Module – 1

INTRODUCTION TO OPERATING SYSTEM:

An Operating system is an important part of almost every computer system. The purpose of an operating system is to provide an environment in which a user can execute programs.

The primary goal of an Operating system is to make the computer system convenient to use. The secondary goal is to use the computer hardware in an efficient manner. An operating system is a program that acts as an inter mediator between a user of a computer and the computer hardware.

A Computer system can be divided roughly into four components: the hardware, the operating system, the applications programs, and the users. There may be many different users trying to solve different application programs. The Operating System controls and coordinates the use of the hardware among the various application programs for the various users.

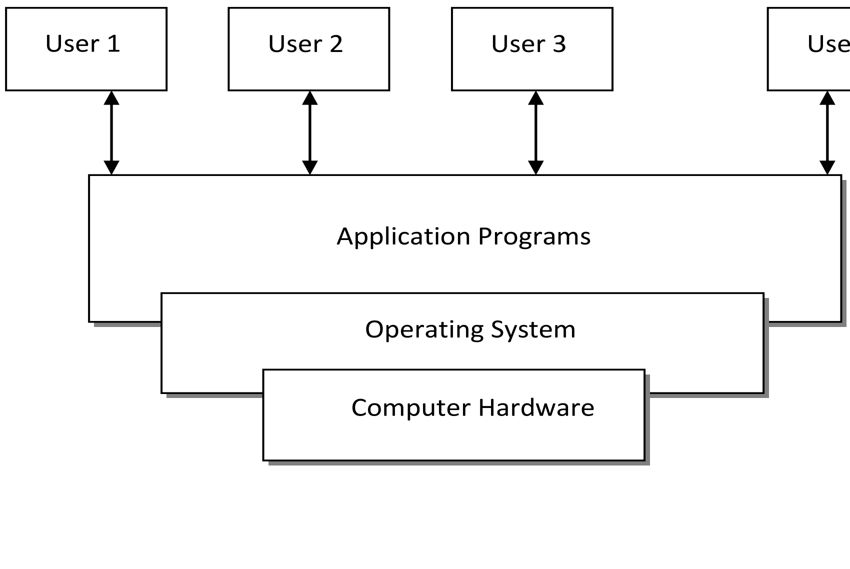


Figure 1.17 Abstract view of the components of a computer system

FUNCTIONS OF AN OPERATING SYSTEM:

An operating system is similar to a **government**. It provides a means for the proper use of the resources in the operation of the computer system. Like a government in a country, the Operating system provides an environment within which other programs can do useful work.

An operating system can be viewed as a **resource allocator**. A Computer System has many resources (hardware & software) such as CPU time, memory space, file storage space, I/O devices and so on. It acts as the manager of these resources of a computer system and allocates them to specific programs and users as necessary for tasks. In case of many conflicting requests for resources, the operating system must see that the resources are allocated efficiently and fairly.

An operating system is a **control program**. A control program controls the execution of user programs to prevent errors and improper use of the computer. It is concerned with the operation and control of I/O devices.

The common functions of controlling and allocating resources are then brought together into one piece of software - the operating system. A more common definition is that OS is one program running at all times on the computer (usually called the kernel), with all else being application programs.

System Goals:

It is easier to define an operating system by what it does than by what it is. The primary goal of some operating system is convenience for the user. Operating systems exist because they are supposed to make it easier to compute with them than without them.

The primary goal of other operating systems is efficient operation of the computer system. This is the case for large, shared, multiuser systems. These systems are expensive, so it is desirable to make them as efficient as possible. These two goals **- convenience and efficiency** - are sometimes contradictory.

The design of an operating system is a complex task. Designers face many tradeoffs in the design and implementation, and many people are involved not only in bringing an operating system to fruition, but also constantly revising and updating it.

MAINFRAME SYSTEMS :

Main frame computer systems were the first computers used to tackle many commercial and scientific applications. The following explains the growth of mainframe systems from simple batch systems, where the computer runs one and only one - applications, to time-shared systems, which allow for user interaction with the computer system.

1. BATCH PROCESSING SYSTEMS:

Early computers were physically large (enormous) machines run from a console. They used card readers, tape drives as input devices and line printers, card punches as output devices. The user did not interact directly with the computer systems. Rather, the user prepared a job - which consisted of the program, the data, and some control information about the nature of the job and submitted it to the computer operator. Batch processing systems came into existence at a time when punched cards were used to record user jobs and job consists of program, the data and some control information about the nature of the job. Processing of a job involved physical actions by the system operator (E.g. loading a deck of cards into the card reader, pressing switches on the computer's console to initiate a job Etc) all of which wasted a lot of computer time. Automating the processing of a batch of jobs could reduce this wastage. The task of Operating System in early computers was to transfer control automatically from one job to the next. The OS was always resident in memory.

The users of batch processing system did not interact directly with the computer system. A batch is a sequence of user jobs. To speed up processing jobs with similar needs were batched together and were run through the computer as a group. A computer operator forms a batch by arranging user jobs in a sequence and inserting special markers to indicate the start and end of the batch. After forming a batch, the operator submits it to the OS for processing. The primary function of the Batch Processing system is to implement the processing of a batch of jobs without requiring any intervention of the operator. The OS achieves this by making an automatic transition from the execution of one job to that of the next job in the batch. The definitive feature of a batch system is the lack of interaction between the user and the job while that job is executing. The job is prepared and submitted, and at some later time, the output appears.

In this execution environment, the CPU is idle. This idleness occurs because the speeds of the mechanical I/O devices are intrinsically slower than those of electronic devices. The introduction of a method called spooling has helped to reduce the CPU idle time. Rather than the cards being read from the card reader directly into memory, and then the job being processed cards are read directly from the card reader onto the disk. The location of the card images is recorded in a table kept by the operating system. When the job is executed, the OS satisfies its requests for card reader input by reading from the disk. Similarly, when the job requests the printer to output a line, that line is copied into a system buffer and is written to the disk.

Batch processing is implemented by locating a component of the batch processing OS called the batch monitor or supervisor, permanently in one part of the main memory. The remaining part of the memory is occupied by the current job of the batch. The batch monitor accepts a command from the system operator for initiating the processing of a batch and sets up the processing of the first job of the batch. At the end of the job, it performs job termination processing & initiates execution of the next job. At the end of the batch, it performs batch termination processing & awaits initiation of the next batch by the operator. Schematic of a Batch processing system is shown below.

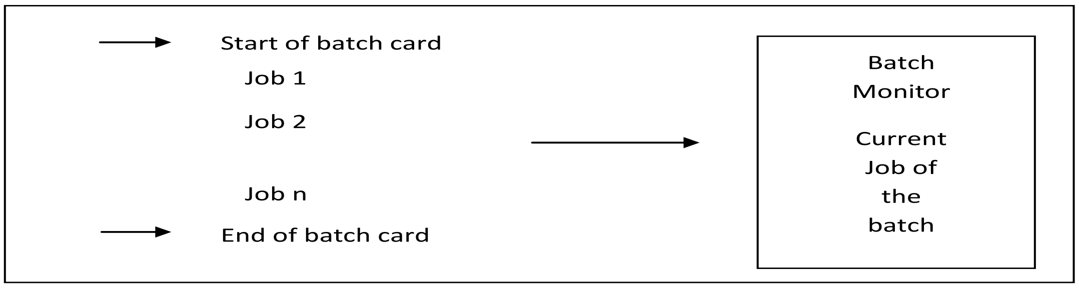
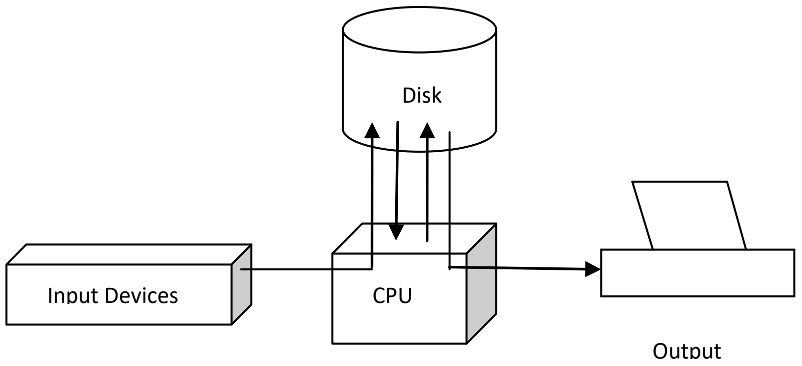


Figure 1.18 Schematic diagram of Batch Processing System

SPOOLING - Spooling is the acronym for Simultaneous Peripheral Operation On-line. Spooling uses the disk as a huge buffer (Job pool), and result in several jobs that have already been read (from input devices) waiting on disk, ready to run and for storing output files until the output devices are able to accept them.

Spooling has a direct beneficial effect on the performance of the system. The computation of one job can overlap with the I/O of other jobs. Thus spooling can keep both CPU and I/O devices working at much higher speeds. Spooling is also used for processing data at remote sites.

A pool of jobs allows OS to select which jobs to run next to increase CPU utilization. When several jobs are on a direct-access device, such as a disk, job scheduling becomes possible. Job scheduling introduces multiprogram and multiprogramming increases CPU utilization. The idea is as follows: The OS keeps several jobs in memory at a time. The OS picks and begins to execute one of the jobs in the memory. The job may have to wait for some task, such as a tape to be mounted, or an I/O operation to complete. In a non-multiprogramming system, the CPU would sit idle. In a multiprogramming system, the OS simply switches to and executes another job. When that job needs to wait, the CPU is switched to another job, and so on. As long as there is always some job to execute, the CPU will never be idle.



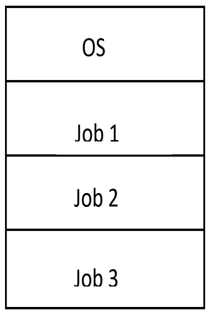


Figure 1.19 Spooling

Figure 1.20 Memory Layout for a multiprogrammed system

2. MUITI PROGRAMMED SYSTEMS:

Spooling results in several jobs that have already been read waiting on disk, ready to be run. A pool of jobs on disk allows the operating system to select which job to run next, to increase CPU utilization, which is not possible when jobs come directly on tapes or on cards. This is called job scheduling. The most important aspect of job scheduling is the ability to multi program. Multiprogramming increases CPU utilization by organizing jobs such that the CPU has always one to execute. The idea of multiprogramming is as follows;

The operating system keeps several jobs in memory at a time. This set of jobs is a subset of the jobs kept in the job pool. The operating system picks and begins to execute one of the jobs in the memory. Eventually, the job may have to wait for some task, such as a tape to be mounted, or an I/O operation to complete. In a non-multi programmed system, the CPU would sit idle. In a multiprogramming system, (see Fig 1.20) the operating system simply switches to and executes another job. When that job needs to wait, the CPU is switched to another job, and so on. Eventually, the first job finishes waiting and gets the CPU bock- As long as there is always some job to execute, the CPU will never be idle.

Multiprogramming is the first instance where the operating system must make decisions for the users. Multi programmed operating systems are therefore fairly sophisticated. All the jobs that enter the system are kept in the job pool. This pool consists of all processes residing on mass storage awaiting allocation of main memory. If several jobs are ready to be loaded into memory and if there is not enough memory, then the system must choose among them. This is the function of job scheduling. The presence of several jobs in the memory at the same time requires some form of memory management. If several jobs are ready to run on the same time the system must choose among them. This function is called CPU scheduling. Finally the operating system must take care to minimize the effect of one job on another when several jobs are running concurrently.

3. TIME-SHARING SYSTEMS:

Multi programmed batched system provides an environment where the various system resources are utilized effectively. There are some difficulties with a batch system from the point of view of the user.

- Since the user cannot interact with the job when it is executing, the user must set up control cards to handle all possible outcomes.
- In multi step job, subsequent steps may depend on the result of earlier ones. It can be difficult to define completely what to do in all cases.
- The programs must be debugged statically, from snapshot dumps. A programmer cannot modify a program as it executes to study its behavior A long turn around time inhibits experimentation with a program.

Time-sharing, or multitasking, is a logical extension of multiprogramming. The CPU switching between them executes multiple jobs, but the switches occur so frequently that the users may interact with each program while it is running. An interactive computer system provides an on-line communication between the user and the system. The user gives instructions to the operating system or to a program directly, and receives an immediate response. An interactive system is used when short response time is required. Early computers with a single user were interactive systems. But the problem with them was the large CPU idle time. Batch OS were developed to avoid this problem. Batch systems improved system utilization for the owners of the computer systems.

Time-sharing systems were developed to provide interactive use of a computer system at a reasonable cost. A time-shared operating system uses CPU scheduling and multiprogramming to provide each user with a small portion of a time-shared computer. Each user has at least one separate program in memory. A program that is loaded into memory and is executing is known as a process. When a process executes, it typically executes for only a short time before it either finishes or needs to perform I/O. Since this interactive input takes a long time to complete, the operating system will rapidly switch the CPU to the program of some other user.

A time-shared operating system allows the many users to share the computer simultaneously. Since each action or command in a time-shared system tends to be short, only a little CPU time is needed for each user. As the system switches rapidly from one user to the next, each user is given the impression that she has her own computer, whereas actually one computer is being shared among many users.

Time-sharing operating systems are even more complex than are multi programmed operating systems. As in multiprogramming, several jobs must be kept simultaneously in memory, which requires some form of memory management and protection. Time-sharing systems must also provide an on-line file system. The file system resides on a collection of disks; hence disk management must also be provided. Also time-sharing systems provide a mechanism for concurrent execution, which requires sophisticated CPU scheduling schemes. To ensure orderly execution, the system must provide mechanisms for job synchronization and communication and must ensure that jobs do not get stuck in a deadlock.

PARALLEL SYSTEMS:

Single-Processor systems have only one main CPU. In a multiprocessor system, there will be more than one processor in close communication, sharing the computer bus, the clock, and sometimes memory and peripheral devices. Such systems are called tightly coupled systems. There are several reasons for building such systems.

- **Increased throughput:** By increasing the number of processors, we hope to get more work done in a shorter period of time.
- **Less cost:** Since multiprocessor systems can share peripherals, cabinets, and power supplies, maintenance cost is less compared to using multiple single systems.
- **Increased reliability:** If functions can be distributed properly among several processors, then the failure of one processor will not halt the system, but rather will only slow it down. If we have 10 processors and one fails, then each of the remaining nine processors must pick-up a share of the work of the failed processor. Thus, the entire system runs only 10% slower, rather than failing altogether. This ability to continue providing service proportional to the level of surviving hardware is called graceful degradation. Systems that are designed for **graceful degradation** are also **called fault-tolerant.**

The most-common multiple-processor systems now use the **symmetric-multiprocessing model**, in which each processor runs an identical copy of the operating system, and these copies communicate with each other when needed. An example is Encore's version of UNIX for the Multimax computer. This computer can be configured to employ dozens of processors, all running a copy of UNIX. The benefit of this model is that many processors can run at once without causing a deterioration of performance. A multiprocessor system of this form will allow jobs and resources to be shared dynamically among the various processors and can lower the variance among the systems.

Some system use **asymmetric multiprocessing**, in which each processor is assigned a specific task. A master processor controls the system; the other processors either look to the master for instructions to have predefined tasks. This is master-slave relationship where the master processor schedules and allocates work to the slave processors. This multiprocessing is more common in extremely large systems where one of the most time-consuming activities is simply processing I/O. In older batch systems, small processors, located at some distance from the main CPU, were used to run card readers and line printers and to transfer these jobs to and from the main computer. These locations are called remote-job-entry sites.

DESKTOP SYSTEMS:

Personal Computers appeared in 1970s. CPUs in PCs lacked the features needed to protect an operating system from user programs. PC operating systems therefore were neither multiuser nor multitasking. The goals of this operating system are to maximize user convenience and responsiveness instead of maximizing CPU and peripheral utilization. These systems include PCs running Microsoft Windows and the Apple Macintosh. Apple Macintosh operating system has been ported to more advanced hardware, and includes new features, such as virtual memory and multitasking. Linux, a UNIX-like operating system available for PCs, has also become popular recently.

Operating systems for these computers have benefited in several ways from the development of operating systems for mainframes. Microcomputers were able to adopt some of the technology developed for larger operating systems. On the other hand, the hardware costs for microcomputers are sufficiently low that individuals have sole use of the computer, and CPU utilization is no longer a prime concern. Thus, some of the design decisions made in operating systems for mainframes may not be appropriate for smaller systems.

Other decisions still apply - for example, file protection was, at first, not necessary on a personal machine. However these computers are tied into other computers over local-area networks or other Internet connections. When other users on other computes/can access the files, file protection again becomes a necessary feature of the operating system. The lack of such protection has made it easy for mailicious programs to destroy data on systems such as MS-DOS and the Macintosh operating system. The programs may be self-replicating, and may spread rapidly via worm or virus mechanisms and disrupt entire companies or even worldwide networks. Advanced time-sharing features such as protected memory and file permission are not enough, on their own, to safeguard a system from attack.

MULTIPROCESSOR SYSTEMS:

Uniprocessor has one main CPU. A true multiprocessor system would have more than one CPU, sharing memory and peripherals. The advantage of multiprocessor system is that they have greater computing power and reliability, have more than one processor in close communication, sharing the computer bus, the clock, and sometimes memory and peripheral devices.

More commonly one of the following approaches is used. The most common multiple processor systems assign to each processor a specific task. A master processor controls the system; the other processors either look to the master for instruction or have predefined tasks. This scheme defines a master/slave relationship. Small processors, located at some distance from the main cpu, may be used to run card readers and line printers and transfer these jobs to and from the main computer. These locations are called Remote Job Entry (RJE) sites. Time-sharing systems are composed of a large computer like a DEC-20) which is the main computer and smaller front-end computer (like a PDP-11) which is simply responsible for the terminal I/O.

The second multiple processor is a computer network. In a network, multiple independent computer systems can communicate, sending files and information between them. However, each computer system has its own operating system and operates independently. Computer networks allow new possibilities in distributed processing. Multiprocessor systems have three main advantages.

1. **Increased throughput:** By increasing the number of processors, more work is done at less time. The speed-up ratio with N processors is less than N. When multiple processors cooperate on a task, a certain amount of overhead is incurred in keeping all the parts working correctly. This overhead, plus contention for shared resources, lowers the expected gain from additional processsors.
2. **Economy of scale:** Multiprocessor systems can save more money than multiple single-processor systems, because they can share peripherals, mass storage, and power supplies. If several programs operate on the same set of data, it is cheaper to store those data on one disk and to have all the processors share them, than to have many computers with local disks and many copies of the data.

3. Increased reliability: If functions can be distributed properly among several processors, then the failure of one processor will not halt the system, only slow it down. If there are 10 processors, and one fails, then each of the remaining nine processors must pick up a share of the work of the failed processor. Thus, the entire system runs only 10 percent slower, rather than failing altogether.

This ability to continue providing service proportional to the level of surviving hardware is called **graceful degradation**. Systems designed for graceful degradation are also called **fault tolerant.**

Some systems use asymmetric multiprocessing, in which each processor is assigned a specific task. A master processor controls the system; the other processors either look to the master-slave relationship. The master processor schedules and allocates work to the slave processors. The most common multiple-processor systems use symmetric multiprocessing (SMP), in which each processor runs an identical copy of the operating system, and these copies communicate with one another. SMP means all processors are peers; no master-slave relationship exists between processors. Each processor concurrently runs a copy of the operating system.

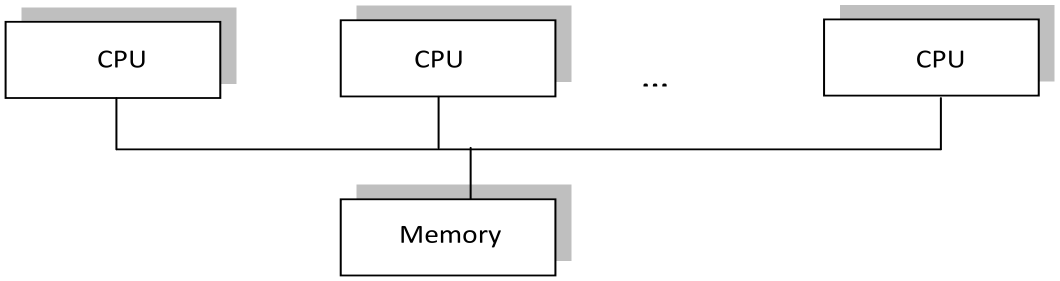


Figure 1.21 Symmetric Multiprocessing System

Figure 1.21 illustrates a typical SMP architecture. An example of the SMP system is Encore's version of UNIX for the Multimax computer. This computer employs dozens of processors, all running copies of UNIX. The benefit of this model is that many processes can run simultaneously - N processes can run if there are N CPUs - without causing a significant deterioration of performance. We must carefully control I/O to ensure that the data reach the appropriate processor. Also since the CPUs are separate, one may be sitting idle while another is overloaded, resulting in inefficiencies. These inefficiencies can be avoided if the processors share certain data structures. A multiprocessor system of this form will allow processes and resources - such as memory - to be shared dynamically among the various processors, and can lower the variance among the processors.

The difference between symmetric and asymmetric multiprocessing may be the result of either hardware or software. Special hardware can differentiate the multiple processors, or the software can be written to allow only one master and multiple slaves.

DISTRIBUTED SYSTEM:

A recent trend in computer systems is to distribute computation among several processors. In contrast to the tightly coupled systems, the processors do not share memory or a clock. Instead, each processor has its own local memory. The processors communicate with one another through various communication lines, such as high-speed buses or telephone lines. These systems are usually referred to as **loosely coupled systems, or distributed system**.

The processors in a distributed system may vary in size and function. They may include small microprocessors, workstations, minicomputers, and large general-purpose computer systems. These processors are either called sites, computer or nodes defending on the context.

Some of the reasons for building distributed systems are

- **Resource sharing:** Resource sharing in a distributed system provides mechanisms for sharing files at remote sites, processing information in a distributed database, printing files at remote sites, using remote specialized hardware devices and performing other operations. For example, a user at site A may be using a laser printer available only at site B. Meanwhile, a user at B may access a file that resides at A.
- **Computation speedup:** Distributed system allows distributing the computation among the various sites. A particular computation can be partitioned into a number of subcomputations that can run concurrently. If a particular site is currently overloaded with jobs, some of them may be moved to other, lightly loaded sites. This movement of jobs is called load sharing
- **Reliability:** If one site fails in a distributed system, the remaining sites can potentially continue operating. The failure of one site will not affect the rest. In general, if sufficient redundancy exists in the system (in both hardware and data), the system can continue with its operation, even if some of its sites have failed.

- **Communication:** When a communication network connects many sites to one another, the processes at different sites have the opportunity to exchange information. Window systems are one example, since they frequently share data or transfer data between displays. Users may initiate file transfers or communicate with one another via electronic mail. A user can send mail to another user at the same site or a different site.

Distributed systems depend on networking for their functionality. Distributed Systems are able to share computational tasks, and provide a rich set of features to users. These systems also vary by their performance and reliability.

(i) Client-Server Systems: As PCs have become faster, more powerful, and cheaper, designers have shifted away from the centralized system architecture. As a result, centralized systems today act as server systems to satisfy requests generated by client systems. The general structure of the client-server system is depicted in Figure 1.22.

Server systems can be broadly categorized as compute servers and file servers.

- Compute-server systems provide an interface to which clients can send requests to perform an action, in response to which they execute the action and send back results to the client.
- File-server systems provide a file-system interface where clients can create, update, read, and delete files.

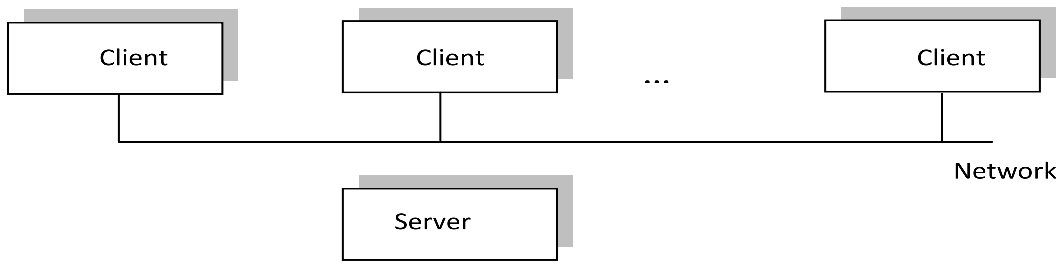


Figure 1.22 General Structure of a client-server System

(ii) Peer-to-Peer Systems: The growth of computer networks - especially the Internet and World Wide Web (WWW) - has had a profound influence on the recent development of operating-systems.

Tightly coupled systems have more than one processor in close communication, sharing the computer bus, the clock, and sometimes memory and peripheral devices. **Loosely coupled systems (or distributed systems)** consist of a collection of processors that do not share memory or a clock. Instead each processor has its own local memory. The processors communicate with one another through various communication lines, such as high-speed buses or telephone lines.

CLUSTERED SYSTEMS: Clustered systems gather together multiple CPUs to accomplish computational work. These systems are composed of two or more individual systems coupled together. The generally accepted definition is that clustered computers share storage and are closely linked via LAN networking.

Clusturing is performed to provide high availability. A layer of cluster software runs on the cluster nodes. Each node can monitor one or more of the others (over the LAN). If the monitored machine fails, the monitoring machine can take ownership of its storage, and restart the application(s) that were running on the failed machine. The failed machine can remain down. In asymmetric clustering, one machine is in hot standby mode while the other is running the applications. The hot standby host machine does nothing but monitor the active server. If that server fails, the hot standby host becomes the active server. In this, two or more hosts are running applications, and they are monitoring each other. This is more efficient as it uses all the available hardware.

Parallel clusters allow multiple hosts to access the same data on the shared storage. These are accomplished by special versions of software and special releases of applications. For example, Oracle Parallel Server is a version of Oracle's database that has been designed to run on parallel clusters. Each machine runs Oracle, and a layer of software tracks access to the shared disk. Each machine has full access to all data in the database. In distributed computing, most clusters do not allow shared access to data on the disk. For this, distributed file systems must provide access control and locking to the files to ensure no conflicting operations occur. This type of service is known as a distributed lock manager (DLM). Cluster technology is rapidly changing. Cluster directions include global clusters, in which the machines could be anywhere in the world.

REAL TIME SYSTEMS:

A real-time system is a special-purpose operating system. A real-time system is used when there are rigid time requirements on the operation of a processor or the flow of data, and thus is often used as a control device in a dedicated application. Sensors bring data into the computer. The computer must analyze the data and possibly adjust controls to modify the sensor inputs. Systems that control scientific experiments, industrial control systems are real time systems.

A real time operating system has well-defined, fixed time constraints. Processing must be done within the defined constraints, or the system will fail. For instance, it would not do for a robot arm to be instructed to halt after it had smashed into the car it was building. A real-time system is considered to function correctly only if it returns the correct result within any time constraints. Contrast this requirement to a time-sharing system, where it is desirable to respond quickly, or to a batch system, where there may be no time constraints at all.

There are two types of real-time systems. A **hard real-time system** guarantees that critical tasks complete on time. This goal requires that all delays in the system are bounded, from the retrieval of stored data to the time that it takes the operating system to finish any request made of it. Such time constraints dictate the facilities that are available in hard real-time systems. None of the existing general-purpose operating systems support hard real -time facility. A less restrictive type of real-time system is a **soft real-time system**, where a critical real-time task gets priority over other tasks, and retains that priority until it completes. Soft real-time systems do not have deadline scheduling. Soft real-time, have more limited utility than hard real-time systems. Given their lack of deadline support, they are risky to use for industrial control and robotics. They are very useful in areas like multimedia, virtual reality, advanced scientific projects - such as undersea exploration and planetary rovers.

HANDHELD SYSTEMS: Handheld systems include personal digital assistants (PDAs) such as Palm-Pilots or cellular telephones with connectivity to a network such as the Internet. Developers of handheld systems and applications face many challenges, most of which are due to the limited size of such devices. For example, a PDA is about 5 inches in height and 3 inches in width and it weighs less than one-half pound. Due to this size, they have limited memory, include slow processors, and small display screens.

Many handheld devices have memory between 512 KB and 8MB. So the operating system and applications must manage memory efficiently. This includes returning all allocated memory back to the memory manager once the memory is no longer being used. Currently, many handheld devices do not use virtual memory techniques, and are confined to limited physical memory.

A second issue of concern to developers of handheld devices is the speed of the processor used in the device. Processors for most handheld devices run at a fraction of the speed of a process in a PC. Faster processors require more power. To include a faster processor in a handheld device would require a larger battery that would have to be replaced (or recharged) more frequently. To minimize the size of handheld devices, smaller, slower processors, which consume less power, are used. Therefore, the operating system and applications must be designed not to tax the processor.

Handheld devices also need small display screens and the size is not more than 3 inches square. Familiar tasks, such as reading e-mail or browsing web pages, must be condensed onto smaller displays. One approach for displaying the content in web pages is web clipping, where only a small subset of a web page is delivered and displayed on the handheld device.

Some handheld devices may use wireless technology, such as BlueTooth, allowing remote access to e-mail and web browsing. Cellular telephones with connectivity to the Internet fall into this category. Many PDAs currently do not provide wireless access. To download data to these devices, one first downloads the data to a PC or workstation, and then downloads the data to the PDA. Some PDAs allow data to be directly copied from one device to another using an infrared link.

OPERATING SYSTEM STRUCTURES

OS STRUCTURES – COMPONENTS OF AN OPERATING SYSTEM

A system can be created as large and complex as an operating system only by partitioning it into smaller pieces. Each piece should be a well –delineated portion of the system, with carefully defined inputs, outputs and functions. The following are the different components :

1. Process Management:

A process is a program in execution. A compiler is a process, a word-processing program run on a PC is a process, a system task, such as sending output to a printer, is a process. A process needs certain resources – including CPU time, memory files, and I/O devices to accomplish its task. These resources are either given to the process when it is created or allocated to it while it is running.

A program by itself is not a process; a program is a passive entity, where as a process is an active entity, with a program counter specifying the next instruction to execute.

A process is the unit of work in a system. Such a system consists of a collection of processes, some of which are operating system processes (those that execute system code) and the rest of which are user processes (those that execute user code). All these processes potentially execute concurrently, by multiplexing the CPU among them.

The operating system is responsible for the following activities in connection with process 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 process communication
- The provision of mechanisms for deadlock handling

2. Main-Memory Management:

The main memory is central to the operation of a modern computer system. Main memory is a large array of words or bytes, ranging in size from hundreds of thousands to hundreds of millions. Each word has its own address. Main memory is a repository of quickly accessible data shared by the CPU and I/O devices. The central processor reads instructions from main memory during the instruction-fetch cycle, and both reads and writes data into memory during the data-fetch cycle. The I/O operations implemented via DMA also read and write data in memory. The main memory is generally the only large storage device that the CPU is able to address and access directly.

For a program to be executed, it must be mapped to absolute addresses and loaded into memory. As the program executes, it accesses program instructions and data from memory by generating these absolute addresses. Eventually, the program terminates, its memory space is declared available, and the next program can be loaded and executed.

To improve both the utilization of CPU and the speed of the computer's response to its users, we must keep track of several programs in memory. There are many different memory-management schemes. Selection of a memory-management scheme for a specific system depends on factors such as h/w 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 required

3. File Management:

File management is one of the most visible components of an operating system. Computers can store information on several different types of physical media. Magnetic tape, magnetic disk and optical disk are the most common media. Each of these media has its own characteristics and physical organization. The operating system maps file onto physical media and accesses these files via the storage devices.

A file is a collection of related information defined. Commonly, files represent programs and data. A file consists of a sequence of bits, bytes, lines, or records.

The operating system implements the abstract concept of a file by managing mass storage media, such as tapes and disks, and the devices, which control them. Also, files are normally organized into directories to ease their use. The operating system is responsible for the following activities in connection with file management:

- The creation and deletion of files
- The creation and deletion of directories.
- The support of primitives for manipulating files and directories
- The mapping of files onto secondary storage
- The backup of files onto Secondary storage
- The backup of files on stable storage media.

4. I/O System Management:

One of the purposes of an operating system is to hide the peculiarities of specific hardware resources from the user. For E.g., in UNIX, the peculiarities of I/O devices are hidden from the bulk of the operating system itself by the I/O subsystem. The I/O subsystem consists of

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

5. Secondary-Storage Management:

The main purpose of a computer system is to execute programs. These programs, with the data they access, must be in main memory during execution. Because main memory is too small and volatile, the computer system must provide a secondary storage to back up main memory. Most modern systems use disk as the secondary storage. Since most programs resides in secondary storage and they use disk as the source and destination of their processing, proper management of disk storage is of central importance to the computer system.

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

- Free-space management
- Storage allocation
- Disk scheduling

6. Networking:

A distributed system is a collection of processors that do not share memory, peripheral devices, or a clock. Instead, each processor has its own local memory and clock, and the processors communicate with one another through various communication lines, such as high-speed buses or telephone lines. The processors in a distributed system vary in size and function. They may include small 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 a 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 contention and security.

A distributed system collects physically separate, possibly heterogeneous systems into a single coherent system, providing the user with access to the various resources that the system maintains. Access to a shared resource allows computation speedup, increased data availability, and enhanced reliability. World Wide Web has created a new access method for information sharing. It improved on existing file-transfer protocol (FTP) and network file-system (NFS). It defined a new protocol, hypertext transfer protocol (http), for use in communication between a web server and web browser. The web browser needs to send a request for information to a remote machine's web server, and the information is returned.

7. Protection system

Protection refers to a mechanism for controlling the access of programs, processes, or users to the resources defined by a computer system. This mechanism must provide a means for specification of the controls to be imposed, together with a means of enforcement.

Protection can improve reliability by detecting latent errors at the interfaces between component subsystems. Early detection of such errors can prevent contamination of a healthy subsystem by another subsystem that is malfunctioning.

8. Command-interpreter system:

Command-interpreter is the interface between the user and operating system. Some operating systems include the command interpreter in the kernel. Some others treat the command interpreter as a special program that is running when a job is initiated, or when a user first logs on.

Many commands are given to the operating system by control statements. When a job is started in a batch system or when a user logs on to a time-shared system, a program that reads and interprets control statements is executed automatically. This program is often called the shell. Its function is - Get the next command and execute it.

Operating systems are frequently differentiated in the area of the shell, with a user-friendly command interpreter making the system more agreeable to some users. One style is the mouse-based window and menu system in the Macintosh and MS-Windows. In some others, commands are typed on a keyboard and displayed on a screen or printing terminal, with the enter key signaling that a command is complete and is ready to be executed. The MS-DOS and UNIX shells are like this.

The command statements themselves deal with process creation and management, I/O handling, secondary-storage management, main memory management, file-system access, protection, and networking.

OPERATING SYSTEM SERVICES:

An operating system provides an environment for the execution of programs. The operating system provides certain services to programs and to the users of those programs. The services provided differ from system to system, but there are some common services, which we can identify. These operating system services are provided for the convenience of the programmer, to make the programming function easier.

1. Program Execution: The system must be able to load a program into memory and run it. The program must be able to end its execution, either normally or abnormally.

2. I/O Operation: A running program may require I/O. This I/O may involve a file or an I/O device. For specific devices, special functions may be desired. For efficiency and protection, users may be able to control I/O devices directly. Therefore, an operating system must provide some means to do I/O.

3. File-system manipulation: The programs need to read and write files. They also need to create and delete files by name.

4. Communications: There are many circumstances in which one process needs to exchange information with another process. There are two ways in which communication can occur:

- between processes executing on the same computer
- between processes executing on different computer systems that are tied together by a computer network.

Communications may be implemented via shared memory, or by the technique of message passing, in which packets of information are moved between processes by the operating systems.

5. Error Detection: The operating system constantly needs to be aware of possible errors. Errors may occur in the CPU and memory hardware (such as a memory failure or a power failure), in I/O devices (connection failure in a network etc) or in the user program (such as arithmetic overflow etc). For each type of error, the operating system should take the appropriate action to ensure correct and consistent computing.

Another set of operating system functions exists not for helping the users but rather for ensuring the efficient operation of the system.

1. Resource allocation: When there are multiple users or multiple jobs running at the same time, resources must be allocated to each of them. Many different types of resources are managed by the operating system. Some (CPU cycles, main memory, file storage) may have special allocation code, whereas others (I/O devices) may have much more general request and release code. For instance, in determining how best to use the CPU, operating system have CPU-scheduling routines that take into account the speed of the CPU, the jobs that must be executed, the number of registers available, and other factors.

2. Accounting: Keeping track of the amount of resources used by different users is required either for accounting or simply for accumulating usage statistics. Usage statistics is a valuable tool for researchers who wish to reconfigure the system to improve computing services.

3. Protection: Protection involves ensuring that all access to system resources is controlled. Security from outsiders is also important.

4. Control program: An operating system is a control program. A control program controls the execution of user programs to prevent errors and improper use of the computer. It is concerned with the operation and control of I/O devices.

SYSTEM CALLS:

System calls provide the interface between the process and the operating system. These calls are generally available as assembly language instructions, and are used by assembly language programmers. Some systems may allow system calls to be made directly from a higher-level language program, and calls resemble predefined function or subroutine calls.

The following is an example of how the system calls are used. Consider writing a program to read data from one file and to copy them to another file. The first input the program will need is the names of the two files : the input file and the output file. The names can be specified in the following ways.

a. program can ask the user for the names of two files. In an interactive approach it will require a sequence of system calls. First to write a prompting message on the screen, and then to read the characters that define the two files.

b. In the batch processing system, the names of the files can be specified with control statements. The mechanism is passing the parameters from the control statements to executing program.

Once the two files are obtained the program must open the input file and create the output file. Both these operations need the System call. There may also be possible errors. The input file may not exist. So the program should print the message. If file exists, then the output file has to be created. If the output file with the same name is existing then the system call can abort the program, or to delete the existing file (system call) and create a new one (another system call).

In an Interactive system, the user is asked whether to replace the existing file or to abort the program. Now it reads from the input file (system call) and writes to output file (system call). Each read and write must return status information regarding various possible errors. The write operation may encounter various errors depending on the output device. Finally after the entire file is copied, the program may close both files (another system call), writes a message to console (system call) and finally terminate normally (system call). To get input, we may need to specify the file or device to use as the source, and the address and length of memory buffer into which input should be read. So the device or file and length may be implicit in the call.

System calls can be roughly grouped into five major categories; process control, file manipulation, device manipulation, information maintenance, and, communication. The following are the functions of each major category of system calls:

- Process control
  - End, abort
  - Load, execute
  - Create process, terminate process
  - Set process attributes, set process attributes
  - Wait for time
  - Wait event. Signal event
  - Allocate, free memory
- File manipulation
  - Create file, delete file
  - Open, close
  - Read, write, reposition
  - Set file attributes, set file attributes

- Device manipulation
  - Request device, release device
  - Read, write, reposition
  - Set device attributes, set device attributes
  - Logically attach or detach devices
- Information maintenance
  - Set time or date, set time or date
  - Set system data, set system data
  - Set process, file, or device attributes
  - Set process, file, or device attributes
- Communications
  - Create, delete communication connection
  - Send, receive messages
  - Transfer status information
  - Attach or detach remote devices

1. Process and Job control:

A running program needs to be able to halt its execution either normally (**end**) or abnormally (**abort**). Under either normal or abnormal circumstances, the operating system must transfer control to the command interpreter. In an interactive system, the command interpreter continues with the next command where as in a batch system, the command interpreter usually terminates the entire job and continues with the next command.

A process or job executing one program may want to **load** and **execute** other program. This feature allows the command interpreter to execute a program as directed by the user. An important decision to be made here is where to return control when the loaded program terminates. There is a system call specifically for this purpose (**create process or submit job**).

If we create a new job or process, or a set of jobs or processes, we should be able to control its execution. This control requires the ability to determine and reset the attributes of a job or process, including the job's priority, its maximum allowable execution time etc (**get process attributes**). We may also want to terminate a job or process that we created (**terminate process**) if we find that it is incorrect or no longer needed.

Once a job or process is created, we may need to wait for a certain amount of time either to finish their execution (**wait time**) or for a specific event to occur (**wait event**). The job or processes should then signal when that event has occurred (**signal event**). Many operating systems provide system calls to **dump memory**, which are useful for debugging.

2. File manipulation:

Two common system calls associated with files are **Create** and **Delete**. Either system calls require the name of the file and perhaps some of the file's attributes. Once the file is created, we need to **open** it and to use it. Operations like read, write or reposition (**rewinding or skipping to the end**) can also be used. Finally we need to **close** the file, indicating that we are no longer using it.

If we have a directory structure for organizing files, we may need the same set of operations. In addition, for either files or directories we need to determine the various attributes and reset them if necessary. File Attributes include file name, file type, protection codes and Accounting information. At least two system calls, **get file attribute** and **set file attribute**, are required for this function.

3. Device Management:

A program while it is running may need additional resources to proceed. If they are available they are readily granted. If not, the program has to wait until sufficient resources are available. If there are multiple users of the systems, we must first **request** the device, and after we are finished with the device, we must **release** it. These functions are similar to **open** and **close** system calls for files, since files can be thought of as abstract or virtual devices. Once the device has been allocated, we can **read**, **write** and **reposition** the device, just as we can with ordinary files.

4. Information Maintenance:

Many system calls exists for the purpose of transferring information between the user program and the operating system. For E.g. most systems have a system call to return the **current time and date**. Other System calls may return about the system, such as the number of the current users, version of OS, amount of free spaces. In addition the OS keeps information about all its processes, and there are system calls to access this information. There are also calls to reset the process information (**get process attributes and set process attributes**).

5. Communication:

There are two common models of communication

1. Message-passing model
2. Shared-memory model

In message-passing model, information is exchanged through an inter¬process communication facility provided by the operating system. Before communication can take place, a connection must be opened and the name of the Communicator must be known (it can be another process of the same CPU, or a process of another computer connected by a communication network). The **gethostid** and **getprocessid** system calls do this function. These identifiers are then passed to the general-purpose open and close calls provided by the file system, or to specific **open connection** and **close connection** system calls, depending on the system's model of communications. The recipient process usually must give its permission for communication to take place with an **accept connection** call. Most processes that will be receiving connections are system programs dedicated to that purpose. They execute a **wait for connection** call and are awakened when a connection is made. The source of communication, called client, and the receiver called server, then exchange messages by **read message** and **write message** system calls. The close connection call terminates the communication.

In the shared memory model, processes use **map memory** system calls to gain access to regions of memory owned by other processes. They may then exchange information by reading and writing data in the shared areas. The form of data and the location are determined by these processes and are not under the operating system's control. The processes are also responsible for ensuring that they are not writing to the same location simultaneously.

Message passing is useful when useful when smaller numbers of data need to be exchanged, because no conflicts need to be avoided. It is also easier to implement than shared memory for inter computer communication. Shared memory allows maximum speed and convenience of communication.

SYSTEM PROGRAMS:

Modern system is the collection of system programs. At the lowest level is the hardware. Next is the operating system, then the system programs and finally the application programs. System programs provide a convenient environment for program development and execution. Some of them are simply user interfaces to system calls; others are considerably more complex. They can be divided into these categories:

- **File management:** These programs create, delete, copy, rename, print, dump, list and generally manipulate files and directories.
- **Status information:** Information like date, time, amount of available memory or disk space, number of users, or similar status information. The information is formatted and is printed to the terminal or other output device or file.
- **File modification:** Several text editors may be available to create and modify the content of files stored on disk or tape.
- **Programming-language support:** Compilers, assemblers and interpreters for common programming languages are provided to the user with the operating system.
- **Program loading and execution:** Once a program is assembled or compiled, it must be loaded into memory to be executed. The system may provide absolute loaders,

relocatable loaders, linkage editors, and overlay loaders. Debugging systems for either higher-level languages or machine language are needed.

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

Most operating systems are supplied with programs that solve common problems, or perform common operations. Such programs include web browsers, word processors and text formatters, spread sheets, database systems, compilers, plotting and statistical-analysis packages and games. These programs are known as system utilities or application programs.

Most important system program for an operating system is the command interpreter, the main function of which is to get and execute the next user-specified command. Command Interpreter contains the code to execute the command. For example, a command to delete a file may cause the command interpreter to jump to a section of its code that sets up the parameters and makes the appropriate system call.

PROCESS MANAGEMENT

2.1 Process

2.1.1. Introduction

A process is the basic unit of execution in an operating system.

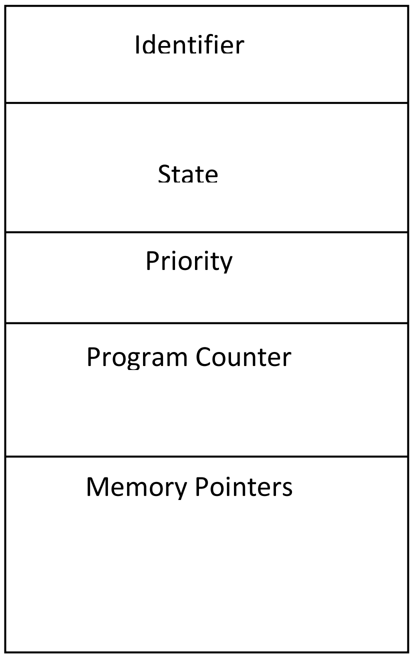
- A process is an instance of a computer program that is being sequentially executed by a computer system that has the ability to run several computer programs concurrently.
- A process is a program in execution which can be assigned to and executed on a processor
- Process is an entity that consists of a number of elements - program code and a set of data associated with that code.

2.1.2 Process Control Block

- All the information needed to keep track of a process when switching, is kept in a data package called a Process Control Block (PCB).
- While the program is executing, the process can be uniquely characterized by a number of elements. The process control block typically contains the following information which is stored in a data structure called Process Control Block that is created and managed by OS.

- **Identifier**: A unique identifier to distinguish it from all other processes.
- **State**: It contains state of process. The state may be new, ready, running waiting and halted.
- **Priority**: It contains the priority of the process. That is Priority level relative to other processes.
- **Program counter**: Program counter indicates the address of the next instruction to be executed for current process.
- **Memory pointers**: Includes pointers to the program code and data associated with this process, plus any memory blocks shared with other processes.
- **Context data**: These are data that are present in registers in the processor while the process is executing.

- **CPU Registers**: The registers vary in number and type depending on the computer architecture. They include accumulators, index registers, stack pointers and general-purpose registers.
- **CPU Scheduling information**: This information includes a process priority, pointers to scheduling queues and any other scheduling parameters.
- **I/O status information**: This information includes the list of I/O requests, I/O devices (e.g., tape drives) assigned to this process, a list of files in use by the process, and so on.
- **Accounting information**: May include the amount of processor time and clock time used, time limits, account numbers, and so on.
  - When a process is interrupted, the current values of the program counter and the processor registers (context data) are saved in the appropriate fields of the corresponding process control block, and the state of the process is changed to blocked or ready.
  - The OS is now free to put some other process in the running state. The program counter and context data for this process are loaded into the processor registers and this process now begins to execute.



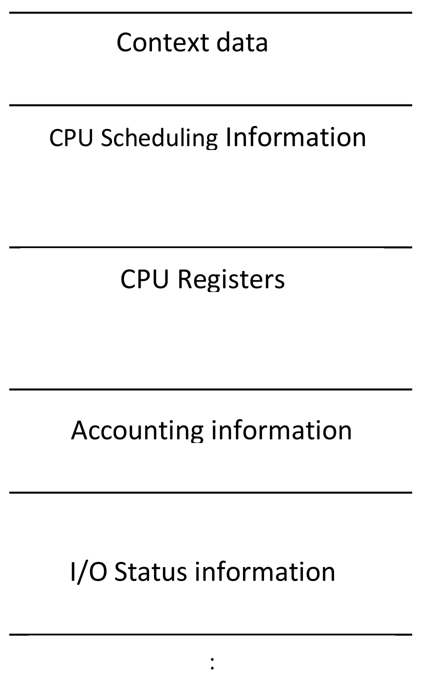


Fig 2.1 Process Control Block

2.1.3 Process States:

As a process executes, it changes state. The state of a process is defined as the current activity of that process. The state includes whether the process has executed or whether the process is waiting for some input and output from the user and whether the process is waiting for the CPU to run the program after the completion of the process. Following are the state models:

1. **Two-state Process model** 2. **Five-state model**

1. Two-State process model:

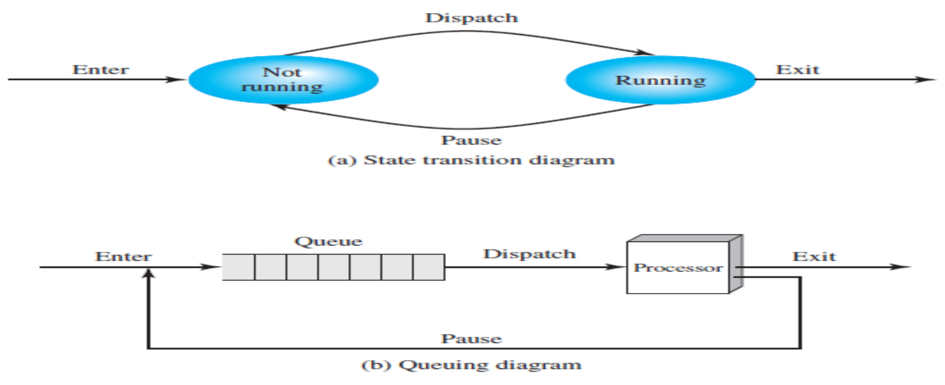


Fig 2.2 Two-State Transition diagram

- The operating system’s principal responsibility is controlling the execution of processes
- In the Two-state model, a process is either being executed by a processor or not. A process may be in one of two states: Running or Not Running, as shown in Figure 2.2.
- When the OS creates a new process, it creates a process control block for the process and enters that process into the system in the Not Running state.
- From time to time, the currently running process will be interrupted and the dispatcher portion of the OS will select some other process to run.
- The former process moves from the Running state to the Not Running state, and one of the other processes moves to the Running state.
- Information relating to each process, including current state and location in memory; is stored in the process control block.

Five-state model:

In the Two-state model some processes in the Not Running state are ready to execute, while others are blocked, waiting for an I/O operation to complete. In Five-state model, the Not

Running state is split into two states: Ready and Blocked. This is shown in Figure 2.3. The five states in this state are as follows:

- **New :** A process that has just been <u>created</u> but has not yet been admitted to the pool of executable processes by the OS. A new process has not yet been loaded into main memory, although its process control block has been created.
- **Ready :** This state indicate process is waiting to be assigned to a processor for execution
- **Running :** The process that is currently being executed.
- **Blocked/Waiting** : A process that cannot execute until some event occurs, such as the completion of an I/O operation.
- **Exit/Terminated :** A process that has been released from the pool of executable processes by the OS, either because it halted or because it aborted for some reason.

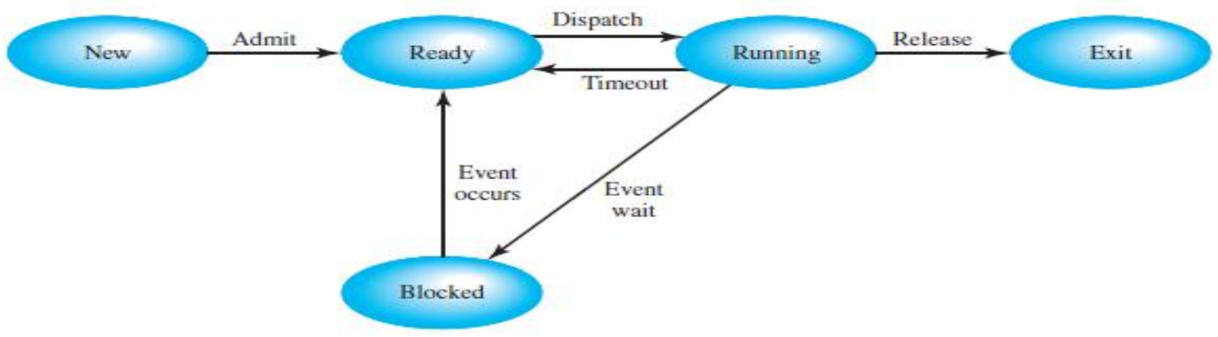


Fig 2.3 Five-state model

- The New state corresponds to a process that has just been defined.
- While a process is in the new state, information concerning the process that is needed by the OS is maintained in control tables in main memory. The process itself is not in main

memory. That is, the code of the program to be executed is not in main memory, and no space has been allocated for the data associated with that program. While the process is in the New state, the program remains in secondary storage, typically disk storage.

- Similarly, a process exits a system in two stages. First, a process is terminated when it reaches a natural completion point, when it aborts due to an unrecoverable error, or when another process with the appropriate authority causes the process to abort. Termination moves the process to the exit state. At this point, the process is no longer eligible for execution.

2.5 CPU Scheduling:

2.5.1 Introduction:

- The objective of multiprogramming is to have some process running at all times, so as to maximize CPU utilization.
- The objective of time-sharing is to switch the CPU among processes so frequently that users can interact with each program while it is running.
- Scheduling refers to a set of policies and mechanisms built into the operating system that govern the order in which work is to be done by a computer system.
- In multi-programming systems, when there is more than one running process, the Operating system must decide which one to run first. The part of Operating System concerned with this decision is called Scheduler and the algorithm it uses is called scheduling algorithm.

2.5.2 CPU-I/O Burst Cycle:

Process execution consists of a cycle of CPU execution and I/O wait. Processes alternate back and forth between these two states. Process execution begins with a CPU burst. That is followed by an I/O burst, which is followed by another CPU burst, then another I/O burst, and so on. Eventually, the last CPU burst will end with a system request to terminate execution.

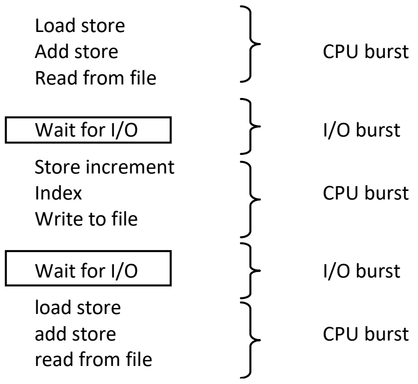


Figure 2.14 Alternating sequence of CPU and I/O bursts

The durations of these CPU bursts have been extensively measured. They vary from process to process and computer to computer. An I/O-bound program typically have many very short CPU bursts. A CPU-bound program might have a few very long CPU bursts. This distribution can be important in the selection of an appropriate CPU scheduling algorithm.

2.5.3 CPU Schedulers: The process migrates between the various scheduling queues. The operating system must select the processes from these queues in some fashion. The selection process is carried out by the scheduler. Scheduling is the activity of determining which service request should be handled next by the server. When the server is CPU it is called CPU scheduling. The objective of multiprogramming is to have some process running at all times to maximize CPU utilization. For a uniprocessor system, there will never be more than one running process. In multiprogramming the time is used productively. Several processes are kept in memory at one time. When one process has to wait, the OS takes the CPU away from that process and gives the CPU to another process. This pattern continues.

Types of Schedulers: In general there are three different types of schedulers.

a. Long term scheduler: In batch systems, processes are spooled to a mass-storage device, where they are kept for later execution. The **long-term scheduler**, or job scheduler, selects processes from this pool and loads them into memory for execution. Most processes can be described as either I/O bound or CPU bound. An I/O bound process spends more of its time doing I/O than it spends doing computations. A CPU bound process, generates I/O requests infrequently, using more of its time doing computation than an I/O bound process uses. The long-term scheduler should select a good process mix of I/O bound and CPU bound processes. If all processes are CPU bound, the I/O waiting queue will almost always be empty, devices will go unused, and again the system will be unbalanced. The system with the best performance will have a combination of CPU-bound and I/O bound processes. The long-term scheduler executes much less frequently and controls the degree of multiprogramming.

b. Short-term scheduler: Short term scheduler, selects from among the processes that are ready to execute, and allocates the CPU to one of them. It must select a new process for the CPU frequently. Whenever the CPU becomes idle, the operating system must select one of the processes in the ready queue to be executed. This selection process is carried out by short-term scheduler or CPU scheduler. The scheduler selects from among the processes in memory that are ready to execute, and allocates the CPU to one of them. The ready queue may be implemented as a FIFO queue, a priority queue, a tree, or simply an unordered linked list.

c. Medium-term scheduler: Medium term scheduler is the part of operating system that performs the swapping function.

i) It determines which waiting process should be swapped out to disk

ii) It determines which ready process should be moved to ready suspended if there is no other way to free memory and running process requires more memory

iii) It determines which ready suspended process should be moved to ready state when there are no more ready processes in memory.

2.5.4 Scheduling Policies:

There are two general categories of scheduling policies: They are Non-preemptive and Preemptive Scheduling.

Non-Preemtptive Scheduling: A scheduling discipline is non-preemptive if, once the CPU has been allocated to a process, the process keeps the CPU until it releases the CPU, either by terminating or by switching to wait state.

Characteristics of Non-preemptive scheduling:

1. Short jobs are made to wait by longer jobs
2. Response time is more predictable because incoming high priority jobs cannot displace waiting jobs
3. A scheduler executes jobs in the following two situations:

   i) When a process switches from running state to the waiting state

ii) When a process terminate

Preemtptive Scheduling: In preemptive scheduling the CPU can be switched to a new process before the processing of an earlier scheduled process is complete. This is contrast to "run to completion".

A scheduler executes jobs in the following two situations:

i) When a process switches from running state to the ready state (e.g. when an interrupt occurs)

ii) When a process switches from the waiting state to the ready state (e.g., completion of I/O)

A process can be interrupted because of the following reasons:

i) The allocated service interval (time slice) expires

ii) Another process with a higher priority has arrived into the ready queue

2.5.5 Scheduling Criteria:

Different CPU scheduling algorithms have different properties and may favor one class of process over other. Many criteria have been suggested for comparing CPU scheduling algorithms. Scheduling Criteria includes the following:

CPU utilization: The CPU should be as busy as possible. CPU utilization may range from 0 to 100%. In real systems it ranges from 40 to 90%.

Throughput: It is defined as the number of process that are completed per unit time. For long processes, the rate may be one process/hour: for short transactions, throughput might be 10 processes/second.

Turnaround time: The interval from the time of submission of a process to the time completion is the turnaround time. It 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 affects only the amount 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: It is the time from the submission of a request until the first response is produced i.e., response time is the amount of time taken to start responding, but not the time that takes to output that response. It is desirable to maximize CPU utilization & throughput and minimize turnaround time, waiting time and response time.

2.5.6 SCHEDULING ALGORITHMS:

These algorithms decide which of the process in the ready queue is to be allocated the CPU. There are various scheduling algorithms.

1. **First-Come, First-Served scheduling (FCFS):**

This is the simplest CPU scheduling algorithm. According to this scheme the process that requests the CPU first is allocated the CPU first. The implementation of the FCFS policy is managed with a FIFO queue. When the CPU is free it is allocated to the process at head of the queue. The running process is then removed from the queue. When a new process enters, it is added to the tail. FCFS scheduling algorithm uses Nonpremptive Scheduling. Once the CPU has been allocated to a process, that process keeps the CPU until it releases the CPU, either by terminating or by requesting I/O.

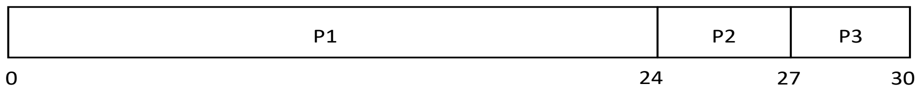
Characteristics:

- The FCFS scheduling algorithm is simple to write and understand
- The FCFS scheduling is non-preemptive
- Average waiting time under FCFS policy is not minimal but vary substantially if the process CPU-Burst times vary greatly.

For e.g. consider the following set of processes that arrive at time 0, with the length of the CPU-Burst time given in milliseconds.

Process:	P1	P2	P3
Burst time:	24	3	3

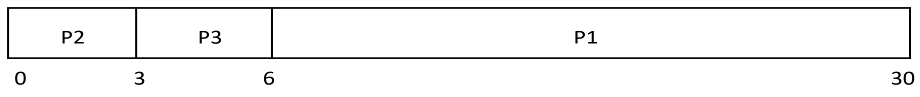
If the processes arrive in the order P1, P2, P3 and are served in FCFS order, we get the result shown in the following Gantt chart-



Average waiting time = (0+ 24 + 27) / 3 = 17

Average turnaround time = ((24) + (24+3) + (24+3+3))/3 = 27

If the arrival order is P2, P3, P1 then the following is the Gantt chart.



Average waiting time = (6 + 0 + 3) / 3 = 3

Average turnaround time = ((6+24) + (3) + (3+3) +)/3 = 13

Thus, the average waiting time under a FCFS policy is generally not minimal, and may vary substantially if the process CPU-burst times vary greatly.

Advantages

- Simple, low overhead, minimal interference to processes

Disadvantages

- Low utilization, difficult to match CPU burst of processes with I/Os
- Difficult to guarantee response time.
- It is troublesome for time-sharing systems.
- Not efficient when measured in terms of average wait time.
- In dynamic situation, it results in lower CPU and device utilization if long process is received first. Then shorter processes may need to wait a long time. This is called Convoy effect where all other processes must wait for one long process to finish.

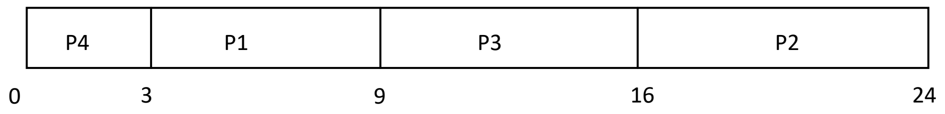
2. Shortest-Job-First scheduling:

This algorithm associates with each process the length of the latter's next CPU burst. When the CPU is available, it is assigned to the process that has the smallest next CPU burst. FCFS scheduling is used to break the tie if two processes have the same length next CPU burst. It is also called as shortest next CPU burst, because the scheduling is done by examining the length of the next CPU-burst of a process. This algorithm may be either preemptive or non-preemptive. The choice arises when a new process arrives at the ready queue while a previous process is executing.

Consider the following set of processes, with the length of the CPU burst time given in milliseconds.

Process:	P1	P2	P3	P4
Burst time:	6	8	7	3

Let the processing order be P4, P1, P3, P2



Average waiting time = (3 + 16 + 9 + 0) / 4 = 7 ms

Average turnaround time = ((3+6) + (16+8) + (9+7) + (0+3))/4 = 13

If we were using FCFS scheduling the average waiting time would be 10.25 ms.

The SJF scheduling algorithm is a special case of the general priority-scheduling algorithm. An SJF algorithm is simply a priority algorithm where the priority (p) is the inverse of the next CPU burst. The larger the CPU burst, the lower the priority and vice versa. Preemptive SJF scheduling is sometimes called shortest-remaining-time-first scheduling. A preemptive SJF will preempt the currently executing process, whereas the nonpreemptive SJF will allow the currently running process to finish its CPU burst.

Advantages

- Reduces average wait time and number of waiting processes

- Gives the minimum average waiting time for a set of processes

Disadvantages

- Difficult to determine "shortest" for the next burst
- Possible indefinite postponement of long processes, bias towards short processes
- Waiting time is even more unpredictable than FCFS
- Indefinite postponement

3. Priority scheduling:

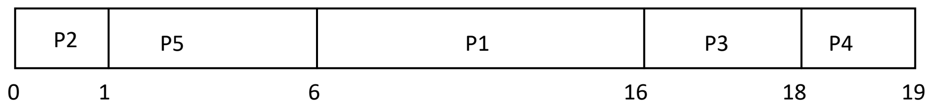
In this type of algorithm a priority is associated with each process and the CPU is allocated to the process with the highest priority. Equal priority processes are scheduled in FCFS order.

Priorities are generally expressed in some fixed range of numbers like 0 to 7 or 0 to 4095. Some systems use low numbers to represent low priority; others use low numbers for high priority. Priorities can be defined either internally or externally. Internally defined priorities use some measurable quantity to compute the priority of process. External priorities are set by criteria that are external to the operating system.

Priority scheduling can be either preemptive or non-preemptive. When a process arrives at the ready queue its priority is compared with the priority of the currently running process. A preemptive priority-scheduling algorithm will preempt the CPU if the priority of the newly arrived process is higher than the priority of the currently running process. A non-preemptive priority-scheduling algorithm will simply put the new process at the end of the ready queue.

Consider the following set of processes, assumed to have arrived at time 0, in the order of P1, P2, ..., P5, with the length of the CPU burst time given in milliseconds.

Process:	P1	P2	P3	P4	P5
Burst time:	10	1	2	1	5
Priority	3	1	3	4	2



The average waiting time is = (6+0+16+18+1) = 41/5= 8.2 ms

The average turnaround time is = ((6+10) + (0+1)+ (16+2) + (18+1) + (1+5) = 60/5 = 12 ms

The major problem with priority scheduling algorithms is indefinite blocking or starvation. A process that is ready to run but lacking the CPU can be considered blocked, waiting for the CPU. A priority scheduling algorithm can leave some low-priority processes waiting indefinitely for the CPU. In a heavily loaded computer system, a steady stream of higher-priority processes can prevent a low-priority processes waiting indefinitely for the CPU.

Advantages

- Can tailor to site specific attributes - giving priority is flexible.

Disadvantages

- Indefinite postponement due to late arriving process with high priority. Include dynamic priority scheme to overcome this. Technique is called aging. This increases the priority of the processes proportional to the waiting time.

4. Round robin scheduling:

This scheduling algorithm is designed especially for time-sharing systems. Here a small unit of time, called a time quantum or time slice is defined. A time quantum is generally from 10 to 100 milliseconds. The CPU scheduler moves around the ready queue, allocating the CPU to each process for a time interval up to one time quantum.

In this algorithm the ready queue is implemented as FIFO queue. New processes are added to the tail of ready queue. The CPU scheduler picks the first process from the ready queue, sets a timer to interrupt after one time quantum and dispatches the process. After this any of the following things will happen:

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

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

Characteristics:

1. The average waiting time under round robin is long.

2. The turnaround time depends on the size of the time quantum.

3. If there are n processes in the ready queue and the time quantum is q, then each process gets 1/n of the CPU time in chunks of at most q time units. Each process must wait no longer than (n-1) x q time units until its next time quantum.
4. If the time quantum is very large (infinite), the RR policy is same as FCFS, where as if the time quantum is very small, the RR approach is called processor sharing. This appears to the users as though each of n processes has its own processor running at 1/n the speed of the real processor.
5. The quantum size should be large compared to context switching time. Also it should be large enough such that 80% CPU burst are within 1 quantum.

Advantages

- Gives every one a share, better response time, no bias to a particular type of process.
- Can prevent infinite loop process to tie up the whole system.

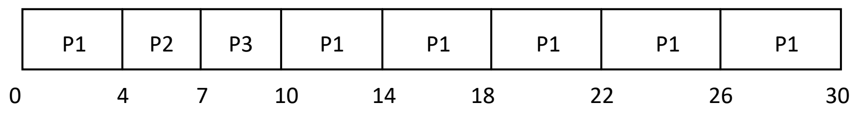
Disadvantages

- Overhead in performing preemption.
- A single fixed quantum may not fit different types of processing.

Consider the following set of processes that arrive at time 0, with the length of the CPU-burst time given in milliseconds.

Process	P1	P2	P3
Burst time	24	3	3

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



The average waiting time is = ((0+6)+4+7)/3 = 17/3 = 5.66 ms

The average turnaround time is = ((6+24)+(4+3)+(7+3)) = 47/3 = 15.67 ms

In the RR scheduling algorithm, no process is allocated the CPU for more than 1 time quantum in a row. If a CPU burst exceeds 1 time quantum, that process is preempted and is put back in the ready queue. The RR scheduling algorithm is preemptive. The performance of RR algorithm depends heavily on the size of the time quantum. If the time quantum is very large, the RR policy is the same as the FCFS policy. If the time quantum is very small, the RR approach is called processor sharing and appears to the users as though each of n processes has its own processor running at 1/n the speed of the real processor.

5. Multilevel Queue Scheduling:

Another class of scheduling algorithms has been created for situations in which the processes are easily classifieds into different groups. A division is made between foreground (or interactive) processes and background (or batch) processes. These two types of processes have different response-time requirements.

A multilevel queue-scheduling, partitions the ready queue into several separate queues.

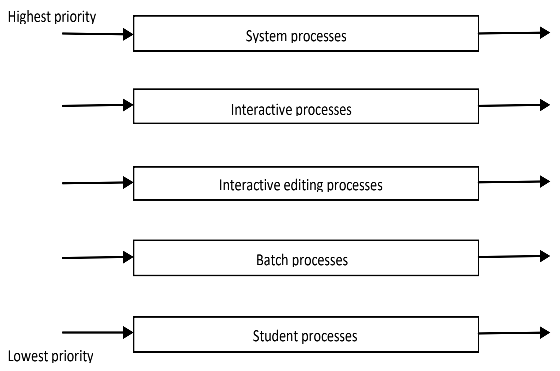


Figure: 2.15 Multilevel queue scheduling

The processes are permanently assigned to one queue, 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 is scheduled by an RR algorithm, while the background queue is scheduled by an FCFS algorithm. Figure 2.15 is an example of a multilevel queue-scheduling algorithm with five queues - System processes, Interactive processes, Interactive editing processes, Batch processes and Student processes

Each queue has absolute priority over lower-priority queues. No process in the batch queue, could run unless the queues for system processes, interactive processes and interactive editing processes were all empty. If an interactive editing process entered the ready queue while a batch process was running, the batch process would be preempted. Another possibility is to time slice between the queues. Each queue gets a certain portion of the CPU time, which it can then schedule among the various processes in its queue. For instance, the foreground queue can be given 80% of the CPU time for RR scheduling among its processes, whereas the background queue receives 20% of the CPU to give to its processes in a PCPS manner.

6. Multilevel Feedback Queue Scheduling:

In Multilevel queue-scheduling algorithm, processes are permanently assigned to a queue on entry to the system. Processes do not move between queues. If there are separate queues for foreground and background processes, processes do not move from one queue to the other, since processes do not change their foreground or background nature. Advantage of this is low scheduling overhead, but the disadvantage it is inflexible.

Multilevel feedback queue scheduling, allows a process to move between queues. The idea is to separate processes with different CPU-burst characteristics. If a process uses too much CPU time it will be moved to a lower-priority queue. This scheme leaves I/O bound and interactive processes in the higher-priority queues. A process that waits too long in a lower-priority queue may be moved to a higher-priority queue. This form of aging prevents starvation.

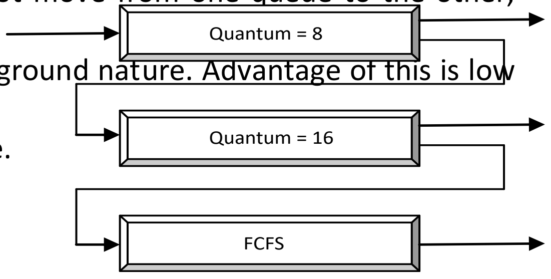


Figure 2.16 Multilevel feedback queues

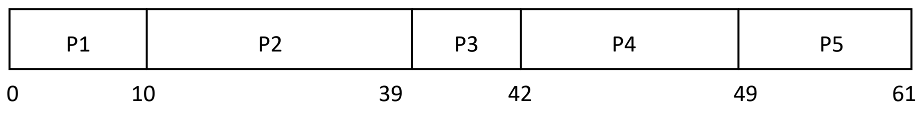
For example, consider a multilevel feedback queue scheduler with three queues, numbered from 0 to 2. The scheduler first executes all processes in queue 0. Only when queue 0 is empty, it will execute processes in queue 1. Similarly, processes in queue 2 will be executed only if queues 0 and 1 are empty.

A process entering the ready queue is put in queue 0. A process in queue 0 is given a time quantum of 8 milliseconds. If it does not finish within this time, it is moved to the tail of queue 1. If queue 0 is empty, the process at the head of queue 1 is given a quantum of 16 milliseconds. If it does not complete, it is preempted and is put into queue 2. Processes in queue 2 are run on an PCPS basis, only when queues 0 and 1 are empty. In general, a multilevel feedback queue scheduler is defined by the following parameters - The number of queues, the scheduling algorithm for each queue, the method used to determine when to upgrade a process to a higher-priority queue, the method used to determine when to demote a process to a lower-priority queue and the method used to determine which queue a process will enter, when that process needs service

For Eg., all five processes arrive at time 0, in the order given, with the length of the CPU-burst time given in milliseconds :

Process	P1	P2	P3	P4	P5
Burst time	10	29	3	7	12

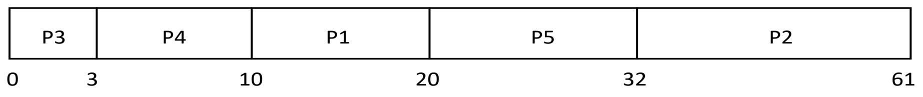
For **FCFS algorithm**, we would execute the processes as



Average waiting time is (0 + 10 + 39 + 42 + 49)/5 = 28 ms.

Average turnaround time is ((0+10)+(10+29)+(39+3)+(42+7)+(49+12)) = 201/5 = 40.2 ms

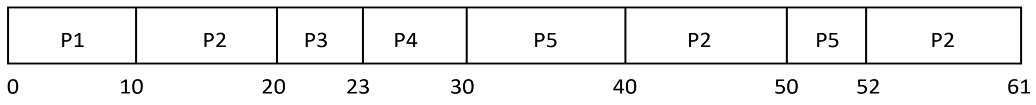
With **nonpreemptive SJF scheduling**, we execute the processes as



Average waiting time is (10 + 32 + 0 + 3 + 20)/5 = 13 ms.

Average turnaround time is ((10+10)+(32+29)+(0+3)+(3+7)+(20+12)) = 126/5 = 25.2 ms

With the **RR algorithm**, we start process P2, but preempt it after 10 ms., i.e., quantum = 10 ms.



Average waiting time is (0 + 32 + 20 + 23 + 40) = 115/5 = 23 ms.

Average turnaround time is ((0+10) + (32+29) + (20+3) + (23+7) + (40+12) = 176/5 = 35.2 ms.

SJF policy results in less than one-half the average waiting time obtained with FCF5 scheduling; the RR algorithm gives an intermediate value.

3.5 DEADLOCKS

3.5.1 Deadlocks

In a multiprogramming environment, several processes may compete for a finite number of resources. A process requests resources; if the resources are not available at that time, the process enters a wait state. Waiting processes may never again change state, because the resources they have requested are held by other waiting processes. This situation is called a deadlock. Eg., "When two trains approach each other at a crossing, both shall come to a full stop and neither shall start up again until the other has gone."

3.5.1.1. System Model

A system consists of a finite number of resources to be distributed among a number of competing processes. The resources are partitioned into several types, each of which consists of some number of identical instances. Memory space, CPU cycles, files, and I/O devices (such as printers and tape drives) are examples of resource types. If a system has two CPUs, then the resource type CPU has two instances. Similarly, the resource type printer may have five instances.

A process must request a resource before using it, and must release the resource after using it. The number of resources requested may not exceed the total number of resources available in the system. In other words, a process cannot request three printers if the system has only two. Under the normal mode of operation, a process may utilize a resource in only the following sequence:

1. **Request:** If the request cannot be granted immediately (for example, the resource is being used by another process), then the requesting process must wait until it can acquire the resource.
2. **Use:** The process can operate on the resource (for example, if the resource is a printer, the process can print on the printer).

3. **Release:** The process releases the resource.

A set of processes is in a deadlock state when every process in the set is waiting for an event that can be caused only by another process in the set. The events with which we are mainly concerned here are resource acquisition and release. The resources may be either physical resources (for example, printers, tape drives, memory space, and CPU cycles) or logical resources (for example, files, semaphores, and monitors).

Example of a Deadlock:

To illustrate a deadlock state, we consider a system with three tape drives. Suppose each of three processes holds one of these tape drives. If each process now requests another tape drive, the three processes will be in a deadlock state. Each is waiting for the event "tape drive is released," which can be caused only by one of the other waiting processes. This example illustrates a deadlock involving the same resource type.

Deadlocks may also involve different resource types. For example, consider a system with one printer and one tape drive. Suppose that process Pi is holding the tape drive and process Pj is holding the printer. If Pi requests the printer and Pj requests the tape drive, a deadlock occurs.

3.5.1.2 Deadlock Characterization

In a deadlock, processes never finish executing and system resources are tied up, preventing other jobs from starting. The following are the features that characterize deadlocks.

Necessary Conditions

A deadlock situation can arise if the following four conditions hold simultaneously in a system:

1. **Mutual exclusion**: At least one resource must be held in a non-sharable mode; that is, only one process at a time can use the resource. If another process requests that resource, the requesting process must be delayed until the resource has been released.
2. **Hold and wait**: A process must be holding at least one resource and waiting to acquire additional resources that are currently being held by other processes.
3. **No preemption**: Resources cannot be preempted; that is, a resource can be released only voluntarily by the process holding it, after that process has completed its task.
4. **Circular wait**: A set {Po, P_1, ..., Pn,) of waiting processes must exist such that Po is waiting for a resource that is held by P_1, P_1 is waiting for a resource that is held by P_2, ..., P_{n-1} is waiting for a resource that is held by Pn and Pn is waiting for a resource that is held by Po.

All four conditions must hold for a deadlock to occur. The circular-wait condition implies the hold-and-wait condition, so the four conditions are not completely independent.

3.5.1.3. Resource-Allocation Graph

Deadlocks can be described in terms of a directed graph called a system resource-allocation graph. This graph consists of a set of vertices V and a set of edges E. The set of vertices V is partitioned into two different types of nodes P = {P_1, P_2, ..., P_n}, the set consisting of all the active processes in the system, and R = {R_1, R_2, ..., R_m), the set consisting of all resource types in the system.

A directed edge from process P_i to resource type R_j is denoted by $P_i \rightarrow R_j$; it signifies that process P_i requested an instance of resource type R_j and is currently waiting for that resource. A directed edge from resource type R_j to process P_i is denoted by $R_j \rightarrow P_i$; it signifies that an instance of resource type R_j has been allocated to process P_i. A directed edge $P_i \rightarrow R_j$ is called a request edge; a directed edge $R_j \rightarrow P_i$ is called an assignment edge.

Pictorially, we represent each process P_i as a circle, and each resource type R_j as a square. Since resource type R_j may have more than one instance, we represent each such instance as a dot within the square. Request edge points to only the square R_j, whereas an assignment edge must also designate one of the dots in the square.

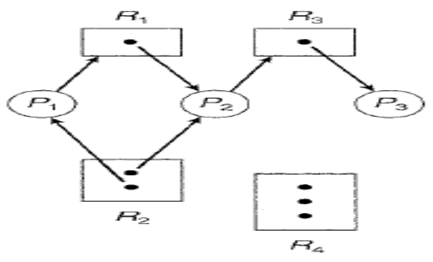


Figure 3.9 Resource-allocation graph

When process Pi requests an instance of resource type R_j, a request edge is inserted in the resource-allocation graph. When this request can be fulfilled, the request edge is instantaneously transformed to an assignment edge. When the process no longer needs access to the resource it releases the resource, and as a result the assignment edge is deleted. The resource-allocation graph shown in Figure 3.9 depicts the following situation.

The sets *P, R* and *E:*

P = {P1, P2, P3}

R = {R1, R2, R3, R4}

E = {Pl→ Rl, P2→R3, R1→P2, R2→P2, R2→P1, R3→ P3}

Resource instances:

One instance of resource type R1

Two instances of resource type R2

One instance of resource type R3

Three instances of resource type R4

Process states:

- Process PI is holding an instance of resource type R2, and is waiting for an instance of resource type R1.
- Process P2 is holding an instance of R1 and R2, and is waiting for an instance of resource type R3.
- Process P3 is holding an instance of R3.

From resource-allocation graph, it can be shown that, if the graph contains no cycles, then no process in the system is deadlocked. If the graph does contain a cycle, then a deadlock may exist. If each resource type has exactly one instance, then a cycle implies that a deadlock has occurred. In this case, a cycle in the graph is both a necessary and a sufficient condition for the existence of deadlock. If each resource type has several instances, then a cycle does not necessarily imply that a deadlock has occurred. In this case, a cycle in the graph is a necessary but not a sufficient condition for the existence of deadlock.

To illustrate this concept, we will take the resource-allocation of figure 3.8. Suppose that process P3 requests an instance of resource type R2. Since no resource instance is currently available, a request edge P3 →R2 is added to the graph (Figure 3.10). At this point, two minimal cycles exist in the system:

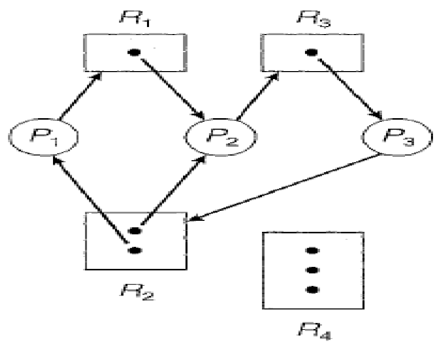


Figure 3.10 Resource-allocation graph with a deadlock

P1 → R 1 → P2 → R3 → P3 → R2 → P1

P2 → R3 → P3 → R2 → P2

Processes P1, P2, and P3 are deadlocked. Process P2 is waiting for the resource R3, which is held by process P3. Process P3, on the other hand, is waiting for either process PI or process P2 to release resource R2. In addition, process P1 is waiting for process P2 to release resource R1.

In summary, if a resource-allocation graph does not have a cycle, then the system is not in a deadlock state. On the other hand, if there is a cycle, then the system may or may not be in a deadlock state. This observation is important when we deal with the deadlock problem.

3.5.2 Methods for Handling Deadlocks

Principally, we can deal with the deadlock problem in one of three ways:

- We can use a protocol to prevent or avoid deadlocks, ensuring that the system will never enter a deadlock state.
- We can allow the system to enter a deadlock state, detect it, and recover.
- We can ignore the problem altogether, and pretend that deadlocks never occur in the system.

To ensure that deadlocks never occur, the system can use either a deadlock prevention or a deadlock-avoidance scheme. **Deadlock prevention** is a set of methods for ensuring that at least one of the necessary conditions cannot hold. These methods prevent deadlocks by constraining how requests for resources can be made. **Deadlock avoidance**, requires that the operating system be given in advance additional information concerning which resources a process will request and use during its lifetime. With this additional knowledge, we can decide for each request whether or not the process should wait. To decide whether the current request can be satisfied or must be delayed, the system must consider the resources currently available, the resources currently allocated to each process, and the future requests and releases of each process. If a system does not employ either a deadlock-prevention or a deadlockavoidance algorithm, then a deadlock situation may occur. In this environment, the system can provide an algorithm that examines the state of the system to determine whether a deadlock has occurred, and an algorithm to recover from the deadlock. If a system does not ensure that a deadlock will never occur, and also does not provide a mechanism for deadlock detection and recovery, then we may arrive at a situation where the system is in a deadlock state.

3.5.2.1 Deadlock Prevention

For a deadlock to occur, each of the four necessary conditions must hold. By ensuring that at least one of these conditions cannot hold, we can prevent the occurrence of a deadlock.

a) Mutual Exclusion

The mutual-exclusion condition must hold for nonsharable resources. For example, a printer cannot be simultaneously shared by several processes. Sharable resources, on the other hand, do not require mutually exclusive access, and thus cannot be involved in a deadlock. Read-only files are a good example of a sharable resource. If several processes attempt to open a read-only file at the same time, they can be granted simultaneous access to the file. A process never needs to wait for a sharable resource. In general, however, we cannot prevent deadlocks by denying the mutual-exclusion condition: Some resources are intrinsically nonsharable.

b) Hold and Wait

To ensure that the hold-and-wait condition never occurs in the system, we must guarantee that, whenever a process requests a resource, it does not hold any other resources.

One protocol that can be used requires each process to request and be allocated all its resources before it begins execution.

An alternative protocol allows a process to request resources only when the process has none. A process may request some resources and use them. Before it can request any additional resources, it must release all the resources that it is currently allocated.

To illustrate the difference between these two protocols, we consider a process that copies data from a tape drive to a disk file, sorts the disk file, and then prints the results to a printer. If all resources must be requested at the beginning of the process, then the process must initially request the tape drive, disk file, and printer. It will hold the printer for its entire execution, even though it needs the printer only at the end.

The second method allows the process to request initially only the tape drive and disk file. It copies from the tape drive to the disk, then releases both the tape drive and the disk file. The process must then again request the disk file and the printer. After copying the disk file to the printer, it releases these two resources and terminates.

These protocols have two main disadvantages. First, resource utilization may be low, since many of the resources may be allocated but unused for a long period. Second, starvation is possible. A process that needs several popular resources may have to wait indefinitely, because at least one of the resources that it needs is always allocated to some other process.

c) No Preemption

The third necessary condition is that there should be no preemption of resources that have already been allocated. To ensure that this condition does not hold, we can use the following protocol. If a process is holding some resources and requests another resource that cannot be immediately allocated to it (that is, the process must wait), then all resources currently being held are preempted. The preempted resources are added to the list of resources for which the process is waiting.

Alternatively, if a process requests some resources, we first check whether they are available. If they are, we allocate them. If they are not available, we check whether they are allocated to some other process that is waiting for additional resources. If so, we preempt the desired resources from the waiting process and allocate them to the requesting process.

If the resources are not either available or held by a waiting process, the requesting process must wait. While it is waiting, some of its resources may be preempted, but only if another process requests them. This protocol is often applied to resources whose state can be easily saved and restored later, such as CPU registers and memory space. It cannot generally be applied to such resources as printers and tape drives.

d) Circular Wait

One way to ensure that circular wait condition never holds is to impose a total ordering of all resource types, and to require that each process requests resources in an increasing order of enumeration.

Let R = {R1, R2, ..., Rm,) be the set of resource types. We assign to each resource type a unique integer number, which allows us to compare two resources and to determine whether one precedes another in our ordering. A one-to-one function is defined as F: R → N, where N is the set of natural numbers. For example, if the set of resource types R includes tape drives, disk drives, and printers, then the function F might be defined as follows:

F(tape drive) = 1, F(disk drive) = 5, F(printer) = 12.

The following protocol can be used to prevent deadlocks: Each process can request resources only in an increasing order of enumeration. That is, a process can initially request any number of instances of a resource type, say Ri. After that, the process can request instances of resource type Rj if and only if F(Rj) > F(Ri). Alternatively, we can require that, whenever a process requests an instance of resource type Rj, it has released any resources Ri such that F(Ri) >= F(Rj).

If these two protocols are used, then the circular-wait condition cannot hold. Let the set of processes involved in the circular wait be {Po, P1, ..., Pn), where Pi is waiting for a resource Ri, which is held by process Pi+1. Then, since process Pi+1 is holding resource Ri while requesting resource Ri+l, we must have F(Ri) < F(Ri+1), for all i. But this condition means that F(Ro) < F(R1) < ... < F(R,) < F(Ro). By transitivity, F(Ro) < F(Ro), which is impossible. Therefore, there can be no circular wait.

3.5.2.2 Deadlock Avoidance

Deadlock-prevention algorithms, prevent deadlocks by restraining how requests can be made. The restraints ensure that at least one of the necessary conditions for deadlock cannot occur, and, hence, that deadlocks cannot hold.

An alternative method for avoiding deadlocks is to require additional information about how resources are to be requested. For example, in a system with one tape drive and one printer, we might be told that process P will request first the tape drive, and later the printer, before releasing both resources. Process Q, on the other hand, will request first the printer, and then the tape drive. With this knowledge of the complete sequence of requests and releases for each process, we can decide for each request whether or not the process should wait.

Each request requires that the system consider the resources currently available, the resources currently allocated to each process, and the future requests and releases of each process, to decide whether the current request can be satisfied or must wait to avoid a possible future deadlock.

The simplest and most useful model requires that each process declare the maximum number of resources of each type that it may need. Given a priori information about the maximum number of resources of each type that may be requested for each process, it is possible to construct an algorithm that ensures that the system will never enter a deadlock state. This algorithm defines the **deadlock-avoidance approach**. A deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that a circular wait condition can never exist.

3.5.2.2.1 Safe State

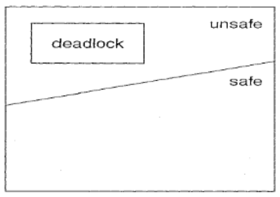


Figure 3.11 Safe, unsafe, and deadlock state spaces

A state is safe if the system can allocate resources to each process (up to its maximum) in some order and still avoid a deadlock. A system is in a safe state only if there exists a safe sequence. A sequence of processes <P1, P2, ..., Pn,> is a safe sequence for the current allocation state if, for each Pi, the resources that Pi can still request can be satisfied by the currently available resources plus the resources held by all the Pi, with j < i. If the resources that process Pi needs are not immediately available, then Pi can wait until all Pi have finished. When they have finished, Pi can obtain all of its needed resources, complete its designated task, return its allocated resources, and terminate. When Pi terminates, Pi+1 can obtain its needed resources, and so on. If no such sequence exists, then the system state is said to be unsafe.

A safe state is not a deadlock state. Conversely, a deadlock state is an unsafe state. Not all unsafe states are deadlock (Figure 3.11). An unsafe state may lead to a deadlock. As long as the state is safe, the operating system can avoid unsafe (and deadlock) states.

To illustrate, we consider a system with 12 magnetic tape drives and 3 processes: P0, P1, and P2. Process P0 requires 10 tape drives, process P1 may need as many as 4, and process P2 may need up to 9 tape drives. Suppose that, at time to, process P0 is holding 5 tape drives, process P1 is holding 2, and process P2 is holding 2 tape drives. (Thus, there are 3 free tape drives.)

	Maximum Needs	Current Needs
P0	10	5
P1	4	2
P2	9	2

At time to, the system is in a safe state. The sequence <P1, P0, P2> satisfies the safety condition, since process P1 can immediately be allocated all its tape drives and then return them (the system will then have 5 available tape drives), then process Po can get all its tape drives and return them (the system will then have 10 available tape drives), and finally process P2 could get all its tape drives and return them (the system will then have all 12 tape drives available).

A system may go from a safe state to an unsafe state. Suppose that, at time t1, process P2 requests and is allocated 1 more tape drive. The system is no longer in a safe state. At this point, only process P1 can be allocated all its tape drives. When it returns them, the system will have only 4 available tape drives. Since process Po is allocated 5 tape drives, but has a maximum of 10, it may then request 5 more tape drives. Since they are unavailable, process Po must wait. Similarly, process P2 may request an additional 6 tape drives and have to wait, resulting in a deadlock.

3.5.2.2.2 Resource-Allocation Graph Algorithm

Resource-Allocation graph algorithm can be used for deadlock avoidance. In addition to the request and assignment edges, new type of edge exists, called a claim edge. A claim edge Pi → Rj indicates that process Pi may request resource Rj at some time in the future. This edge resembles a request edge in direction, but is represented by a dashed line. When process Pi requests resource Rj, the claim edge Pi → Rj is converted to a request edge. Similarly, when a resource Rj is released by Pi, the assignment edge Rj → Pi is reconverted to a claim edge Pi →Rj. The resources must be claimed a priori in the system. That is, before process Pi starts executing, all its claim edges must already appear in the resource-allocation graph.

Suppose that process Pi requests resource Rj. The request can be granted only if converting the request edge Pi → Rj to an assignment edge Rj → Pi does not result in the formation of a cycle in the resource-allocation graph. Safety is checked by using a cycle-detection algorithm. An algorithm for detecting a cycle in this graph requires an order of n2 operations, where n is the number of processes in the system.

If no cycle exists, then the allocation of the resource will leave the system in a safe state. If a cycle is found, then the allocation will put the system in an unsafe state. Therefore, process Pi will have to wait for its requests to be satisfied.

To illustrate this algorithm, we consider the resource-allocation graph of Figure 3.12. Suppose that P2 requests R2. Although R2 is currently free, we cannot allocate it to P2, since this action will create a cycle in the graph (Figure 3.13). A cycle indicates that the system is in an unsafe state. If P1 requests R2, and P2 requests R1, then a deadlock will occur.

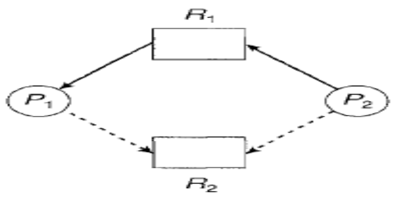


Figure 3.12 Resource allocation graph for deadlock avoidance

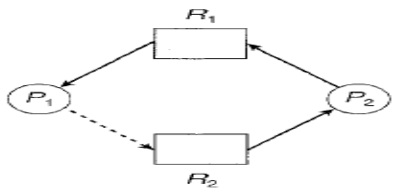


Figure 3.13 An unsafe state in a resource-allocation graph

3.5.2.2.3 Banker's Algorithm

The resource-allocation graph algorithm is not applicable to a resource allocation system with multiple instances of each resource type. The deadlock-avoidance algorithm is applicable to resource allocation system with multiple instances of each resource type, but is less efficient than the resource-allocation graph scheme. This algorithm is commonly known as the banker's algorithm. The name was chosen because this algorithm could be used in a banking system to ensure that the bank never allocates its available cash such that it can no longer satisfy the needs of all its customers.

When a new process enters the system, it must declare the maximum number of instances of each resource type that it may need. This number may not exceed the total number of resources in the system. When a user requests a set of resources, the system must determine whether the allocation of these resources will leave the system in a safe state. If it will, the resources are allocated; otherwise, the process must wait until some other process releases enough resources.

Several data structures must be maintained to implement the banker's algorithm. These data structures encode the state of the resource-allocation system. Let n be the number of processes in the system and m be the number of resource types. We need the following data structures:

Available: A vector of length m indicates the number of available resources of each type. If Available[j] = k, there are k instances of resource type Rj available.

Max: An n x m matrix defines the maximum demand of each process. If Max[i,j] = k, then process Pi may request at most k instances of resource type Ri.

Allocation: An n x m matrix defines the number of resources of each type currently allocated to each process. If Allocation[i,j] = k, then process Pi is currently allocated k instances of resource type Rj.

Need: An n x m matrix indicates the remaining resource need of each process. If Need[i,j] = k, then process Pi may need k more instances of resource type Ri to complete its task.

Need[i,j] = Max[i,j] - Allocation[i,j].

Safety Algorithm

The algorithm for finding out whether or not a system is in a safe state can be described as follows:

1. Let Work and Finish be vectors of length m and n, respectively. Initialize Work := Available and Finish[i] :=false for i = 1,2, ..., n.
2. Find an i such that both
   a. Finish[i] =false

b. Needi 5 Work.

If no such i exists, go to step 4.

3. Work := Work + Allocation$_i$

   Finish[i] := true

   go to step 2.

4. If Finish[i] = true for all i, then the system is in a safe state.

   This algorithm may require an order of m x n^2 operations to decide whether a state is safe.

Resource-Request Algorithm

Let Request$_i$ be the request vector for process Pi. If Request$_i$ [j] = k, then process Pi wants k instances of resource type Rj. When a request for resources is made by process Pi, the following actions are taken:

1. If Request$_i$ <= Needi, go to step 2. Otherwise, raise an error condition, since the process has exceeded its maximum claim.
2. If Request$_i$ <= Available, go to step 3. Otherwise, Pi must wait, since the resources are not available.
3. Modifing the state of Pi as follows:

   Available := Available - Request;

   Allocationi := Allocation; + Request;

   Needi := Needi - Request$_i$;

If the resulting resource-allocation state is safe, the transaction is completed and process Pi is allocated its resources. However, if the new state is unsafe, then Pi must wait for $Request_i$ and the old resource-allocation state is restored.

An Illustrative Example

Consider a system with five processes P0 through P4 and three resource types A, B, C. Resource type A has 10 instances, resource type B has 5 instances, and resource type C has 7 instances. Suppose that, at time to, the following snapshot of the system has been taken:

	Allocation	*Ma x*	*Need*	*Available*
A B C	*A B C*	*A B C*	*A B C*	*A B C*
Po	0 1 0	7 5 3	7 4 3	3 3 2
P1	2 0 0	3 2 2	1 2 2	
P2	3 0 2	9 0 2	6 0 0	
P3	2 1 1	2 2 2	0 1 1	
P4	0 0 2	4 3 3	4 3 1	

The system is currently in a safe state. The sequence <P1, P3, P4, P2, P0> satisfies the safety criteria.

3.5.3 Deadlock Detection

If a system does not employ either a deadlock-prevention or a deadlock avoidance algorithm, then a deadlock situation may occur. In this environment, the system must provide:

- An algorithm that examines the state of the system to determine whether a deadlock has occurred

- An algorithm to recover from the deadlock

3.5.3.1 Single Instance of Each Resource Type

If all resources have only a single instance, then we can define a deadlock detection algorithm that uses a variant of the resource-allocation graph, called a wait-for graph. Wait-for graph is obtained from the resource-allocation graph by removing the nodes of type resource and collapsing the appropriate edges. An edge from Pi to Pj in a wait-for graph implies that process Pi is waiting for process Pj to release a resource that Pi needs. An edge Pi → Pj exists in a wait-for graph if and only if the corresponding resource allocation graph contains two edges Pi → Rq and Rq → Pj for some resource Rq. For example, in Figure 3.14, we present a resource-allocation graph and the corresponding wait-for graph.

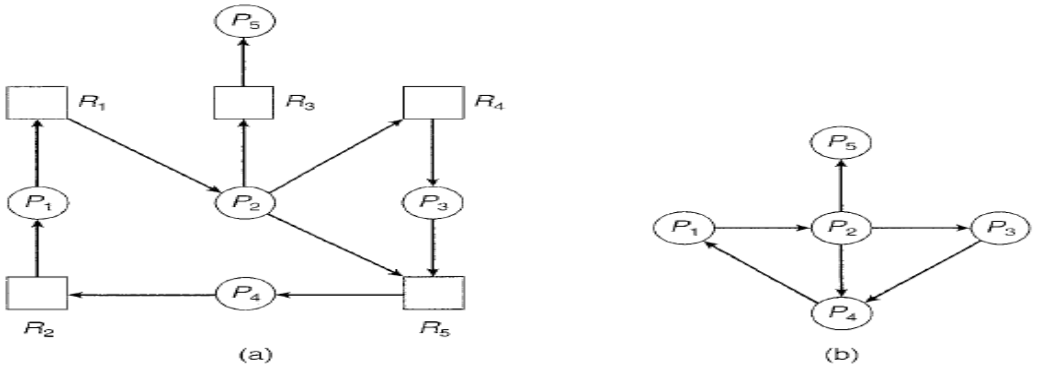


Fig 3.14 (a) Resource-allocation graph **(b) Corresponding wait-for graph**

A deadlock exists in the system if and only if the wait-for graph contains a cycle. To detect deadlocks, the system needs to maintain the wait-for graph and periodically to invoke an algorithm that searches for a cycle in the graph. An algorithm to detect a cycle in a graph requires an order of n^2 operations, where n is the number of vertices in the graph.

3.5.3.2 Several Instances of a Resource Type

The wait-for graph scheme is not applicable to a resource-allocation system with multiple instances of each resource type. The deadlock-detection algorithm is applicable to resource-allocation system with multiple instances of each resource type. The algorithm employs several time-varying data structures that are similar to those used in the banker's algorithm:

Available: A vector of length m indicates the number of available resources of each type.

Allocation: An n x m matrix defines the number of resources of each type currently allocated to each process.

Request: An n x m matrix indicates the current request of each process. If Request[i,j] = k, then process Pi is requesting k more instances of resource type Rj.

The detection algorithm simply investigates every possible allocation sequence for the processes that remain to be completed.

1. Let Work and Finish be vectors of length m and n, respectively. Initialize Work := Available. For i = 1, 2, ..., n, if Allocationi ≠0, then Finish[i] :=false; otherwise, Finish[i] := true.
2. Find an index i such that both
   a. Finish[i] =false.
   b. Requesti ≤ Work.
   If no such i exists, go to step 4.
3. Work := Work + Allocation$_i$
   Finish[i] := true

go to step 2.

4. If Finish[i] = false, for some i, 1 ≤ i ≤ n, then the system is in a deadlock state.

   If Finish[i] = false, then process Pi is deadlocked.

This algorithm requires an order of m x n2 operations to detect whether the system is in a deadlocked state.

To illustrate this algorithm, we consider a system with five processes Po through P4 and three resource types A, B, C. Resource type A has 7 instances, resource type B has 2 instances, and resource type C has 6 instances. Suppose that, at time To, we have the following resource-allocation state:

	Allocation	*Request*	*Available*
	A B C	*A B C*	*A B C*
Po	0 1 0	0 0 0	0 0 0
P1	2 0 0	2 0 2	
P2	3 0 3	0 0 0	
P3	2 1 1	1 0 0	
P4	0 0 2	0 *0* 2	

We claim that the system is not in a deadlocked state. Indeed, if we execute our algorithm, we will find that the sequence <Po, P2, P3, P1, P4> will result in Finish[i] = true for all i.

Suppose now that process P2 makes one additional request for an instance of type C. The Request matrix is modified as follows:

Request

	A B C
Po	0 0 0
P1	2 0 2
P2	0 0 1
P3	1 0 0
P4	0 0 2

We claim that the system is now deadlocked. Although we can reclaim the resources held by process Po, the number of available resources is not sufficient to fulfill the requests of the other processes. Thus, a deadlock exists, consisting of processes PI, P2, P3, and P4.

3.5.4 Recovery from Deadlock

When a detection algorithm determines that a deadlock exists, the following options are used for breaking a deadlock.

- One solution is simply to abort one or more processes to break the circular wait.
- The second option is to preempt some resources from one or more of the deadlocked processes.

3.5.4.1 Process Termination

To eliminate deadlocks by aborting a process, we use one of two methods. In both methods, the system reclaims all resources allocated to the terminated processes.

Abort all deadlocked processes: This method will break the deadlock cycle, but at a great expense; these processes may have computed for a long time, and the results of these partial computations must be discarded and probably recomputed later.

Abort one process at a time until the deadlock cycle is eliminated: This method incurs considerable overhead, since, after each process is aborted, a deadlock-detection algorithm must be invoked to determine whether any processes are still deadlocked.

Aborting a process may not be easy. If the process was in the midst of updating a file, terminating it will leave that file in an incorrect state. Similarly, if the process was in the midst of printing data on the printer, the system must reset the printer to a correct state before printing the next job. Also we must determine which process (or processes) should be terminated in an attempt to break the deadlock.

3.5.4.2 Resource Preemption

To eliminate deadlocks using resource preemption, the resources are successively preempted from processes and give these resources to other processes until the deadlock cycle is broken.

If preemption is required to deal with deadlocks, then three issues need to be addressed:

1. Selecting a victim: Which resources and which processes are to be preempted?

The order of preemption has to be determined to minimize cost. Cost factors may include such parameters as the number of resources a deadlock process is holding, and the amount of time a deadlocked process has consumed during its execution.

2. Rollback: If resources are preempted from a process, it cannot continue with its normal execution; as the resources are missing. The process has to be rolled back to some safe state, and restart it from that state. It is more effective to roll back the process only as far as necessary to break the deadlock.

3. Starvation: In a system where victim selection is based primarily on cost factors, it may happen that the same process is always picked as a victim. As a result, this process never completes its designated task. Such a situation is called starvation. We must ensure that a process can be picked as a victim only a (small) finite number of times.