientist is the processor (CPU), and the cake ingredients are the input data. The process is the activity consisting of our baker reading the recipe, fetching the ingredients, and baking the cake.

Now imagine that the computer scientist's son comes running in crying, saying that he has been stung by a bee. The computer scientist records where he was in the recipe (the state of the current process is saved), gets out a first aid book, and begins following the directions in it. Here we see the processor being switched from one process (baking) to a higher-priority process (administering medical care), each having a different program (recipe versus first aid book). When the bee sting has been taken care of, the computer scientist goes back to his cake, continuing at the point where he left off.

The key idea here is that a process is an activity of some kind. It has a program, input, output, and a state. A single processor may be shared among several processes, with some scheduling algorithm being used to determine when to stop work on one process and service a different one.

3.1.3 Process states

As defined, a process is an independent entity with its own input values, output values, and internal state. A process often needs to interact with other processes. One process may generate some outputs that other process uses as input. For example, in the shell command ***cat file1 file2 file3 | grep tree***

The first process, running *cat*, concatenates and outputs three files. Depending on the relative speed of the two processes, it may happen that *grep* is ready to run, but there is no input waiting for it. It must then block until some input is available. It is also possible for a process that is ready and able to run to be blocked because the operating system is decided to allocate the CPU to other process for a while.

A process state may be in one of the following:

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

The transition of the process states are shown in *Figure* and their corresponding transition is described below:

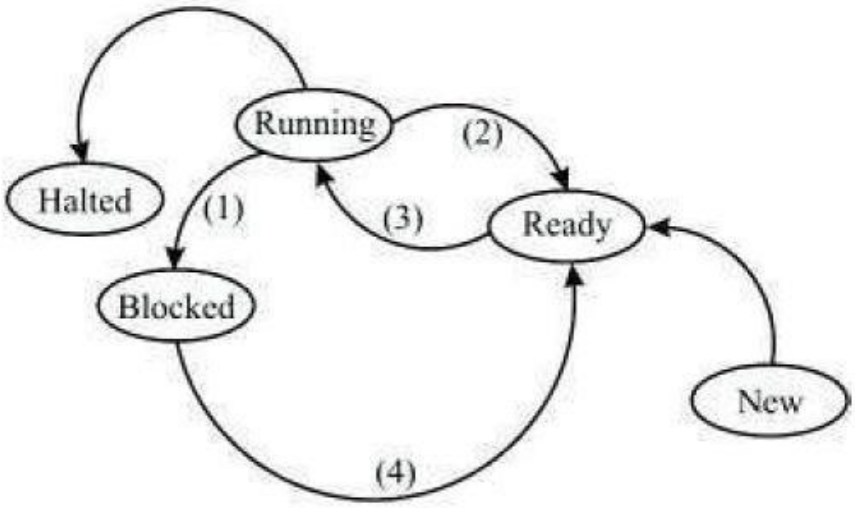


Fig: Typical Process States

As shown in *Figure* , four transitions are possible among the states. Transition 1 appears when a process discovers that it cannot continue. In order to get into blocked state, some systems must execute a system call *block*. In other systems, when a process reads from a pipe or special file and there is no input available, the process is automatically blocked.

Transition 2 and 3 are caused by the process scheduler, a part of the operating system. Transition 2 occurs when the scheduler decides that the running process has run long enough,

and it is time to let another process have some CPU time. Transition 3 occurs when all other processes have had their share and it is time for the first process to run again.

Transition 4 appears when the external event for which a process was waiting was happened. If no other process is running at that instant, transition 3 will be triggered immediately, and the process will start running. Otherwise it may have to wait in ready state for a little while until the CPU is available.

Using the process model, it becomes easier to think about what is going on inside the system. There are many processes like user processes, disk processes, terminal processes, and so on, which may be blocked when they are waiting for some thing to happen. When the disk block has been read or the character typed, the process waiting for it is unblocked and is ready to run again.

The process model, an integral part of an operating system, can be summarized as follows. The lowest level of the operating system is the scheduler with a number of processes on top of it. All the process handling, such as starting and stopping processes are done by the scheduler. More on the schedulers can be studied is the subsequent sections.

3.1.4 Process state transition

- **Null -> New:** A new process is created to execute a program. This event occurs for any of the following reasons:
  1. An interactive logon to the system by a user
  2. Created by OS to provide a service on behalf of a user program
  3. Spawn by existing program
  4. The OS is prepared to take on a new batch job
- **New -> Ready:** The OS moves a new process to a ready state when it is prepared to take on additional process (Most systems set some limit on the number of existing processes)
- Ready -> Running: OS chooses one of the processes in the ready state and assigns CPU to it.
- Running -> Terminated: The process is terminated by the OS if it has completed or aborted.
- Running -> Ready: The most common reason for this transition are:
  1. The running process has expired his time slot.
  2. The running process gets interrupted because a higher priority process is in the ready state.
- **Running -> Waiting (Blocked):** A process is put to this state if it requests for some thing for which it must wait:
  1. A service that the OS is not ready to perform.
  2. An access to a resource not yet available.
  3. Initiates I/O and must wait for the result.
  4. Waiting for a process to provide input.
- **Waiting -> Ready:** A process from a waiting state is moved to a ready state when the event for which it has been waiting occurs.
- **Ready -> terminated:** Not shown on the diagram. In some systems, a parent may terminate a child process at any time. Also, when a parent terminates, all child processes are terminated.

- **Blocked -> terminated:** Not shown. This transition occurs for the reasons given above
- Another state, Suspend, can also be included in the model. The operating system may move a process from a blocked state to suspend state by temporarily taking them out of memory.

3.1.5 The process control block

To implement the process model, the operating system maintains a table, an array of structures, called the *process table or process control block (PCB) or Switch frame.* Each entry identifies a process with information such as process state, its program counter, stack pointer, memory allocation, the status of its open files, its accounting and scheduling information. In other words, it must contain everything about the process that must be saved when the process is switched from the running state to the ready state so that it can be restarted later as if it had never been stopped. The following is the information stored in a PCB.

- Process state, which may be new, ready, running, waiting or halted;
- Process number, each process is identified by its process number, called process ID;
- Program counter, which indicates the address of the next instruction to be executed for this process;
- CPU registers, which vary in number and type, depending on the concrete microprocessor architecture;
- Memory management information, which include base and bounds registers or page table;
- I/O status information, composed I/O requests, I/O devices allocated to this process, a list of open files and so on;
- Processor scheduling information, which includes process priority, pointers to scheduling queues and any other scheduling parameters;
- List of open files.

A process structure block is shown in Figure

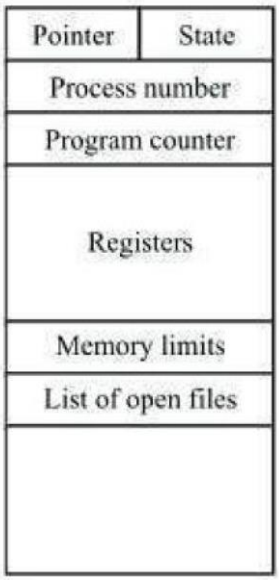


Fig: Process Control Block Structure

Context Switch

A *context switch* (also sometimes referred to as a *process switch* or a *task switch*) is the switching of the CPU (central processing unit) from one process or to another. A *context* is the contents of a CPU's registers and *program counter* at any point in time.

A context switch is sometimes described as the kernel suspending *execution of one process* on the CPU and resuming *execution of some other process* that had previously been suspended.

Context Switch: Steps

In a context switch, the state of the first process must be saved somehow, so that, when the scheduler gets back to the execution of the first process, it can restore this state and continue normally.

The state of the process includes all the registers that the process may be using, especially the program counter, plus any other operating system specific data that may be necessary. Often, all the data that is necessary for state is stored in one data structure, called a process control block (PCB). Now, in order to switch processes, the PCB for the first process must be created and saved. The PCBs are sometimes stored upon a per-process stack in the kernel memory, or there may be some specific operating system defined data structure for this information.

Let us understand with the help of an example. Suppose if two processes A and B are in ready queue. If CPU is executing Process A and Process B is in wait state. If an interrupt occurs for Process A, the operating system suspends the execution of the first process, and stores the current information of Process A in its PCB and context to the second process namely Process B. In doing so, the program counter from the PCB of Process B is loaded, and thus execution can continue with the new process. The switching between two processes, Process A and Process B is illustrated in the Figure *given* below:

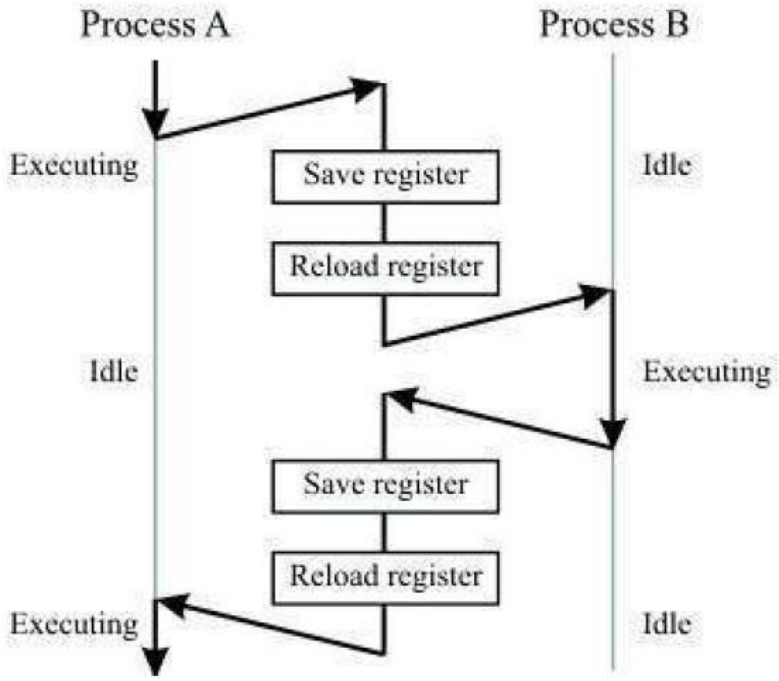


Fig: Process switching between two processes

3.1.6 Operations on processes

Process Creation:

Operating systems need some way to make sure all the necessary processes exist. In very simple systems, or in systems designed for running only a single application (e.g., the controller in a microwave oven), it may be possible to have all the processes that will ever be needed be present when the system comes up. In general-purpose systems, however, some way is needed to create and terminate processes as needed during operation. We will now look at some of the issues.

There are four principal events that cause processes to be created:

1. System initialization.
2. Execution of a process creation system call by a running process.
3. A user request to create a new process.
4. Initiation of a batch job.

When an operating system is booted, typically several processes are created. Some of these are foreground processes, that is, processes that interact with (human) users and perform work for them. Others are background processes, which are not associated with particular users, but instead have some specific function. For example, one background process may be designed to accept incoming email, sleeping most of the day but suddenly springing to life when email arrives. Another background process may be designed to accept incoming requests for Web pages hosted on that machine, waking up when a request arrives to service the request. Processes that stay in the background to handle some activity such as email, Web pages, news, printing, and so on are called **daemons.** Large systems commonly have dozens of them. In UNIX, the *ps* program can be used to list the running processes. In Windows 95/98/Me, typing CTRL-ALT-DEL once shows what's running. In Windows 2000, the task manager is used.

In addition to the processes created at boot time, new processes can be created afterward as well. Often a running process will issue system calls to create one or more new processes to help it do its job. Creating new processes is particularly useful when the work to be done can easily be formulated in terms of several related, but otherwise independent interacting processes. For example, if a large amount of data is being fetched over a network for subsequent processing, it may be convenient to create one process to fetch the data and put them in a shared buffer while a second process removes the data items and processes them. On a multiprocessor, allowing each process to run on a different CPU may also make the job go faster.

Process Termination:

After a process has been created, it starts running and does whatever its job is. However, nothing lasts forever, not even processes. Sooner or later the new process will terminate, usually due to one of the following conditions:

1. Normal exit (voluntary).
2. Error exit (voluntary).
3. Fatal error (involuntary).
4. Killed by another process (involuntary).

Most processes terminate because they have done their work. When a compiler has compiled the program given to it, the compiler executes a system call to tell the operating system that it is finished. This call is exit in UNIX and ExitProcess in Windows. Screen-oriented programs also support voluntary termination. Word processors, Internet browsers and similar programs always have an icon or menu item that the user can click to tell the process to remove any temporary files it has open and then terminate.

Process Hierarchies:

In some systems, when a process creates another process, the parent process and child process continue to be associated in certain ways. The child process can itself create more processes, forming a process hierarchy. Note that unlike plants and animals that use sexual reproduction, a process has only one parent (but zero, one, two, or more children).

In UNIX, a process and all of its children and further descendants together form a process group. When a user sends a signal from the keyboard, the signal is delivered to all members of the process group currently associated with the keyboard (usually all active processes that were created in the current window). Individually, each process can catch the signal, ignore the signal, or take the default action, which is to be killed by the signal.

As another example of where the process hierarchy plays a role, let us look at how UNIX initializes itself when it is started. A special process, called *init,* is present in the boot image. When it starts running, it reads a file telling how many terminals there are. Then it forks off one new process per terminal. These processes wait for someone to log in. If a login is successful, the login process executes a shell to accept commands. These commands may start up more processes, and so forth. Thus, all the processes in the whole system belong to a single tree, with *init* at the root.

In contrast, Windows does not have any concept of a process hierarchy. All processes are equal. The only place where there is something like a process hierarchy is that when a process is created, the parent is given a special token (called a **handle)** that it can use to control the child. However, it is free to pass this token to some other process, thus invalidating the hierarchy. Processes in UNIX cannot disinherit their children.

Implementation of Processes:
To implement the process model, the operating system maintains a table (an array of structures), called the **process table**, with one entry per process. (Some authors call these entries **process control blocks**.) This entry contains information about the process' state, its program counter, stack pointer, memory allocation, the status of its open files, its accounting and scheduling information, and everything else about the process that must be saved when the process is switched from *running* to *ready* or *blocked* state so that it can be restarted later as if it had never been stopped.

The fields in the first column relate to process management. The other two columns relate to memory management and file management, respectively. It should be noted that precisely which fields the process table has is highly system dependent, but this figure gives a general idea of the kinds of information needed.

Process management	**Memory management**	**File management**
Registers	Pointer to text segment	Root directory
Program counter	Pointer to data segment	Working directory
Program status word	Pointer to stack segment	File descriptors
Stack pointer		User ID
Process state		Group ID
Priority		
Scheduling parameters		
Process ID		
Parent process		
Process group		
Signals		
Time when process started		
CPU time used		
Children's CPU time		
Time of next alarm		

3.1.7 Cooperating processes

There are two kinds of processes: cooperating and independent processes. A process is independent if it cannot affect or be affected by other processes. A process is said to be a cooperating process if it can affect or be affected by other processes in the system. A process that shares data with other processes is cooperating and a process that does not share data is independent.

Advantages of Cooperating Processes:
There are some advantages of cooperating processes:

Information Sharing: Several users may which to share the same information e.g. a shared file. The O/S needs to provide a way of allowing concurrent access.

Computation Speedup: Some problems can be solved quicker by sub-dividing it into smaller tasks that can be executed in parallel on several processors.

Modularity: The solution of a problem is structured into parts with well-defined interfaces, and where the parts run in parallel.

Convenience: A user may be running multiple processes to achieve a single goal, or where a utility may invoke multiple components, which interconnect via a pipe structure that attaches the stdout of one stage to stdin of the next etc.

If we allow processes to execute concurrently and share data, then we must either provide some mechanisms to handle conflicts e.g. writing and reading the same piece of data. We must also be prepared to handle inconsistent or corrupted data.

3.1.8 System calls

The interface between the operating system and the user programs is defined by the set of system calls that the operating system provides. To really understand what operating systems do, we must examine this interface closely. The system calls available in the interface vary from operating system to operating system (although the underlying concepts tend to be similar).

Modern operating systems have system calls that perform the same functions, even if the details differ. Since the actual mechanics of issuing a system call are highly machine dependent and often must be expressed in assembly code, a procedure library is provided to make it possible to make system calls from C programs and often from other languages as well.

Any single-CPU computer can execute only one instruction at a time. If a process is running a user program in user mode and needs a system service, such as reading data from a file, it has to execute a trap or system call instruction to transfer control to the operating system. The operating system then figures out what the calling process wants by inspecting the parameters. Then it carries out the system call and returns control to the instruction following the system call. In a sense, making a system call is like making a special kind of procedure call, only system calls enter the kernel and procedure calls do not.

In the following sections, we will examine some of the most heavily used system calls:

Process management

Call	Description
pid = fork()	Create a child process identical to the parent
pid = waitpid(pid, &statloc, options)	Wait for a child to terminate
s = execve(name, argv, environp)	Replace a process' core image
exit(status)	Terminate process execution and return status

File management

Call	Description
fd = open (file, how,...)	Open a file for reading, writing or both
s = close(fd)	Close an open file
n = read(fd, buffer, nbytes)	Read data from a file into a buffer
n = write(fd, buffer, nbytes)	Write data from a buffer into a file
position = lseek(fd, offset, whence)	Move the file pointer
s = stat(name, &buf)	Get a file's status information

Directory and file system management

Call	Description
s = mkdir(name, mode)	Create a new directory
s = rmdir(name)	Remove an empty directory
s = link(name1, name2)	Create a new entry, name2, pointing to name1
s = unlink(name)	Remove a directory entry
s = mount(special, name, flag)	Mount a file system
s = umount(special)	Unmount a file system

Miscellaneous

Call	Description
s = chdir(dirname)	Change the working directory
s = chmod(name, mode)	Change a file's protection bits
s = kill(pid, signal)	Send a signal to a process
seconds = time(&seconds)	Get the elapsed time since Jan. 1, 1970

3.2 Threads

In traditional operating systems, each process has an address space and a single thread of control. In fact, that is almost the definition of a process. Nevertheless, there are frequently situations in which it is desirable to have multiple threads of control in the same address space running in quasi-parallel, as though they were separate processes (except for the shared address space).

3.2.1 Definitions of threads

Threads, sometimes called lightweight processes (LWPs) are independently scheduled parts of a single program. We say that a task is *multithreaded* if it is composed of several

independent sub-processes which do work on common data, and if each of those pieces could (at least in principle) run in parallel. If we write a program which uses threads – there is only one program, one executable file, one task in the normal sense. Threads simply enable us to split up that program into logically separate pieces, and have the pieces run independently of one another, until they need to communicate. In a sense, threads are a further level of *object orientation* for multitasking systems. They allow certain *functions* to be executed in parallel with others.

On a truly parallel computer (several CPUs) we might imagine parts of a program (different subroutines) running on quite different processors, until they need to communicate. When one part of the program needs to send data to the other part, the two independent pieces must be synchronized, or be made to wait for one another. But what is the point of this? We can always run independent procedures in a program as separate *programs*, using the process mechanisms we have already introduced. They could communicate using normal interprocesses communication. Why introduce another new concept? Why do we need threads?

The point is that threads are cheaper than normal processes, and that they can be scheduled for execution in a user-dependent way, with less overhead. Threads are cheaper than a whole process because they do not have a full set of resources each. Whereas the process control block for a heavyweight process is large and costly to context switch, the PCBs for threads are much smaller, since each thread has only a stack and some registers to manage. It has no open file lists or resource lists, no accounting structures to update. All of these resources are shared by all threads within the process. Threads can be assigned priorities – a higher priority thread will get put to the front of the queue.

Similarities between process and threads:

- Like process, threads share CPU and only one thread active (running) at a time.
- Like process, thread can create children.
- Like process, threads within process execute sequentially.
- Like the process, if one thread is blocked, another thread can run.

Difference between Process and Thread

S.N.	Process	Thread
1	Process is heavy weight or resource intensive.	Thread is light weight taking lesser resources than a process.
2	Process switching needs interaction with operating system.	Thread switching does not need to interact with operating system.
3	In multiple processing environments each process executes the same code but has its own memory and file resources.	All threads can share same set of open files, child processes.
4	If one process is blocked then no other process can execute until the first process is unblocked.	While one thread is blocked and waiting, second thread in the same task can run.

5	Multiple processes without using threads use more resources.	Multiple threaded processes use fewer resources.
6	In multiple processes each process operates independently of the others.	One thread can read, write or change another thread's data.

Advantages of threads:

- The process becomes co-operative.
- The internal process becomes strong by the use of thread.
- Each thread assign their respective work so that, three is possibility of work mismatch.
- The thread works as a small part of process, so that when the thread becomes efficient then whole process becomes efficient and reliable.
- Threads allow multiple execution streams.
- Threads are popular way to improve application through parallelism.
- Context switching.

Disadvantage of thread:

- The process becomes complex in multithreading system.
- When the threads are necessary for modification, then the whole process would be affected.
- There may not be the complete process by single thread.
- When the error occurs to thread then the entire process may kill.
- Threads are not isolated as they don't have their own address space.
- Not reusable.
- Less security.

3.2.2 Types of thread process (single and multithreaded process)

One way of looking at a process is that it is way to group related resources together. A process has an address space containing program text and data, as well as other resources. These resources may include open files, child processes, pending alarms, signal handlers, accounting information, and more. By putting them together in the form of a process, they can be managed more easily.

The other concept a process has is a thread of execution, usually shortened to just **thread.** The thread has a program counter that keeps track of which instruction to execute next. It has registers, which hold its current working variables. It has a stack, which contains the execution history, with one frame for each procedure called but not yet returned from. Although a thread must execute in some process, the thread and its process are different concepts and can be treated separately. Processes are used to group resources together; threads are the entities scheduled for execution on the CPU.

What threads add to the process model is to allow multiple executions to take place in the same process environment, to a large degree independent of one another. Having multiple

threads running in parallel in one process is analogous to having multiple processes running in parallel in one computer. In the former case, the threads share an address space, open files, and other resources. In the latter case, processes share physical memory, disks, printers, and other resources. Because threads have some of the properties of processes, they are sometimes called **lightweight processes**. The term **multithreading** is also used to describe the situation of allowing multiple threads in the same process.

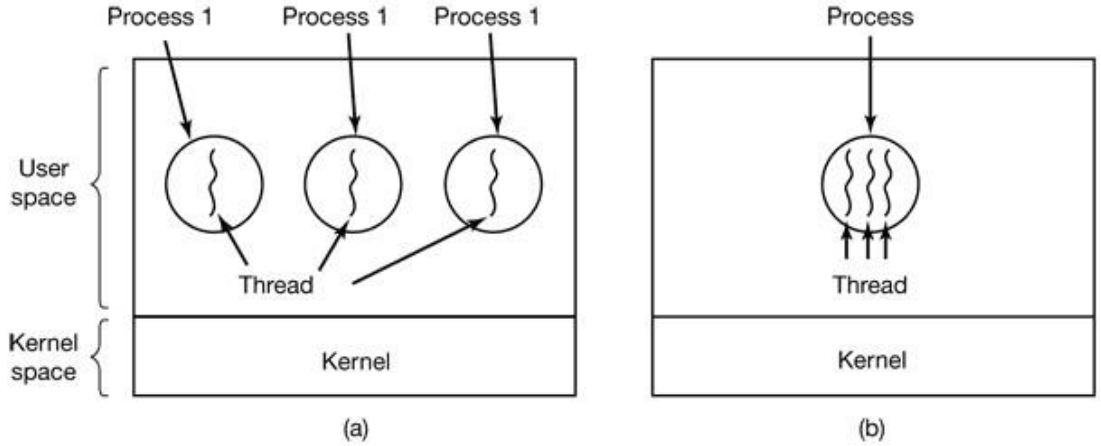


Fig: (a) Three processes each with one thread. (b) One process with tree threads.

When a multithreaded process is run on a single-CPU system, the threads take turns running. By switching back and forth among multiple processes, the system gives the illusion of separate sequential processes running in parallel. Multithreading works the same way. The CPU switches rapidly back and forth among the threads providing the illusion that the threads are running in parallel, albeit on a slower CPU than the real one. With three compute-bound threads in a process, the threads would appear to be running in parallel, each one on a CPU with one-third the speed of the real CPU.

Different threads in a process are not quite as independent as different processes. All threads have exactly the same address space, which means that they also share the same global variables. Since every thread can access every memory address within the process' address space, one thread can read, write, or even completely wipe out another thread's stack. There is no protection between threads because (1) it is impossible, and (2) it should not be necessary. Unlike different processes, which may be from different users and which may be hostile to one another, a process is always owned by a single user, who has presumably created multiple threads so that they can cooperate, not fight. In addition to sharing an address space, all the threads share the same set of open files, child processes, alarms, and signals, etc.

Like a traditional process (i.e., a process with only one thread), a thread can be in any one of several states: running, blocked, ready, or terminated. A running thread currently has the CPU and is active. A blocked thread is waiting for some event to unblock it. For example, when a thread performs a system call to read from the keyboard, it is blocked until input is typed. A thread can block waiting for some external event to happen or for some other thread to unblock it. A ready thread is scheduled to run and will as soon as its turn comes up. It is important to realize that each thread has its own stack, as shown in Fig. 2-8. Each thread's stack contains

one frame for each procedure called but not yet returned from. This frame contains the procedure's local variables and the return address to use when the procedure call has finished. For example, if procedure *X* calls procedure *Y* and this one calls procedure *Z*, while *Z* is executing the frames for *X*, *Y* and *Z* will all be on the stack. Each thread will generally call different procedures and a thus a different execution history. This is why thread needs is its own stack.

When multithreading is present, processes normally start with a single thread present. This thread has the ability to create new threads by calling a library procedure, for example, *thread_create.* A parameter to *thread_create* typically specifies the name of a procedure for the new thread to run. It is not necessary (or even possible) to specify anything about the new thread's address space since it automatically runs in the address space of the creating thread. Sometimes threads are hierarchical, with a parent-child relationship, but often no such relationship exists, with all threads being equal. With or without a hierarchical relationship, the creating thread is usually returned a thread identifier that names the new thread.

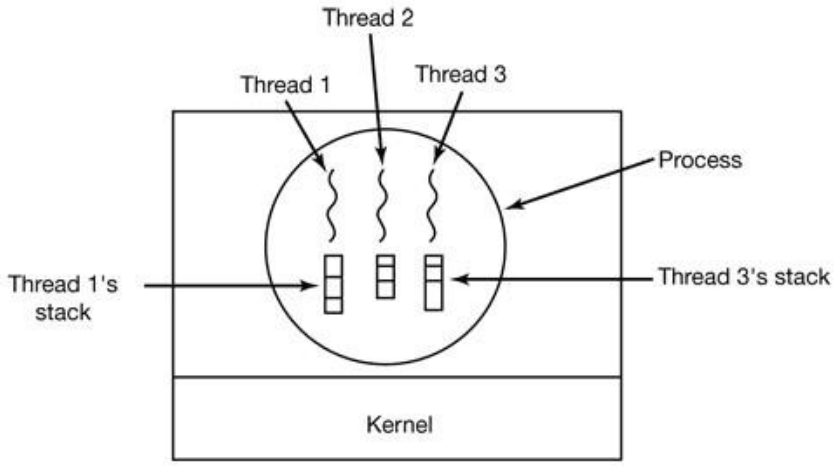


Fig: Each thread has its own stack.

3.2.3 Benefits of multithread

1. Responsiveness
2. Resource sharing, hence allowing better utilization of resources.
3. Economy. Creating and managing threads become 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

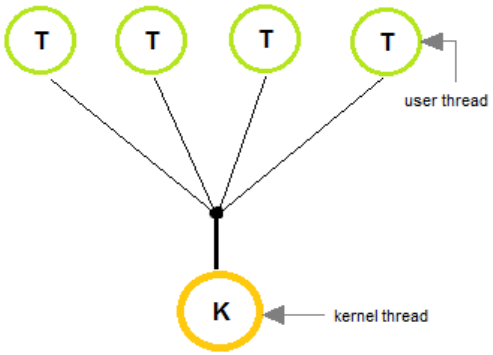
the target thread immediately. The other is **Deferred cancellation** allows the target thread to periodically check if it should be cancelled.

2. **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.
3. **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?
4. **Security Issues** because of extensive sharing of resources between multiple threads.

3.2.4 Multithreading models

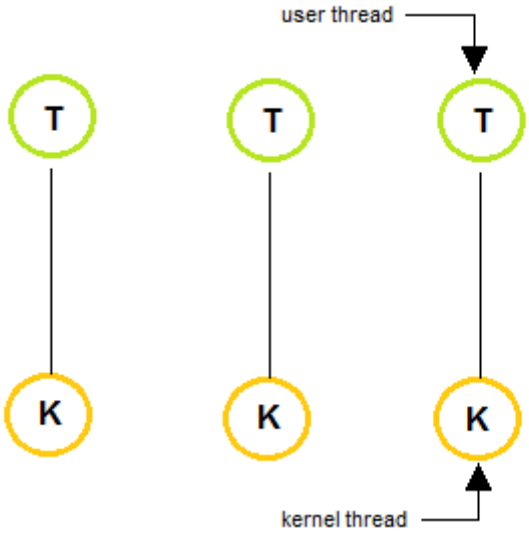
1. Many to one model:

- In 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.



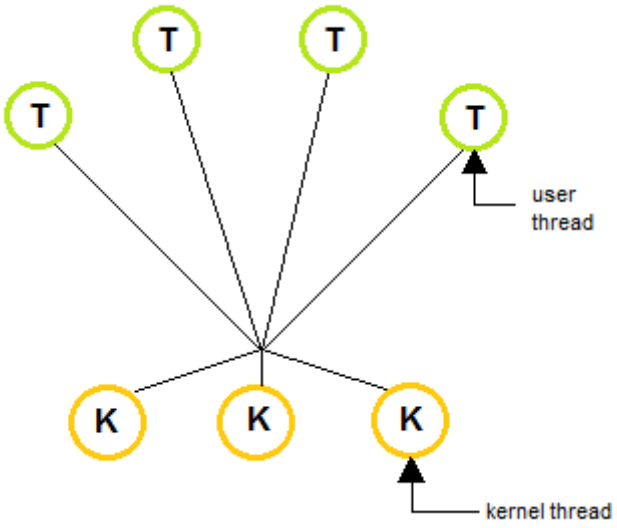
2. One to One model:

- 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.



3. 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 many 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.



3.3 Inter-process communication and synchronization

3.3.1 Introduction

Processes frequently need to communicate with other processes. For example, in a shell pipeline, the output of the first process must be passed to the second process, and so on down

the line. Thus there is a need for communication between processes, preferably in a well-structured way not using interrupts.

Very briefly, there are three issues here. The first was: how one process can pass information to another. The second has to do with making sure two or more processes do not get into each other's way when engaging in critical activities (suppose two processes each try to grab the last 1 MB of memory). The third concerns proper sequencing when dependencies are present: if process *A* produces data and process *B* prints them, *B* has to wait until *A* has produced some data before starting to print.

3.3.2 Race condition

In some operating systems, processes that are working together may share some common storage that each one can read and write. The shared storage may be in main memory (possibly in a kernel data structure) or it may be a shared file: the location of the shared memory does not change the nature of the communication or the problems that arise.

To see how interprocess communication works in practice, let us consider a simple but common example: a print spooler. When a process wants to print a file, it enters the file name in a special **spooler directory**. Another process, the **printer daemon**, periodically checks to see if there are any files to be printed, and if there are, it prints them and then removes their names from the directory.

Imagine that our spooler directory has a very large number of slots, numbered 0, 1, 2, …, each one capable of holding a file name. Also imagine that there are two shared variables, *out*, which points to the next file to be printed, and *in*, which points to the next free slot in the directory. These two variables might well be kept on a two-word file available to all processes. At a certain instant, slots 0 to 3 are empty (the files have already been printed) and slots 4 to 6 are full (with the names of files queued for printing). More or less simultaneously, processes *A* and *B* decide they want to queue a file for printing.

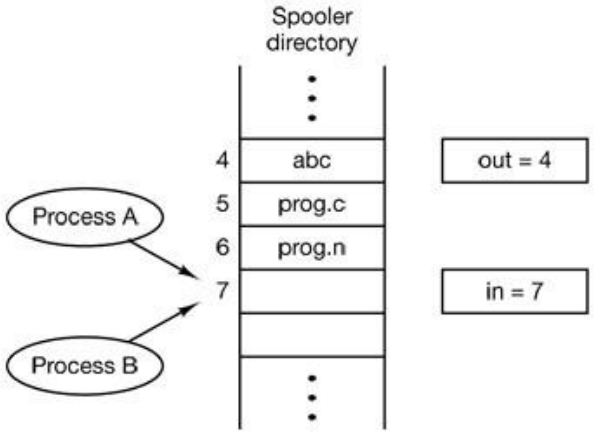


Fig: Two processes want to access shared memory at the same time.

Process *A* reads *in* and stores the value, 7, in a local variable called *next_free_slot*. Just then a clock interrupt occurs and the CPU decides that process *A* has run long enough, so it switches to process *B*, Process *B* also reads *in*, and also gets a 7. It too stores it in *its* local variable *next_free_slot*. At this instant both processes think that the next available slot is 7.

Process *B* now continues to run. It stores the name of its file in slot 7 and updates *in* to be an 8. Then it goes off and does other things.

Eventually, process *A* runs again, starting from the place it left off. It looks at *next_free_slot*, finds a 7 there, and writes its file name in slot 7, erasing the name that process *B* just put there. Then it computes *next_free_slot* + 1, which is 8, and sets *in* to 8. The spooler directory is now internally consistent, so the printer daemon will not notice anything wrong, but process *B* will never receive any output. User *B* will hang around the printer room for years, wistfully hoping for output that never comes.

Situations like this, where two or more processes are reading or writing some shared data and the final result depends on who runs precisely when, are called **race conditions**. Debugging programs containing race conditions is no fun at all. The results of most test runs are fine, but once in a rare while something weird and unexplained happens.

How do we avoid race conditions? The key to preventing trouble here and in many other situations involving shared memory, shared files, and shared everything else is to find some way to prohibit more than one process from reading and writing the shared data at the same time. Put in other words, what we need is **mutual exclusion**, that is, some way of making sure

that if one process is using a shared variable or file, the other processes will be excluded from doing the same thing. The difficulty above occurred because process *B* started using one of the shared variables before process *A* was finished with it. The choice of appropriate primitive operations for achieving mutual exclusion is a major design issue in any operating system

3.3.3 Critical regions

Sometimes processes have to access shared memory or files, or doing other critical things that can lead to races. That part of the program where the shared memory is accessed is called the **critical region** or **critical section**. If we could arrange matters such that no two processes were ever in their critical regions at the same time, we could avoid races. We need four conditions to hold to have a good solution:

1. No two processes may be simultaneously inside their critical regions.
2. No assumptions may be made about speeds or the number of CPUs.
3. No process running outside its critical region may block other processes.
4. No process should have to wait forever to enter its critical region.

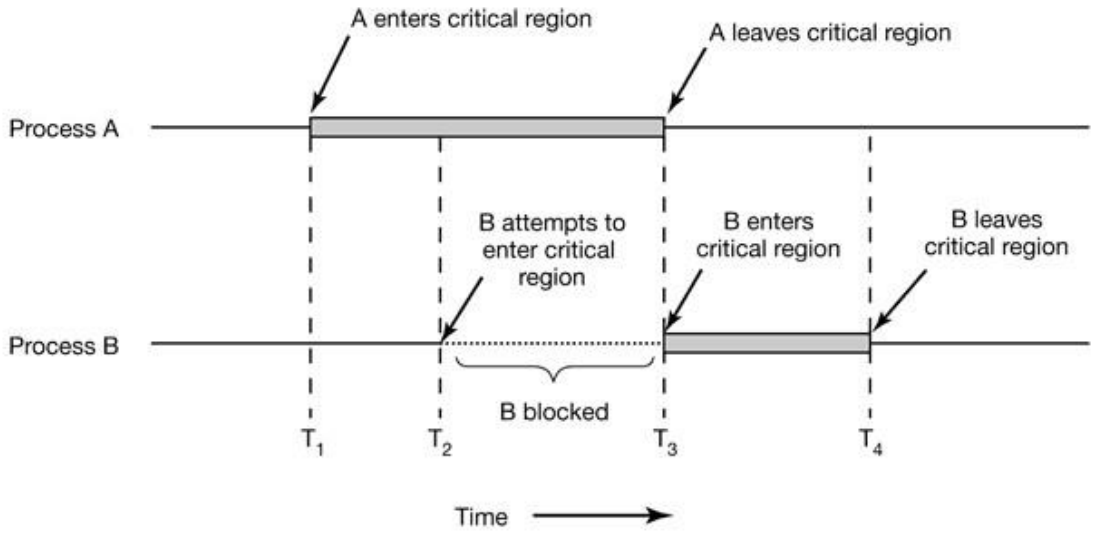


Fig: Mutual exclusion using critical regions.

Here process *A* enters its critical region at time T_1, A little later, at time T_2 process *B* attempts to enter its critical region but fails because another process is already in its critical region and we allow only one at a time. Consequently, *B* is temporarily suspended until time T_3 when *A* leaves its critical region, allowing *B* to enter immediately. Eventually *B* leaves (at T_4) and we are back to the original situation with no processes in their critical regions.

3.3.4 Mutual exclusion

A way of making sure that if one process is using a shared modifiable data, the other processes will be excluded from doing the same thing.

Formally, while one process executes the shared variable, all other processes desiring to do so at the same time moment should be kept waiting; when that process has finished executing the

shared variable, one of the processes waiting; while that process has finished executing the shared variable, one of the processes waiting to do so should be allowed to proceed. In this fashion, each process executing the shared data (variables) excludes all others from doing so simultaneously. This is called Mutual Exclusion.

Note that mutual exclusion needs to be enforced only when processes access shared modifiable data - when processes are performing operations that do not conflict with one another they should be allowed to proceed concurrently.

Mutual Exclusion conditions

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

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

3.3.5 Proposals for achieving mutual exclusion:

In this section we will examine various proposals for achieving mutual exclusion, so that while one process is busy updating shared memory in its critical region, no other process will enter *its* critical region and cause trouble.

Disabling Interrupts

The simplest solution is to have each process disable all interrupts just after entering its critical region and re-enable them just before leaving it. With interrupts disabled, no clock interrupts can occur. The CPU is only switched from process to process as a result of clock or other interrupts, after all, and with interrupts turned off the CPU will not be switched to another process. Thus, once a process has disabled interrupts, it can examine and update the shared memory without fear that any other process will intervene.

Lock Variables

As a second attempt, let us look for a software solution. Consider having a single, shared (lock) variable, initially 0. When a process wants to enter its critical region, it first tests the lock. If the lock is 0, the process sets it to 1 and enters the critical region. If the lock is already 1, the process just waits until it becomes 0. Thus, a 0 means that no process is in its critical region and a 1 means that some process is in its critical region.

Strict Alternation

The integer variable *turn*, initially 0, keeps track of whose turn it is to enter the critical region and examine or update the shared memory. Initially, process 0 inspects *turn*, finds it to be 0, and enters its critical region. Process 1 also finds it to be 0 and therefore sits in a tight loop

continually testing *turn* to see when it becomes 1. Continuously testing a variable until some value appears is called **busy waiting**. It should usually be avoided, since it wastes CPU time. Only when there is a reasonable expectation that the wait will be short is busy waiting used. A lock that uses busy waiting is called a **spin lock**.

```
while (TRUE) {                          while (TRUE) {
while (turn != 0)   /* loop */ ;        while (turn != 1);  /* loop */ ;
    critical_region();                  critical_region();     turn = 0;
turn = 1;                               noncritical_region();
    noncritical_region();               }
        }
    (a)                                 (b)
```

Fig: A proposed solution to the critical region problem. (a) Process 0. (b) Process 1.

When process 0 leaves the critical region, it sets *turn* to 1, to allow process 1 to enter its critical region. Suppose that process 1 finishes its critical region quickly, so both processes are in their noncritical regions, with *turn* set to 0. Now process 0 executes its whole loop quickly, exiting its critical region and setting *turn* to 1. At this point *turn* is 1 and both processes are executing in their noncritical regions.

Peterson's Solution

In 1981, G.L. Peterson discovered a much simpler way to achieve mutual exclusion. Peterson's algorithm is shown in Fig. This algorithm consists of two procedures written in ANSI C, which means that function prototypes should be supplied for all the functions defined and used.

```
#define FALSE 0
#define TRUE  1
#define N     2      /* number of processes */
 int turn;           /* whose turn is it? */
int interested[N];  /* all values initially 0 (FALSE) */
 void enter_region(int process)      /* process is 0 or 1 */
{
   int other;                   /* number of the other process */
    other = 1 − process;            /* the opposite of process */
interested[process] = TRUE;     /* show that you are interested */
turn = process;                 /* set flag */
while (turn == process && interested[other] == TRUE)/* null statement */;  }
void leave_region (int process)     /* process, who is leaving */
{
  interested[process] = FALSE;/* indicate departure from critical region */
 }
```

Fig: Peterson's solution for achieving mutual exclusion.

Before using the shared variables (i.e., before entering its critical region), each process calls *enter_region* with its own process number, 0 or 1, as parameter. This call will cause it to wait, if need be, until it is safe to enter. After it has finished with the shared variables, the process calls *leave_region* to indicate that it is done and to allow the other process to enter, if it so desires.

Initially neither process is in its critical region. Now process 0 calls *enter_region.* It indicates its interest by setting its array element and sets *turn* to 0. Since process 1 is not interested, *enter_region* returns immediately. If process 1 now calls *enter_region,* it will hang there until *interested*[0] goes to *FALSE,* an event that only happens when process 0 calls *leave_region* to exit the critical region.

The TSL Instruction

Now let us look at a proposal that requires a little help from the hardware. Many computers, especially those designed with multiple processors in mind, have an instruction

TSL RX,LOCK

(Test and Set Lock) that works as follows. It reads the contents of the memory word *lock* into register RX and then stores a nonzero value at the memory address *lock.* The operations of reading the word and storing into it are guaranteed to be indivisible—no other processor can access the memory word until the instruction is finished. The CPU executing the TSL instruction locks the memory bus to prohibit other CPUs from accessing memory until it is done.

To use the TSL instruction, we will use a shared variable, *lock,* to coordinate access to shared memory. When *lock* is 0, any process may set it to 1 using the TSL instruction and then read or write the shared memory. When it is done, the process sets *lock* back to 0 using an ordinary move instruction.

```
enter_region:
   TSL REGISTER,LOCK   | copy lock to register and set lock to 1
   CMP REGISTER,#0     | was lock zero?
   JNE enter_region    | if it was non zero, lock was set, so loop
   RET | return to caller; critical region entered
leave_region:
   MOVE LOCK,#0        | store a 0 in lock
   RET | return to caller
```

Fig: Entering and leaving a critical region using the TSL instruction.

One solution to the critical region problem is now straightforward. Before entering its critical region, a process calls *enter_region,* which does busy waiting until the lock is free; then it acquires the lock and returns. After the critical region the process calls *leave_region*, which stores a 0 in *lock.* As with all solutions based on critical regions, the processes must call *enter_region* and *leave_region* at the correct times for the method to work.

3.3.6 Sleep and Wakeup

Both Peterson's solution and the solution using TSL are correct, but both have the defect of requiring busy waiting. In essence, what these solutions do is this: when a process wants to enter its critical region, it checks to see if the entry is allowed. If it is not, the process just sits in a tight loop waiting until it is.

Not only does this approach waste CPU time, but it can also have unexpected effects. Consider a computer with two processes, *H*, with high priority and *L*, with low priority. The scheduling rules are such that *H* is run whenever it is in ready state. At a certain moment, with *L* in its critical region, *H* becomes ready to run (e.g., an I/O operation completes). *H* now begins busy waiting, but since *L* is never scheduled while *H* is running, *L* never gets the chance to leave its critical region, so *H* loops forever. This situation is sometimes referred to as the **priority inversion problem.**

Now let us look at some inter process communication primitives that block instead of wasting CPU time when they are not allowed to enter their critical regions. One of the simplest is the pair sleep and wakeup. Sleep is a system call that causes the caller to block, that is, be suspended until another process wakes it up. The wakeup call has one parameter, the process to be awakened. Alternatively, both sleep and wakeup each have one parameter, a memory address used to match up sleeps with wakeups.

The Producer-Consumer Problem

As an example of how these primitives can be used, let us consider the **producer-consumer** problem (also known as the **bounded-buffer** problem). Two processes share a common, fixed-size buffer. One of them, the producer, puts information into the buffer, and the other one, the consumer, takes it out.

Trouble arises when the producer wants to put a new item in the buffer, but it is already full. The solution is for the producer to go to sleep, to be awakened when the consumer has removed one or more items. Similarly, if the consumer wants to remove an item from the buffer and sees that the buffer is empty, it goes to sleep until the producer puts something in the buffer and wakes it up.

This approach sounds simple enough, but it leads to the same kinds of race conditions we saw earlier with the spooler directory. To keep track of the number of items in the buffer, we will need a variable, *count*. If the maximum number of items the buffer can hold is *N*, the producer's code will first test to see if *count* is *N*. If it is, the producer will go to sleep; if it is not, the producer will add an item and increment *count*.

The consumer's code is similar: first test *count* to see if it is 0. If it is, go to sleep, if it is nonzero, remove an item and decrement the counter. Each of the processes also tests to see if the other should be awakened, and if so, wakes it up.

```
#define N 100       /* number of slots in the buffer */
int count = 0;      /* number of items in the buffer */

void producer (void)
{
   int item;
while (TRUE) {                /* repeat forever */
  item = produce_item();     /* generate next item */
 if (count == N) sleep();      /* if buffer is full, go to sleep */
insert_item(item);         /* put item in buffer */
 count = count + 1;         /* increment count of items in buffer */
  if (count == 1) wakeup(consumer);  /* was buffer empty? */
   }
}

void consumer(void)
{
   int item;
   while (TRUE) {                /* repeat forever */
      if (count == 0) sleep();   /* if buffer is empty, got to sleep */
  item = remove_item();      /* take item out of buffer */
 count = count − 1;         /* decrement count of items in buffer */
   if (count == N − 1) wakeup(producer); /* was buffer full? */
consume_item(item);        /* print item */
   }
}
```

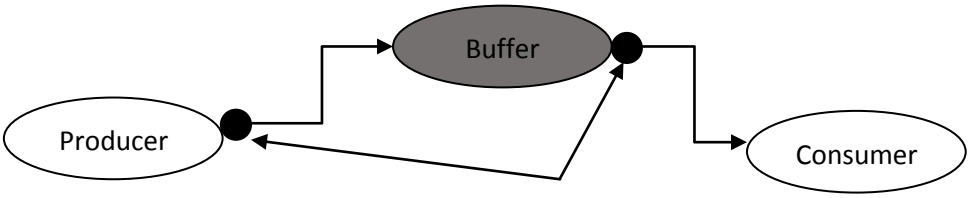
Fig: The producer-consumer problem with a fatal race condition.

3.3.7 Types of mutual exclusion

Semaphores

This was the situation in 1965, when E. W. Dijkstra (1965) suggested using an integer variable to count the number of wakeups saved for future use. In his proposal, a new variable type, called a semaphore, was introduced. A semaphore could have the value 0, indicating that no wakeups were saved, or some positive value if one or more wakeups were pending. Dijkstra proposed having two operations, down and up (generalizations of sleep and wakeup, respectively). The down operation on a semaphore checks to see if the value is greater than 0. If so, it decrements the value (i.e., uses up one stored wakeup) and just continues. If the value is 0, the process is put to sleep without completing the down for the moment. Checking the value, changing it and possibly going to sleep, is all done as a single, indivisible **atomic**

action. It is guaranteed that once a semaphore operation has started, no other process can access the semaphore until the operation has completed or blocked. This atomicity is absolutely essential to solving synchronization problems and avoiding race conditions.



Semaphore variable

Solving the Producer-Consumer Problem using Semaphores

Semaphores solve the lost-wakeup problem, as shown in Fig. It is essential that they be implemented in an indivisible way. The normal way is to implement up and down as system calls, with the operating system briefly disabling all interrupts while it is testing the semaphore, updating it, and putting the process to sleep, if necessary. As all of these actions take only a few instructions, no harm is done in disabling interrupts. If multiple CPUs are being used, each semaphore should be protected by a lock variable, with the TSL instruction used to make sure that only one CPU at a time examines the semaphore. Be sure you understand that using TSL to prevent several CPUs from accessing the semaphore at the same time is quite different from busy waiting by the producer or consumer waiting for the other to empty or fill the buffer. The semaphore operation will only take a few microseconds, whereas the producer or consumer might take arbitrarily long.

```
#define N 100            /* number of slots in the buffer */
 typedef int semaphore;     /* semaphores are a special kind of int */
semaphore mutex = 1;       /* controls access to critical region */
semaphore empty = N;       /* counts empty buffer slots */
semaphore full = 0;        /* counts full buffer slots */

void producer(void)
{
   int item;
   while (TRUE) {            /* TRUE is the constant 1 */
   item = produce_item(); /* generate something to put in buffer */
down(&empty);          /* decrement empty count */
 down(&mutex);          /* enter critical region */
  insert_item(item);     /* put new item in buffer */
up(&mutex);           /* leave critical region */
up(&full);            /* increment count of full slots */      }
}

void consumer(void)
{
   int item;
   while (TRUE) {            /* infinite loop */
down(&full);                    /*  decrement  full  count  */
down(&mutex);          /* enter critical region */
  item a= remove_item(); /* take item from buffer */
up(&mutex);           /* leave critical region */
 up(&empty);           /* increment count of empty slots */
consume_item(item);    /* do something with the item */
   }
}
```

Fig: The producer-consumer problem using semaphores.

Introduction to message passing

This method of inter process communication uses two primitives, send and receive, which, like semaphores and unlike monitors, are system calls rather than language constructs. As such, they can easily be put into library procedures, such as

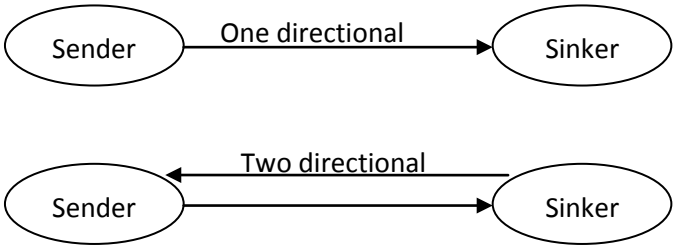
send(destination, &message);
and
receive(source, &message);

The former call sends a message to a given destination and the latter one receives a message from a given source (or from *ANY*, if the receiver does not care). If no message is available, the receiver can block until one arrives. Alternatively, it can return immediately with an error code.

Message passing systems have many challenging problems and design issues that do not arise with semaphores or monitors, especially if the communicating processes are on different machines connected by a network. For example, messages can be lost by the network. To guard against lost messages, the sender and receiver can agree that as soon as a message has been received, the receiver will send back a special **acknowledgement** message. If the sender has not received the acknowledgement within a certain time interval, it retransmits the message.

Now consider what happens if the message itself is received correctly, but the acknowledgement is lost. The sender will retransmit the message, so the receiver will get it twice. It is essential that the receiver be able to distinguish a new message from the retransmission of an old one. Usually, this problem is solved by putting consecutive sequence numbers in each original message. If the receiver gets a message bearing the same sequence number as the previous message, it knows that the message is a duplicate that can be ignored. Successfully communicating in the face of unreliable message passing is a major part of the study of computer networks.

Message passing is commonly used in parallel programming systems. One well-known message-passing system, for example, is **MPI** (**Message-Passing Interface**). It is widely used for scientific computing.



The communication link in this scheme has following properties:

1. A link is established automatically between every pair of processes that want to communicate. The process need to know only each other's identify to communicate.
2. A link is associated with exactly two processes.
3. Exactly one link exists between each pair of processes.

Mutexes:

When the semaphore's ability to count is not needed, a simplified version of the semaphore, called a mutex, is sometimes used. Mutexes are good only for managing mutual exclusion to some shared resource or piece of code. They are easy and efficient to implement, which makes them especially useful in thread packages that are implemented entirely in user space.

A **mutex** is a variable that can be in one of two states: unlocked or locked. Consequently, only 1 bit is required to represent it, but in practice an integer often is used, with 0 meaning unlocked and all other values meaning locked. Two procedures are used with mutexes. When a thread (or process) needs access to a critical region, it calls *mutex_lock*. If the mutex is current unlocked (meaning that the critical region is available), the call succeeds and the calling thread is free to enter the critical region.

On the other hand, if the mutex is already locked, the calling thread is blocked until the thread in the critical region is finished and calls *mutex_unlock*. If multiple threads are blocked on the mutex, one of them is chosen at random and allowed to acquire the lock.

Because mutexes are so simple, they can easily be implemented in user space if a TSL instruction is available. The code for *mutex_lock* and *mutex_unlock* for use with a user-level threads package are shown in Fig.

```
mutex_lock:
   TSL REGISTER,MUTEX | copy mutex to register and set mutex to 1
   CMP REGISTERS,#0   | was mutex zero?
   JZE ok             | if it was zero, mutex was unlocked, so return
   CALL thread_yield  | mutex is busy; schedule another thread
   JMP mutex_lock     | try again later
ok:   RET | return to caller; critical region entered

mutex_unlock:
   MOVE MUTEX,#0      | store a 0 in mutex
   RET | return to caller
```

Monitors

A monitor is a set of multiple routines which are protected by a mutual exclusion lock. None of the routines in the monitor can be executed by a thread until that thread acquires the lock. This means that only ONE thread can execute within the monitor at a time. Any other threads must wait for the thread that's currently executing to give up control of the lock.

However, a thread can actually suspend itself inside a monitor and then wait for an event to occur. If this happens, then another thread is given the opportunity to enter the monitor. The thread that was suspended will eventually be notified that the event it was waiting for has now occurred, which means it can wake up and reacquire the lock.

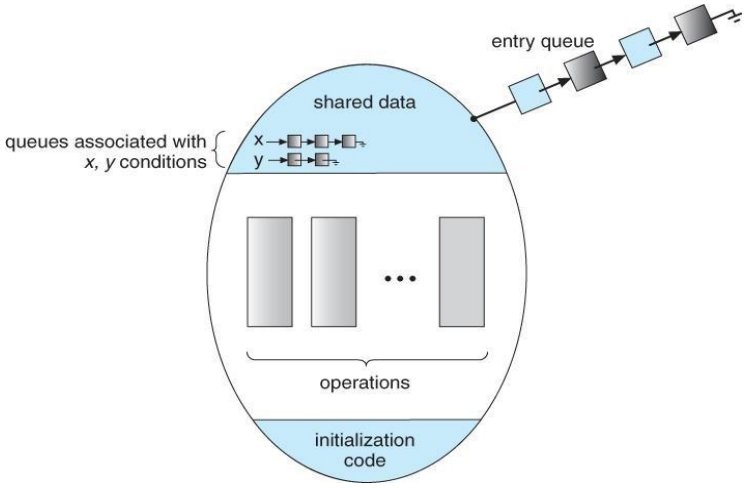


Fig: Monitor

```
monitor example
    integer i;
    condition c;
     procedure producer( );
    .
    .
    .
    end;

    procedure consumer( );
    .    .    .
    end;
end monitor;
```

3.3.8 Serializability: Locking protocols and time stamp protocols

1. Locking Protocols:

A transaction follows the two phase locking protocol, if all first unlock operations precede the first unlock operation in the transaction. This protocol ensures that each data item used by a transaction should be locked. These locks are of two types:

a) Shared
b) Exclusive

If a transaction T_1 has locked a data item which transaction T_2 also wants to lock then the following possibilities are there:

i) If T_1 has a shared lock on data-item, T_2 can also lock it in shared mode.
ii) If T_1 has exclusive lock on a data item, T_2 cannot lock it unless T_1 releases this data item.
iii) If T_1 unlocked the data item then T_2 can lock it either in shared or in exclusive mode, which ever is required.

Please note here that a transaction cannot unlock a data item as long as it access that data item. Even after the transaction has undergone last access of that data item, it may not release it to ensure serializability.

The two phase locking protocol ensures serializability. This protocol requires that each transaction should issue lock and unlock requests in two phases:

a. Growing phase: In this phase, a transaction may obtain locks but may not release any lock. Please note that in this phase, the number of locks increases from zero to the maximum for the transaction.

b. Shrinking phase: In this phase, a transaction may release locks but may not obtain any new locks. Please note that in this phase, the number of locks decreases from maximum to zero.
Initially, a transaction is in the growing phase. The transaction an acquire all the needed locks in this phase. Once, the transaction releases a lock, it enters the shrinking phase in which no new locks can be obtained. This protocol ensures serializability but is not free from deadlock.

2. Time stamp based protocols:
Time stamping is a concurrency control protocol in which the fundamental goal is to order transactions globally in such a way that older transactions get priority in the event of a conflict. This protocol requires that we must have prior knowledge about the order in which the transactions will be accessed. For ordering the transactions, we associate a unique fixed timestamp. These timestamps are increasing numbers only. And they are denoted by $TS(T_1)$ for each transaction. If a transaction T_i has been assigned timestamp as $TS(T_1)$ and a new transaction T_j enters system then,
$TS(T_i) < TS(T_j)$
This timestamp may be implemented by using the value of a system clock i.e. a transactions timestamp is equal to the value of the clock when the transaction enters the system. Or, one may use a logical counter that is incremented after a new timestamp has been assigned.

The timestamps of transactions determine the serializability order. Thus, if $TS(T_i) < TS(T_j)$ then the system must ensure that the produced schedule is equivalent to a serial schedule in which transaction T_i appears before transaction T_j.

3.3.9 Classical IPC problems

The Dining Philosophers Problem

In 1965, Dijkstra posed and solved a synchronization problem he called the **dining philosophers problem**. Since that time, everyone inventing yet another synchronization primitive has felt obligated to demonstrate how wonderful the new primitive is by showing how elegantly it solves the dining philosopher's problem. The problem can be stated quite simply as follows. Five philosophers are seated around a circular table. Each philosopher has a plate of spaghetti. The spaghetti is so slippery that a philosopher needs two forks to eat it. Between each pair of plates is one fork. The layout of the table is illustrated in Fig.

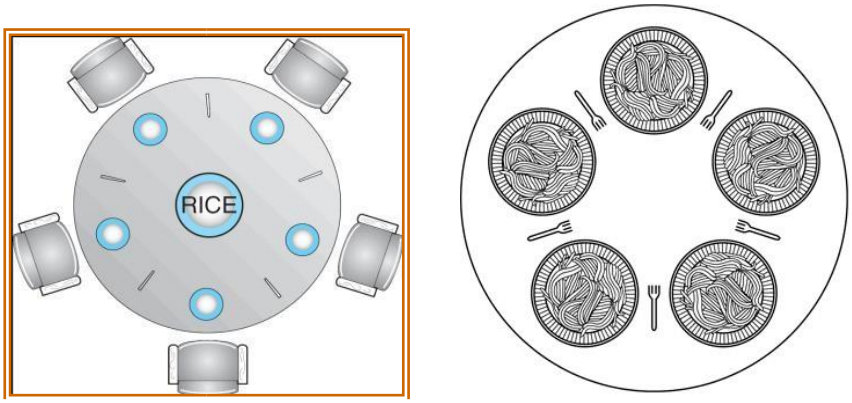


Fig: Lunch time in the Philosophy Department.

The life of a philosopher consists of alternate periods of eating and thinking. When a philosopher gets hungry, she tries to acquire her left and right fork, one at a time, in either order. If successful in acquiring two forks, she eats for a while, then puts down the forks, and continues to think.
The procedure *take_fork* waits until the specified fork is available and then seizes it. Unfortunately, the obvious solution is wrong. Suppose that all five philosophers take their left forks simultaneously. None will be able to take their right forks, and there will be a deadlock.

```
#define N 5                 /* number of philosophers */
  void philosopher(int i)    /* i: philosopher number, from 0 to 4 */
{
   while (TRUE) {
      think( );             /* philosopher is thinking */
take_fork(i);          /* take left fork */
      take_fork((i+1) % N);  /* take right fork; % is modulo operator */
eat();                /* yum-yum, spaghetti */
      put_fork(i);          /* Put left fork back on the table */
      put_fork((i+1) % N);   /* put right fork back on the table */
   }
}
```

Fig: A non solution to the dining philosopher's problem.

We could modify the program so that after taking the left fork, the program checks to see if the right fork is available. If it is not, the philosopher puts down the left one, waits for some time, and then repeats the whole process. With a little bit of bad luck, all the philosophers could start the algorithm simultaneously, picking up their left forks, seeing that their right forks were not available, putting down their left forks, waiting, picking up their left forks again simultaneously, and so on, forever. A situation like this, in which all the programs continue to run indefinitely but fail to make any progress is called **starvation**.

"If the philosophers would just wait a random time instead of the same time after failing to acquire the right-hand fork, the chance that everything would continue in lockstep for even an

hour is very small." This observation is true, and in nearly all applications trying again later is not a problem. For example, in the popular Ethernet local area network, if two computers send a packet at the same time, each one waits a random time and tries again; in practice this solution works fine. However, in a few applications one would prefer a solution that always works and cannot fail due to an unlikely series of random numbers.

The solution presented in Fig. is deadlock-free and allows the maximum parallelism for an arbitrary number of philosophers. It uses an array, *state*, to keep track of whether a philosopher is eating, thinking, or hungry (trying to acquire forks). A philosopher may move only into eating state if neither neighbor is eating. Philosopher *i*'s neighbors are defined by the macros *LEFT* and *RICHT*. In other words, if *i* is 2, *LEFT* is 1 and *RIGHT* is 3.

The program uses an array of semaphores, one per philosopher, so hungry philosophers can block if the needed forks are busy. Note that each process runs the procedure *philosopher* as its main code, but the other procedures, *take_forks*, *put_forks*, and *test* are ordinary procedures and not separate processes.

```
#define N            5   /* number of philosophers */
#define LEFT         (i+N−1)%N /* number of i's left neighbor */
#define RIGHT        (i+1)%N /* number of i's right neighbor */
#define THINKING     0   /* philosopher is thinking */
#define HUNGRY       1   /* philosopher is trying to get forks */
#define EATING       2   /* philosopher is eating */
typedef int semaphore;     /* semaphores are a special kind of int */
int state[N];              /* array to keep track of everyone's state */
semaphore mutex = 1;       /* mutual exclusion for critical regions */
semaphore s[N];            /* one semaphore per philosopher */

void philosopher (int i)   /* i: philosopher number, from 0 to N−1 */
{     while (TRUE) {       /* repeat forever */
think();                   /*  philosopher  is  thinking  */
take_forks(i);     /* acquire two forks or block */
```

```
      eat();            /* yum-yum, spaghetti */
      put_forks(i);     /* put both forks back on table */
}
}

void take_forks(int i)     /* i: philosopher number, from 0 to N−1 */
{
   down(&mutex);         /* enter critical region */
   state[i] = HUNGRY;    /* record fact that philosopher i is hungry */
test(i);             /* try to acquire 2 forks */
 up(&mutex);          /* exit critical region */
   down(&s[i]);         /* block if forks were not acquired */
 }

void put_forks(i)         /* i: philosopher number, from 0 to N−1 */
{
   down(&mutex);         /* enter critical region */
state[i] = THINKING;   /* philosopher has finished eating */
test(LEFT);              /* see if left neighbor can now eat */
test(RIGHT);             /* see if right neighbor can now eat */
up(&mutex);          /* exit critical region */
}

void test(i)             /* i: philosopher number, from 0 to N−1 */
{
  if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING)
{
   state[i] = EATING;
   up(&s[i]);
  }
}
```

Fig: A solution to the dining philosopher's problem.

The Readers and Writers Problem

In this solution, the first reader to get access to the database does a down on the semaphore *db*. Subsequent readers merely increment a counter, *rc*. As readers leave, they decrement the counter and the last one out does an up on the semaphore, allowing a blocked writer, if there is one, to get in.

The solution presented here implicitly contains a subtle decision that is worth commenting on. Suppose that while a reader is using the database, another reader comes along. Since having two readers at the same time is not a problem, the second reader is admitted. A third and subsequent readers can also be admitted if they come along.

Now suppose that a writer comes along. The writer cannot be admitted to the database, since writers must have exclusive access, so the writer is suspended. Later, additional readers show up. As long as at least one reader is still active, subsequent readers are admitted. As a consequence of this strategy, as long as there is a steady supply of readers, they will all get in as soon as they arrive. The writer will be kept suspended until no reader is present. If a new reader arrives, say, every 2 seconds, and each reader takes 5 seconds to do its work, the writer will never get in.

To prevent this situation, the program could be written slightly differently: when a reader arrives and a writer is waiting, the reader is suspended behind the writer instead of being admitted immediately. In this way, a writer has to wait for readers that were active when it arrived to finish but does not have to wait for readers that came along after it. The disadvantage of this solution is that it achieves less concurrency and thus lower performance. Courtois et al. present a solution that gives priority to writers.

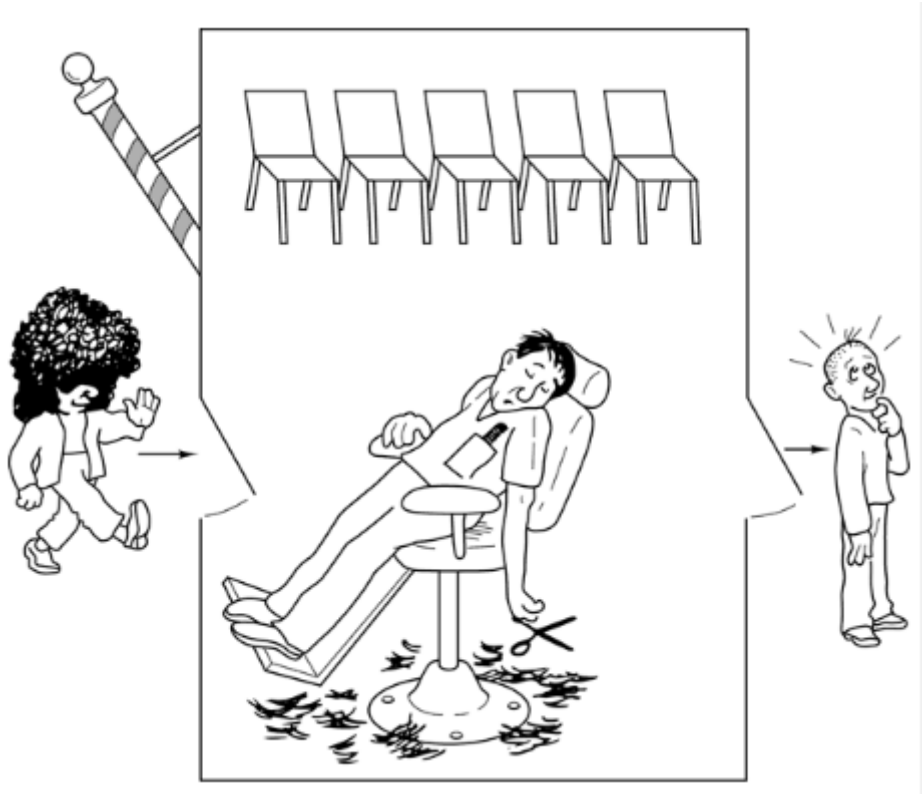
```
typedef int semaphore;        /* use your imagination */
semaphore mutex = 1;          /* controls access to 'rc' */
semaphore db = 1;             /* controls access to the database */
int rc = 0;                   /* # of processes reading or wanting to */

void reader(void)
{
   while (TRUE) {             /* repeat forever */
      down(&mutex);           /* get exclusive access to 'rc' */
      rc = rc + 1;            /* one reader more now */
      if (re == 1) down(&db);    /* if this is the first reader… */
      up{&mutex);             /* release exclusive access to 'rc' */
      read_data_base();       /* access the data */
      down(&mutex);           /* get exclusive access to 'rc' */
      rc = rc − 1;            /* one reader fewer now */
      if (rc == 0) up(&db);   /* if this is the last reader… */
      up(&mutex);             /* release exclusive access to 'rc' */
      use_data_read();        /* noncritical region */
   }
}

void writer(void)
{
   while (TRUE) {             /* repeat forever */
      think_up_data();        /* noncritical region */
      down(&db);              /* get exclusive access */
      write_data_base();      /* update the data */
      up(&db);                /* release exclusive access */
   }
}
```

The Sleeping Barber Problem

Another classical IPC problem takes place in a barber shop. The barber shop has one barber, one barber chair, and *n* chairs for waiting customers, if any, to sit on. If there are no customers present, the barber sits down in the barber chair and falls asleep, as illustrated in Fig. When a customer arrives, he has to wake up the sleeping barber. If additional customers arrive while the barber is cutting a customer's hair, they either sit down (if there are empty chairs) or leave the shop (if all chairs are full). The problem is to program the barber and the customers without getting into race conditions. This problem is similar to various queueing situations, such as a multiperson helpdesk with a computerized call waiting system for holding a limited number of incoming calls.



Our solution uses three semaphores: *customers*, which counts waiting customers (excluding the customer in the barber chair, who is not waiting), *barbers*, the number of barbers (0 or 1) who are idle, waiting for customers, and *mutex*, which is used for mutual exclusion. We also need a variable, *waiting*, which also counts the waiting customers. It is essentially a copy of *customers*. The reason for having *waiting* is that there is no way to read the current value of a semaphore. In this solution, a customer entering the shop has to count the number of waiting customers. If it is less than the number of chairs, he stays; otherwise, he leaves.

```
#define CHAIRS 5              /* # chairs for waiting customers */
typedef int semaphore;        /* use your imagination */
semaphore customers = 0;      /* # of customers waiting for service */
semaphore barbers = 0;        /* # of barbers waiting for customers */
semaphore mutex = 1;          /* for mutual exclusion */
int waiting = 0;              /* customers are waiting (not being cut) */
```

```
void barber(void)
{
   white (TRUE) {
      down(&customers);      /* go to sleep if # of customers is 0 */
      down(&mutex);          /* acquire access to 'waiting' */
      waiting = waiting − 1; /* decrement count of waiting customers */
      up(&barbers);          /* one barber is now ready to cut hair */
      up(&mutex);            /* release 'waiting' */
      cut_hair();           /* cut hair (outside critical region) */
   }
}

void customer(void)
{
   down(&mutex);             /* enter critical region */
   if (waiting < CHAIRS) {    /* if there are no free chairs, leave */
      waiting = waiting + 1; /* increment count of waiting customers */
      up(&customers);        /* wake up barber if necessary */
      up(&mutex);            /* release access to 'waiting' */
      down(&barbers);        /* go to sleep if # of free barbers is 0 */
      get_haircut();         /* be seated and be serviced */
   } else {
      up(&mutex);            /* shop is full; do not wait */
   }
}
```

3.4 Process Scheduling

When a computer is multi programmed, it frequently has multiple processes competing for the CPU at the same time. This situation occurs whenever two or more processes are simultaneously in the ready state. If only one CPU is available, a choice has to be made which process to run next. The part of the operating system that makes the choice is called the **scheduler** and the algorithm it uses is called the **scheduling algorithm.**

3.4.1 Basic concept

Back in the old days of batch systems with input in the form of card images on a magnetic tape, the scheduling algorithm was simple: just runs the next job on the tape. With timesharing systems, the scheduling algorithm became more complex because there were generally multiple users waiting for service. Some mainframes still combine batch and timesharing service, requiring the scheduler to decide whether a batch job or an interactive user at a terminal should go next. Because CPU time is a scarce resource on these machines, a good scheduler can make a big difference in perceived performance and user satisfaction.

Consequently, a great deal of work has gone into devising clever and efficient scheduling algorithms.

With the advent of personal computers, the situation changed in two ways. First, most of the time there is only one active process. A user entering a document on a word processor is unlikely to be simultaneously compiling a program in the background. When the user types a command to the word processor, the scheduler does not have to do much work to figure out which process to run—the word processor is the only candidate.

In addition to picking the right process to run, the scheduler also has to worry about making efficient use of the CPU because process switching is expensive. To start with, a switch from user mode to kernel mode must occur. Then the state of the current process must be saved, including storing its registers in the process table so they can be reloaded later. In many systems, the memory map (e.g., memory reference bits in the page table) must be saved as well. Next a new process must be selected by running the scheduling algorithm. After that, the MMU must be reloaded with the memory map of the new process. Finally, the new process must be started. In addition to all that, the process switch usually invalidates the entire memory cache, forcing it to be dynamically reloaded from the main memory twice (upon entering the kernel and upon leaving it). All in all, doing too many process switches per second can chew up a substantial amount of CPU time, so caution is advised.

3.4.2 Types of scheduling

Categories of Scheduling Algorithms

Not surprisingly, in different environments different scheduling algorithms are needed. This situation arises because different application areas (and different kinds of operating systems) have different goals. In other words, what the scheduler should optimize for is not the same in all systems. Three environments worth distinguishing are

1. Batch.
2. Interactive.
3. Real time.

In batch systems, there are no users impatiently waiting at their terminals for a quick response. Consequently, no preemptive algorithms, or preemptive algorithms with long time periods for each process are often acceptable. This approach reduces process switches and thus improves performance.

In an environment with interactive users, preemption is essential to keep one process from hogging the CPU and denying service to the others. Even if no process intentionally ran forever, due to a program bug, one process might shut out all the others indefinitely. Preemption is needed to prevent this behavior.

In systems with real-time constraints, preemption is, oddly enough, sometimes not needed because the processes know that they may not run for long periods of time and usually do

their work and block quickly. The difference with interactive systems is that real-time systems run only programs that are intended to further the application at hand. Interactive systems are general purpose and may run arbitrary programs that are not cooperative or even malicious.

Scheduling Algorithm Goals

In order to design a scheduling algorithm, it is necessary to have some idea of what a good algorithm should do. Some goals depend on the environment (batch, interactive, or real time), but there are also some that are desirable in all cases. Some goals are listed in Fig.

All systems

Fairness - giving each process a fair share of the CPU
Policy enforcement - seeing that stated policy is carried out
Balance - keeping all parts of the system busy

Batch systems

Throughput - maximize jobs per hour
Turnaround time - minimize time between submission and termination
CPU utilization - keep the CPU busy all the time

Interactive systems

Response time - respond to requests quickly
Proportionality - meet users' expectations

Real-time systems

Meeting deadlines - avoid losing data
Predictability - avoid quality degradation in multimedia systems

Fig: Some goals of the scheduling algorithm under different circumstances.

3.4.3 Scheduling criteria or performance analysis

Scheduling Criteria

Different CPU scheduling algorithms have different properties, and the choice of a particular algorithm may favor one class of processes over another. In choosing which algorithm to use in a particular situation, we must consider the properties of the various algorithms. Many criteria have been suggested for comparing CPU scheduling algorithms. Which characteristics are used for comparison can make a substantial difference in which algorithm is judged to be best. The criteria include the following:

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

- **Throughput.** If the CPU is busy executing processes, then work is being done. One measure of work is the number of processes that are completed per time unit, called

throughput. For long processes, this rate may be one process per hour; for short transactions, it may be 10 processes per second.

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

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

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

3.4.4 Scheduling algorithm

Now let's discuss some processor scheduling algorithms again stating that the goal is to select the most appropriate process in the ready queue.

CPU scheduling deals with the problem of deciding which of the processes in the ready queue is to be allocated to the CPU. There are several scheduling algorithms which will be examined in this section. A major division among scheduling algorithms is that whether they support **pre-emptive** or **non-preemptive scheduling** discipline.

Preemptive scheduling: Preemption means the operating system moves a process from running to ready without the process requesting it. An OS implementing these algorithms switches to the processing of a new request before completing the processing of the current request. The preempted request is put back into the list of the pending requests. Its servicing would be resumed sometime in the future when it is scheduled again. Preemptive scheduling is more useful in high priority process which requires immediate response. For example, in Real time system the consequence of missing one interrupt could be dangerous.

Round Robin scheduling, priority based scheduling or event driven scheduling and SRTN are considered to be the preemptive scheduling algorithms.

Non–Preemptive scheduling: A scheduling discipline is non-preemptive if once a process has been allotted to the CPU, the CPU cannot be taken away from the process. A nonpreemptible discipline always processes a scheduled request to its completion. In nonpreemptive systems, jobs are made to wait by longer jobs, but the treatment of all processes is fairer.

First come First Served (FCFS) and Shortest Job First (SJF), are considered to be the nonpreemptive scheduling algorithms.

The decision whether to schedule preemptive or not depends on the environment and the type of application most likely to be supported by a given operating system.

3.4.4.1 First Come First Serve (FCFS):

The simplest scheduling algorithm is First Come First Serve (FCFS). Jobs are scheduled in the order they are received. FCFS is non-preemptive. Implementation is easily accomplished by implementing a queue of the processes to be scheduled or by storing the time the process was received and selecting the process with the earliest time.

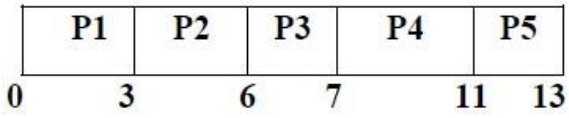
FCFS tends to favour CPU-Bound processes. Consider a system with a CPU-bound process and a number of I/O-bound processes. The I/O bound processes will tend to execute briefly, then block for I/O. A CPU bound process in the ready should not have to wait long before being made runable. The system will frequently find itself with all the I/O-Bound processes blocked and CPU-bound process running. As the I/O operations complete, the ready Queue fill up with the I/O bound processes.

Under some circumstances, CPU utilisation can also suffer. In the situation described above, once a CPU-bound process does issue an I/O request, the CPU can return to process all the I/O-bound processes. If their processing completes before the CPU-bound process's I/O completes, the CPU sits idle. So with no preemption, component utilisation and the system throughput rate may be quite low.

Example:
Calculate the turnaround time, waiting time, average turnaround time, average waiting time, throughput and processor utilization for the given set of processes that arrive at a given arrive time shown in the table, with the length of processing time given in milliseconds:

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

If the processes arrive as per the arrival time, the Gantt chart will be



Time	Process Completed	Turn around Time = t(Process completed) – t(Process submitted)	Waiting Time = Turnaround time – Processing time
0	-	-	-
3	P1	3-0=3	3-3=0
6	P2	6-2=4	4-3=1
7	P3	7-3=4	4-1=3
11	P4	11-5=6	6-4=2
13	P5	13-8=5	5-2=3

Average turn around time = (3+4+4+6+5) / 5 = 4.4
Average waiting time = (0+1+3+2+3) / 5 = 1.8
Processor utilization = (13/13)*100 = 100%
Throughput = 5/13 = 0.38

.

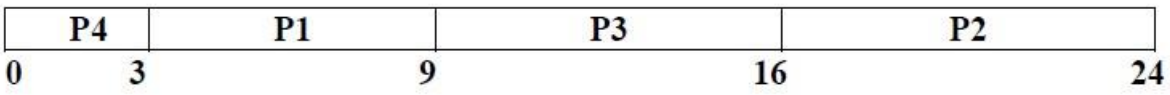
3.4.4.2 Shortest-Job First (SJF)

This algorithm is assigned to the process that has smallest next CPU processing time, when the CPU is available. In case of a tie, FCFS scheduling algorithm can be used. It is originally implemented in a batch-processing environment. SJF relied on a time estimate supplied with the batch job.

As an example, consider the following set of processes with the following processing time which arrived at the same time.

Process	Processing Time
P1	06
P2	08
P3	07
P4	03

Using SJF scheduling because the shortest length of process will first get execution, the Gantt chart will be:



Because the shortest processing time is of the process P4, then process P1 and then P3 and Process P2. The waiting time for process P1 is 3 ms, for process P2 is 16 ms, for process P3 is 9 ms and for the process P4 is 0 ms as:

Time	Process Completed	Turn around Time = t (Process Completed)– t (Process Submitted)	Waiting Time = Turn around time – Processing time
0	-	-	-
3	P4	3 – 0 = 3	3 – 3 = 0
9	P1	9 – 0 = 9	9 – 6 = 3
16	P3	16 – 0 = 16	16 – 7 = 9
24	P2	24 – 0= 24	24 – 8 = 16

Average turnaround time = (3+9+16+24) / 4 = 13
Average waiting time = (0+3+9+16) / 4 = 7
Processor utilization = (24/24)*100 = 100%
Throughput = 4/24 = 0.16

3.4.4.3 Round Robin (RR)

Round Robin (RR) scheduling is a preemptive algorithm that relates the process that has been waiting the longest. This is one of the oldest, simplest and widely used algorithms. The round robin scheduling algorithm is primarily used in time-sharing and a multi-user system environment where the primary requirement is to provide reasonably good response times and in general to share the system fairly among all system users. Basically the CPU time is divided into time slices.

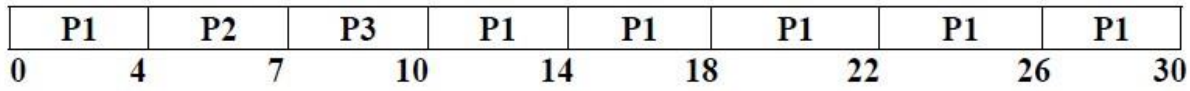
Each process is allocated a small time-slice called quantum. No process can run for more than one quantum while others are waiting in the ready queue. If a process needs more CPU time to complete after exhausting one quantum, it goes to the end of ready queue to await the next allocation. To implement the RR scheduling, Queue data structure is used to maintain the Queue of Ready processes. A new process is added at the tail of that Queue. The CPU schedular picks the first process from the ready Queue, Allocate processor for a specified time Quantum. After that time the CPU schedular will select the next process is the ready Queue.

Consider the following set of process with the processing time given in milliseconds.

Process	Processing Time
P1	24
P2	03
P3	03

If we use a time **Quantum of 4 milliseconds** then process P1 gets the first 4 milliseconds. Since it requires another 20 milliseconds, it is preempted after the first time Quantum, and the

CPU is given to the next process in the Queue, Process P2. Since process P2 does not need and milliseconds, it quits before its time Quantum expires. The CPU is then given to the next process, Process P3 one each process has received 1 time Quantum, the CPU is returned to process P1 for an additional time quantum. The Gantt chart will be:



Process	Processing Time	Turn around time = t(Process Completed) – t(Process Submitted)	Waiting Time = Turn around time – Processing time
P1	24	30 – 0 = 30	30 – 24 = 6
P2	03	7 – 0 = 7	7 – 3 = 4
P3	03	10 – 0 =10	10 – 3 = 7

Average turn around time = (30+7+10)/3 = 47/3 = 15.66
Average waiting time = (6+4+7)/3 = 17/3 = 5.66
Throughput = 3/30 = 0.1
Processor utilization = (30/30) * 100 = 100%

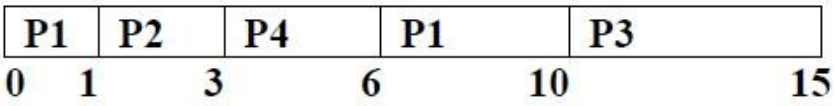
3.4.4.4 Shortest Remaining Time Next (SRTN)

This is the preemptive version of shortest job first. This permits a process that enters the ready list to preempt the running process if the time for the new process (or for its next burst) is less than the *remaining* time for the running process (or for its current burst). Let us understand with the help of an example.

Consider the set of four processes arrived as per timings described in the table:

Process	Arrival time	Processing time
P1	0	5
P2	1	2
P3	2	5
P4	3	3

At time 0, only process P1 has entered the system, so it is the process that executes. At time 1, process P2 arrives. At that time, process P1 has 4 time units left to execute At this juncture process 2's processing time is less compared to the P1 left out time (4 units). So P2 starts executing at time 1. At time 2, process P3 enters the system with the processing time 5 units. Process P2 continues executing as it has the minimum number of time units when compared with P1 and P3. At time 3, process P2 terminates and process P4 enters the system. Of the processes P1, P3 and P4, P4 has the smallest remaining execution time so it starts executing. When process P1 terminates at time 10, process P3 executes. The Gantt chart is shown below:



Turnaround time for each process can be computed by subtracting the time it terminated from the arrival time.

Turn around Time = t(Process Completed)– t(Process Submitted) The turnaround time for each of the processes is:

P1: 10 – 0 = 10
P2: 3 – 1 = 2
P3: 15 – 2 = 13
P4: 6 – 3 = 3

The average turnaround time is (10+2+13+3) / 4 = 7
The waiting time can be computed by subtracting processing time from turnaround time, yielding the following 4 results for the processes as

P1: 10 – 5 = 5
P2: 2 – 2 = 0
P3: 13 – 5 = 8
P4: 3 – 3 = 0

The average waiting time = (5+0+8+0) / 4 = 3.25milliseconds
Four jobs executed in 15 time units, so throughput is 15 / 4 = 3.75 time units/job.

3.4.4.5 Priority Based Scheduling or Event-Driven (ED) Scheduling

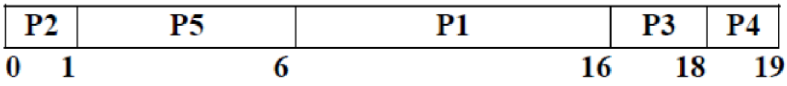
A priority is associated with each process and the scheduler always picks up the highest priority process for execution from the ready queue. Equal priority processes are scheduled FCFS. The level of priority may be determined on the basis of resource requirements, processes characteristics and its run time behaviour.

A major problem with a priority based scheduling is indefinite blocking of a lost priority process by a high priority process. In general, completion of a process within finite time cannot be guaranteed with this scheduling algorithm. A solution to the problem of indefinite blockage of low priority process is provided by aging priority. Aging priority is a technique of gradually increasing the priority of processes (of low priority) that wait in the system for a long time. Eventually, the older processes attain high priority and are ensured of completion in a finite time.

As an example, consider the following set of five processes, assumed to have arrived at the same time with the length of processor timing in milliseconds: –

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

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



Time	Process Completed	Turn around Time = t(Process Completed) – t(Process Submitted)	Waiting Time = Turn around time – Processing time
0	-	-	-
1	P2	1 – 0 = 1	1 – 1 = 0
6	P5	6 – 0 = 6	6 – 2 = 4
16	P1	16 – 0 = 16	16 – 10 = 6
18	P3	18 – 0 = 18	18 – 2 = 16
19	P4	19 – 0 = 19	19 – 1 = 18

Average turn around time = (1+6+16+18+19) / 5 = 60/5 = 12
Average waiting time = (6+0+16+18+1) / 5 = 8.2 Throughput
= 5/19 = 0.26
Processor utilization = (30/30) * 100 = 100%

Priorities can be defined either internally or externally. Internally defined priorities use one measurable quantity or quantities to complete the priority of a process.

3.4.4.6 Guaranteed Scheduling

A completely different approach to scheduling is to make real promises to the users about performance and then live up to them. One promise that is realistic to make and easy to live up to is this: If there are *n* users logged in while you are working, you will receive about 1/*n* of the CPU power. Similarly, on a single user system with *n* processes running, all things being equal, each one should get 1/*n* of the CPU cycles.

To make good on this promise, the system must keep track of how much CPU each process has had since its creation. It then computes the amount of CPU each one is entitled to, namely the time since creation divided by *n.* Since the amount of CPU time each process has actually had is also known, it is straightforward to compute the ratio of actual CPU time consumed to CPU time entitled. A ratio of 0.5 means that a process has only had half of what it should have had, and a ratio of 2.0 means that a process has had twice as much as it was entitled to. The algorithm is then to run the process with the lowest ratio until its ratio has moved above its closest competitor.

3.4.4.7 Lottery Scheduling:

While making promises to the users and then living up to them is a fine idea, it is difficult to implement. However, another algorithm can be used to give similarly predictable results with a much simpler implementation. It is called **lottery scheduling** (Waldspurger and Weihl, 1994).

The basic idea is to give processes lottery tickets for various system resources, such as CPU time. Whenever a scheduling decision has to be made, a lottery ticket is chosen at random, and the process holding that ticket gets the resource. When applied to CPU scheduling, the system might hold a lottery 50 times a second, with each winner getting 20 msec of CPU time as a prize.

To paraphrase George Orwell: "All processes are equal, but some processes are more equal." More important processes can be given extra tickets, to increase their odds of winning. If there are 100 tickets outstanding, and one process holds 20 of them, it will have a 20 percent chance of winning each lottery. In the long run, it will get about 20 percent of the CPU. In contrast to a priority scheduler, where it is very hard to state what having a priority of 40 actually means, here the rule is clear: a process holding a fraction *f* of the tickets will get about a fraction *f* of the resource in question.

Lottery scheduling has several interesting properties. For example, if a new process shows up and is granted some tickets, at the very next lottery it will have a chance of winning in proportion to the number of tickets it holds. In other words, lottery scheduling is highly responsive.

Cooperating processes may exchange tickets if they wish. For example, when a client process sends a message to a server process and then blocks, it may give all of its tickets to the server, to increase the chance of the server running next. When the server is finished, it returns the tickets so the client can run again. In fact, in the absence of clients, servers need no tickets at all.

Lottery scheduling can be used to solve problems that are difficult to handle with other methods. One example is a video server in which several processes are feeding video streams to their clients, but at different frame rates. Suppose that the processes need frames at 10, 20, and 25 frames/sec. By allocating these processes 10, 20, and 25 tickets, respectively, they will automatically divide the CPU in approximately the correct proportion, that is, 10 : 20 : 25.

3.4.4.8 Fair-share scheduling

So far we have assumed that each process is scheduled on its own, without regard to who its owner is. As a result, if user 1 starts up 9 processes and user 2 starts up 1 process, with round robin or equal priorities, user 1 will get 90% of the CPU and user 2 will get only 10% of it.

To prevent this situation, some systems take into account who owns a process before scheduling it. In this model, each user is allocated some fraction of the CPU and the scheduler picks processes in such a way as to enforce it. Thus if two users have each been promised 50% of the CPU, they will each get that, no matter how many processes they have in existence.

As an example, consider a system with two users, each of which has been promised 50% of the CPU. User 1 has four processes, *A*, *B*, *C*, and *D*, and user 2 has only 1 process, *E*. If round-robin scheduling is used, a possible scheduling sequence that meets all the constraints is this one:
A E B E C E D E A E B E C E D E ...

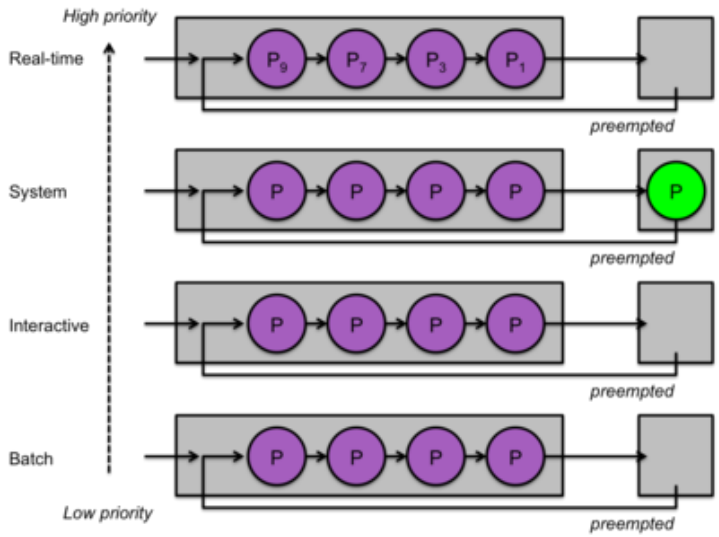
On the other hand, if user 1 is entitled to twice as much CPU time as user 2, we might get
A B E C D E A B E C D E ...

Numerous other possibilities exist of course, and can be exploited, depending on what the notion of fairness is.

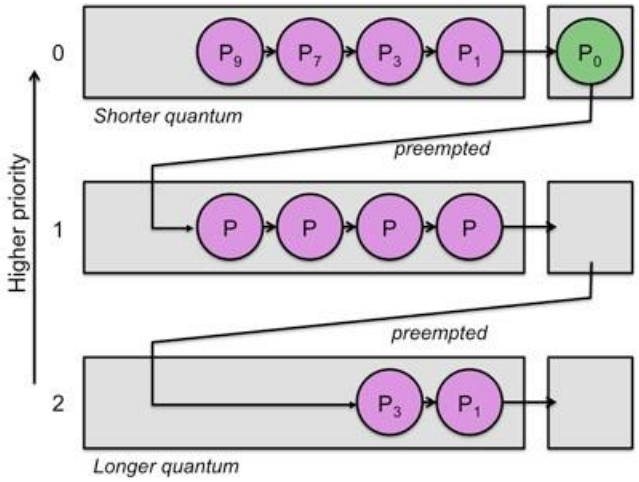
3.4.4.9 Multiple Queues

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

A **multilevel queue scheduling algorithm** partitions the ready queue into several separate queues. The processes are permanently assigned to one queue, generally based on some property of the process, such as memory size, process priority, or process type. Each queue has its own scheduling algorithm. For example, separate queues might be used for foreground and background processes. The foreground queue might be scheduled by an RR algorithm, while the background queue is scheduled by an FCFS algorithm. In addition, there must be scheduling among the queues, which is commonly implemented as fixed-priority preemptive scheduling. For example, the foreground queue may have absolute priority over the background queue.



Multilevel Feedback Queues



3.5 Performance Evaluation of the scheduling algorithms

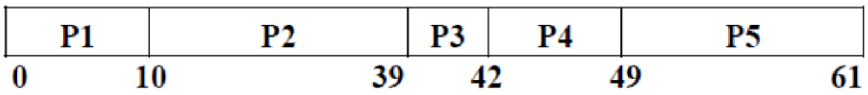
Performance of an algorithm for a given set of processes can be analysed if the appropriate information about the process is provided. But how do we select a CPU-scheduling algorithm for a particular system? There are many scheduling algorithms so the selection of an algorithm for a particular system can be difficult. To select an algorithm there are some specific criteria such as:

- Maximize CPU utilization with the maximum response time.
- Maximize throughput.

For example, assume that we have the following five processes arrived at time 0, in the order given with the length of CPU time given in milliseconds.

Process	Processing time
P1	10
P2	29
P3	03
P4	07
P5	12

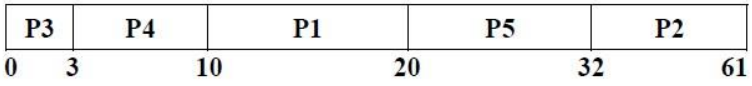
First consider the FCFS scheduling algorithm for the set of processes. For FCFS scheduling the Gantt chart will be:



Process	Processing time	Waiting time
P1	10	0
P2	29	10
P3	03	39
P4	07	42
P5	12	49

Average Waiting Time: (0+10+39+42+49) / 5 = 28 milliseconds.

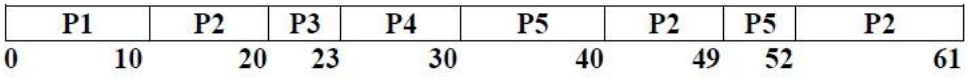
Now consider the SJF scheduling, the Gantt chart will be:



Process	Processing time	Waiting time
P1	10	10
P2	29	32
P3	03	00
P4	07	03
P5	12	20

Average Waiting Time: (10+32+0+3+20)/5 = 13 milliseconds.

Now consider the Round Robin scheduling algorithm with a quantum of 10 milliseconds. The Gantt chart will be:



Process	Processing time	Waiting time
P1	10	0
P2	29	32
P3	03	20
P4	07	23
P5	12	40

Average waiting time = (0+32+20+23+40) / 5 = 23 milliseconds

Now if we compare average waiting time above algorithms, we see that SJF policy results in less than one half of the average waiting time to that of FCFS scheduling; the RR algorithm gives us an intermediate value.

So performance of algorithm can be measured when all the necessary information is provided.

3.6 Exercise

1. What is process threading? Explain the role of threads in cooperative processes.
2. Differentiate between process and thread. Explain the contents of the Process Control Block (PCB)
3. For given processes, their arrival time, service time and process priority is given in the table below.

Process	Service Time	Process Priority
P1	12	3
P2	5	1
P3	2	4
P4	8	5
P5	6	2

Draw a Gantt chart to schedule the above process using

- FCFS
- Round Robin (with Time Quantum = 2)
- Priority scheduling

Also calculate the average waiting time.

4. Define context switching. How does the kernel mode differ from the user mode?
5. What do you mean by light weight process? How does lightweight process help in process sharing and are they economic? Explain in your own words.

6. For given processes, their arrival time, service time and process priority is given in the table below.

Process	Arrival	Service Time	Process Priority
P_1	0	15	2
P_2	1	30	1
P_3	2	20	3

Draw a Gantt chart to schedule the above process using,

i) FCFS ii) SRTN iii) Priority scheduling

Also calculate the average waiting time.

7. What are threads? How threads are different from process? Which do you think are better and why?
8. For given processes, their arrival time, service time and process priority is given in the table below.

Process	Arrival time	Run time	Process Priority

			(5 highest priority)
P1	0	12	3
P2	2	6	5
P3	4	2	2
P4	4	4	4
P5	0	8	1

Draw a Gantt chart to schedule the above process using

i. FCFS
ii. Round Robin (with Time Quantum = 2)
iii. Priority scheduling

Also calculate the average waiting time for each of the above scheduling algorithm.

9. How does Peterson's Algorithm work in achieving exclusion?
10. What is preemptive and non preemptive scheduling? Explain shortest job first algorithm with an example.

11. For given processes, their arrival time, service time and process priority is given in the table below. Calculate the avg. waiting and turnaround time for following algorithm[Where the 4 being highest and 1 being lowest priority]

Job	Arrival Time	Service Time	Process Priority
0	0	40	4
1	0	61	2
2	11	27	5
3	23	41	3
4	25	5	1

i. FCFS
ii. SJF
iii. Priority scheduling
iv. Round robin scheduling

Deadlocks

Computer systems are full of resources that can only be used by one process at a time. Common examples include printers, tape drives, and slots in the system's internal tables. Having two processes simultaneously writing to the printer leads to gibberish. Having two processes using the same file system table slot will invariably lead to a corrupted file system. Consequently, all operating systems have the ability to (temporarily) grant a process exclusive access to certain resources.

For many applications, a process needs exclusive access to not one resource, but several. Suppose, for example, two processes each want to record a scanned document on a CD. Process *A* requests permission to use the scanner and is granted it. Process *B* is programmed differently and requests the CD recorder first and is also granted it. Now *A* asks for the CD recorder, but the request is denied until *B* releases it. Unfortunately, instead of releasing the CD recorder *B* asks for the scanner. At this point both processes are blocked and will remain so forever. This situation is called a **deadlock**.

Deadlocks can also occur across machines. For example, many offices have a local area network with many computers connected to it. Often devices such as scanners, CD recorders, printers, and tape drives are connected to the network as shared resources, available to any user on any machine. If these devices can be reserved remotely (i.e., from the user's home machine), the same kind of deadlocks can occur as described above. More complicated situations can cause deadlocks involving three, four, or more devices and users.

Deadlocks can occur in a variety of situations besides requesting dedicated I/O devices. In a database system, for example, a program may have to lock several records it is using, to avoid race conditions. If process *A* locks record *R1* and process *B* locks record *R2*, and then each process tries to lock the other one's record, we also have a deadlock. Thus deadlocks can occur on hardware resources or on software resources.

4.1 System Model

- For the purposes of deadlock discussion, a system can be modeled as a collection of limited resources, which can be partitioned into different categories, to be allocated to a number of processes, each having different needs.
- Resource categories may include memory, printers, CPUs, open files, tape drives, CD-ROMS, etc.

- By definition, all the resources within a category are equivalent, and a request of this category can be equally satisfied by any one of the resources in that category. If this is not the case (i.e. if there is some difference between the resources within a category), then that category needs to be further divided into separate categories. For example, "printers" may need to be separated into "laser printers" and "color inkjet printers".
- Some categories may have a single resource.
- In normal operation a process must request a resource before using it, and release it when it is done, in the following sequence:
    1. **Request -** If the request cannot be immediately granted, then the process must wait until the resource(s) it needs become available. For example the system calls open( ), malloc( ), new( ), and request( ).
    2. **Use -** The process uses the resource, e.g. prints to the printer or reads from the file.
    3. **Release** - The process relinquishes the resource. so that it becomes available for other processes. For example, close( ), free( ), delete( ), and release( ).
- For all kernel-managed resources, the kernel keeps track of what resources are free and which are allocated, to which process they are allocated, and a queue of processes waiting for this resource to become available. Application-managed resources can be controlled using mutexes or wait( ) and signal( ) calls, (i.e. binary or counting semaphores.)
- A set of processes is deadlocked when every process in the set is waiting for a resource that is currently allocated to another process in the set (and which can only be released when that other waiting process makes progress.)

4.2 System resources: Preemptable and nonpreemptable

Deadlocks can occur when processes have been granted exclusive access to devices, files, and so forth. To make the discussion of deadlocks as general as possible, we will refer to the objects granted as **resources**. A resource can be a hardware device (e.g., a tape drive) or a piece of information (e.g., a locked record in a database). A computer will normally have many different resources that can be acquired. For some resources, several identical instances may be available, such as three tape drives. When several copies of a resource are available, any one of them can be used to satisfy any request for the resource. In short, a resource is anything that can be used by only a single process at any instant of time.

Preemptable and Nonpreemptable Resources

Preemptable Resources: A preemptable resource is one that can be taken away from the process owning it with no ill effects. Memory is an example of a preemptable resource. Consider, for example, a system with 32 MB of user memory, one printer, and two 32-MB processes that each want to print something. Process *A* requests and gets the printer, then starts to compute the values to print. Before it has finished with the computation, it exceeds its time quantum and is swapped out.

Process *B* now runs and tries, unsuccessfully, to acquire the printer. Potentially, we now have a deadlock situation, because *A* has the printer and *B* has the memory, and neither can proceed without the resource held by the other. Fortunately, it is possible to preempt (take away) the memory from *B* by swapping it out and swapping *A* in. Now *A* can run, do its printing, and then release the printer. No deadlock occurs.

Nonpreemptable resources: A **nonpreemptable resource**, in contrast, is one that cannot be taken away from its current owner without causing the computation to fail. If a process has begun to burn a CD-ROM, suddenly taking the CD recorder away from it and giving it to another process will result in a garbled CD, CD recorders are not preemptable at an arbitrary moment.

4.3 Conditions for resource deadlocks

Coffman et al. (1971) showed that four conditions must hold for there to be a deadlock:

1. Mutual exclusion condition. Each resource is either currently assigned to exactly one process or is available.
2. Hold and wait condition. Processes currently holding resources granted earlier can request new resources.
3. No preemption condition. Resources previously granted cannot be forcibly taken away from a process. They must be explicitly released by the process holding them.
4. Circular wait condition. There must be a circular chain of two or more processes, each of which is waiting for a resource held by the next member of the chain.

All four of these conditions must be present for a deadlock to occur. If one of them is absent, no deadlock is possible.

It is worth noting that each condition relates to a policy that a system can have or not have. Can a given resource be assigned to more than one process at once? Can a process hold a resource and ask for another? Can resources be preempted? Can circular waits exist? Later on we will see how deadlocks can be attacked by trying to negate some of these conditions.

4.4 Deadlock modelling

Holt (1972) showed how these four conditions can be modeled using directed graphs. The graphs have two kinds of nodes: processes, shown as circles, and resources, shown as squares. An arc from a resource node (square) to a process node (circle) means that the resource has previously been requested by, granted to, and is currently held by that process. In Fig. (a), resource *R* is currently assigned to process A.

An arc from a process to a resource means that the process is currently blocked waiting for that resource. In Fig. 3-3, process *B* is waiting for resource *S*. In Fig. (c) we see a deadlock: process *C* is waiting for resource *T*, which is currently held by process *D*. Process *D* is not about to

release resource *T* because it is waiting for resource *U*, held by *C*. Both processes will wait forever. A cycle in the graph means that there is a deadlock involving the processes and resources in the cycle (assuming that there is one resource of each kind). In this example, the cycle is *C–T–D–U–C*.

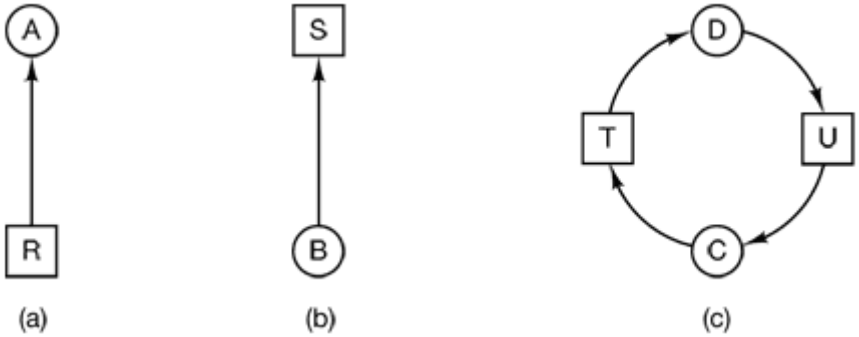
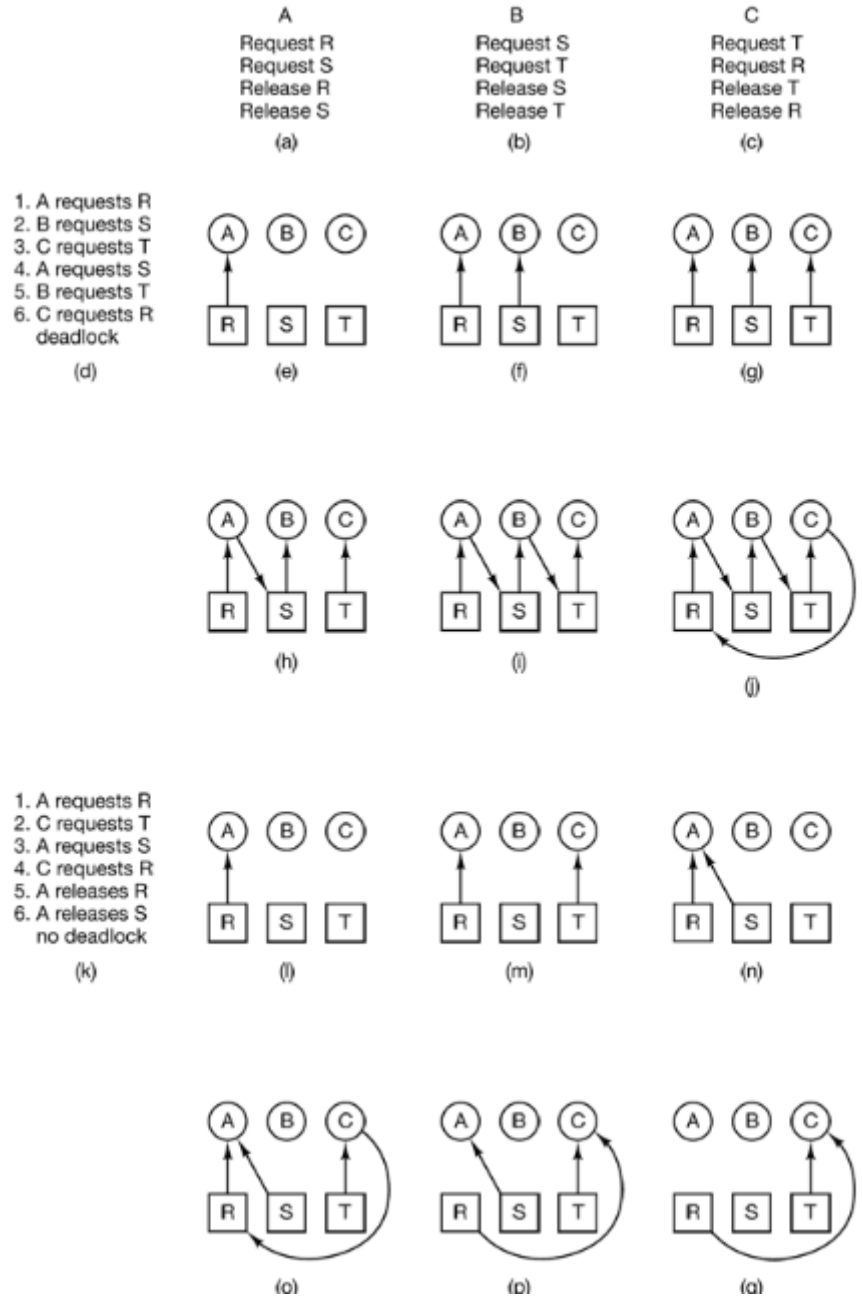


Fig: Resource allocation graphs (a) Holding a resource (b) Requesting a resource (c) Deadlock

An example of how deadlock occurs and it can be avoided.



In general, four strategies are used for dealing with deadlocks.

1. Just ignore the problem altogether. Maybe if you ignore it, it will ignore you.
2. Detection and recovery. Let deadlocks occur, detect them, and take action.
3. Dynamic avoidance by careful resource allocation.
4. Prevention, by structurally negating one of the four conditions necessary to cause a deadlock.

4.5 The OSTRICH algorithm

The simplest approach is the ostrich algorithm: stick your head in the sand and pretend there is no problem at all. Different people react to this strategy in different ways. Mathematicians find it totally unacceptable and say that deadlocks must be prevented at all costs. Engineers ask how

often the problem is expected, how often the system crashes for other reasons, and how serious a deadlock is. If deadlocks occur on the average once every five years, but system crashes due to hardware failures, compiler errors, and operating system bugs occur once a week, most engineers would not be willing to pay a large penalty in performance or convenience to eliminate deadlocks.

To make this contrast more specific, most operating systems potentially suffer from deadlocks that are not even detected, let alone automatically broken. The total number of processes in a system is determined by the number of entries in the process table. Thus process table slots are finite resources. If a fork fails because the table is full, a reasonable approach for the program doing the fork is to wait a random time and try again.

Now suppose that a UNIX system has 100 process slots. Ten programs are running, each of which needs to create 12 (sub)processes. After each process has created 9 processes, the 10 original processes and the 90 new processes have exhausted the table. Each of the 10 original processes now sits in an endless loop forking and failing—a deadlock. The probability of this happening is minuscule, but it *could* happen. Should we abandon processes and the fork call to eliminate the problem?

The maximum number of open files is similarly restricted by the size of the i-node table, so a similar problem occurs when it fills up. Swap space on the disk is another limited resource. In fact, almost every table in the operating system represents a finite resource. Should we abolish all of these because it might happen that a collection of n processes might each claim $1/n$ of the total, and then each try to claim another one?

Most operating systems, including UNIX and Windows, just ignore the problem on the assumption that most users would prefer an occasional deadlock to a rule restricting all users to one process, one open file, and one of everything. If deadlocks could be eliminated for free, there would not be much discussion. The problem is that the price is high, mostly in terms of putting inconvenient restrictions on processes, as we will see shortly. Thus we are faced with an unpleasant trade-off between convenience and correctness, and a great deal of discussion about which is more important, and to whom. Under these conditions, general solutions are hard to find.

4.6 Method of handling deadlocks

- Ensure that the system will *never* enter a deadlock state.
- Allow the system to enter a deadlock state and then recover.
- Ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems, including UNIX.

4.7 Deadlock prevention

Having seen that deadlock avoidance is essentially impossible, because it requires information about future requests, which is not known, how do real systems avoid deadlock? The answer is to go back to the four conditions stated by Coffman et al. (1971) to see if they can provide a clue. If we can ensure that at least one of these conditions is never satisfied, then deadlocks will be structurally impossible (Havender, 1968).

1. Attacking the Mutual Exclusion Condition:

First let us attack the mutual exclusion condition. If no resource were ever assigned exclusively to a single process, we would never have deadlocks. However, it is equally clear that allowing two processes to write on the printer at the same time will lead to chaos. By spooling printer output, several processes can generate output at the same time. In this model, the only process that actually requests the physical printer is the printer daemon. Since the daemon never requests any other resources, we can eliminate deadlock for the printer.

Unfortunately, not all devices can be spooled (the process table does not lend itself well to being spooled). Furthermore, competition for disk space for spooling can itself lead to deadlock. What would happen if two processes each filled up half of the available spooling space with output and neither was finished producing output? If the daemon was programmed to begin printing even before all the output was spooled, the printer might lie idle if an output process decided to wait several hours after the first burst of output. For this reason, daemons are normally programmed to print only after the complete output file is available. In this case we have two processes that have each finished part, but not all, of their output, and cannot continue. Neither process will ever finish, so we have a deadlock on the disk.

2. Attacking the Hold and Wait Condition:

The second of the conditions stated by Coffman et al. looks slightly more promising. If we can prevent processes that hold resources from waiting for more resources, we can eliminate deadlocks. One way to achieve this goal is to require all processes to request all their resources before starting execution. If everything is available, the process will be allocated whatever it needs and can run to completion. If one or more resources are busy, nothing will be allocated and the process would just wait.

An immediate problem with this approach is that many processes do not know how many resources they will need until they have started running. In fact, if they knew, the banker's algorithm could be used. Another problem is that resources will not be used optimally with this approach. Take, as an example, a process that reads data from an input tape, analyzes it for an hour, and then writes an output tape as well as plotting the results. If all resources must be requested in advance, the process will tie up the output tape drive and the plotter for an hour.
A slightly different way to break the hold-and-wait condition is to require a process requesting a resource to first temporarily release all the resources it currently holds. Then it tries to get everything it needs all at once.

3. Attacking the No Preemption Condition:
Attacking the third condition (no preemption) is even less promising than attacking the second one. If a process has been assigned the printer and is in the middle of printing its output, forcibly taking away the printer because a needed plotter is not available is tricky at best and impossible at worst.

4. Attacking the Circular Wait Condition:
The circular wait can be eliminated in several ways. One way is simply to have a rule saying that a process is entitled only to a single resource at any moment. If it needs a second one, it must release the first one. For a process that needs to copy a huge file from a tape to a printer, this restriction is unacceptable.

Another way to avoid the circular wait is to provide a global numbering of all the resources, as shown in Fig. 3-13(a). Now the rule is this: processes can request resources whenever they want to, but all requests must be made in numerical order. A process may request first a printer and then a tape drive, but it may not request first a plotter and then a printer.

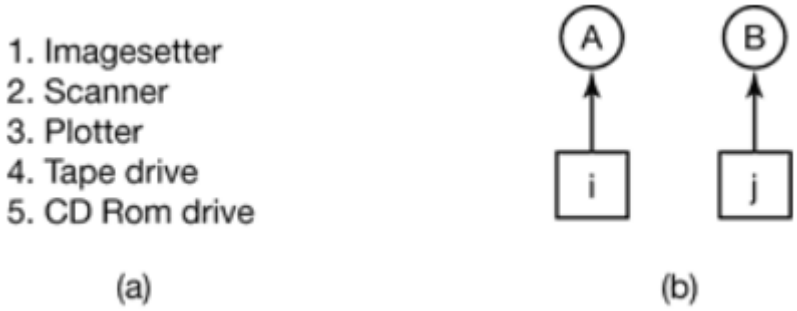


Fig: (a) Numerically ordered resources (b) A resource graph

Summary of approaches to prevent deadlock:

Condition	Approach
Mutual exclusion	Spool everything
Hold and wait	Request all resources initially
No preemption	Take resources away
Circular wait	Order resources numerically

4.8 Deadlock avoidance

In most systems, however, resources are requested one at a time. The system must be able to decide whether granting a resource is safe or not and only make the allocation when it is safe. Thus the question arises: Is there an algorithm that can always avoid deadlock by making the right choice all the time? The answer is a qualified yes—we can avoid deadlocks, but only if certain information is available in advance. In this section we examine ways to avoid deadlock by careful resource allocation.

1. Resource Trajectories:

The main algorithms for doing deadlock avoidance are based on the concept of safe states. Before describing the algorithms, we will make a slight digression to look at the concept of safety in a graphic and easy-to-understand way. Although the graphical approach does not translate directly into a usable algorithm, it gives a good intuitive feel for the nature of the problem.

In Fig. we see a model for dealing with two processes and two resources, for example, a printer and a plotter. The horizontal axis represents the number of instructions executed by process *A*. The vertical axis represents the number of instructions executed by process *B*. At I_1 *A* requests a printer; at I_2 it needs a plotter. The printer and plotter are released at I_3 and I_4, respectively. Process *B* needs the plotter from I_5 to I_7 and the printer from I_6 to I_8.

Every point in the diagram represents a joint state of the two processes. Initially, the state is at *p*, with neither process having executed any instructions. If the scheduler chooses to run *A* first, we get to the point *q*, in which *A* has executed some number of instructions, but *B* has executed none. At point *q* the trajectory becomes vertical, indicating that the scheduler has chosen to run *B*. With a single processor, all paths must be horizontal or vertical, never diagonal. Furthermore, motion is always to the north or east, never to the south or west (processes cannot run backward).

When *A* crosses the I_1 line on the path from *r* to *s*, it requests and is granted the printer. When *B* reaches point *t*, it requests the plotter.

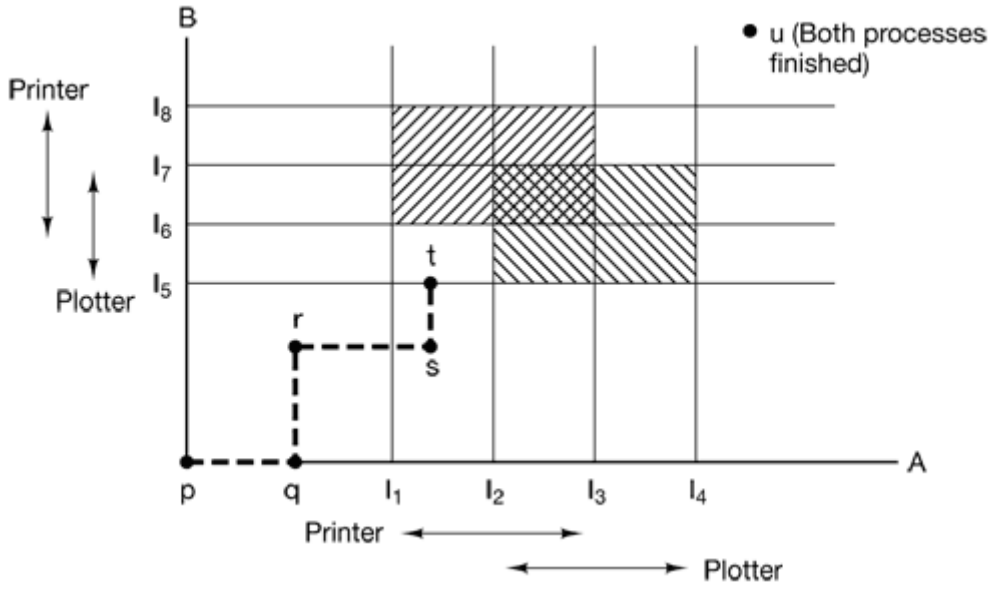


Fig: Two process resource trajectories

The regions that are shaded are especially interesting. The region with lines slanting from southwest to northeast represents both processes having the printer. The mutual exclusion rule makes it impossible to enter this region. Similarly, the region shaded the other way represents both processes having the plotter, and is equally impossible.

If the system ever enters the box bounded by I_1 and I_2 on the sides and I_5 and I_6 top and bottom, it will eventually deadlock when it gets to the intersection of I_2 and I_6. At this point *A* is requesting the plotter and *B* is requesting the printer, and both are already assigned. The entire box is unsafe and must not be entered. At point *t* the only safe thing to do is run process *A* until it gels to I_4. Beyond that, any trajectory to *u* will do.

The important thing to see here is at point t B is requesting a resource. The system must decide whether to grant it or not. If the grant is made, the system will enter an unsafe region and eventually deadlock. To avoid the deadlock, B should be suspended until A has requested and released the plotter.

Safe and Unsafe States:

A state is said to be safe if it is not deadlocked and there is some scheduling order in which every process can run to completion even if all of them suddenly request their maximum number of resources immediately. It is easiest to illustrate this concept by an example using one resource. In Fig. (a) we have a state in which *A* has 3 instances of the resource but may need as many as 9 eventually. *B* currently has 2 and may need 4 altogether, later. Similarly, *C* also has 2 but may need an additional 5. A total of 10 instances of the resource exist, so with 7 resources already allocated, there are 3 still free.

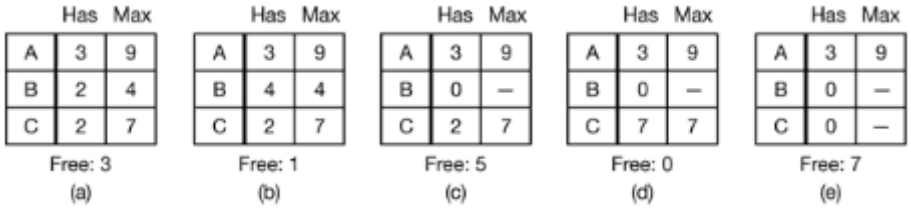


Fig: Demonstration that the states in (a) is safe.

The state of Fig.(a) is safe because there exists a sequence of allocations that allows all processes to complete. Namely, the scheduler could simply run *B* exclusively, until it asked for and got two more instances of the resource, leading to the state of Fig.(b). When *B* completes, we get the state of Fig. (c). Then the scheduler can run *C* leading eventually to Fig.(d). When *C* completes, we get Fig. (e). Now *A* can get the six instances of the resource it needs and also complete. Thus the state of Fig. (a) is safe because the system, by careful scheduling, can avoid deadlock.

Now suppose we have the initial state shown in Fig. (a), but this time *A* requests and gets another resource, giving Fig. (b). Can we find a sequence that is guaranteed to work? Let us try. The scheduler could run *B* until it asked for all its resources, as shown in Fig. (c).

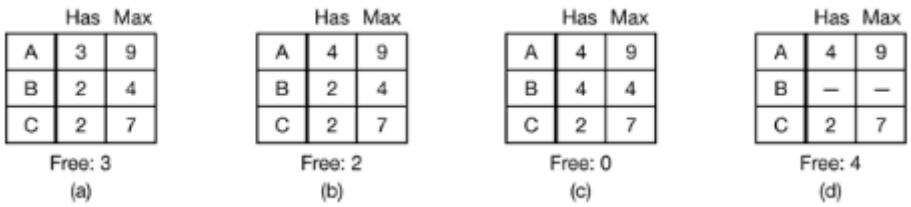


Fig: Demonstration that the state in (b) is not safe.

Eventually, *B* completes and we get the situation of Fig.(d). At this point we are stuck. We only have four instances of the resource free, and each of the active processes needs five. There is no sequence that guarantees completion. Thus the allocation decision that moved the system from Fig.(a) to Fig.(b) went from a safe state to an unsafe state. Running *A* or C next starting at Fig. (b) does not work either. In retrospect, *A*'s request should not have been granted.

It is worth noting that an unsafe state is not a deadlocked state. Starting at Fig.(b), the system can run for a while. In fact, one process can even complete. Furthermore, it is possible that *A* might release a resource before asking for any more, allowing *C* to complete and avoiding deadlock altogether. Thus the difference between a safe state and an unsafe state is that from a safe state the system can *guarantee* that all processes will finish; from an unsafe state, no such guarantee can be given.

The Banker's Algorithm:

This approach to the deadlock problem anticipates a deadlock before it actually occurs. This approach employs an algorithm to access the possibility that deadlock could occur and act accordingly. This method differs from deadlock prevention, which guarantees that deadlock cannot occur by denying one of the necessary conditions of deadlock. The most famous deadlock avoidance algorithm, from Dijkstra [1965], is the Banker's algorithm. It is named as Banker's algorithm because the process is analogous to that used by a banker in deciding if a loan can be safely made a not.

The Banker's Algorithm is based on the banking system, which never allocates its available cash in such a manner that it can no longer satisfy the needs of all its customers. Here we must have the advance knowledge of the maximum possible claims for each process, which is limited by the resource availability. During the run of the system we should keep monitoring the resource allocation status to ensure that no circular wait condition can exist.

If the necessary conditions for a deadlock are in place, it is still possible to avoid deadlock by being careful when resources are allocated. The following are the features that are to be considered for avoidance of the deadlock s per the Banker's Algorithms.

- Each process declares maximum number of resources of each type that it may need.
- Keep the system in a safe state in which we can allocate resources to each process in some order and avoid deadlock.
- Check for the safe state by finding a safe sequence: <P1, P2, ..., Pn> where resources that Pi needs can be satisfied by available resources plus resources held by Pj where j < i.
- Resource allocation graph algorithm uses claim edges to check for a safe state.

The resource allocation state is now defined by the number of available and allocated resources, and the maximum demands of the processes. Subsequently the system can be in either of the following states:

- **Safe state:** Such a state occurs when the system can allocate resources to each process (up to its maximum) in some order and avoid a deadlock. This state will be

characterized by a safe sequence. It must be mentioned here that we should not falsely conclude that all unsafe states are deadlocked although it may eventually lead to a deadlock.

- **Unsafe State:** If the system did not follow the safe sequence of resource allocation from the beginning and it is now in a situation, which may lead to a deadlock, then it is in an unsafe state.
- **Deadlock State:** If the system has some circular wait condition existing for some processes, then it is in deadlock state.

Let us study this concept with the help of an example as shown below:
Consider an analogy in which 4 processes (P1, P2, P3 and P4) can be compared with the customers in a bank, resources such as printers etc. as cash available in the bank and the Operating system as the Banker.

Processes	Resources used	Maximum resources
P1	0	6
P2	0	5
P3	0	4
P4	0	7

Let us assume that total available resources = 10
In the above table, we see four processes, each of which has been granted a number of maximum resources that can be used. The Operating system reserved only 10 resources rather than 22 units to service them. At a certain moment, the situation becomes:

Processes	Resources used	Maximum resources
P1	1	6
P2	1	5
P3	2	4
P4	4	7

Available resources = 2

Safe State: The key to a state being safe is that there is at least one way for all users to finish. In other words the state of Table is safe because with 2 units left, the operating system can delay any request except P3, thus letting P3 finish and release all four resources. With four units in hand, the Operating system can let either P4 or P2 have the necessary units and so on.
Unsafe State: Consider what would happen if a request from P2 for one more unit was granted in Table. We would have following situation as shown in Table.

Processes	Resources used	Maximum resources
P1	1	6
P2	2	5
P3	2	4
P4	4	7

Available resources = 1
This is an unsafe state.

If all the processes request for their maximum resources respectively, then the operating system could not satisfy any of them and we would have a deadlock.

Limitations of Banker's Algorithm:
There are some problems with the Banker's algorithm as given below:

- It is time consuming to execute on the operation of every resource.
- If the claim information is not accurate, system resources may be underutilised.
- Another difficulty can occur when a system is heavily loaded. Lauesen states that in this situation "so many resources are granted away that very few safe sequences remain, and as a consequence, the jobs will be executed sequentially". Therefore, the Banker's algorithm is referred to as the "Most Liberal" granting policy; that is, it gives away everything that it can without regard to the consequences.
- New processes arriving may cause a problem.
    - The process's claim must be less than the total number of units of the resource in the system. If not, the process is not accepted by the manager.
    - Since the state without the new process is safe, so is the state with the new process. Just use the order you had originally and put the new process at the end.
    - Ensuring fairness (starvation freedom) needs a little more work, but isn't too hard either (once every hour stop taking new processes until all current processes finish).
- A resource becoming unavailable (e.g., a tape drive breaking), can result in an unsafe state.

4.9 Deadlock detection: Resource allocation graph

Let us begin with the simplest case: only one resource of each type exists. Such a system might have one scanner, one CD recorder, one plotter, and one tape drive, but no more than one of each class of resource. In other words, we are excluding systems with two printers for the moment. We will treat them later using a different method.

As an example of a more complex system than the ones we have looked at so far, consider a system with seven processes, *A* though *G*, and six resources, *R* through *W*. The state of which resources are currently owned and which ones are currently being requested is as follows:

1. Process *A* holds *R* and wants *S*.
2. Process *B* holds nothing but wants *T*.
3. Process *C* holds nothing but wants *S*.
4. Process *D* holds *U* and wants *S* and *T*.
5. Process *E* holds *T* and wants *V*.
6. Process *F* holds *W* and wants *S*.
7. Process *G* holds *V* and wants *U*.

The question is: "Is this system deadlocked, and if so, which processes are involved?"
To answer this question, we can construct the resource graph of Fig. (a). This graph contains one cycle, which can be seen by visual inspection. The cycle is shown in Fig. (b). From this cycle, we can see that processes *D*, *E*, and *G* are all deadlocked. Processes *A*, *C*, and *F* are not deadlocked because *S* can be allocated to any one of them, which then finishes and returns it. Then the other two can take it in turn and also complete.

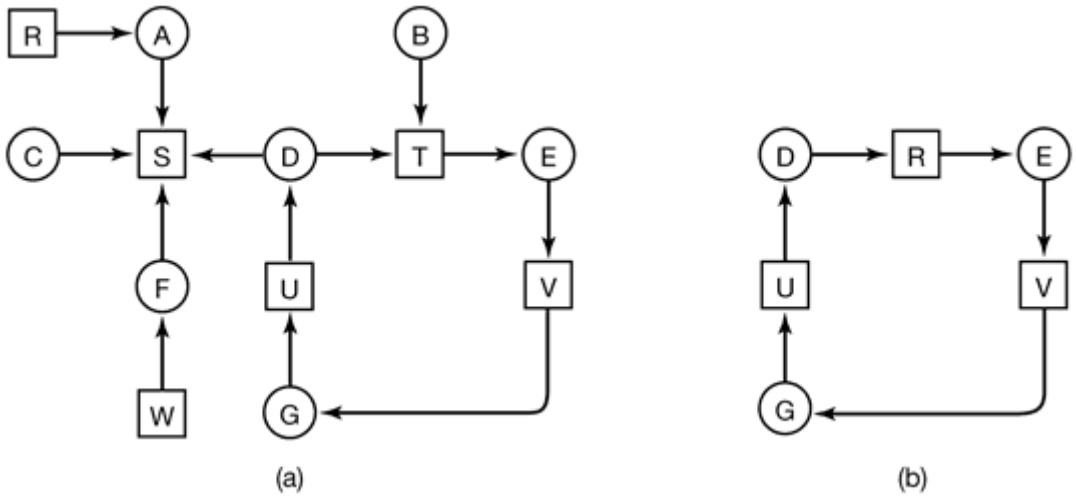


Fig: (a) A resource graph (b) A cycle extracted from (a)

Although it is relatively simple to pick out the deadlocked processes by eye from a simple graph, for use in actual systems we need a formal algorithm for detecting deadlocks. Many algorithms for detecting cycles in directed graphs are known. Below we will give a simple one that inspects a graph and terminates either when it has found a cycle or when it has shown that none exist. It uses one data structure, *L*, a list of nodes. During the algorithm, arcs will be marked to indicate that they have already been inspected, to prevent repeated inspections.

The algorithm operates by carrying out the following steps as specified:

1. For each node, *N* in the graph, perform the following 5 steps with *N* as the starting node.
2. Initialize *L* to the empty list, and designate all the arcs as unmarked.
3. Add the current node to the end of *L* and check to see if the node now appears in *L* two times. If it does, the graph contains a cycle (listed in *L*) and the algorithm terminates.

4. From the given node, see if there are any unmarked outgoing arcs. If so, go to step 5; if not, go to step 6.
5. Pick an unmarked outgoing arc at random and mark it. Then follow it to the new current node and go to step 3.
6. We have now reached a dead end. Remove it and go back to the previous node, that is, the one that was current just before this one, make that one the current node, and go to step 3. If this node is the initial node, the graph does not contain any cycles and the algorithm terminates.

4.10 Recovery from deadlock

Suppose that our deadlock detection algorithm has succeeded and detected a deadlock. What next? Some way is needed to recover and get the system going again. In this section we will discuss various ways of recovering from deadlock. None of them are especially attractive, however.

Recovery through pre-emption

In some cases it may be possible to temporarily take a resource away from its current owner and give it to another process. In many cases, manual intervention may be required, especially in batch processing operating systems running on mainframes.

For example, to take a laser printer away from its owner, the operator can collect all the sheets already printed and put them in a pile. Then the process can be suspended (marked as not runnable). At this point the printer can be assigned to another process. When that process finishes, the pile of printed sheets can be put back in the printer's output tray and the original process restarted.

The ability to take a resource away from a process, have another process use it, and then give it back without the process noticing it is highly dependent on the nature of the resource. Recovering this way is frequently difficult or impossible. Choosing the process to suspend depends largely on which ones have resources that can easily be taken back.

Recovery through Rollback

If the system designers and machine operators know that deadlocks are likely, they can arrange to have processes **checkpointed** periodically. Checkpointing a process means that its state is written to a file so that it can be restarted later. The checkpoint contains not only the memory image, but also the resource state, that is, which resources are currently assigned to the process. To be most effective, new checkpoints should not overwrite old ones but should be written to new files, so as the process executes, a whole sequence of checkpoint files are accumulated.
When a deadlock is detected, it is easy to see which resources are needed. To do the recovery, a process that owns a needed resource is rolled back to a point in time before it acquired some other resource by starting one of its earlier checkpoints. All the work done since the checkpoint is lost (e.g., output printed since the checkpoint must be discarded, since it will be printed

again). In effect, the process is reset to an earlier moment when it did not have the resource, which is now assigned to one of the deadlocked processes. If the restarted process tries to acquire the resource again, it will have to wait until it becomes available.

Recovery through Killing Processes

The crudest, but simplest way to break a deadlock is to kill one or more processes. One possibility is to kill a process in the cycle. With a little luck, the other processes will be able to continue. If this does not help, it can be repeated until the cycle is broken.

Alternatively, a process not in the cycle can be chosen as the victim in order to release its resources. In this approach, the process to be killed is carefully chosen because it is holding resources that some process in the cycle needs. For example, one process might hold a printer and want a plotter, with another process holding a plotter and wanting a printer. These two are deadlocked. A third process may hold another identical printer and another identical plotter and be happily running. Killing the third process will release these resources and break the deadlock involving the first two.

Where possible, it is best to kill a process that can be rerun from the beginning with no ill effects. For example, a compilation can always be rerun because all it does is read a source file and produce an object file. If it is killed part way through, the first run has no influence on the second run.

On the other hand, a process that updates a database cannot always be run a second time safely. If the process adds 1 to some record in the database, running it once, killing it, and then running it again will add 2 to the record, which is incorrect.

4.11 Starvation

A problem closely related to deadlock is **starvation**. In a dynamic system, requests for resources happen all the time. Some policy is needed to make a decision about who gets which resource when. This policy, although seemingly reasonable, may lead to some processes never getting service even though they are not deadlocked.

As an example, consider allocation of the printer. Imagine that the system uses some kind of algorithm to ensure that allocating the printer does not lead to deadlock. Now suppose that several processes all want it at once. Which one should get it?

One possible allocation algorithm is to give it to the process with the smallest file to print (assuming this information is available). This approach maximizes the number of happy customers and seems fair. Now consider what happens in a busy system when one process has a huge file to print. Every time the printer is free, the system will look around and choose the process with the shortest file. If there is a constant stream of processes with short files, the process with the huge file will never be allocated the printer. It will simply starve to death (be postponed indefinitely, even though it is not blocked).

Starvation can be avoided by using a first-come, first-serve, resource allocation policy. With this approach, the process waiting the longest gets served next. In due course of time, any given process will eventually become the oldest and thus get the needed resource.

4.12 Exercises

1. What are the different methods for handling deadlock? Explain deadlock prevention method.
2. A system has four processes and five allocable resources. The current allocation and maximum needs are as follows.

	Allocated	**Maximum**	**Available**
Process A	1 0 2 1 1	1 1 2 1 3	0 0 x 1 1
Process B	2 0 1 1 0	2 2 2 1 0	
Process C	1 1 0 1 0	2 1 3 1 0	
Process D	1 1 1 1 0	1 1 2 2 1	

   Calculate the smallest value of x for which this is a safe state.
3. How disabling interrupt can help in achieving Mutual Exclusion? Does this again bring up further problem?
4. What is deadlock condition? Describe the Semaphores solution to achieve mutual exclusion without busy waiting.
5. What is deadlock? How could you prevent it?