Bachelor of Computer Application Part-III

Paper : BCA-304

Operating System

Unit No. 1

Unit - I

Lesson Nos :

Department of Distance Education
Punjabi University, Patiala

Computers and Software

Contents

1.1. Objectives

In this lesson, we will understand systems software, their distinction from application software, resources, resource abstraction and resource sharing. It further defines an operating system and its goals. It further discusses the batch processing environment and the role of an operating system in such environment. Students will be explained about the exact role of an operating system through illustrated diagrams.

1.1.1 General Systems Software

Software is generally divided into systems software and applications software.

Applications software comprises programs designed for an end user, such as word processors, database systems and spreadsheet programs.

System software is any computer software that manages and controls computer hardware so that application software can perform a task. Specific kinds of system software include Operating systems, device drivers, programming tools, compilers, assemblers, linkers and loaders. System software can be classified as operating system and language processors.

Operating system creates an interface between user and the system hardware.

Operating System is the software that communicates with the hardware and allows other programs to run. Language processors are those which help to convert computer language (Assembly and high level Languages) to machine level language. Assemblers, compilers and interpreters are examples of language processors.

System software covers a broad spectrum of functionality. An important class of system software is the run time system for a programming language. In the UNIX system software, important parts of this runtime functionality are implemented in C libraries. Other examples of system software are a window system and database management system. A window system is system software that provides a virtual terminal to an application program. The window is termed "virtual" because the programmer constructs the application software using functions to read and write the window as if it were a terminal device, even though there is no physical terminal uniquely associated with the window. The system software maps these virtual terminal operations on the physical region on a screen. It then translates the software's operations on the virtual terminal to appropriate operations on the physical terminal. One physical terminal can support several virtual terminals.

A database management system is a full system that can be used to store information on the computer's permanent storage devices such as magnetic tapes and disks. The database system provides abstract data types called 'schema' and creates new application specific software optimized for efficient queries/updates on the data according to the schema definition.

Not all system software applies equally to all application domains. Some system software, such as graphics library is specific to a particular application domain and may not be useful in others.

Other system software like the relational database is very general. It can support programs written for different application domains. In the case of databases, different kinds of database management systems can be designed for many different application domains. Once a database technology is chosen to support the domain, it may be further specialized to better support a subdomain such as image processing or an artificial intelligence expert system. And even within the image processing database system software, further specialization of the system software may be appropriate so that it supports specific applications. For example, the image database may be designed to support only monochrome topographic images.

An operating system interacts directly with the hardware to provide an interface to other system software and with application software whenever it wants to use the system resources. It is largely domain dependent.

1.1.2. Resource Abstraction

Resource abstraction and sharing are two key aspects of the operating system. There are many different kinds of hardware components –referred to as resources that can be used by an application program. Disk drive, Memory, CPU cycles etc. are examples of resources.

System software hides the details of how the hardware operates, thereby making computer hardware relatively easy for an application programmer to use. It does this by providing an abstract model of the operation of the hardware components.

For example, an application programmer must be aware of the general behaviour of a disk drive, it is generally preferable to avoid learning details of disk input/output. Abstraction is the perfect approach because a previously implemented abstraction can be used to read and write the disk drive. Thus, a disk software package is an example of the system software.

In designing system software you must first define a set of abstractions that will be general across resources, yet intuitive for the programmer and suited to the target application domain. A good abstraction will be easy for the programmer to understand and use and will allow the programmer to easily perform every kind of operation on the resources required in the domain.

1.1.3. Resource Sharing

Abstract and physical resources must be shared among the programs. There are two types of sharing:

- Space-multiplexed sharing
- Time-multiplexed sharing

Space multiplexed sharing means that the resources can be divided into two or more distinct units of the resources. Different processes (programs in execution) can be allocated exclusive control of different units of a resource at the same time. Memory and disk are the examples of space –multiplexed resources.

When the process is allocated exclusive control of the entire resource for a short period of time and then another process is allocated control of it, the mechanism is known as **time-multiplexed sharing**. Processor is the resource allocated in this manner.

There are two important aspects of sharing the resources:-

-The system must be able to isolate the resource access according to the allocation policy.
-The system must be able to cooperatively share resources when that is desired.
Resource sharing should be done by authorized entities only. Unauthorized sharing should be stopped by the operating system. **Resource isolation** is done by the operating system to prevent unauthorized sharing and still allowing for the authorized sharing.

For example, operating system never allows user programs to be loaded in the main memory area marked for the operating system components. Similarly, the processor isolation mechanism forces the processes to sequentially share the system's processor. If one process is getting executed by the processor, another has to wait till the processor becomes free.

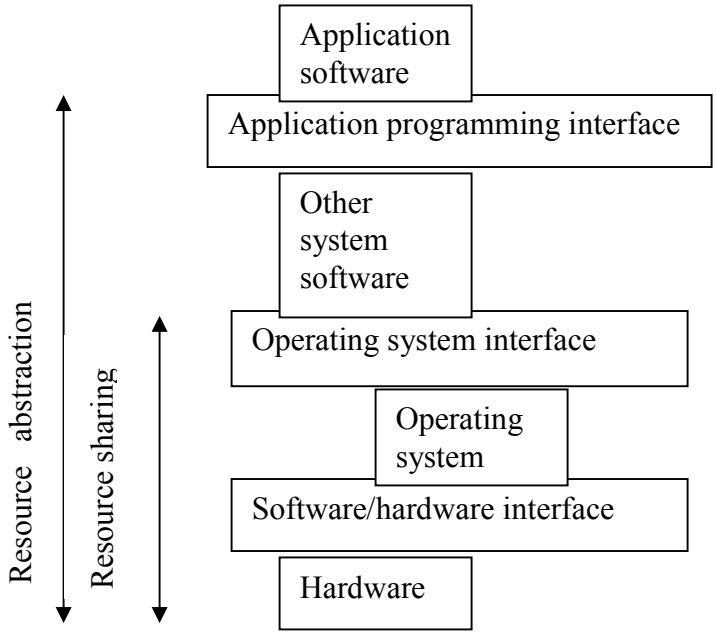


Fig 1.1.1 : System software and the operating system

The system software also must explicitly enable two or more executing applications to share resource access when that is desired. If the programmer intends for two

executing programs to share resources explicitly, the operating system must ensure that the isolation mechanism does not preclude this intended resource sharing.

There can be no guarantee of resource isolation without a guarantee that the system software correctly implements the isolation mechanism. In turn, part of the system software must be trusted to implement the resource isolation in a way that cannot be violated by other programs. This trusted part of the software is encapsulated in the operating system. Even the operating system software must depend on the hardware to implement key parts of the mechanism to ensure resource isolation. While all system software is concerned with some form of resource abstraction, operating systems implement the software part of the trusted mechanisms that manage sharing.

Figure 1.1.1 shows the distinction between system software and operating system. The system software and operating system have following differences:-

- All system software implements an abstraction of resources used by application programmers but the operating system implements the abstraction directly from the physical resources.
- The operating system provides the fundamental methods to manage resource sharing.

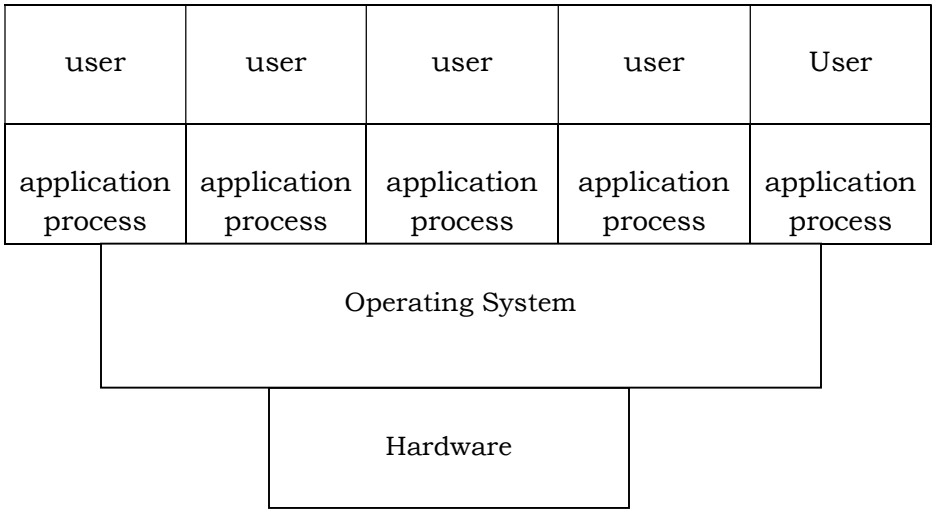


Fig 1.1.2: Position of operating system

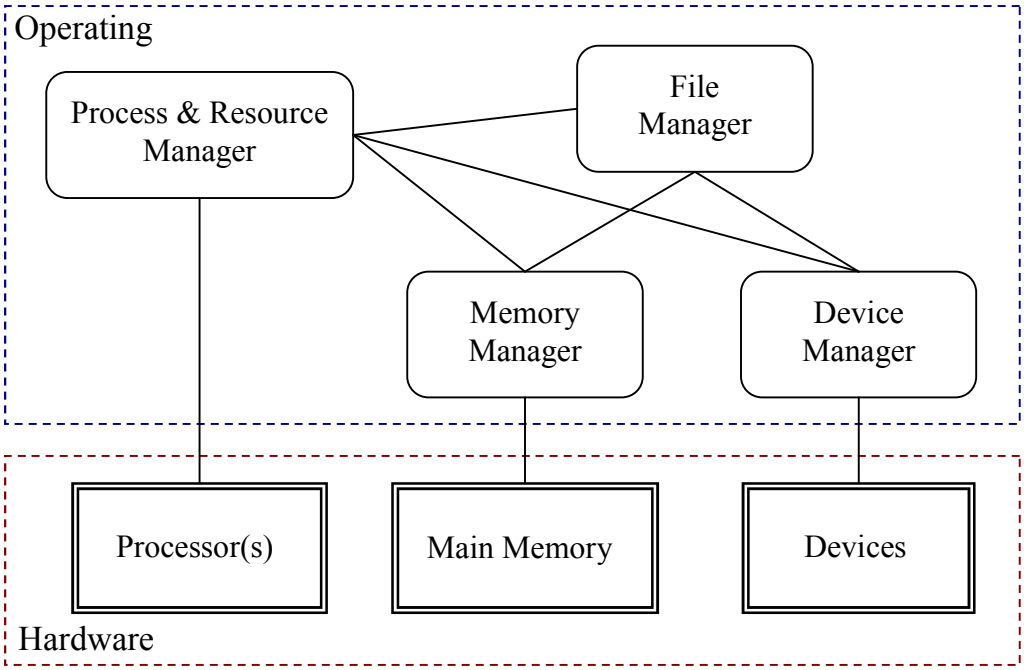


Fig 1.1.3: Goals of operating system

1.1.4. Definition and Goals of an Operating System

It has been rightly said that operating system is the first software we see when we turn on the computer and the last software we see when the computer is turned off. In the simplest words, as shown in Fig 1.1.2, it makes the access to the hardware easier.

More appropriately, we can define operating system as a program (in fact a system program) that, after being initially loaded into the computer by a boot program, manages all other programs in a computer. The other programs managed by it are usually application programs.

The application programs make use of the operating system by making requests for services through a defined Application Program Interface (API). In addition, users can interact directly with the operating system through a user interface, such as a command language or a graphical user interface.

The most common operating systems are the windows family of operating systems (Windows 95, 98, 2000, NT), the UNIX family of operating systems (which includes Linux, BSD UNIX and many other derivatives) and the Macintosh

operating system. Many other operating systems are available for special purpose applications including specializations for mainframes, robotics, manufacturing, real-time control systems and so on.

Operating system has two major goals:-

User Environment – Operating System layer transforms bare hardware machine into higher level abstractions.

It facilitates the following mechanisms for providing a good user environment:-

- Execution environment - process management, file manipulation, interrupt handling, I/O operations, language.
- Error detection and handling
- Protection and security
- Fault tolerance and failure recovery

Resource Management- an Operating system should be efficient, transparent and often feature rich. It facilitates the following mechanisms for providing an efficient resource management:-

- Time management - CPU and disk transfer scheduling
- Space management - main and secondary storage allocation
- Synchronization and deadlock handling - IPC, critical section, coordination

-Accounting and status information - resource usage tracking

Self review

1. (T/F) Operating systems manage only hardware.
2. What are the primary purposes of an operating system?

Answers

1)False.

2)Operating systems manage applications and other software abstractions, such as virtual machines The primary purposes of an operating system are to enable applications to interact with a computer's hardware and to manage a system's hardware and software resources.

1.1.4.1 Operating system components

A user interacts with the operating system via one or more user applications and often through a special application called a shell or command interpreter. Most of today's shells are implemented as text-based interfaces that enable the user to issue commands from a keyboard or as GUIs that allow the user to point and click and drag and drop icons to request services from the operating system (e.g., to open an application). For example, Microsoft Windows XP provides a GUI through which users can issue commands; alternatively, the user can open a command prompt window that accepts typed

commands. The software that contains the core components of the operating system is referred to as the **kernel**.

Typical operating system core components include:

- the **process scheduler** -which determines when and for how long a process executes on a processor**.**
- the **memory manager**- which determines when and how memory is allocated to processes and what to do when main memory becomes full.
- the **I/O manager**- which services input and output requests from and to hardware devices, respectively**.**
- the **interprocess communication (IPC) manager-** which allows processes to communicate with one another.
- the **file system manager**, which organizes named collections of data on storage devices and provides an interface for accessing data on those devices.

Almost all modern operating systems support a multiprogrammed environment in which multiple applications can execute concurrently. One of the most fundamental responsibilities of an operating system is to determine which processor executes a process and for how long that process executes. A program may contain several elements that share data and that can be executed concurrently. For example, a Web browser may contain separate components to read a Web page's HTML, retrieve the page's media (e.g., images, text and video) and render the page by laying out its content in the browser window. Such program components, which execute independently but perform their work in a common memory space are called threads.

1.1.4.2 Operating system characteristics

Users have come to expect certain characteristics of operating systems, such as:

- efficiency
- robustness
- scalability
- extensibility
- portability
- security
- interactivity
- usability

An **efficient operating system** achieves high throughput and low average turnaround time. Throughput measures the amount of work a processor can complete within a certain time period. Recall that one role of an operating system is to provide services to many applications. An efficient operating system minimizes the time spent providing these services.

1.1.4.3 Operating Systems Services

Following are the five services provided by an operating systems to the convenience of the users.

1. Program Execution : The purpose of a computer systems is to allow the user to execute programs. So the operating systems provides an environment where the user can conveniently run programs. The user does not have to worry about the memory allocation or multitasking or anything. These things are taken care of by the operating systems.

Running a program involves the allocating and deallocating memory, CPU scheduling in case of multiprocess. These functions cannot be given to the user-level programs. So user-level programs cannot help the user to run programs independently without the help from operating systems.

2. I/O Operations : Each program requires an input and produces output. This involves the use of I/O. The operating systems hides the user the details of underlying hardware for the I/O. All the user sees is that the I/O has been performed without any details. So the operating systems by providing I/O makes it convenient for the users to run programs.

For efficiently and protection users cannot control I/O so this service cannot be provided by user-level programs.

3. File System Manipulation : The output of a program may need to be written into new files or input taken from some files. The operating systems provides this service. The user does not have to worry about secondary storage management. User gives a command for reading or writing to a file and sees his her task accomplished. Thus operating systems makes it easier for user programs to accomplished their task.

This service involves secondary storage management. The speed of I/O that depends on secondary storage management is critical to the speed of many programs and hence I think it is best relegated to the operating systems to manage it than giving individual users the control of it. It is not difficult for the user-level programs to provide these services but for above mentioned reasons it is best if this service s left with operating system.

4. Communications : There are instances where processes need to communicate with each other to exchange information. It may be between processes running on the same computer or running on the different computers. By providing this service the operating system relieves the user of the worry of passing messages between processes. In case where the messages need to be passed to processes on the other computers through a network it can be done by the user programs. The user program may be customized to the specifics of the hardware through which the message transits and provides the service interface to the operating system.

5. Error Detection : An error is one part of the system may cause malfunctioning of the complete system. To avoid such a situation the operating

system constantly monitors the system for detecting the errors. This relieves the user of the worry of errors propagating to various part of the system and causing malfunctioning.

This service cannot allowed to be handled by user programs because it involves monitoring and in cases altering area of memory or deallocation of memory for a faulty process. Or may be relinquishing the CPU of a process that goes into an infinite loop. These tasks are too critical to be handed over to the user programs. A user program if given these privileges can interfere with the correct (normal) operation of the operating systems.

A **robust operating system** is fault tolerant and reliable—the system will not fail due to isolated application or hardware errors and if it fails , it does so gracefully (i.e., by minimizing loss of work and by preventing damage to the system's hardware).Such an operating system will provide services to each application unless the hardware it relies on fails.

A **scalable operating system** is able to use resources as they are added. If an operating system is not scalable, then it will quickly reach a point where additional resources will not be fully utilized. A scalable operating system can readily adjust its degree of multiprogramming. Scalability is a particularly important attribute of multiprocessor systems—as more processors are added to a system, ideally the processing capacity should increase in proportion to the number of processes, though, in practice, that does not happen.

A **portable operating system** is designed such that it can operate on many hardware configurations. Application portability is also important, because it is costly to develop applications, so the same application should run on a variety of hardware configurations to reduce development costs. The operating system is crucial to achieving this kind of portability.

A **secure operating system** prevents users and software from accessing services and resources without authorization. Protection refers to the mechanisms that implement the system's security policy.

An **interactive operating system** allows applications to respond quickly to user actions, or events.

A **usable operating system** is one that has the potential to serve a significant user base. These operating systems generally provide an easy-to-use user interface. Operating systems such as Linux, Windows XP and MacOS Xare characterized as usable operating systems, because each supports a large set of applications and provides standard user interfaces. Many experimental and academic operating systems do not support a large number of applications or provide user-friendly interfaces and therefore are not considered to be usable.

Self Review

1. Which operating system goals correspond to each of the following characteristics?
   a. Users cannot access services or information without proper authorization.
   b. The operating system runs on a variety of hardware configurations.
   c. System performance increases steadily when additional memory and processors are added.
   d. The operating system supports devices that were not available at the time of its design.
   e. Hardware failure does not necessarily cause the system to fail.
2. How does device driver support contribute to an operating system's extensibility?

Answers

1) a) security; b) portability; c) scalability; d) extensibility; e) robustness.
2) Device drivers enable developers to add support for hardware that did not exist when the operating system was designed. With each new type of device that is added to a system a corresponding device driver must be installed.

1.1.5. Batch processing

The computers in the past were very large in size and their input output devices were very different from those that are there today. There never used to be an interactive job processing as such. In fact the programmers had nothing to do with the computer's input-output operation. The programmers' responsibility was to prepare a program, data, control information etc. and to submit it to a computer operator. This computer operator's job was to submit this program (usually in the form of punch cards) to the computer, take out the result of the program, dump of memory and registers in case of errors.

The role of operating system in these days was to transfer control automatically from one job to the next. In order to use this system effectively, a batch of these programmed cards instead of the single card would be given to the computer operator. Usually, batch jobs would be stored up during working hours and then executed during off time or whenever the computer is idle.

A batch job is a predefined collection of commands that are executed without any interaction with the users.

Advantages of Batch Processing

- Batch processing is particularly useful for operations that require the computer or a peripheral device for an extended period of time and very little user interaction.

Disadvantages of Batch Processing

- No interaction is possible with the user while the program is being executed.

- Job setup time was real problem. It took a lot of time for mounting of tapes or cards.
- Processor (CPU) sat idle during the transition from one job to another.

In order to reduce the wastage of time during job transition, automatic job sequencing was developed. First operating system known as Resident monitor came into existence. Initially, resident monitor was invoked and then it would transfer control to a job. When the job terminated, it would return control to the resident monitor. Then the control will go to the next job in the memory. Thus, resident monitor would automatically sequence the execution of jobs one after the other.

As shown in Fig 1.1.4, an operator would create a job queue of many jobs. Then memory allocator would manage the primary memory space. When space would become available, a job would be selected from job queue and put into memory. Then it would compete for the processor. When the processor became available, the processor scheduler would allocate the processor to the job in the memory and it will be executed. After job completion its memory would be released and its output would be directed for printing.

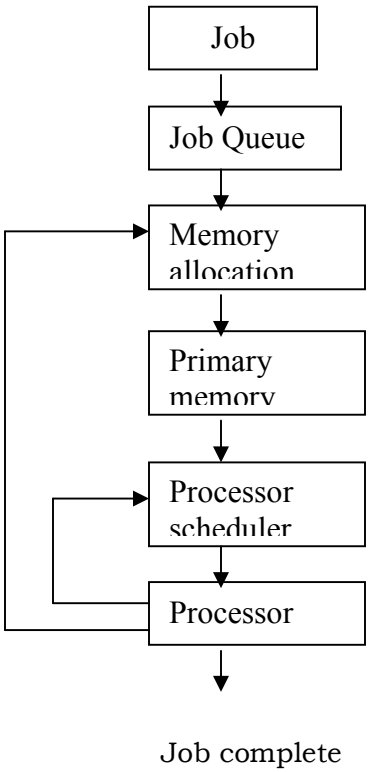


Fig 1.1.4: Batch processing

Resident monitor had many identifiable parts - Major part was control-card interpreter that was responsible for reading and carrying out the instructions on the cards at the point of execution.

But even in this set up with Resident monitor, processor (CPU) used to be mostly idle because Input-Output devices are slower as compared to the CPU.

In order to overcome the problem of speed-mismatch, the concept of SPOOLing (Simultaneous Peripheral Operation On-line) was introduced. To free the CPU for useful work, the output was sent to a magnetic tape drive, which was much faster than a printer and much cheaper than a computer. After the job was finished, the tape was removed from the tape drive attached to the computer and mounted on a tape drive connected to a printer. The printer could print the data without holding up the computer. Similarly, instead of inputting the program from the card reader, it was first copied to a tape and the computer then read the tape. This system solved the problem of speed mismatch and reduced the unnecessary hold of computer by the slow peripheral devices.

1.1.6. Keywords

System Software : a software that manages hardware of a computer system

Resources : Different kinds of hardware components like disk drive, memory and processor

Resource Abstraction : It is a mechanism that helps an application programmer to access the resources without going into much details

Resource Sharing : Sharing of resources amongst concurrently executing programs

Operating System : Resource Manager, Interface between application software and hardware

Shell : Command Interpreter

Kernel : Core component of an operating system

Batch processing systems: They are used to service a collection of jobs called a batch.

1.1.7 Summary

1. The system software and hardware are transparent to the end user (who is using application software to process information). System software enables the application programmer to use system resources with relatively less effort.
2. An operating system is a special kind of system software. It is known as resource manager also.

3. The operating system ensures that private resources are protected from unauthorized access and shared resources are accessible to all relevant processes.
4. Operating systems have evolved from batch processing systems to present sophisticated systems. The mechanism known as Spooling was used to solve the problem of CPU-device speed mismatch.

1.1.8 Short Answer Type Questions

Q.1. What is the resource manager of a computer system called?
Q.2. "Microsoft Word" is an example of application software or system software?
Q.3. Give one example of system software.
Q.4. Give two examples of computer system resources.

1.1.9 Long Answer Type Questions

Q.1. Differentiate between application software and system software.
Q.2. What is an operating system and what are its goals?
Q.3. What are the advantages and disadvantages of batch processing?
Q.4. Explain the term SPOOLing and state the advantages of this technique
Q.5. What are resources? Explain the term "Resource abstraction" and "Resource sharing".

1.1.10 Suggested Readings

1. Nutt Gary, "Operating Systems" Addison Wesley Publication, 2000.
2. Silberschatz and Galvin, "Operating System Concepts" Sixth Edition, Addison Wesley Publishing Co., 1999.
3. Ekta Walia, "Operating System Concepts" , Khanna Publishing Co., 2002
4. William Stallings, Operating Systems, Internals & Design Principles, 4th edition, Prentice-Hall, 2001.
5. Deitel H.M., "Operating Systems", 3rd Edition, Prentice Hall of India
6. Dhamdhare D.M., "Systems Programming and Operating Systems", Tata McGraw Hill, Second Edition,1999.
7. Shaw, "Logical Design of Operating Systems", Prentice Hall of India, 1978.
8. Andrew S. Tannenbaum, "Modern Operating Systems", Pearson Education Asia, Second Edition, 2001.
9. A.S. Godbole **"**Operating Systems**"** Tata McGraw Hill 2002

Web Resources :

www.personal.kent.edu/rmuhamma/opsystems/os.html
www.wiley.com/colleges/silberschatz6e/0471417432/slides/slides.html

Types of Operating systems

Contents

1.2.0 Objectives

In this lesson, students will understand the different flavours of operating sytems : Multiprogramming operating system, timesharing operating system, real time operating system, multiprocessing operating system, network operating system, personal computer operating system, parallel processing operating system and distributed operating systems.

1.2.1 Multiprogramming

Multiprogramming is a rudimentary form of parallel processing in which several programs are run at the same time as a uniprocessor. Since there is only one processor, there can be no true simultaneous execution of different programs. Instead the OS executes part of one program, then part of the another and so on. To the user it appears that all programs are executing at the same time. Let us try to understand the reasons behind the development of operating systems from one form to another and so on.

Though SPOOLing solved the problem of speed mismatch between CPU and peripheral devices, still CPU was not fully utilized by a single user. Therefore, aroused the need of a system that could keep the CPU always busy with either the input-output processing or with the execution of other instructions of one program or the other. This system was termed multi-programming system.

The key idea was: As shown in Fig 1.2.2, the operating system could support for keeping several jobs in memory at one time. The operating system could pick and start the execution of one of the jobs in the memory. Whenever the job does not need CPU i.e. the job has to do only with the input-output devices, and the CPU is idle at that time, the operating system switches to another job in the memory and CPU executes a portion of it till this job issues a request for Input/Output or the first one has finished its Input/Output and so on.

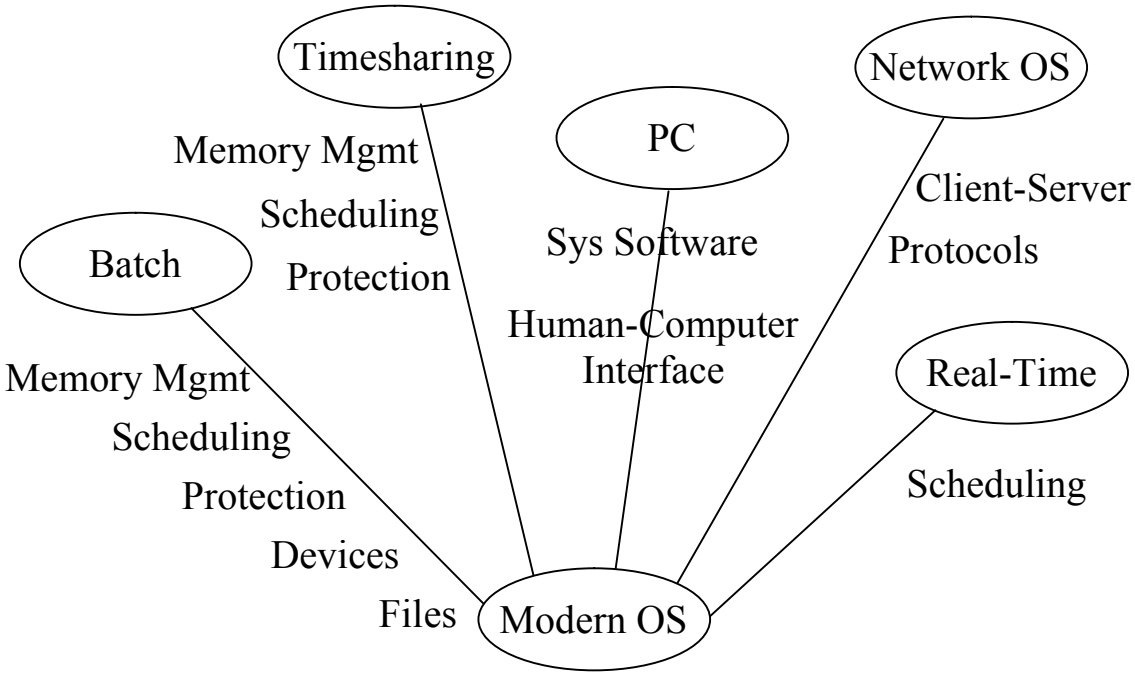


Fig 1.2.1: Various types of Operating Systems

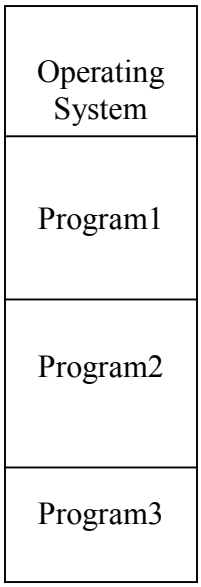


Fig 1.2.2: Many user programs which are ready to run have been loaded in the memory

Here, the operating system in this case had to be very complex. A very high-level memory management operation was expected from this operating system. Switching from one job to another, involves a huge effort on the part of operating system as operating system has to keep track of where did it left the execution of previous job and from where did it take up the execution of next job.

Advantages of multiprogramming

- High CPU utilization
- It appears that many programs are allotted CPU almost simultaneously.

Problems with Multiprogramming

- Jobs may have different sizes therefore, memory management is needed to accommodate them in memory.
- Many jobs may be ready to run on the CPU, which implies that we need CPU scheduling.

1.2.2. Time Sharing

Time sharing is an operating system feature that allows several users to run several tasks concurrently on one processor or on many processors, usually providing each user with his own terminal for input and output and an impact that only he/she is the one who is running his/her processes on the system (the presence of other user processes is transparent to a user). This implies that CPU's time is

shared between many user processes. These processes can be of one user or of different users.

Multitasking is a term closely related to the concept of time-sharing. Multitasking implies that multiple tasks can be handled simultaneously by the system. Multitasking is also possible on a system for a single user. For example, in windows environment, one can open more than one tasks in the running mode. This is how multitasking works for a single user. Multitasking is possible on a system for multiple users i.e. multiple users can submit a request (each through his/her own Input/Output terminal) to do specific tasks on a system. This is known as multitasking for multiple users and also known as Timesharing. UNIX operating system is known for its multi-user and timesharing feature.

There were four early systems that essentially defined timesharing operating strategy:

- CTSS(Compatible timesharing system): CTSS was developed for the IBM 7090 in the mid -1960s at M.I.T. It was the vehicle that supported the initial research on radical scheduling algorithms and modern memory management techniques.
- Multics replaced CTSS early on. It was intended to be more like a utility than a timesharing computer. It was designed to be extremely capable and reliable, whereas its contemporaries were often unreliable. Mutlics was the operatig system used to develop fundamental knowledge about virtual memory.
- The Cal timesharing system was designed and implemented about the same time as CTSS and Multics. The research results attributable to Cal generally concern generic timesharing technology, protection and security.
- AT&T Bell Labs UNIX designers had been associated with Multics but wished to have a far less complex operating to manage a minicomputer. Hence, they developed UNIX in 1970. Note that “minicomputer” meant something different then that it does now. In 1970, a minicomputer was a machine that had about 4KB of RAM, a few serial ports, and a small disk drive.

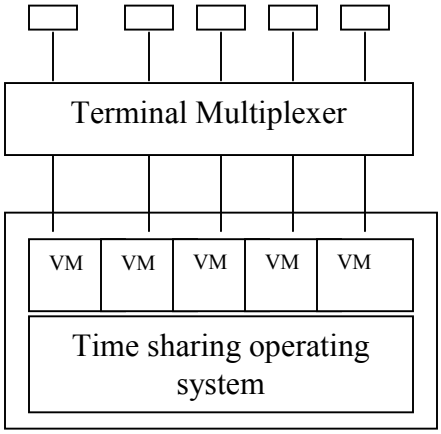


Fig 1.2.3: Timesharing System

As shown in Fig 1.2.3, time sharing systems focus on policies to implement equitable processor sharing. This allows users to treat the machine as if they have exclusive control of a comparatively slow computer- known as virtual machine(VM). As long as the timesharing system does not become overloaded, the relative response time is usually so small that no user ever comes to know about any comparative slowness in the computer's performance.

1.2.3 Real Time Systems

There are several definitions of real-time systems and most of them are contradictory. Unfortunately the topic is controversial, and there does not seem to be 100 per cent agreement over the terminology. We can state that:

- ◊ "A real-time system is one in which the correctness of the computations not only depends upon the logical correctness of the computation but also upon the time at which the result is produced. If the timing constraints of the system are not met, system failure is said to have occurred."
- ◊ "Hence, it is essential that the timing constraints of the system are guaranteed to be met. **Guaranteeing timing** (Deadline) behaviour requires that the system should be predictable. It is also desirable that the system should attain a high degree of utilization while satisfying the timing constraints of the system".

1.2.3.1 Real Time Operating System (RTOS)

Real time operating system has well defined fixed time limit i.e. processing must be done within the defined time limit otherwise the system will fail.

Areas where RTOS is useful

A Real time system is often used as a central device in a dedicated application like those that have been listed here:

1. Systems that control scientific experiments
2. Medical-imaging systems
3. Some display systems, e.g. display screen of a airplane
4. Fuel-injection systems
5. Robotics
6. Air-traffic control
7. Control systems

Next generation system will include the autonomous land rover, controller of robots with elastic joints, the space stations and undersea exploration.

Example of Real Time Systems

A good example is a robot that has to pick up something from a conveyor belt. The piece is moving, and the robot has a small window to pick up the object. If the robot is late, the piece won't be there anymore and thus the job will not be done correctly,

despite the fact that the robot went to the right place. If the robot is early, the piece won't be there yet and the robot may block it.

Let us first identify, what is a real time task?

A task, which is attempting to control or react to the events (occurring in real time) and is able to keep up with the event with which it is concerned, is known as Real Time Task.

This control is possible only when task has a deadline (which specifies either start time or completion time). Most of the time deadline refers to the completion time.

Now, these real time tasks can be classified into two categories - Hard Real Time Tasks and Soft Real Time Tasks.

In case of Hard Real Time task, meeting the deadline is must otherwise, undesirable and unforeseen damages can occur. Whereas, in case of Soft Real Time Task, meeting the deadline is desirable but not mandatory. For such tasks, if the deadline could not be met, there is a possibility to reschedule the task and complete it.

Real time operating system should support the functioning of a system, which is specialized to cater to the deadline requirements of a particular real time task. Examples of real time operating system are: Harmony (Developed by National Research Council of Canada), Maruti (University of Maryland), OS9 (Microware Systems Corporation), RTEMs (Real Stone Military Arsenal), etc.

The operating system should make best effort to meet the deadline, but if it fails to meet the deadline, it should continue to provide service rather than simply abandon the service request. Today, real time computing has extended beyond deadline scheduling to address various other quality of service issues. For example, application processes may require that information be delivered over a network in a prescribed amount of time or with a minimized deviation in its delivery rate. The solution to these requirement s are difficult, and they are subject of considerable effort in operating system designs.

1.2.4 Multiprocessing

An old saying "many hands make light work", expresses the precise that led to the development of multi-processor systems. At any given time, there is a technological limit on the speed with which a single processor chip can operate. If a system's workload cannot be handled satisfactory by a single processor, one response is to apply multiple processors to the problem and this is known as multiprocessing environment.

The success of this response depends upon:

Not only the skill of the system designers but also on whether the workload is amenable to multiprocessing.

If improved performance is the objective of a proposed migration from uniprocessor to a multiprocessor system, then the following conditions should be true:

- The workload is processor limited and has saturated its uniprocessor system.
- The workload contains multiple processor-intensive elements, such as transactions or complex computations that can be performed simultaneously and independently.

In case the condition number 2 is not true, the performance of a multiprocessor can sometimes actually be worse than that of a comparable uniprocessor.

1.2.4.1 Symmetrical Multiprocessing System

In symmetrical multiprocessor system, all of the processors are essentially identical and perform identical functions. Characteristics of such system can be specified as:

- Any processor can initiate an I/O operation, can handle any external interrupt, and can run any process in the system.

All of the processors are potentially available to handle whatever needs to be done next.

1.2.4.2 Asymmetric Multiprocessing System

Asymmetry implies imbalance, or some difference between processors. Thus, in asymmetric multiprocessing, different processors do different things. From design point of view, this is often implemented so that one processors job is to control the rest, or it is the supervisor of the others. Sometimes, a processor is specialized to the point where it is responsible for doing only one thing in the system (like handling a task like sound, video or some other specialized input or output).

Some advantages and disadvantages of this approach are:

- In some situations, I/O operation or application-program processing may be faster because it does not have to contend with other operations or programs for access to a processor i.e. many processors may be available for a single job.
- In other situations, I/O operation or application program processing can be slowed down because not all of the processors are available to handle peak loads.
- In asymmetric multiprocessing, if the supervisor processor handling a specific work fails, the entire system will go down.

1.2.5 Network Operating System

In the late 1960s ARPA —the Advanced Research Projects Agency of the Department of Defense rolled out the blueprints for networking the main computer systems of about a dozen ARPA-funded universities and research institutions. They were to be connected with communications lines operating at a then-stunning 56 kilobits per second Kbps)—1 Kbps is equal to 1,000 bits per second—at a time

when most people (of the few who could be) were connecting over telephone lines to computers at a rate of 110 bits per second. Researchers at Harvard talked about communicating with the Univac 1108 "supercomputer" across the country at the University of Utah to handle the massive computations related to their computer graphics research. Academic research was about to take a giant leap forward. Shortly after this conference, ARPA proceeded to implement what quickly became called the ARPAnet —the grandparent of today's Internet. Although the ARPAnet did enable researchers to network their computers, its chief benefit proved to be its capability for quick and easy communication via what came to be known as electronic mail (e-mail). This is true even on the Internet today, with e-mail, instant messaging and file transfer facilitating communications among hundreds of millions of people worldwide and growing rapidly. TheARPAnet was designed to operate without centralized control. This meant that if aportion of the network should fail, the remaining working portions would still be able to route data packets from senders to receivers over alternative paths.

The protocols (i.e., sets of rules) for communicating over the ARPAnet became known as the Transmission Control Protocol/Internet Protocol (TCP/IP). TCP/IP was used to manage communication between applications. The protocols ensured that messages were routed properly from sender to receiver and that those messages arrived intact. The advent of TCP/IP promoted worldwide computing growth. Initially, Internet use was limited to universities and research institutions; later, the military adopted the technology.

Eventually, the government decided to allow access to the Internet for commercial purposes. This decision led to some concern among the research and military communities—it was felt that response times would suffer as "the Net" became saturated with users. Infact, the opposite occurred. Businesses rapidly realized that they could use the Internet to tune their operations and to offer new and better services to their clients. Companies spent vast amounts of money to develop and enhance their Internet presence. This generated intense competition among communications carriers, hardware suppliers and software suppliers to meet the increased infrastructure demand. The result is that bandwidth (i.e., the information carrying capacity of communications lines) on the Internet has increased tremendously, and hardware and communications costs have plummeted. The World Wide Web(WWW)allows computer users to locate and view multimedia-based documents (i.e., documents with text, graphics, animation, audio or video) on almost any subject.

Although the Internet was developed more than three decades ago, the introduction of the World Wide Web(WWW) was a relatively recent event. In 1989, TimBerners-Lee of CERN (the European Center for Nuclear Research) began to

develop a technology for sharing information via hyperlinked text documents. To implement this new technology, Berners-Lee created the HyperText Markup Language (HTML).Berners-Lee also implemented the Hypertext Transfer Protocol(HTTP)to form the communications backbone of his new hypertext information system, which he called the World Wide Web.

Surely, historians will list the Internet and the World Wide Web among the most important and profound creations of humankind. In the past, most computer applications ran on "stand-alone" computers (computers that were not connected to one another). Today's applications can be written to communicate among the world's hundreds of millions of computers. The Internet and World Wide Web merge computing and communications technologies, expediting and simplifying our work. They make information instantly and conveniently accessible to large numbers of people. They enable individuals and small businesses to achieve worldwide exposure. They are changing the way we do business and conduct our personal lives. And they are changing the way we think of building operating systems.

Today's operating systems provide GUIs that enable users to "access the world" over the Internet and the Web as seamlessly as accessing the local system whereas the operating systems of the 1980s were concerned primarily with managing resources on the local computer.

An operating system, which includes software to communicate with other computers via a network, is called **network operating system**. This allows resources such as files, application programs and printers to be shared between computers.

Such operating systems are specialized to provide the networking services. The first network operating systems really were add-on packages that supplied the networking software for existing operating systems such as MS-DOS or OS/2. More recent operating systems come with the networking components built in to them.

Network operating system examples are :

BSD(Berkeley System Distribution) Unix, Novell, Lantastic, MS LAN Manager, Windows NT etc. Network operating system can be specialized to serve as peer-to-peer operating system or as client/server operating system.

1.2.5.1 Client / Server support

As shown in Fig 1.2.4, client server machines require specific software components. Redirector and Server services are two types of services that must be provided by client and server operating system respectively. Redirector service forwards the I/O requests (of the clients) for accessing remote files, where a server satisfies them. A computer that is strictly a server often cannot provide any client functionality. On a Novell server or a Banyan server, a user cannot run word

processing application whereas on Microsoft's NT server and UNIX server, user can run client programs.

1.2.5.2 Peer-to-peer support

A computer in a peer-to-peer network functions as both client and a server, thus, it requires both client and server software. Operating systems such as Windows NT Workstation and Windows 95, both of which are peer-to-peer network operating systems, include dozen of services and utilities that facilitate networking.

1.2.6 Personal Computer System

For a personal computer (PC) the operating system should perform some basic tasks such as recognizing information from the keyboard and mouse, sending information to the monitor, storing and retrieving information from memory and controlling peripheral devices such as printers and scanners.

Operating systems are responsible for running assistive technology applications such as screen magnifiers, and applications that read text aloud. They are the basis for running common applications such as word processors and Internet browsers.

A variety of operating systems have evolved for PCs. Some of them have more sophisticated user interface whereas others have more emphasis on maximizing CPU and peripheral utilization. UNIX operating system has a lot of variants, which emphasize on the different aspects that have led to the continuous process of developing new operating systems. Protection, multitasking, multi-user, user interaction, multiprocessing are few issues that have now become a mandatory part of the operating system's role.

Window NT, Windows 95, Windows 98, Window XP, OS/2, Windows 2000, Linux, Mac OS, are few operating systems commonly used on personal computers.

Microsoft Corporation became dominant in the 1990s. In1981, Microsoft released the first version of its DOS (Disk operating system) for the IBM personal computer. In the mid-1980s,Microsoft developed the Windows operating system, a graphical user interface built on top of the DOS operating system. Microsoft released Windows 3.0 in 1990; this new version featured a user-friendly interface and rich functionality. The Windows operating system became incredibly popular after the 1993 release of Windows 3.1, whose successors, Windows 95 and Windows 98, virtually cornered the desktop operating system market by the late 90s.These operating systems, which borrowed from many concepts (such as icons, menus and windows) popularized by early Macintosh operating systems, enabled users to navigate multiple concurrent applications with ease. Microsoft also entered the corporate operating system market with the 1993 release of Windows NT, which quickly became the operating system of choice for corporate workstations.

The journey of Microsoft in this regard has therefore been quoted to range from Disk Operating System to Windows XP. However, therefore, this evolution has witnessed a lot of change in design criterion for an operating system.

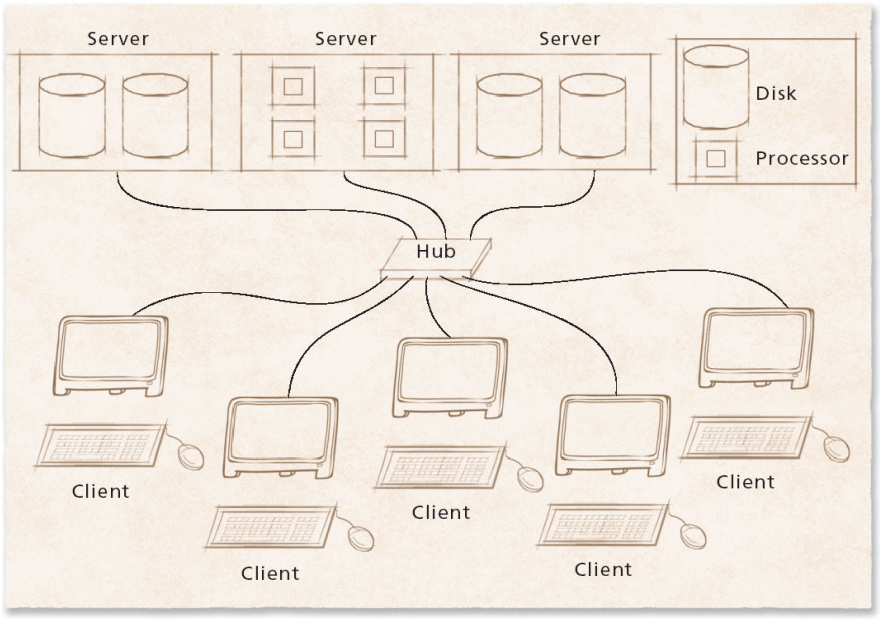


Fig 1.2.4: Client/server networked operating system model

1.2.7. Parallel Processing Systems

Many people conceive that the notion of parallel processing is very complicated stuff for them.... but here, an attempt has been made to explain it in very simple words...

In layman's terms, if you have lot of clothes piled up for washing, you can wash them in your washing machine at home by spending several hours on this job. Another way to do this job is to distribute these clothes among many laundries, which will do their job simultaneously. Thus, you can shrink your time commitment by several hours.

These same principles are behind parallel processing, in which lots of processors - either in one computer or inside several linked machines gang up to work on a single problem at one time. A typical desktop PC has one processor; a computer built to handle parallel processing can have several hundred of processors.

Specific areas where parallel processing is required are scientific industries, although its reach is slowly extending to the business world. The best candidates for parallel processing are projects that require many different computations. Single processor computers perform each computation sequentially. Using parallel processing, a computer can perform several computations simultaneously, drastically

reducing the time it takes to complete a project. The medical community uses parallel processing supercomputers to analyze MRI images and study models of bone implant systems. Airline uses parallel processing to process customer information, to forecast demand and to decide what fares to charge.

Parallel processing is not that easy to implement, a special operating system, a special programming language is few things that are required to implement such a system.

The task of an operating system designed to support parallel processing is to manage how to divide a program so that it runs on more than one processor at a time.

Similarly, the programming language meant to code parallel processing programs must have special constructs to specify different parts of a program that should run on different processors for achieving parallelism.

1.2.8 Distributed Processing

A large central computer with a number of remote terminals connected to it is sometimes conceived as distributed processing environment. However, it is not like that until it has a component of processing that is distributed or data that is distributed over a number of computers connected across a network.

The concept of distributed processing will be more clear if we consider the examples of a bank having its branches throughout India and its head office in Chandigarh. Let us assume that this bank maintains the local data at the local branch and the copy of data of all branches at Chandigarh. Thus, we say that the data is distributed throughout the country. This is basically to facilitate the query processing for local customers of a branch and also for global customer (one who moves from Bangalore to Coimbatore).

Distributed processing is thus, a form of on-line processing that allows a single transaction to be composed of application programs that access one or more databases on one or more computers across a network. This type of transaction in which multiple applications programs cooperate, is called a distributed transaction. The computer in distributed system can vary in size and even in their platforms. They may include microprocessors, workstations, minicomputers and large general-purpose computer systems. Resource sharing, increased throughput, communication, reliability are few reasons to have distributed processing systems. Using distributed processing, related data at regional and branch locations can be synchronized; updated simultaneously with guaranteed results.

The job of distributed operating system is to allow the user to access remote resources in the same manner as they do local resources, control the data and process migration from one site to another, detecting idle CPUs on the network etc.

Many operating systems have been developed to support distributed processing. Alpha Kernel, Amoeba, Angle, Chorus, Mach are few examples of distributed operating systems.

Self Review

Identify certain operating systems in your laboratory and classify them into various categories.

1.2.9 Keywords

Multiprogramming: capability of having multiple programs in main memory. Such programs are ready to execute

Timesharing: sharing of processor time among the jobs of different users

Multitasking: execution of multiple tasks all of which ma belong to one user only

Multiprocessing systems: system having more than one processor

Real Time systems: systems where deadlines are specified for execution of a task

Network operating systems: operating system which provides networking support that may be client/server or peer to peer

Parallel processing systems: systems capable of running different tasks or pieces of one task in parallel

Distributed systems: networked system of autonomous computers that appears to the users of the system as a single computer

Distributed operating systems: operating systems specialized to handle distributed systems

1.2.10 Summary

1. Operating systems have evolved from single-user computers, to multiprogramming systems, to timesharing systems, to personal computers and workstations interconnected with networks. Multiprogramming systems introduced technology to support concurrency among jobs.
2. Time sharing operating systems extended multiprogramming so that each job could have multiple processes executing on behalf of the user at any one time.
2. Multitasking is a term closely related to the concept of time-sharing. Multitasking implies that multiple tasks can be handled simultaneously by the system. Multitasking is also possible on a system for a single user.
3. An operating system, which includes software to communicate with other computers via a network, is called network operating system.
4. Operating systems that facilitate the execution of programs in parallel are parallel processing operating systems.
5. Operating systems that facilitate the execution of real time tasks are called real time operating system.
6. Operating systems that facilitate the execution of distributed applications are called distributed operating systems.

1.2.10 Short Answer Questions

Q.1. Give one example each of the operating system that incorporates multitasking, time sharing, multiprogramming and networking facilities.
Q.2. Name any two distributed operating systems.
Q.3. In what all categories do the UNIX operating system fall?
Q.4. Does the user of timesharing system come to know that many other users are also accessing the processor that the same time?
Q.5. What is virtual machine?
Q.6. Differentiate between hard real time task and soft real time task.

1.2.11 Long Answer Questions

Q.1. Explain the evolution of the operating systems from a simple operating system to today's popular operating systems.
Q.2. What are supposed to be special features of a network operating system?
Q.3. Explain how multiprogramming contributes towards higher CPU utilization and increased throughput.
Q.4. Identify some situations where multiprocessing becomes mandatory.
Q.5. What is the fundamental difference between network and distributed operating systems?
Q.6. Compare symmetric and asymmetric multiprocessing systems.

1.2.13. Suggested Readings

1. Nutt Gary, "Operating Systems" Addison Wesley Publication, 2000.
2. Silberschatz and Galvin, "Operating System Concepts" Sixth Edition, Addison Wesley Publishing Co., 1999.
3. Ekta Walia, "Operating System Concepts" , Khanna Publishing Co., 2002
4. William Stallings, Operating Systems, Internals & Design Principles, 4th edition, Prentice-Hall, 2001.
5. Deitel H.M., "Operating Systems", 3rd Edition, Prentice Hall of India
6. Dhamdhare D.M., "Systems Programming and Operating Systems", Tata McGraw Hill, Second Edition,1999.
7. Shaw, "Logical Design of Operating Systems", Prentice Hall of India, 1978.
8. Andrew S. Tannenbaum, "Modern Operating Systems", Pearson Education Asia, Second Edition, 2001.
9. A.S. Godbole **"**Operating Systems" Tata McGraw Hill 2002

Web Resources :

www.personal.kent.edu/rmuhamma/opsystems/os.html
www.wiley.com/colleges/silberschatz6e/0471417432/slides/slides.html

Processes and Threads

Contents

1.3.0 Objectives

The objectives of this lesson are to familiarize the students with the concept of process and make them understand the basic difference between a program and a process. It further explains in which all states a process can be at any time and how the state of a process changes. The concept of Process control block and various kinds of schedulers have been described next. The various operations permissible on a process are discussed. The cooperating processes have been put forth next which is followed by important concept of threads and their differences and similarities with processes. Objects have been described very briefly just to make the students aware that object oriented technology has also made its place in the context of operating systems.

1.3.1 Concept of Process

A process is a program in execution.

The environment you interact with when you use a computer is built out of processes.

- ◊ Your command interpreter is a process
- ◊ When you execute a program you just compiled, the operating system generates a process to run the compiler.

As mentioned "process" is not the same as "program", it is infact a program in execution. We can distinguish between a passive program on disk and an actively executing process, very easily. Many people can run the same program; and each copy of this program corresponds to a distinct process.

Let us now see, what does a process contain other than a program? A program is only one part of a process; the process also contains the execution state of the program. Let us now identify the reasons for process creation and process termination. **Reasons for process creation can be:**

- ◊ User logs on the system
- ◊ User starts a program
- ◊ Operating system creates a process to provide a service (e.g. printer daemon to manage printer)
- ◊ Program starts another process (e.g. netscape calls XV to display a picture)

Reasons for process termination can be:

- ◊ Normal completion of process (Exit)
- ◊ Arithmetic error, or data misuse (e.g. wrong type of data)
- ◊ Invalid instruction execution
- ◊ Insufficient memory available, or memory bounds violation
- ◊ Resource protection (e.g.: memory protection error, CPU protection error (e.g. Infinite looping process))
- ◊ I/O failure

All the above stated reasons fall in the category of internal completion. It can be external completion (forced completion: by operator or by parent process of the child process).

1.3.2 Process States and their transitions

At any time, a process is in one of the states mentioned below:

1. Running
2. Ready to Run
3. Blocked

Running state implies that process is currently run by the CPU. Ready to run means that it needs CPU attention and time to run, otherwise it is not blocked in any sense or in other words, process is waiting to be assigned to a processor. Blocked state implies that process is not running currently and is waiting for some event to occur (such as I/O completion or reception of a signal).

Figure 3.1 shows state transitions of a process, in general. Some possible reasons for these transitions are:

1. **Ready to Running:** A process is selected out of the ready queue (i.e. scheduled to be run) and is dispatched to the running state.

2. **Running to Blocked**: A process starts a time consuming I/O operation or is swapped out or makes a spontaneous request to sleep or is blocked due to synchronization operations. I/O operation is infact the most common reason for this type of transition. Here, since a process is waiting for I/O completion and thus do not need CPU for executing any other instructions, CPU is freed by it and CPU in the meantime takes up some other ready process and starts its execution.
3. **Blocked to Ready:** The reason for a process to remain blocked is no longer applicable e.g. I/O completes, timer goes off etc. Thus, the process becomes ready for its execution.
4. **Running to Ready:** The running process is pre-empted because some other process of higher priority has become ready.

1.3.3 Process Control Block

Each process is represented by a process control block, also known as task control block. It contains information regarding the execution state of the program, memory management information and CPU scheduling information etc. Figure 1.3.2 shows the contents of a process control block.

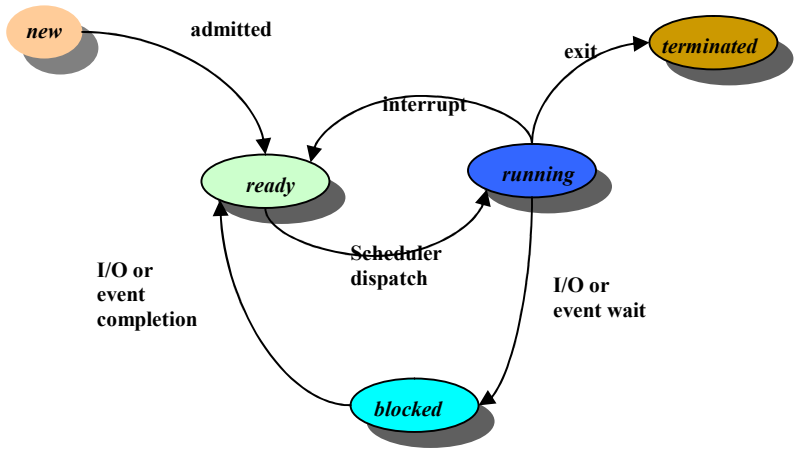


Fig 1.3.1: Process transitions

The description of this information is as follows:

(i) **Process State:** It can be ready, running and blocked. It is useful to indicate the current state of the process.

(ii) **Process Number:** Unique number is allocated to a process at the time of its execution. This is helpful in referring to the process through shell

commands or other commands of the command language of the operating system.

(iii) **Parent Process Number:** If this process has been originated as a sub-process of a process, then its parent's process number is also mentioned so that the output of this process can be returned back to the parent process.

(iv) **Program Counter:** contains the address of the next instruction to be executed and thus is useful in indicating the status of a process.

(v) **CPU Registers:** CPU registers depend upon the computer architecture. They can be index registers, accumulators, general-purpose registers etc. They are used to save other information e.g. when an interrupt occurred, whether the current instruction was fully executed or not etc., at the time of state transition of a process.

(vi) **CPU scheduling information:** Priority of the process is mostly indicated here.

(vii) **Memory Management Information:** Values of base and limit registers, page or segment tables can be indicated here.

(viii) **Accounting Information:** The information like amount of CPU time used, process numbers, time limits etc. fall into this category.

(ix) **I/O Status Information:** It includes the names of I/O devices used by a process, list of open files etc.

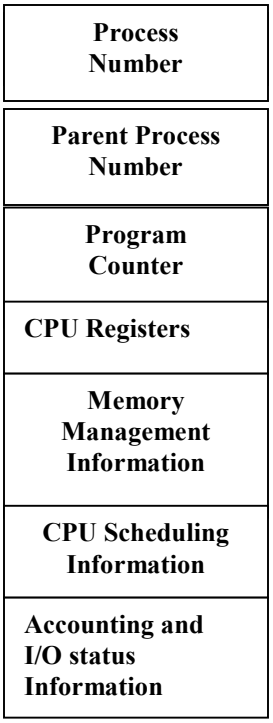


Fig 1.3.2: Contents of Process Control block of a Process

1.3.4 Process Scheduling

Let us first understand why and where do we need to do process scheduling? In a uniprogramming environment (the environment where only one process is ready for its execution at one moment of time) there is no question of process scheduling. It becomes a necessary activity in a multiprogramming environment.

Refer to state transitions in previous section where it has been mentioned that a process, which is ready, becomes a running process when it is selected out of a ready queue. Let us first concentrate on this concept of ready queue and device queue.

Ready Queue: It is a doubly linked list of processes, which are ready to run. They are ordered by priority, the highest priority process is at the front of the queue. As we all know queue has FIFO structure, therefore, whenever the CPU takes up a new process, it actually picks up the process with the highest priority.

When we say processes are listed together, it means that actually their PCBs (Process Control Blocks) are linked together. The header of the queue contains two pointers. The first pointer points to the first PCB and the second points to the last PCB in the list. Each PCB has a pointer to point to the next process in the ready queue. Fig 1.3.4 shows the situations in which a process enters the ready queue.

Device Queue: The processes, which are blocked due to unavailability of an input/ output device, constitute this queue. There is a separate queue for each I/O device. Figure 1.3.3 shows the structure of a ready queue. A process when created is initially put in the ready queue. Once it is picked up by the CPU for its execution, there may occur one of the many events that cause a transition in its state from running to ready or running to blocked or running to final completion (exit). If its state is changed to blocked, it is added to the device queue of the device whose services are required by it.

In case, its state is changed to ready again, it is put back to the ready queue. In case, it terminates after its successful completion, it is removed from all the queues and its PCB and other resources are deallocated.

Now, let us find out who is responsible for taking a process out of a ready queue and assign it to CPU for its execution? The answer is process scheduler. Process scheduler is a part of the operating system that is responsible for changing the state of processes. In other words, it is that component, which is responsible for scheduling of the processes to be executed by the CPU.

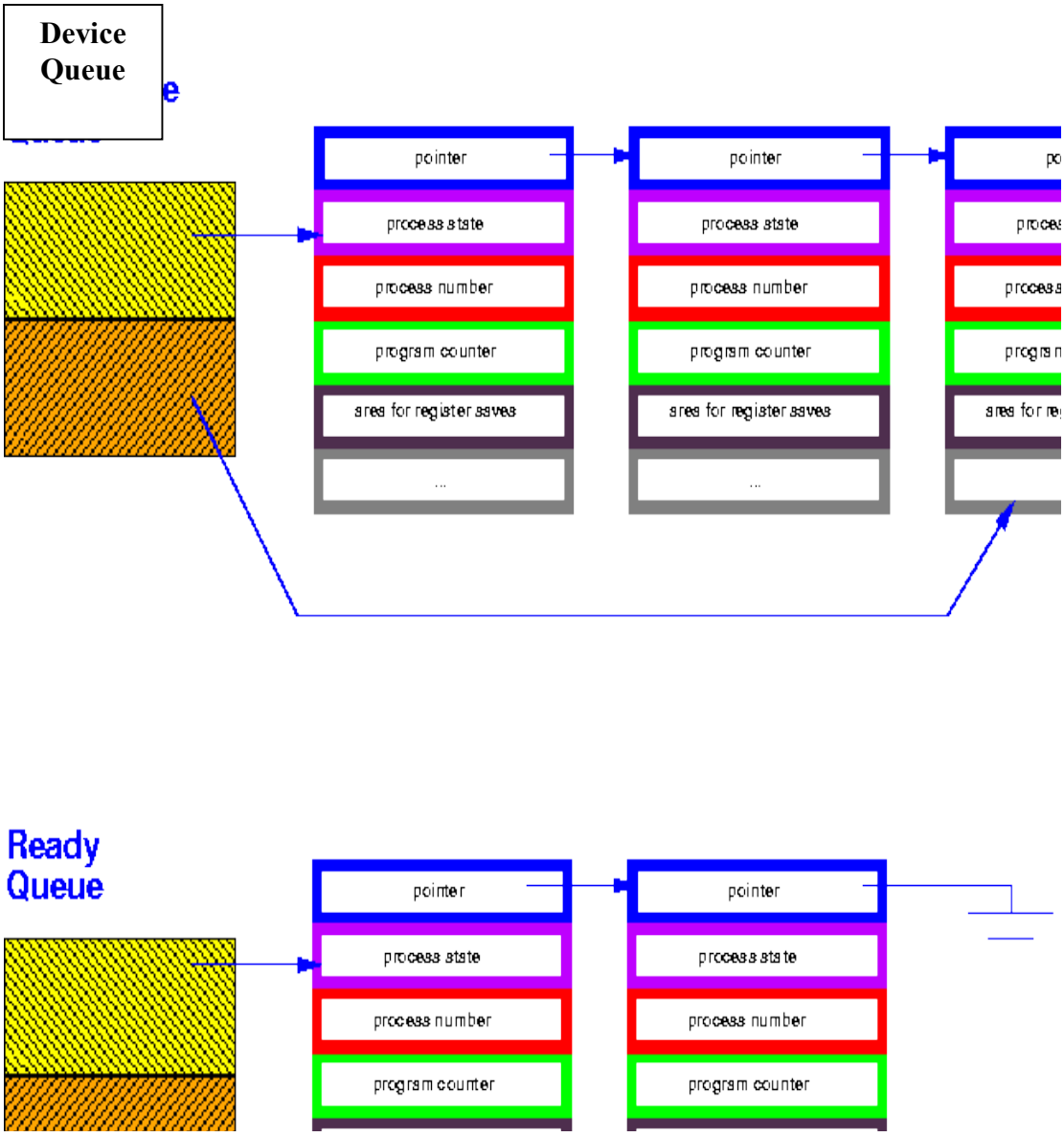


Fig 1.3.3: Device Queue, Ready Queue

A scheduler infact should compromise between certain desirable things:

- ◊ Make sure that each process gets a fair share of the CPU
- ◊ Keep the CPU busy 100% of the time
- ◊ Minimize Response Delays

Schedulers are defined to be of three types:

(i) Long-term scheduler

(ii) Short-term scheduler

(iii) Medium-term scheduler

Long-term scheduler selects processes from secondary storage device e.g. disk and loads them into memory for execution.

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

Long-term scheduler is known as "long-term" because time for which the scheduling is valid is long. In other words, it executes less frequently as it takes from minutes to hours for the creation of new processes in the system. The primary job of long-term scheduler is to run the multi-programming environment smoothly. Thus, its job is to ensure that there are always a fixed number of ready processes in the memory of the system.

Short-term scheduler on the other hand, must have to work very often. A process must be executing for only a very short duration (e.g. in few milliseconds) before it gets blocked (due to some I/O operation etc.). This requires that CPU should immediately take up another process for execution and thus the scheduler should immediately select a process for the CPU. Therefore, it is needed that a short-term scheduler should be fast as compared to the long-term scheduler.

Medium-term scheduler is required at the times when a process is to be removed from the memory (due to some performance reasons) and thus to reduce the degree of multiprogramming. Later on, this process may be put into the memory again and its execution can be again started from where it was left off. This process of moving a process in and out of the memory is called **swapping** (shown in Fig 1.3.5). Sometimes, swapping becomes essential to deal with the situation when a running process requests more RAM. It can be given by moving (swapping) a ready process to the disk and making RAM available to the requesting process. Later on, when this process's requirement is over, the swapped out process can be swapped back into the memory. All versions of Windows operating system use swapping.

1.3.5 Operations on a Process

A process is usually sequential and consists of a sequence of actions that take place one at a time. When a set of processes is run on a single CPU, using time slicing to multitask them, this is referred to as concurrent sequential processing or pseudo-parallel processing. If there is more than one CPU available, then processes may be run in parallel. **The set of operations on processes includes:**

- ◊ Create a process
- ◊ Destroy a process
- ◊ Run a process
- ◊ Suspend a process
- ◊ Get process information

◊ Set process information

The operating system will supply mechanisms for at least some of these. Let us discuss process creation and deletion mechanisms provided by the operating systems.

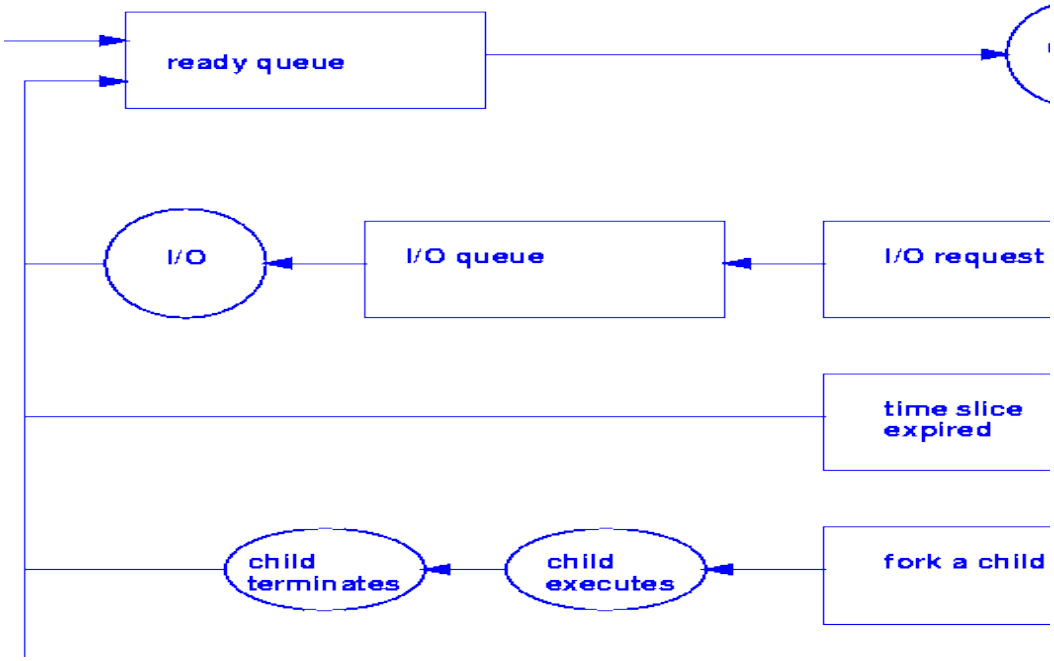


Fig 1.3.4: Situations in which a process enters ready queue

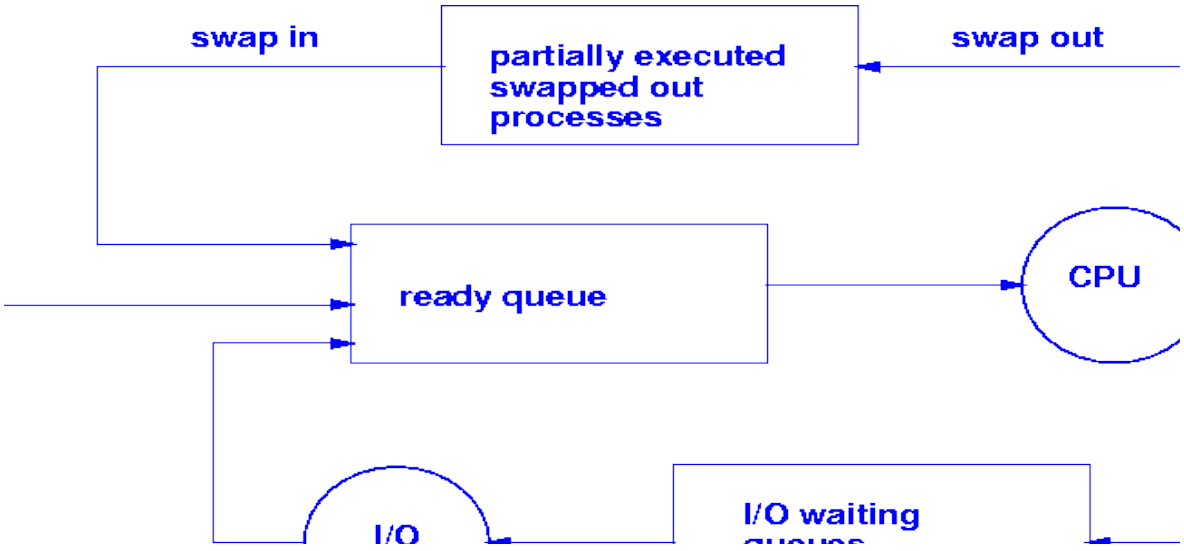


Fig 1.3.5: Swapping

1.3.5.1 Process Creation

There must be a mechanism for creating processes by the operating system. A new process will be created using an existing one. There are several possible ways of doing this.

Synchronous

If a process is created from another as a synchronous process then the new process must complete execution before the old one can resume.

Asychronous

If the new process is created asynchronously, then the two processes may be run concurrently (in pseudo-parallel way).

Parent

When a new process is created, it may use the old one as "parent". This is done in MSDOS and Unix. Alternatively, there may be no relation between the original and the new process, as in windows NT.

As each newly created process will use some of the physical and logical resources (e.g. input data to the program), therefore, we should also analyze how these resources will be shared by the parent and child processes? There are infact three possibilities:

- Parent and children share all resources
- Children share a subset of parent's resources
- Parent and child share no resources.

Similarly, there are two possibilities regarding the address space of the new process:

- The child is duplicate of parent
- The child has a program loaded into it.

Let us see, how a process is created in UNIX.

Unix's Application Programming Interface (API)

Unix API allows creation of asynchronous processes where each process has a parent. Each process has a process ID (a number) that can be used to tell them apart. This PID (Process ID) is shown by executing the user-level command "ps". Because the parent process can resume execution before a child terminates, the parent can continue to create processes. The children can also create processes as shown in Figure 1.3.6.

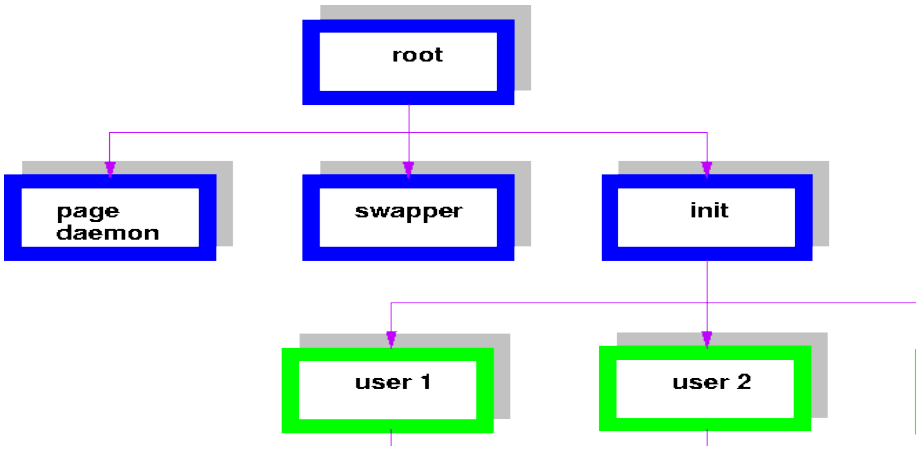


Fig 1.3.6: UNIX process hierarchy

Fork() is infact provided as a system call by Unix. This splits the current process into two almost identical copies. A copy is made of the user and system stacks, of the allocated data space and of the registers. The major difference is that one has the PID of the parent; the other has a new PID. The fork system call returns a value that is a PID. The return value of fork() is zero if a process is child, or is PID of the child if a process is parent.

Exec

It is common to want to replace one of these processes so that it uses a different program. For example, suppose the process creates a file that it wants to be printed. In Unix it cannot print it itself, because it does not have access to the line printer device. The program "lpr" must be used. One way is

system ("lpr myfile");

This runs synchronously, so until the line printer program terminates, the calling program suspends. A preferable way may be to fork a new process and let the parent continue with the main piece of work, while the child prints the file asynchronously. This is possible but may require good length of code to be written like :

```
if (fork ( ) == 0)
        system ("lpr myfile");
else { / * parent * / }
```

But it does nothing and waits for the "system" to finish. Better is to replace the child with the "lpr" program. This is the job of the "exec" functions.

```
        int exec lp   (   (char * file
                          char * arg0,
                          char * arg1,
                            ‘
                            ‘
```

‘

char * argn)
(char *) 0);

The file argument here is the name of the program to execute. “argo” is often the same as “file”. This is used as the name of the process, as reported by “ps”. “arg1” to “argn” are the arguments(1 to n) to the command. In normal operation “exec” never returns. If it fails, it returns - 1.

1.3.5.2 Process Termination

Normal Termination

Process terminates by using exit system call. It returns some output data to its parent process. All the resources allocated to the process are freed by it, at this time. This is all that happens when a process terminates by executing its last statement i.e. completion on its own.

Forced Termination

On the other hand, it can be a forced termination. A process may be forcefully terminated by its parent process by executing an abort system call. A parent may want to abort its child’s execution due to the following reasons:

- Child has exceeded allocated resources
- Task assigned to child is no longer required
- Parent is terminating and operating system does not allow a child to continue if its parent terminates (this is known as cascading termination).

In case of UNIX, a process may terminate by using exit system call and its parent process may wait for that event by using wait system call. UNIX supports cascading termination.

In UNIX, there is a possibility that user can straight away terminate a process provided he knows the process ID of that process. This is possible by a system call “kill”. It takes two arguments -

(i) Process ID (pid) of the process that user wants to kill, and

(ii) Signal (sig) that you want to send to that process.

If kill is successful, it returns 0; otherwise it returns a negative value.

1.3.6 Cooperating Process

Let us now understand the concept of cooperating processes.

“The processes which execute concurrently and affect or get affected by other processes executing in the system are called cooperating processes”

The processes can affect or get affected by other processes by sharing data or by sharing other resources in the system. Thus, any process which shares data with other processes is a cooperating process. The need of such processes is obvious.

They will be required to speed up the computation, sharing of information, user convenience, etc.

Breaking up of one task into subtasks such that each of them can run in parallel and thus give faster computation results may be one of the requirements. Similarly, it may be required to allow several users to share the same piece of information (e.g. some data may be common to many user programs). This makes it compulsory for the operating system to provide support for the execution, synchronization and communication amongst cooperating processes.

To illustrate the concept of cooperating processes, let us have a look at figure 1.3.7 indicating "Pizza" problem. Let us consider two persons - Person A and Person B. Both of them are family members and one is producing while another is consuming pizzas.

- PERSON A (producer process)- creates pizzas

```
repeat
                …
  produce a pizza in nextp
    …
  while in+1 mod n = out do noop;
  buffer[in] := nextp;
  in := in+1 mod n;
until false;
```

- PERSON B (consumer process)– eats pizzas

```
repeat
        while in = out do noop;
        nextc := buffer[out] ;
        out:= out+1 mod n;
  …
  consume the next item in nextc
  …
until false
```

Figure 1.3.7: Pizza Problem

Here, person A is a producing process whereas person B is a consuming process. In order that both can do their jobs concurrently, we must have a buffer (here fridge) of pizzas that can be filled in by the person A (producing process) and emptied by the person B (consuming process). The producer and consumer should be synchronized in their actions. This implies that person B (consumer) should not

try to eat pizza that has not been bought from the store yet, by the person A. In such situation, the person B (consumer) should wait. Now this buffer (fridge in our example) can be taken to be bounded or unbounded. In case of unbounded buffer there is no limit on the size of buffer and therefore, it is possible that consumer has to wait sometime for new pizzas but the producer can always keep on producing new pizzas. Thus, we can say that person A will never fall short of fridge space, whereas in case of bounded buffer when buffer size is fixed, both person A and B can have to wait if buffer is full and if buffer is empty respectively.

If person A and person B are processes executing in the operating system and have a producer-consumer relation, then the buffer that has been talked about here, is provided by either the operating system or by the code written by the application programmer in the form of shared memory.

1.3.7 Threads

All programmers are familiar with writing sequential programs. You have probably written a program that displays or sorts a list of names or computes a list of prime numbers. These are sequential programs: each has a beginning, an end, a sequence, and at any given time during the runtime of the program there is a single point of execution.

A thread is similar to a sequential program: a single thread also has a beginning, an end, a sequence, and at any given time during the runtime of the thread, there is a single point of execution. However, a thread itself is not a program -- it cannot run on its own -- but runs within a program.

Definition: A thread is a single sequential flow of control within a program.

There is nothing new in the concept of a single thread. The real concern surrounding threads is not about a single sequential thread, but rather about the use of multiple threads in a single program all running at the same time and performing different task. This is known as multithreading.

The HotJava browser is an example of a multithreaded application; within the HotJava browser you can scroll a page while it is downloading an applet or image, play animation and sound concurrently, print a page in the background while you download a new page or watch three sorting algorithms race to the finish. You are used to life operating in a concurrent fashion ... so why not your browser? This concurrent execution is possible by virtue of multithreading.

Thread is also known as lightweight process. A thread is similar to a real process as a thread and a running program are both a single sequential flow of control. However, a thread is considered lightweight because it runs within the context of a full-blown program and takes advantage of the resources allocated for that program and the program's environment.

As a sequential flow of control, a thread must carve out some of its own resources within a running program (it must have its own execution stack and program counter for example). The code running within the thread only works within that context. Thus, other texts use "execution context" as a synonym for "thread".

1.3.7.1 Threads and Processes

An idea of how threads and processes can be related to each other is depicted in Figure 1.3.8. Threads are almost similar to processes in many ways.

Similarities between Threads and Processes are :

(i) They also have almost the same states as processes - ready, running, blocked.
(ii) They also share CPU
(iii) Each thread has its own stack and program counter.
(iv) Threads can create child threads.
(v) If one thread is blocked, the other thread can run.
(vi) A thread (within a process) executes sequentially.

Dissimilarities between Threads and Processes are

(i) Unlike processes, threads are not independent of one another.
(ii) There is no concept of protection as in processes. All threads share the address space of a task and therefore, a thread can read/write over any other thread's stack.
(iii) Processes may originate from different users and may be hostile to one another whereas an individual task with multiple threads will always be owned by a single user.
(iv) It takes less time to create a new thread in an existing process than to create a new process.
(v) It takes less time to terminate a thread than to terminate a process.
(vi) It takes less time to switch between two threads within the same process than to switch between two processes.
(vii) Threads enhance efficiency in communication between different executing programs - since threads within the same process share memory and files, they can communicate with each other without invoking the operating system's kernel.

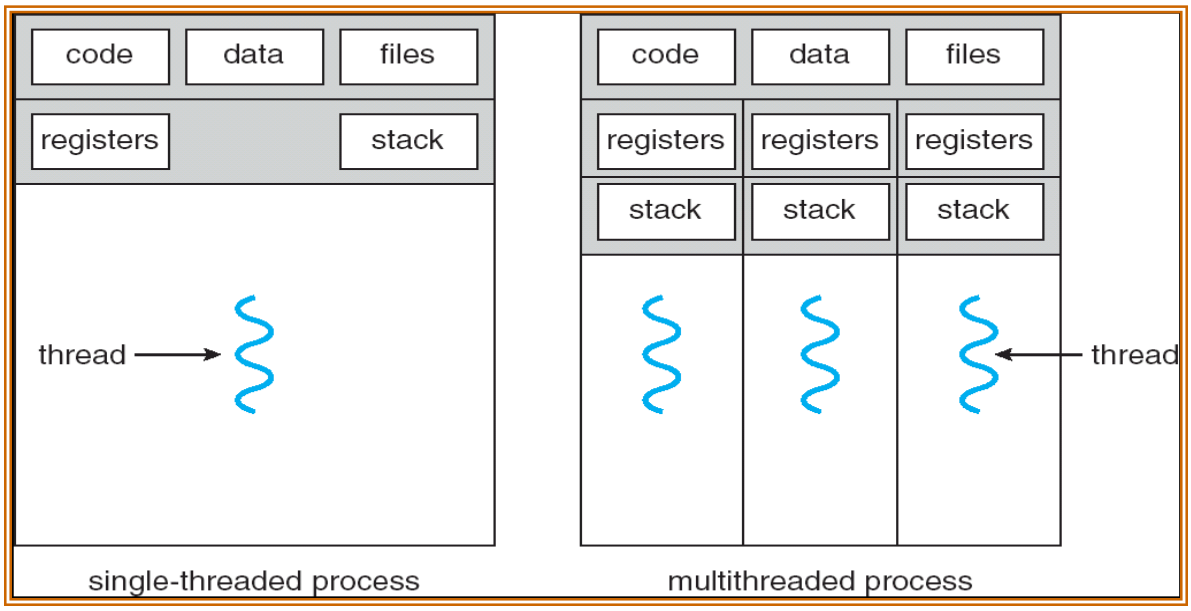


Fig 1.3.8: Threads and Processes

1.3.8 Keywords

Process: program in execution

Process Control Block: A block which contains information regarding the execution state of the program, memory management information and CPU scheduling information

Thread: A thread is a single sequential flow of control within a program

Object: Instance of a class

Cooperating process: which executes concurrently and affect or get affected by other processes executing in the system

Ready Queue: queue containing processes ready for their execution

Device Queue: queue containing processes waiting for their turn to access certain device

Long term scheduler: brings processes from secondary memory to the main memory

Short term scheduler: selects a process (from main memory) for execution. It is also known as CPU scheduler

Medium term scheduler: swaps out the process from main memory to a secondary storage and vice versa

1.3.9 Summary

A Process is a program in execution. Its execution proceeds in a sequential fashion. A process contains program counter, stack and data section. The processes which execute concurrently and affect or get affected by other processes executing in the system are called cooperating processes. Long term scheduler brings processes

from secondary memory to the main memory. Short term scheduler selects a process (from main memory) for execution. Medium term scheduler acts a swapper which swaps out the process from main memory to a secondary storage and vice versa. A thread is (or lightweight process) basic unit of CPU utilization; it consists of program counter, register set and stack space. A thread shares the following with its peer threads: code section, data section and OS resources. The objects are used in almost all application domains. Objects are therefore used in operating systems particularly for distributed computing environment.

1.3.10 Short Answer Type Questions

Q.1. What is the basic difference between a program and a process?
Q.2. What is the basic difference between a process and a thread?
Q.3. List the contents of process control block.
Q.4. Give an example other than operating systems where objects are used.
Q.5. Give two examples of multithreaded applications.
Q.6. List the situations when a process enters a ready queue.
Q.7. List the possible reasons of process termination and process creation.

1.3.11 Long Answer Type Questions

Q.1. What is a process? What are the operations defined on a process?
Q.2. What are the different states a process can be in at a particular time? What factors can lead to the transition of process states?
Q.3. List the differences among short-term, medium term and long term schedulers.
Q.4. What will you create (if you are given a choice out of a thread and a process) as a subset of a current process and why?
Q.5. What is the purpose of fork () and exec () system call in UNIX?
Q.6. What are cooperating processes? What support do you think the operating system should provide for the concurrent execution of processes?
Q.7. List the names of operating systems that support multithreading.
Q.8. Differentiate between single threaded and multithreaded system.
Q.9. What do you understand by the following terms?

(i) Swapping
(ii) Cascading termination
(iii) Synchronous process creation
(iv) Asynchronous process creation
(v) Ready Queue

Q.10. Justify that multithreading can take the best advantage of multiple processors.

1.3.12 Suggested Readings

1. Nutt Gary, “Operating Systems” Addison Wesley Publication, 2000.
2. Silberschatz and Galvin, “Operating System Concepts” Sixth Edition, Addison Wesley Publishing Co., 1999.

3. Ekta Walia, "Operating System Concepts" , Khanna Publishing Co., 2002
4. William Stallings, Operating Systems, Internals & Design Principles, 4th edition, Prentice-Hall, 2001.
5. Deitel H.M., "Operating Systems", 3rd Edition, Prentice Hall of India
6. Dhamdhare D.M., "Systems Programming and Operating Systems", Tata McGraw Hill, Second Edition,1999.
7. Shaw, "Logical Design of Operating Systems", Prentice Hall of India, 1978.
8. Andrew S. Tannenbaum, "Modern Operating Systems", Pearson Education Asia, Second Edition, 2001.
9. A.S. Godbole **"**Operating Systems" Tata McGraw Hill 2002.

Web Resources :

www.personal.kent.edu/rmuhamma/opsystems/os.html
www.wiley.com/colleges/silberschatz6e/0471417432/slides/slides.html

CPU Scheduling

Contents

1.4.0 Objectives

The objectives of this lesson are to discuss the various components of a CPU scheduler, their functions and working. The concept of scheduling mechanisms has been presented for the first time and various types of scheduling strategies have been discussed. The criteria for strategy selection have been laid down. Finally, various preemptive and non preemptive scheduling strategies have been discussed with examples. Scheduling strategies specific to multiprocessor environment have been described in detail.

1.4.1 Introduction

Before going on to understanding various CPU scheduling criteria and algorithms, let us first try to understand **what is CPU scheduling? It is the process of determining which processes will actually run when there are multiple runnable processes.** The next thing that comes to our mind is, is it really that important? Yes, it is. Because, it can have a big effect on the resource utilization and overall performance of the system.

By the way, world went through a long period (late 40's to early 90's) in which the most popular operating systems (DOS, Mac) had no sophisticated CPU scheduling algorithms. They were single threaded and ran one process at a time until the user directed them to run another process. But now as the operating systems became more sophisticated (e.g. Windows NT), they started supporting multiprogramming for maximizing CPU utilization. Multiprogramming environment led to the development and implementation of sophisticated CPU scheduling algorithms.

Basic assumptions behind most of the scheduling algorithms are:

* There is a pool of runnable processes contending for the CPU.
* The processes are independent and compete for resources.
* The job of the scheduler is to distribute the scarce resource of CPU to the different processes "fairly" and in a way that optimizes some performance criteria.

The behaviour of a process plays a key role in CPU scheduling. Let us now see, how do processes behave? It is depicted by CPU-I/O burst cycle. A process will normally run for a while (known as the CPU burst), perform some I/O, (known as the I/O burst) then run for a while more (the next CPU burst) and so on. The duration of interval between CPU burst and I/O burst depends upon the following:

* If the process is I/O bound - (A process is an I/O bound process if it performs lots of I/O operations as compared to other computation operations). Each I/O operation is followed by a short CPU burst to process the I/O. Thus, more I/O happens.
* If the process is CPU bound - (A process is known as CPU bound process if it performs lots of computation and has very less I/O operations to do), there will be long CPU bursts with I/O bursts of very short duration.

1.4.2. Scheduling Mechanisms

The scheduling mechanism is the part of the process manager that handles the removal of the running process from the CPU and the selection of another process on the basis of a particular strategy.

1.4.2.1 The process scheduler organization

The scheduler is responsible for multiplexing processes on the CPU. When it is time for the running process to be removed from the CPU, a different process is

selected from the set of the processes in the ready state. Then the selected process is allocated the CPU. The *scheduling policy* determines when it is time for a process to be removed from the CPU and which ready process should be allocated the CPU next. The *scheduling mechanism* determines how the process manager knows it is time to multiplex the CPU, and how a process is allocated to and deallocated from the CPU.

The scheduling mechanism is composed of several distinct parts, depending on exactly how it is implemented in any particular operating system. Fig 1.4.1 shows three conceptually different parts incorporated into a scheduler: the enqueuer, the dispatcher and the context switcher.

Enqueuer

When a process is changed to the ready state, the process descriptor is updated and the enqueuer places a pointer to the process descriptor into a list of processes that desire the CPU (called the ready list). Ready list is a queue containing processes in ready state. The enqueuer may compute the priority for allocating the CPU to the process when it is inserted into the ready list or the priority may be determined when the process is considered for removal from the ready list.

Dispatcher

This is another component involved in CPU scheduling. It is a program responsible for assigning the CPU to the process, which has been selected by the short-term scheduler. Assigning the CPU to a ready process involves three major steps: context switching, switching to user mode from monitor mode and restarting the execution of the process.

1. Context Switching – As the name indicates, it implies the switching of CPU from one process to another. It is done by context switcher. This involves saving the state of the old process (one that was running previously) and loading the saved state, if any, for this ready process (scheduled to be run now). This implies that there will be no saved state in case this ready process has been newly created and is being allocated the CPU for the first time. But if this ready process was previously in the blocked state (due to I/O etc.), then the process' registers and memory must be loaded with all the previously saved data.

In voluntary techniques, the context switcher is invoked by the running process that intends to release the CPU. In involuntary techniques, an interrupt causes the running process to be removed from the CPU, so it is the interrupt handler code that performs the context switch.

2. Switching to user mode from monitor mode - As we know, a user process must be run in user mode therefore; mode must be changed from monitor mode to user mode.

3. Restart the execution of process – The execution of process should be restarted by jumping to the instruction that was supposed to be executed when this process was last interrupted or to the first instruction if this ready process is to be executed for the first time after its creation.

Out of the three operations mentioned above, context switching consumes more time and therefore an effort should be made to minimize context switch time. The overall time taken by the dispatcher is the sum of time consumed in all these operations.

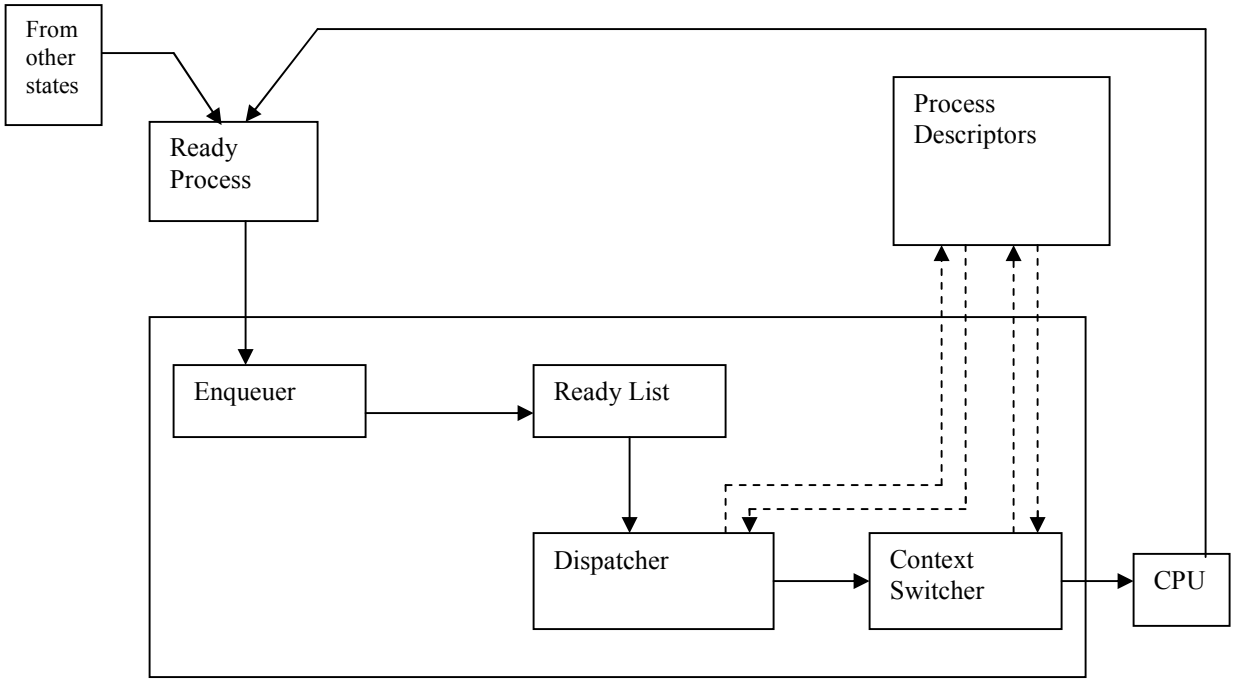


Figure 1.4.1**:** The Scheduler

1.4.2.2 Saving the process context

When the CPU is multiplexed, two pairs of context switches occur. In the first, the original running process has its context saved by the operating system and the dispatcher's context is then loaded. The second pair is for the dispatcher to be removed and the selected application process to be loaded onto the CPU. Context switching can significantly affect performance, since modern computers have a lot of general and status registers to be saved. A modern CPU, for example, incorporates 32 or more 32-or-64 –bit registers, plus status registers. The context switching part of the scheduler ordinarily uses normal software store and load operations to save the register contents. Here, a context switch requires $(n+m)bxK$ time units to save the state of a processor with n general registers and m status registers, assuming b store operations are required to save a single register and each store instruction requires K time units.

1.4.2.3 Voluntary CPU sharing

The CPU can be voluntarily relinquished by a process so that it can be shared among other processes also. Some hardware is designed to include a special *yield* machine instruction to allow a process to release the CPU. This instruction is similar to a procedure call instruction in that it saves the address of the next instruction to be executed and then branches to an arbitrary address. It differs from a procedure call instruction in that the address of the next instructions is not saved on the process's stack but in a designated memory location.

1.4.2.4 Involuntary CPU sharing

The interrupt system can enforce periodic involuntary interruption of any process's execution; that is, it can force a process to effectively execute a *yield* instruction. This is done by incorporating an interval timer device, which produces an interrupt whenever the timer expires. The basic operation of the interval timer is summarized by the *IntervalTimer* procedure given in Fig 1.4.2.

```
IntervalTimer()
{
InterruptCount = InterruptCount-1;
If(InterruptCount <= 0)
{
InterruptRequest = True;
InterruptCount = K
}
}

SetInterval(<programmableValue>)
{
K = <programmableValue>;
InterruptCount = K;
}
```

Figure 1.4.2 : The Interval Timer

The interval timer hardware completes an operation each time a real-time clock ticks- for example, for all T oscillations of a crystal. Setting the *InterruptRequest* variable represents the timer hardware's setting the interrupt request flag. The effect is that every KxT time units, the interrupt request flag will be set. The interval (K = number of ticks) can be specified by the *SetInterval* call shown in the fig 1.4.2, and it is thus called a *programmable interval timer*.

An interrupt will occur every K clock ticks, thus causing the hardware clock's controller to execute the logical equivalent of the *yield* instruction to invoke the interrupt handler. A scheduler that uses involuntary CPU sharing is called a ***preemptive scheduler.***

1.4.3 Types of scheduling strategies

As we already know, that there is a short-term scheduler which assigns a ready job to the CPU for its execution. Now, let us examine how will the scheduler do the job of assigning a process to the CPU? The scheduler will always switch the CPU to another process when one is busy with the I/O operation. This is actually the fundamental way in which multiprogramming should work. The basic aim that scheduler should achieve is that CPU should never be allowed to sit idle.

Another question that comes to our mind is when should the scheduler take the scheduling decisions? There are many possibilities like:

* When process switches from running to blocked state.
* When process switches from running to ready state.
* When process switches from blocked to ready state.
* When a process terminates.

Basically, there are two types of scheduling algorithms - Preemptive and Non-preemptive algorithms.

Non preemptive scheduling strategies - *Once the CPU has been allocated to a process, the process keeps the CPU until its termination or its transition to the blocked state.* This means that once CPU is allocated to a process, this process can use the CPU for its own execution till it willingly surrenders or leaves the CPU. FCFS scheduling, Deadline scheduling, shortest job first scheduling fall into this category.

Preemptive scheduling strategies- *Here, even if the CPU has been allocated to a certain process, it can be snatched from this process any time either due to time constraint or due to priority reasons.* It implies that if a process with higher priority becomes ready for its execution, the process which is currently using the CPU will be forced to give up the CPU so that higher priority job can run first. Round robin scheduling and priority scheduling strategies fall into this category.

Preemptive scheduling is costly as compared to Non-preemptive scheduling. For time constraints, time support is needed. Also when two processes share some data and one is in the midst of updating the data when the second process preempts it, then second process might access the inconsistent data left in between by the first process. For coordinating access to the shared data, special mechanisms must be provided by the operating system and this is again a costly affair.

Strategies like shortest job first, priority scheduling can be implemented as both preemptive and non preemptive strategies.

1.4.4 Strategy Selection

The possible parameters for determining “How good a scheduling algorithm is ?” are:

* **CPU utilization** - CPU should remain as busy as possible.

* **Throughput** - Throughput is defined as the number of processes that are completed per unit of time and it should be as high as possible. It is usually dependent on the job mix-i.e. the number of CPU bound and I/O bound processes. If all the processes are CPU bound, throughput will be less as compared to I/O based processes (which require CPU for less time and hence more of them can be finished in a given time). Thus, the two types of the jobs should be properly mixed to improve the throughput.
* **Turnaround time** - Turn around time is defined as interval between the time of submission and completion of the job and it should be as less as possible. It is a metric for batch systems.
* **Waiting time** - It is the sum of the time intervals for which the process has to wait in the ready queue. The CPU scheduling algorithm should try to make this time as less as possible for a given process.
* **Response time** - In an interactive system, response time is the best metric. It is defined as the time interval between the job submission and the first response produced by the job. Response time should be minimized in an interactive system.

As we have seen, most of these metrics are applicable to one process e.g., waiting time of a process, response time of a process etc. but actually, we have to consider the general performance of a scheduling algorithm. Therefore, we optimize the average measure. For example, average of waiting times of all the processes should be kept minimum and this can be taken to be one criterion to prove that the performance of a scheduling algorithm is good. On the other hand, the variance not average of the response time of the processes should be minimized, because the idea is not to have a process having very long response time as compared to all other processes.

After having discussed the various types of scheduling and the strategy selection, let us explore the scheduling strategies (algorithms) in detail.

1.4.5 Scheduling Strategies

1.4.5.1 First-come First-served scheduling algorithm (FCFS)

It is the simplest of all the scheduling algorithms. Key concept of this algorithm is "allocate the CPU in the order in which the processes arrive". It assumes that ready queue is managed as FIFO (First in first out). When the CPU is free, it is allocated to the process, which is occupying the front of the queue. Once this process goes into running state, its PCB is removed from the queue. This algorithm is non-preemptive.

Advantages of this algorithm are:

1. It is simple to understand and code.
2. Suitable for Batch Systems

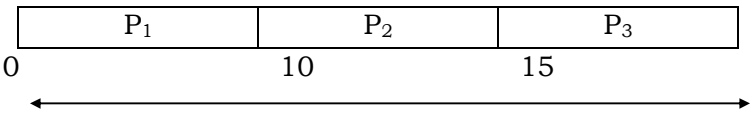
Disadvantages of this algorithm are:

1. Waiting time can be large if short requests wait behind the long ones.
2. It is not suitable for time sharing systems where it is important that each user should get the CPU for an equal amount of time interval.
3. A proper mix of jobs (I/O based and CPU based jobs) is needed to achieve good results from FCFS scheduling.

Example: Consider the following set of processes having their CPU-burst time (mentioned in milli-seconds). CPU-burst time indicates that for how much time, the process needs the CPU.

Process	**Time for which CPU is required CPU-Burst(Milli- seconds)**
P_1	10
P_2	5
P_3	5

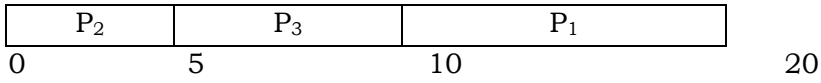
If the processes have arrived in order P_1, P_2, and P_3, then the average waiting time for the process will be obtained from the chart shown below :



Waiting time for P_1 = 0 milli-second
Waiting time for P_2 = 10 milli-seconds
Waiting time for P_3 = 15 milli-seconds

$$\text{Average waiting time} = \frac{0 + 10 + 15}{3} = \frac{25}{3} = 1.4.33 \text{ milli-seconds}$$

In case, the processes have arrived in the order P_2, P_3, P_1, then



$$\text{Average waiting time} = \frac{0 + 5 + 10}{3} = \frac{15}{3} = 5 \text{ milli-seconds}$$

Thus, average waiting time will always depend upon the order in which the processes arrive and will greatly vary depending upon whether the processes having less CPU-burst arrive first in the ready queue. Overall, this algorithm is never recommended wherever performance is a major issue.

1.4.5.2 Shortest -Job First Scheduling

Key concept of this algorithm is :

"CPU is allocated to the process with least CPU-burst time"

Amongst the processes in the ready queue, CPU is always assigned to the process with least CPU burst requirement. If there are two processes with same CPU burst, the one which arrived first, will be taken up first by the CPU. This algorithm can be either preemptive or non-preemptive.

Advantage of this algorithm is:

1. This is usually considered to be an optimal algorithm, as it gives the minimum average waiting time.

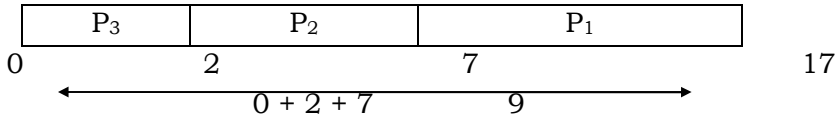
Disadvantage of this algorithm is:

1. The problem is to know the length of time for which CPU is needed by a process. A prediction formula can be used to predict the amount of time for which CPU may be required by a process.

Example: Let us consider, the following set of processes having their CPU-burst time (mentioned in milli-seconds) and having arrived almost at the same time.

Process	CPU-burst (Milli-seconds)
P_1	10
P_2	5
P_3	2

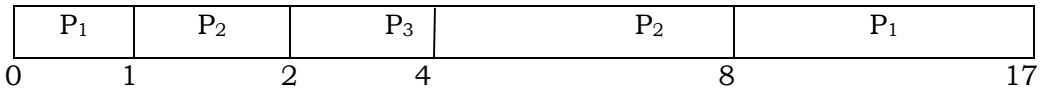
Using, the non-preemptive SJF scheduling, the waiting times for each of the processes will be as shown below :



Average waiting time = $\frac{0 + 2 + 7}{3} = \frac{9}{3}$ = 3 milli-seconds

If we consider that the processes arrived in the order P_1, P_2, P_3, and at the time mentioned below, then the average waiting time using preemptive SJF scheduling algorithm will be calculated as :-

Process	CPU-burst	Time of Arrival
P_1	10	0
P_2	5	1
P_3	2	2



In this, the CPU will be taken away from the currently executing process whenever a process with less CPU burst time (as compared to the time remaining

for the current process) arrives. As in example shown above, the time when process P_2, arrives, P_1 needs 9 milli-seconds more to finish. As P_2's CPU burst is 5 milli-seconds < 9 milli-seconds, therefore, P_1's execution will be preempted and P_2 will be executed but again as P_3 arrives P_2's execution needs 3 more milli-seconds where as P_3 needs only 2 milli-seconds to execute, thus P_3 takes over P_2 and so on.

Waiting time for P_1 = 0 + (8 – 1) = 7 milli-seconds

Waiting time for P_2 = 1 + (4 – 2) = 3 milli-seconds

Waiting time for P_3 = 2 milli-seconds

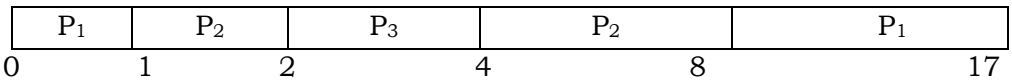
$$\text{Average waiting time} = \frac{7 + 3 + 2}{3} = \frac{12}{3} = 4 \text{ milli-seconds}$$

1.4.5.3 Priority Scheduling

Each process in the system if given a priority, then the scheduling must be done according to the priority of each process. A higher priority job should get CPU whereas lower priority job can be made to wait. Key concept of this algorithm is "Priority scheduling is necessarily a form of preemptive scheduling where priority is the basis of preemption". Some systems implement it is as non-preemptive scheduling also where a higher priority job is added to the front of the ready queue so that it can be executed next, but it cannot preempt the currently executing process.

Example: Let us consider, a set of processes P_1, P_2 and P_3 having priorities ranging from 1 to 3. Let us assume that 1 is the highest priority whereas 3 is the least priority. Let us also assume that P_1 arrives first and P_3 arrives in the last.

Process	Burst time	Priority	Arrival time
P_1	10	3	0
P_2	5	2	1
P_3	2	1	2



$$\text{Average waiting time} = \frac{0 + (4 - 1) + 1 + (4 - 2) + 2}{3}$$

$$= \frac{7 + 1 + 2 + 2}{3} = \frac{12}{3}$$

$$= 4 \text{ milli-seconds}$$

Here, the preemption is based on the priority. When P_1 executes, P_2 arrives with priority 2, which is higher than priority 3 of P_1 and thus P_1 is preempted. Similarly when P_2 executes, P_3 arrives with priority 1 which is the highest priority and thus P_2 is preempted and so on.

A natural question that arises here is - what if two processes have been assigned same priorities? (Priorities may be assigned by the user or by the operating system depending upon the importance of the process). The tie will be broken by comparing their CPU-burst times. The process with short CPU burst time will be executed first.

Problems with this scheduling algorithm are:

Preemptive priority scheduling some times becomes the biggest cause of starvation of a process - If a process is in ready state but its execution is almost always preempted due to the arrival of higher priority processes, it will starve for its execution. Therefore, a mechanism like aging has to be built into the system so that almost every process should get the CPU in a fixed interval of time. This can be done by increasing the priority of a low-priority process after a fixed time interval so that at one moment of time it becomes a high priority job as compared to others and thus, finally gets CPU for its execution.

1.4.5.4 Round - Robin Scheduling

All the above mentioned algorithms are good for non-interactive systems. What about the interactive systems, then? Another scheduling algorithm whose key concept is to give response to the users in a reasonable time is known as Round-Robin Scheduling Algorithm. The basic purpose of this algorithm is to support time-sharing systems. This algorithm is similar to FCFS algorithm but now it is a preemptive FCFS scheduling. The preemption takes place after a fixed interval of time called time quantum or time slice. Its implementation requires timer interrupts based on which the preemption takes place.

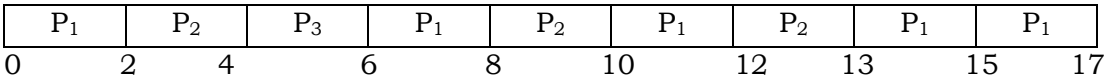
- Key concept of this algorithm is:

Consider a set of processes lined up in the ready queue. The processes are taken out of the ready queue in FCFS order. Let us consider that a process has been taken up for execution now. Keep in mind that the execution cannot go beyond the time-slice (implemented by timer interrupt). This process may either finish up its execution (if its CPU burst is less than time quantum) before the timer goes off or CPU will be preempted from it after the timer goes off and this causes an interrupt to the operating system. At this time, context switching will take place. The next process will be taken up from the ready queue (this process, which is left by the CPU, will be added to the tail of the ready queue now).

Let us first understand the concept of its working with the help of the following example.

Example: Let there be three processes P_1, P_2 and P_3, with their CPU burst times mentioned in milli-seconds. Let us assume a time sharing system in which all of the processes arrive almost simultaneously. Let the time quantum be of 2 milli-seconds.

Process	CPU Burst time
P_1	10
P_2	5
P_3	2



Waiting time for P_1 will be : = 0 + (6 – 2) + (10 – 4) + (13 – 12)

= 4 + 2 + 1 = 7 milli-seconds

Waiting time for P_2 will be : = 2 + (8 – 4) + (12 – 10)

= 2 + 4 + 2 = 8 milli-seconds

Waiting time for P_3 will be 4 milli-seconds

Thus, the average waiting time will be :

$$= \frac{7 + 8 + 4}{3} = \frac{19}{3}$$

= 6.33 milli-seconds.

As shown in the figure above, first P_1 gets the CPU and gets executed for 2 milli-seconds, then context switching occurs and P_2 gets CPU for 2 milli-seconds, then again context switching occurs and P_3 gets CPU for 2 milli-seconds, since its CPU burst time is 2 milli-seconds only, therefore, it completes its execution here and thus do not get the CPU again. P_1 and P_2 similarly continue to share the CPU in the same fashion till they are done.

As we have seen here, this algorithm is purely preemptive.

The performance of this algorithm depends upon many factors:

1. Size of time quantum
   - If time quantum's size is large then this algorithm becomes same as FCFS algorithm and thus performance degrades.
   - If time quantum's size is very small then the number of context switches increases considerably and the time quantum almost equals the time taken to switch the CPU from one process to another. Therefore, 50 percent of overall time spent in the execution of a set of processes will be due to context switching among the processes which is not desirable at all.

   Thus, the time quantum should not be very large and also should not be too small to achieve good system performance.

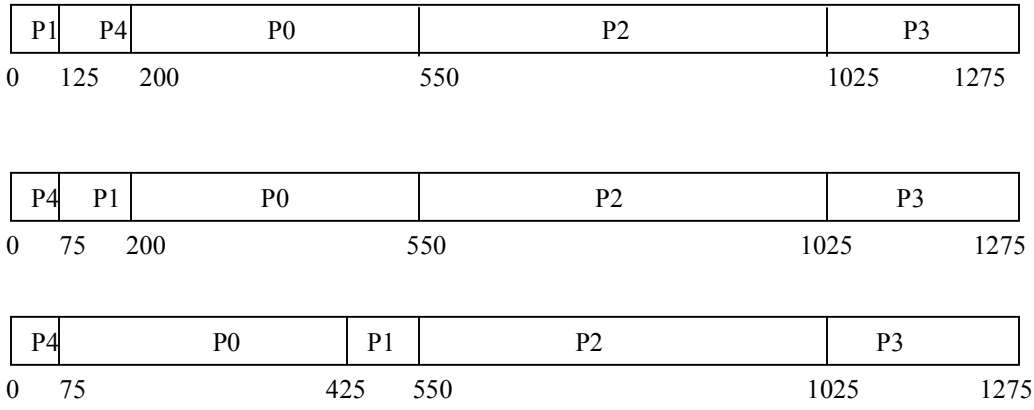
2. Number of context switches - As mentioned above, the number of context-switches should not be too many to slow down the overall execution of all the processes. Time quantum should be large with respect to the context switch time. This is to ensure that a process keeps CPU for a considerable amount of time as compared to the time spent in context switching.

1.4.5.5 Deadline Scheduling

Hard real time systems are often characterized as systems that have certain processes which must complete their execution prior to some deadline. The critical performance measure is whether the system was able to meet all such processes' scheduling deadlines; measures of turnaround and wait time are generally irrelevant. As a result, these schedulers must have very complete knowledge regarding how much time is required to execute each process(or part of process).A process can be admitted to the ready list only if the scheduler can guarantee that it will be able to meet each deadline imposed by the process.

In continuous media systems, the deadline may be required in order to prevent jitter(irregular delivery of information) and latency (delay) in the audio or video processing. In process control systems, the deadline may be established by an external sensor reading.

Example: Suppose our sample set of processes contained deadlines such as those shown below. There may be several different schedules satisfying the deadline. The middle schedule is an example of a strategy known as the earliest deadline first scheduling.



1.4.5.6 Multilevel Queue Scheduling

Key concept of this algorithm is "Multilevel Queue Scheduling was created for situations in which processes are easily classified into different groups".

Multilevel Queue Scheduling has the following characteristics:

- Processes are divided into different queues based on their types. For example, interactive (foreground) versus batch (background) system, these two types of processes have different response-time requirements.

In addition, processes have different priority. The foreground queue processes may have higher priority over background queue processes. In Multilevel Queue Scheduling, processes are permanently assigned to a particular queue.

- Each queue has its own scheduling algorithm. For example, interactive processes at the foreground may use the Round-Robin Scheduling methods while batch jobs at the background may use the FCFS method.
- There must be scheduling algorithm between the queues. Usually this is a fixed-priority preemptive scheduling. For example, the foreground queue may have absolute priority over the background queue. A percentage-based time sliced approach can also be used. Each queue gets a certain portion of the CPU time, which it can then schedule among the various processes in its queue. For instance, in the foreground-background queue examples, the foreground queue can be given 45 percent of the CPU time for Round Robin scheduling among its processes, whereas the background queue receives 15 percent of the CPU time to allocate to its processes in a FCFS manner.

Example: Let us look at an example of a multilevel queue scheduling algorithm with five queues as shown in figure 1.4.3

1. System processes
2. Interactive processes
3. Interactive Editing processes

1.4. Batch processes

5. Student processes

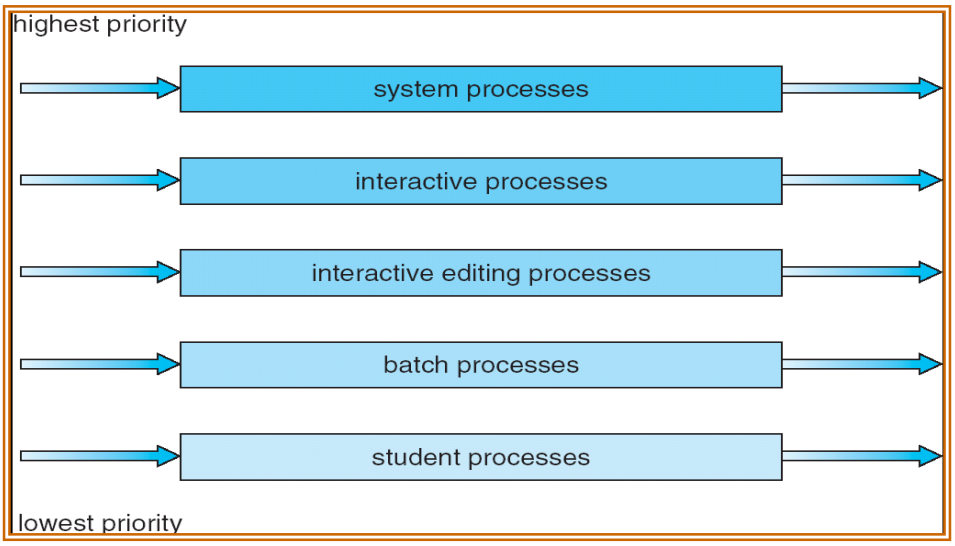


Figure 1.4.3: Multilevel queue scheduling

Here each queue has absolute priority over lower-priority queues. No process in the batch queue, for example, could run unless the queues for system processes and interactive processes were all empty. If an interactive process entered the ready queue while a batch process was running, the batch process would be preempted.

Advantages of this algorithm are:

1. In a Multilevel Queue algorithm, processes are permanently assigned to a queue on entry to the system. Since processes do not change their foreground or background nature, this set up has the advantage of low scheduling overhead.

Disadvantages of this algorithm are:

1. It is inflexible as the processes can never change their queues and thus may have to starve for the CPU if one or the other higher priority queues are never becoming empty.

1.4.5.7 Multilevel Feedback Queue Scheduling

Key concept of this algorithm is:

"It is an enhancement of multilevel queue scheduling where processes can move between queues". In this approach to process scheduling, the ready queue is partitioned into multiple queues of different priorities. The system tends to assign processes to queues based on their CPU burst characteristics: if a process consumes too much CPU time it is placed into a lower priority queue. Thus I/O-bound and interactive processes stay in the higher priority queues and CPU-bound processes gravitate to the lower priority queues. A technique called "aging" promotes lower priority processes to the next higher priority queue after a suitable interval of time.

In figure 1.4.4, the process queues are displayed from top to bottom in order of decreasing priority. The top priority queue has the smallest CPU-time quantum. After a process from the top queue exhausts this time quantum on the CPU, it is placed on the next lower queue. Similarly, a process, which consumes the larger CPU-time quantum of the next-to-highest priority queue, will be demoted to the third queue. The queue's time quota is scaled by a factor of 2 with each decreasing step in priority until the lowest priority queue is reached. This queue has a large time quantum which effects FCFS scheduling in contrast to the round robin scheduling on the other queues. We assume that a new process arriving on the top priority queue will preempt an active process originating from a lower priority queue and require it on its queue of origin. A process that spends an amount of elapsed time exceeding the aging interval will be promoted to the next higher priority queue.
How to define this algorithm?

The following things should be determined for the implementation of this type of algorithm:

- Number of queues
- Scheduling algorithm for each queue
- Method for upgrading a process to a higher-priority queue
- Method for downgrading a process to a lower-priority queue
- Method for assigning a process to a queue initially.

The advantages of Multilevel Feedback Queue Algorithm are:

1. Multilevel feedback queue scheduling allows a process to move between queues. This is fair for I/O-bound processes, they do not have to wait too long.
2. 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.
3. The definition of a multilevel feedback queue scheduling makes it the most general CPU scheduling algorithm. It can be configured to match a specific system under design.
4. It is more flexible.

The disadvantages of this algorithm are:

1. It requires some means of selecting values for all the parameters to define the best scheduler.
2. Moving the processes around the queues produces more CPU overheads.
3. It is the most complex scheduling algorithm.

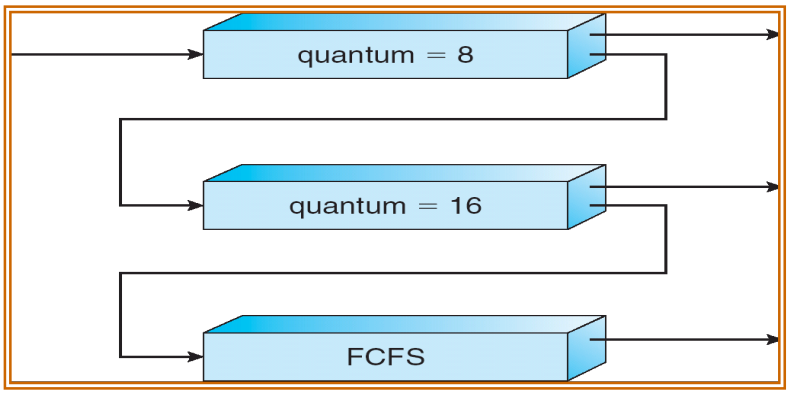


Figure 1.4.4: Multilevel feedback queues

Consider the following set of processes with their CPU-burst times and arrival times mentioned here:

Processes	Arrival time	Burst time
	Time in milli-seconds	
P_1	0	17
P_2	12	25
P_3	24	8
P_4	36	32
P_5	46	14

Process	Arrival time	CPU Burst	Waiting time
P_1	0	17	8
P_2	12	25	55
P_3	24	8	0
P_4	36	32	32
P_5	46	14	27

Consider a multilevel feedback queue scheduling with three queues, numbered as Q1, Q2, Q3. The scheduler first executes processes in Q1, which is given a time quantum of 4 milli-seconds. If a process does not finish within this time, it is moved to the tail of the Q2. The scheduler executes processes in Q2 only when Q1 is empty. The process at the head of Q2 is given a quantum of 16 milli-seconds. If it does not complete, it is preempted and is put into Q3. Processes in Q3 are run on an FCFS basis, only when Q1 and Q2 are empty.

A process that arrives in Q1 will preempt a process in Q2, a process that arrives in Q2 will preempt a process in Q3.

P1	P1	P2	P1	P2	P3	P4	P2	P5	P2	P4	P5	P2	P4
Q1	Q2	Q1	Q2	Q2	Q1	Q1	Q2	Q1	Q2	Q2	Q2	Q3	Q3

Round Robin with multilevel feed back queue scheduling is used in UNIX system.

1.4.6 Multiple - Processor Scheduling

All the algorithms, discussed here were to deal with the scheduling of one processor (CPU) among various processes in the system. What, if multiple CPUs are available in the system? Probably, a very different kind of scheduling mechanism needs to be incorporated. Without looking into the details of such scheduling mechanisms, it can be mentioned that such scheduling will be much more complex.

Terminology

Understanding multiprocessing, like understanding any other technology, requires a familiarity with some important terms. In Symmetric Multiprocessing (SMP) systems, CPUs share system resources, including the bus to a common memory space. CPUs communicate to one another by sharing information in this memory space. This setup is also referred to as a tightly coupled architecture.

Conversely, in Parallel Processing (PP) systems, each CPU has its own memory and often a unique set of system resources. As a result, CPU's must communicate externally over some type of network scheme. This setup is also referred to as a loosely coupled architecture.

One other way to classify multiprocessor systems is by the type of CPUs that are utilized. Heterogeneous systems use different types of CPUs, whereas Homogeneous systems use identical CPUs.

1.4.6.1 Static and Dynamic Scheduling

As we know, concurrent execution of a program on multiple processors used to be the sole domain of large, expensive supercomputers running complex scientific or financial applications. Since there were no general-purpose operating systems to manage task for these machines, the scheduling issues had traditionally been resolved before runtime during the program design phase. Set theory, task graphs, mathematical proofs and other forms of numerical analysis were used to determine the optimum method of allocating processors and ordering events. This form of static scheduling required prior knowledge of the program's behaviour and resource requirements, and was an expensive procedure in terms of clock cycles. Dynamic scheduling of tasks occurs at the time of execution, so it seeks to minimize this overhead. As a result, dynamic scheduling differs from static scheduling in its objectives as well as its operation.

1.4.6.2 Dynamic Scheduling Goals

Whereas the focus of static scheduling was primarily to minimize execution time, dynamic scheduling in a multiprocessor system seeks to :

- minimize execution time
- minimize communication delays between processors
- maximize resource utilization
- maximize processor utilization

Communication delays and resource utilization are primarily relevant in distributed systems. Most distributed systems are difficult to program due to the message passing that is required between processing elements. As a result, communication overhead can become a bottleneck.

Resource utilization becomes an issue in distributed heterogeneous systems, where different processors may have sole access to specific hardware, software, or data. The scheduler must keep track of all these resources and assign the tasks accordingly.

The majority of multiprocessor systems are used for their high performance, fault tolerance (redundancy in case of failure) and scalability (increased performance with additional processors). Each of these requires that the scheduler should maximize processor usage. This is done through a procedure called load balancing, where tasks are redistributed from heavily loaded to lightly loaded processors.

1.4.6.3 Performance Issues

In order for a multiprocessor system to run at all, a program must be divided into smaller steps that can run concurrently. Developing applications for multiprocessing computers requires that a lot of thought be put into this process. If the steps are small (fine grain tasks), the overhead for CPU communication and operating system intervention will severely degrade performance. However, if the

steps are large (coarse grain tasks), parallelism is reduced. The goal then is to find the right ratio for task granularity to maximize overall performance. As with static scheduling, there are many complex algorithms available to generate an ideal size of steps into which a process must be divided but the overhead of these operations makes it impractical for the operating system to incorporate them.

1.4.7 Keywords

Scheduling policy- it determines when it is time for a process to be removed from the CPU and which ready process should be allocated the CPU next.

Scheduling mechanism- it determines how the process manager knows it is time to multiplex the CPU, and how a process is allocated to and deallocated from the CPU.

Enqueuer- When a process is changed to the ready state, the process descriptor is updated and the enqueuer places a pointer to the process descriptor into the ready list.

Dispatcher- it is a program responsible for assigning the CPU to the process, which has been selected by the short-term scheduler.

Context switcher- when the scheduler switches the CPU from executing one process to executing another, it saves the contents of all processor registers for the process being removed from the CPU in its process descriptor.

Ready list- it is a queue of ready processes.

Voluntary CPU sharing- CPU is shared among the processes by their own wish.

Involuntary CPU sharing-CPU is shared among the processes involuntarily.

Throughput-it is defined as the number of processes that are completed per unit of time.

Turnaround time- it is defined as interval between the time of submission and completion of the job and it should be as less as possible.

Response time- it is defined as the time interval between the job submission and the first response produced by the job. Response time should be minimized in an interactive system.

Waiting time- it is the sum of the time intervals for which the process has to wait in the ready queue.

Non preemptive scheduling- here, once the CPU has been allocated to a process, the process keeps the CPU until its termination or its transition to the blocked state.

Preemptive scheduling- here, if the CPU has been allocated to a certain process, it can be snatched from this process any time either due to time constraint or due to priority reasons.

1.4.8 Summary

Scheduling is the heart of the CPU resource manager. The scheduler is responsible for multiplexing the CPU among a set of ready processes. It is invoked

periodically by a timer interrupt or any time that a running process voluntarily releases the processor through a *yield* or resource request. The scheduler selects from among a ready list of processes waiting to use the processor and then allocates the processor to the selected process.

Scheduling strategies can be divided into non-preemptive and preemptive strategies. Non-preemptive strategies allow a process to run to completion once it obtains the processor, while preemptive strategies use the interval timer and scheduler to periodically reallocate the CPU. FCFS, SJF, priority and deadline algorithms are well known non-preemptive algorithms, while RR and multi-level queue algorithms (along with preemptive variants of priority and SJF) are often used to implement preemptive approaches.

1.4.9 Short Answer Type Questions

Q.1. Define the term "scheduling mechanism".

Q.2. What are the different methods of strategy selection?

Q.3. What do you understand by CPU-bound and I/O bound processes? What is the meaning of CPU-burst time?

Q.4. Differentiate between preemptive and non-preemptive scheduling strategies.

Q.5. What is a dispatcher? What are its functions?

Q.6. What is an enqueuer? What are its functions?

Q.7. What is a context switcher? What are its functions?

Q.8 What are the disadvantages of FCFS scheduling algorithm as compared to shortest job-first (SJF) scheduling?

Q.9. What factors can lead to the degradation of performance of RR (Round and Robin) scheduling?

Q.10. With what kind of systems, deadline scheduling is associated?

Q.11. What is the purpose of *yield* machine instruction?

1.4.10 Long Answer Type Questions

Q.1. How is multilevel queue scheduling different from multilevel feed back queue scheduling? What are the advantages of multilevel feed back queue scheduling over multilevel queue scheduling?

Q.2. What is dynamic scheduling in multiprocessor system? How does it differ from static scheduling?

Q.3. Consider a system with a set of processes P_1, P_2, and P_3 and their CPU burst times, priorities and arrival times being mentioned as below :

Process	CPU burst time	Arrival time	Priority
P_1	5	0	2
P_2	15	1	3
P_3	10	2	1

Assume 1 to be the highest priority and calculate the following:

1. Average waiting time using FCFS, SJF (preemptive and non- preemptive) and priority (preemptive and non-preemptive) scheduling mechanisms.
2. Average turn around time using FCFS, SJF (preemptive and non-preemptive) and priority (preemptive and non-preemptive) scheduling mechanisms.
3. Assume time quantum to be 2 units of time. Calculate average waiting time and average turn around time using Round Robin Scheduling.

Q.4. List the advantages and disadvantages of FCFS, SJF and Round Robin scheduling strategies.

Q.5. How does the system implement voluntary CPU sharing and involuntary CPU sharing?

1.4.11 Suggested Readings

1. Nutt Gary, "Operating Systems" Addison Wesley Publication, 2000.
2. Silberschatz and Galvin, "Operating System Concepts" Sixth Edition, Addison Wesley Publishing Co., 1999.
3. Ekta Walia, "Operating System Concepts", Khanna Publishing Co., 2002.
4. William Stallings, Operating Systems, Internals & Design Principles, 4th edition, Prentice-Hall, 2001.
5. Deitel H.M., "Operating Systems", 3rd Edition, Prentice Hall of India.
6. Dhamdhare D.M., "Systems Programming and Operating Systems", Tata McGraw Hill, Second Edition,1999.
7. Shaw, "Logical Design of Operating Systems", Prentice Hall of India, 1974.
8. Andrew S. Tannenbaum, "Modern Operating Systems", Pearson Education Asia, Second Edition, 2001.

Web Resources :

www.personal.kent.edu/~rmuhamma/opsystems/os.html
www.wiley.com/college/silberschatz6e/0471417432/slides/slides.html

DEADLOCKS

Contents

1.5.0 Objectives

This lesson has been written to exclusively deal with the problem of deadlocks. As mentioned in the previous lesson, a state can arise in the system when one process waits for the other to release the required resources and the other process waits for the first one to release the required resources. Such a state is known as

Deadlock State. This lesson describes the methods to prevent and avoid deadlocks. Bankers' algorithm has been given in detail. Second half of this lesson deals with deadlock detection mechanisms and deadlock recovery. Algorithms for deadlock detection have been described with examples.

1.5.1 Problem

A deadlock is a situation in which two processes sharing the same set of resources are effectively preventing each other from accessing the resource, resulting in both programs ceasing to function.

The earliest computer operating systems ran only one program at a time. All of the resources of the system were available to this one program. Later, operating systems with multiprogramming support ran multiple programs at once, interleaving them. Programs were required to specify in advance what resources they needed so that they could avoid conflicts with other programs running at the same time. Eventually, some operating systems offered dynamic allocation of resources. Programs could request further allocations of resources after they had begun running. This led to the problem of the deadlock. Here is the simplest example:

Program 1 requests resource A and receive it.

Program 2 requests resource B and receive it.

Program 1 requests resource B and is queued up, pending the release of B by program 2.

Program 2 requests resource A and is queued up, pending the release of A by program 1.

The resources talked about here could be of the type memory space, CPU cycles, files and I/O devices (such as printers and tape drives etc.). Now neither program can proceed until the other program releases a resource. The operating system cannot know what action to take. At this point the only alternative is to abort (stop) one of the processes.

Learning to deal with deadlocks had a major impact on the development of operating systems and the structure of databases. Thus, data has to be structured and the order of requests has to be constrained to avoid deadlocks.

If there are multiple resources of the same type, each is called an instance of that resource type. A system table records whether or not a resource is free. So, if a request is made, a lookup operation is done in the table. If a requested resource is being used by some other process, then the requesting process is added to a queue for that resource. Thus, a queue is maintained for each resource.

1.5.2 A system deadlock model

Let P be a set of n different processes and R be a set of m different resource types, where c_j is the number of units of resource type R_j in the system. Next, how the resources can be allocated to the various processes must be defined. Since the allocation of resources to processes changes as the process executes, the model will

have to reflect each of these changes. A useful way to show this type of behaviour is to use a state diagram. So let a state in the model represent the pattern of resource requests and allocations that exist in the system at any particular moment –the model state represents the system state. In the system, the state changes when resources are requested, allocated or de-allocated, and in this model, state transitions occur to represent request, allocation and deallocation operations.

Let S be a set of states in the model that represents corresponding states in the system. The initial state is s_0; this represents the situation in which all resources are unallocated. All other states , s_i, represents possible system states, so each model state represents whichever units of every resource are being requested or held by each process. The details of how a specific state can be represented are considered when we examine individual deadlock strategies. For now, the focus is on the state transitions. The pattern in which resources are requested, acquired and de-allocated determines whether the system is deadlocked. This pattern corresponds to the list of transitions occurring in the state-transition model. All other activities of the processes are not relevant to the study of deadlock, so they are ignored in this model.

In a set of processes, P, any process, p_i belonging to P, might cause a state transition depending upon whether p_i

- Requests a resource (designated by a transition with label r_i)
- Is allocated a resource (designated by a transition with the label a_i), or
- De-allocates a resource (designated by a transition with label d_i).

Whenever the system is in state s_j belonging to S, an event x_i occurs, then the system's state changes to a new state, s_k belonging to S, due to occurrence of the event x_i.

Since we are deriving a model to represent system state changes, we are interested in the effect of a series of transitions that take the system from one state to another. Process p_i is blocked in s_j if p_i cannot cause a state transition out of s_j. In other words a blocked process is incapable of changing the state of the system, since it cannot cause any transition out of the current state. In fig 1.5.1, p_2 is blocked in state s_j because all the transitions out of state s_j are caused by other processes; none are caused by p_2. Any state transition is due to actions by p_1 or p_3 but not by p_2.

Even if p_i is blocked in s_j, some process other than pi might change the state of the system from s_j to a new state s_k, where p_i could proceed. If we can determine that there is no series of state transitions leading from the current state to one in which process p_i is unblocked, then the process can never execute again -- p_i is deadlocked in s_j. In other words, p_i is deadlocked in s_j if for every s_k that can be reached through some series of transitions from s_j, p_i is still blocked in s_k. If there is any process deadlocked in a state s_k, then s_k is called a deadlock state.

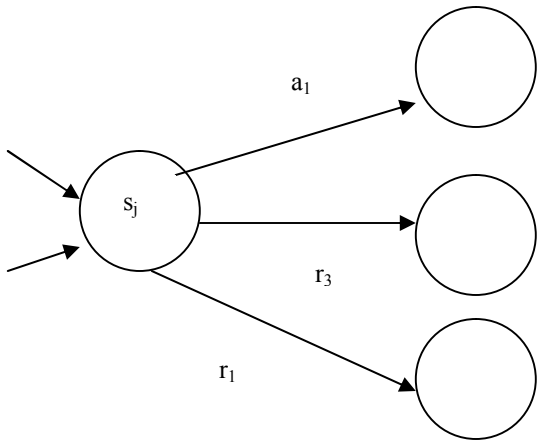


Fig 1.5.1: A state in which p_2 is blocked

1.5.3 Deadlock Characteristics

We can tell when a deadlock will occur. If these four conditions occur simultaneously, then we have deadlock:

1. **Mutual Exclusion:** a resource can be used by only one process at a time.
2. **Hold-and-wait:** at least one process is holding a resource, and needs to acquire other resources that are being held by other processes.
3. **No preemption:** If a process holds a resource, it can not be preempted (forced to release that resource).
4. **Circular wait:** There must exist a set $\{P_0, P_1, ..., P_n\}$ of waiting processes such that P_0 is waiting for a resource that is held by P_1. P_1 is waiting for a resource held by P_2 ... and so on ... and finally P_n is waiting for a resource held by P_0.

1.5.3.1 Recognition methods

A tool for recognizing deadlock is "Resource-allocation graph" (RAG). This graph has vertices representing the processes and the resources in the system. Its edges are of two types:

1. **Request edge:** This edge indicates the request of a process for acquiring a particular resource. It is a directed edge like $P_i \rightarrow R_j$. It indicates that process P_i has requested an instance of resource R_j.
2. **Assignment edge:** This edge indicates the allocation of a resource to a particular process. It is also a directed edge like $R_j \rightarrow P_i$. This edge here indicates that an instance of resource R_j has been allocated to the process P_i. Circles represent processes and rectangles represent resources. Figure 1.5.1 shows a typical resource allocation graph (RAG).

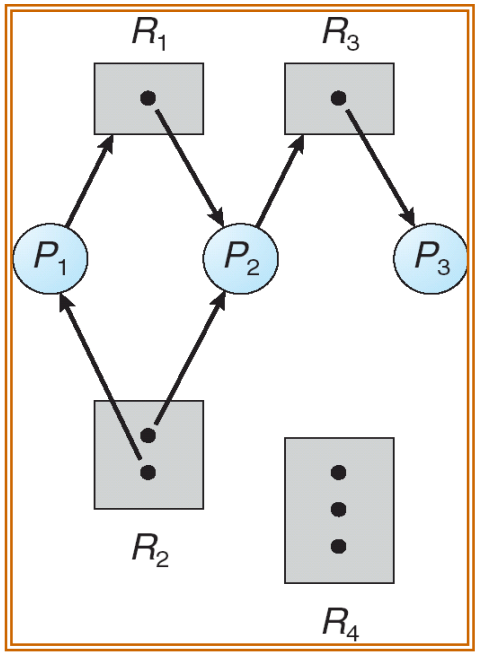


Figure 1.5.2: Example of a Resource Allocation Graph (RAG)

Notice the dots in rectangles of figure 1.5.2. Each is an instance of the resource.

- There is one instance of R_1
- There is two instances of R_2
- There is one instance of R_3
- There are three instances of R_4

Thus, in figure 1.5.2, P_1 is holding an instance of R_2 (since there is the assignment edge R_2 -> P_1) and is waiting for an instance of R_1. Similarly, P_2 is holding an instance of R_1 and is waiting for an instance of R_3, which is now held by P_3.

Can we determine if deadlock exists by using the RAG? Not necessarily, if a cycle exists in the RAG, there may or may not be deadlock. HOWEVER... it is sure that if no cycle exists (that is, if the graph is acyclic) then deadlock will not exist.

Therefore,

Acyclic RAG ⇒ no Deadlock

But it is not true that

No Deadlock ⇒ Acyclic RAG

Resource Allocation Graph with a Deadlock

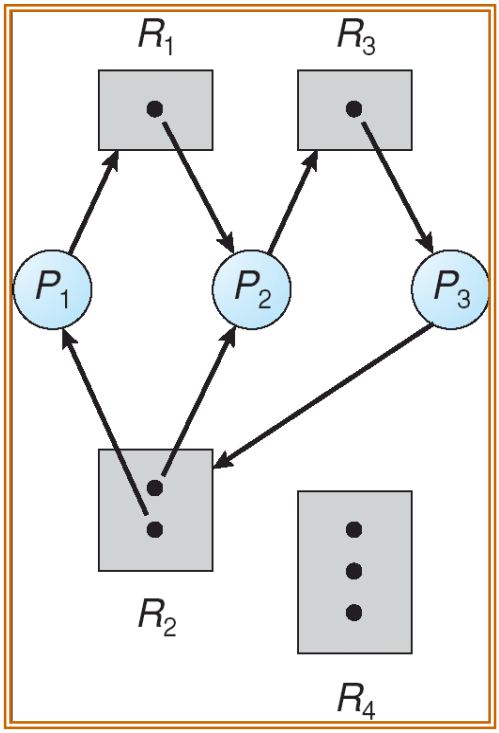


Figure 1.5.3: Deadlocked system

As mentioned above that an acyclic Resource Allocation Graph (RAG) always implies the non-existence of a deadlock whereas if the deadlock does not exist, in the system then, it is not necessary that there are no cycles in the Resource Allocation Graph. This means that cycles can be there even if there is no deadlock. Taking into consideration the number of instances of each resource type can further specify this condition.

Multiple Instances of Resources	One Instance of Every Resource
Cyclic RAG need not always imply Deadlock If there is a cycle in the RAG, it does not necessarily imply that a deadlock has occurred. Figure 1.5.3 shows that even though there is a cycle in the RAG, there is no deadlock.	Cyclic RAG ⇒ Deadlock If the cycle in RAG involves only a set of resource types, each of which has a single instance, then deadlock must have occurred.

Resource Allocation Graph with a Cycle but no Deadlock

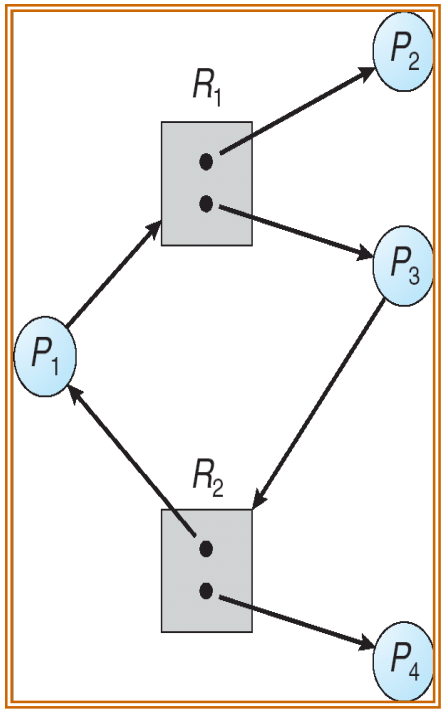


Figure 1.5.4: No Deadlock

In figure 1.5.4, notice that P_2 holds an instance of R_1, and P_4 holds an instance of R_2. Neither of these processes is waiting for a resource in the set R. P_2 and P_4 can release their resources. If they do then R_1 and R_2 will both have free instances, so there will be no deadlock, as those free instances can be assigned to P_1 and P_3, respectively, and the arrows will be reversed. The request edges will turn into assignment edges. Then the graph will be acyclic, and acyclic RAG $\Rightarrow$ no deadlock.

1.5.4 Dealing with Deadlocks

Once we have come to know that a deadlock exists in the system, we should now understand the methods of dealing with the deadlocks.

These methods are:

- **Deadlock prevention** - Do not let the deadlock occur.
- **Deadlock recovery** - Let the deadlock occur in the system and then attempt to recover the system from deadlock.
- **Do not do anything for deadlock prevention and recovery**. Let the deadlocks occur in the system and deteriorate the system performance, leading to a situation where system will have to be restarted manually.

1.5.4.1 Deadlock prevention

Deadlock can be prevented, if one of the necessary conditions (mentioned in section 1.5.3 of the text) is prevented from occurring. Let us see how we can prevent the occurrence of each of these conditions:

1.5.4.1.1 Mutual Exclusion

An effort should be made to prevent the mutual exclusion. This is possible by making all the sharable resources sharable. For example, if several processes would like to access a read-only file, then allow that to happen. No need for Mutual Exclusion in such a case. But few resources g cannot be changed from non-sharable to sharable. For example, if you have several processes that want to update a file, and you allow them to have simultaneous write privileges, then your data might be inconsistent. This is a condition where mutual exclusion is must. As mentioned in the previous chapter, mutual exclusion is one of the requirements of the critical section problem's solution also, thus preventing it may sometimes lead to undesired results. Therefore, overall conclusion is that denying mutual exclusion can create more problems than it solves.

1.5.4.1.2 Hold-and-wait

To prevent hold-and-wait condition from happening, we can have a rule that says, "a process may not request a resource if it is holding another resource". So, to take the print out of the contents of a file, you first request the disk, then you get it, use it and release it. Then you request the printer, you get it, you use it, and then you release it.

Thus, it implies that a process should have released all its resources before it requests for additional resources. Or another rule can be there that "a process should request and acquire all the requested resources before its execution begins". For example, to take the print out of the contents of a file, the disk and printer should be requested before hand.

In either case, there are two problems: -

1. **Low Resource Utilization -** If a process follows the second rule, it will acquire all the resources like disk, tape and printer at the very beginning of its execution, no matter that it might need the printer at the end of its execution. This will stop other processes from accessing the printer and thus printer utilization will be very low.
2. **Starvation -** A process that needs many resources to start its execution, may be waiting for one or the other resource (allocated to other processes) for an indefinite time.

1.5.4.1.3 No preemption

To ensure that this condition does not happen, preemption of the resources should be allowed. A rule like "if a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all the resources currently being held by it are preempted". The process will now have to wait for the allocation of resources that have been preempted in addition to those for which it was already waiting. This process is restarted only when it can regain its old resources as well as the new ones, it is requesting. This method can be applied only to the

resources whose state can be easily saved and restored later, such as CPU registers and memory.

1.5.4.1.4 Circular wait

In order to prevent circular waiting, resources can be numbered and it may be made compulsory that each process can request the resources in the increasing order of numbers. Suppose we have three resources namely, tape drive, disk drive and printer and suppose that the numbers allocated to them are 1, 2 and 6 respectively. A process, which wants to access disk drive and printer, should first request the disk drive and then printer. This rule can be extended to say that once a process has some resources assigned to it, it can acquire a new resource only if numbers of all its acquired resources is less than the number of the newly requested resource.

A process requesting disk drive numbered 2 will be allotted a disk drive only if all its existing resources have numbers less than 2. This implies that a process having a printer cannot acquire disk drive until it releases the printer (because printer's number 6 is greater than disk drive's number 2).

It can be proved that these rules ensure that circular wait do not hold in the system. The numbering of the resources should be done appropriately. Facts like, "tape drive will be needed before a printer is needed by a process" should be kept in mind while numbering the resources.

Thus, we have seen that by preventing one of these conditions, we can prevent the deadlock.

1.5.5 Avoidance Methods

Is there any other way to prevent the deadlocks? Yes, it is. Let us explore a deadlock - avoidance algorithm that ensures that circular-wait condition never happens in the system. This algorithm requires knowing maximum number of resources that each process needs to acquire. Based on this information, it can calculate what resources can be granted immediately and for which of them should the process wait?

Basically, this algorithm checks that after allocating the resources, will the system be in a safe state or not? A system is in safe state if the system can allocate resources to each process, in some order and still avoid a deadlock. This implies that the sequence in which the requests of the processes should be fulfilled, must be a safe sequence. Let us see what is a safe sequence? A sequence of process $<P_1, P_{2, \text{---}} .. P_n>$ is safe if the request of each process P_i can be granted from the pool of currently available resources and the resources held by the processes preceding it i.e. by P_j where $j<i$. In case P_i's request could not be satisfied, it can wait for those resources till all the processes preceding it (all P_j's) finish their job and release their resources. System is said to be in unsafe state, if no safe sequence exists. A deadlock state is a form of unsafe state. This implies, the moment it is ensured that system is in unsafe

state, there arises a probability of deadlock. Preventing the system from entering in an unsafe state can thus prevent deadlock.

Example : Consider that there are total 12 tape drives.

Process P_0 needs total 10, currently has 5 and thus needs 5 more

Process P_1 needs total 4, currently has 2 and thus needs 2 more

Process P_2 needs total 9, currently has 2 and thus needs 7 more

At time t_o, the system is in safe state, as a safe sequence $<P_1, P_0, P_2>$ exists. This implies that first grant 2 tape drives to P_1 (from the pool of 12 – (5+2+2) = 3 available tape drives) so that it has all the four necessary for its execution.

Then, when P_1 also releases its tape drives we will have (4 released by P_1 and 1 left in the pool), 5 available for meeting the requirement of other process. Now P_0 can be granted these 5 tape drives. After P_0 finishes, number of available tape drives will be 10. Now, 7 tape drives can be granted to P_2 for its execution. Thus, the system is in safe state. But it is in safe state at a time point t_o. It is not necessary that the system be in safe state for always. Whenever a new process enters the system, or when an existing process asks for more resources, the safe state can remain to be safe or can be changed to an unsafe state.

If the process P_0 here requests two more tape drive and is allotted two tape drives. Let us see whether the system is still in safe state or not? The status will be like:

Process P_0 needs total 12, currently has 5 and thus needs 7 more

Process P_1 needs total 4, currently has 2 and thus needs 2 more

Process P_2 needs total 9, currently has 2 and thus needs 7 more

Now first P_1's request can be easily granted as available tape drives are 12 – 9 = 3. Now P_1 releases tape drives and we have 5 available for allocation. These 5 cannot serve the purpose of P_0 (which needs 7 more and P_2 (which also needs 7 more). Thus, the system is in an unsafe state.

This is all that should be done by deadlock avoidance algorithm. It should be able to tell whether the system is left in the safe state or not, after the request of particular process is granted. If the algorithm finds that granting the request immediately will lead to an unsafe state, then the process must be asked to wait.

1.5.5.1 Banker's Algorithm

This algorithm's purpose is deadlock - avoidance. It considers each request and examines if it leads to a safe state. It does the following to examine the state of the system.

1. Check if there are enough resources to satisfy at least one process.
2. Sum the resources of the process to those available, marks the process and iterate.

3. If at the end there are processes that are left unmarked, the system would be in an unsafe state.

These steps ensure whether the system is in safe state or not? This part of the algorithm is known as safety algorithm. Another part of this algorithm ensures whether a process requesting a resource should be granted that resource immediately or should wait. This depends upon whether processing of this request leads to a safe or unsafe state? If granting the request leaves the system in an unsafe state, then the request should not be granted and the process must wait. This portion of the algorithm is called Resource Request Algorithm.

Data structures used in this algorithm are:

Let there be n processes and m resource types.

1. **Maximum** - Request matrix of the order of nxm. It indicates the maximum demand of each of the processes.
2. **Alloc** - Matrix of the order nxm. It indicates the number of resources currently allocated to each of the processes.
3. **Avail** - Vector of the order m. It indicates the number of resources (of each type) that are currently available.
4. **Need** - Matrix of the order nxm. If indicates the number of resources that a process may need in the future.
5. **Finish** - Vector of the order n. It contains the Boolean value (True or false) to indicate if a process has been allocated the required resources and has released all of its resources after using them.

The steps of safety part of this algorithm - that ensures whether a system is in a safe state or not are:

1. Initialization

   Set Finish (i) = False for all i ranging from 1 to n.
2. Compute avail and need

   Avail vector can be obtained directly from given conditions

   Set Need (i, j) = Maximum (i, j) - Alloc (i, j)

   for all processes i and for all resource type j.
3. Find a process i such that

   Set Finish (i) = False and need (i, j) $\leq$ Avail (j) for all resource types.

   (This implies that P_i needs the resources, which the system can easily grant as its need is less than available resources).

   If no such i exists, go to step 5.
4. Schedule process i for execution so that it can utilize the resources and then free up its resources by doing the following

   Set Avail (j) = Avail (j) + Alloc(i, j)

   Set Need (i, j) = 0 for all j

   Set Finish (i) = True

Go to step 3.

5. If finish (i) = True for all i, return (safe) else return (unsafe).

Resource - Request part of this algorithm, ensures whether the request of a process P_i on being granted, will leave the system in a safe state or not?

Let Request be a matrix of order nxm. It indicates the number of resources of each type requested by a particular process, at some moment of time.

The steps will be:

1. If Request (i, j) ≤ Need(i, j), for all resource type j, go to step 2 (here 'i' is a process which has made a request)
2. If Request (i, j) ≤ Avail (j), go to step 3. Otherwise process P_i must wait, since the resources are not available.
3. Assume that the requested resources have been allocated to process P_i. Do the following updations:-
   Set Avail (j) = Avail (j) - Request (i, j), for all j
   Set Alloc (i, j) = Alloc (i, j) + Request (i, j), for all j
   Set Need (i, j) = Need (i, j) - Request (i, j), for all j
4. Now apply the safety part of the algorithm, if the resulting state is safe, the transaction shown in step 3 should be completed otherwise, the transaction shown in step 3 should be roll backed and the process P_i must wait.

The Banker's algorithm has good results. This algorithm makes the following assumptions: -

1. The system has a limited number of resources.
2. Each process must use the resource for a finite amount of time.
3. Each process declares in advance, the amount of resources it will need.
4. A process may have to wait for a resource but is guaranteed to eventually get the resource.

An Example: Assume that there are five processes in the system and 4 resources namely. TD - Tape Drive, DD - Disk Drive, PR - Printer, SC - Scanner. Let the number of available resources be TD→1, DD→0,, PR→2 and SC→0.

Need matrix can be obtained as under:

Process	Resources Allocated				Maximum Number of Resources Required			
	TD	DD	PR	SC	TD	DD	PR	SC
P_1	3	0	1	1	4	1	1	1
P_2	0	1	0	0	0	2	1	2
P_3	1	1	1	0	4	2	1	0
P_4	1	1	0	1	1	1	0	1
P_5	0	0	0	0	2	1	1	0

Need Matrix

Process	TD	DD	PR	SC
P_1	1	1	0	0
P_2	0	1	1	2
P_3	3	1	0	0
P_4	0	0	0	0
P_5	2	1	1	0

The system is in safe state because the processes could finish in the order P_4 (which needs no more resources to finish), then P_1, P_2, P_3, and P_5. After P_4 finishes, Avail vector will have the value (2, 1, 2, 1). After P_1 finishes, Avail vector will have (5, 2, 3, 2). After that P_2 finishes to make avail as (5, 2, 3, 2) and finally P_3 finishes to make (6, 3, 4, 2) available. P_5's request is also easily granted and all are done! Thus, a safe sequence < P_4, P_1, P_2, P_3, P_5 > is obtained after the execution of Banker's safety algorithm. This ensures that the system is in safe state.

Now, suppose that a request from P_5 arrives for (1, 0, 1, 0), Can the whole request be safely granted immediately?

Yes, the resulting state is safe if P_5 's request is granted, and therefore, the request should be immediately granted. How do we determine this? As Request of P_5 (1, 0, 1, 0) ≤ need of P_5, which is (2, 1, 1, 0). We move on to check whether Request i.e. (1, 0, 1, 0) is ≤ Avail, which is (1, 0, 2, 0). This is also true. Then, we try to simulate allocating the requested resources to P_5 and find out a safe sequence for the system. This makes Avail to be (0, 0, 1, 0), Alloc to contain (1, 0, 1, 0) for process P_5 and need of P_5 to contain (1, 1, 0, 0). Thus, < P_4, P_1, P_2, P_3, P_5 > is a safe sequence that can be obtained by applying the algorithm. Therefore, granting the request of process P_5 will leave the system in a safe state.

1.5.6 Deadlock Recovery

In case the system takes no measures to prevent and avoid the deadlock, then at some time deadlock may occur in the system. In order to recover from the deadlock state, the first thing that should be done is deadlock detection. There should be mechanisms to detect whether a deadlock exists in the system and if yes, methods to recover the system from such a state should be known.

1.5.6.1 Deadlock detection

There are two algorithms for deadlock detection. These algorithms periodically determine whether a deadlock has occurred in the system? The first algorithm deals with the case where there is only one instance of each resource type and the other algorithm deals with the situation where there are multiple instances of each resource type.

1.5.6.1.1 Deadlock detection algorithm for single instance of each resource type

1. Maintain a wait -for- graph: Nodes in this graph represent processes. If process i is waiting for a resource held by process j, then there is an edge from i to j.
2. Periodically invoke an algorithm that searches for cycles in the graph. Complexity of algorithm, which detects cycle in a graph, will be n^2, where n is the number of vertices in the graph. If there is a cycle in the wait-for-graph a deadlock is said to exist in the system.

Figure 1.5.5 shows a wait-for-graph, where P_1 is waiting for a resource held by P_2. P_2 is waiting for a resource held by P_3 as well as P_4 and P_3 is waiting for a resource held by P_4 which is further waiting for P_1. There are two cycles in the graph, (between P_1, P_2 and P_4) and also (between P_1, P_2, P_3 and P_4), and thus the processes are deadlocked.

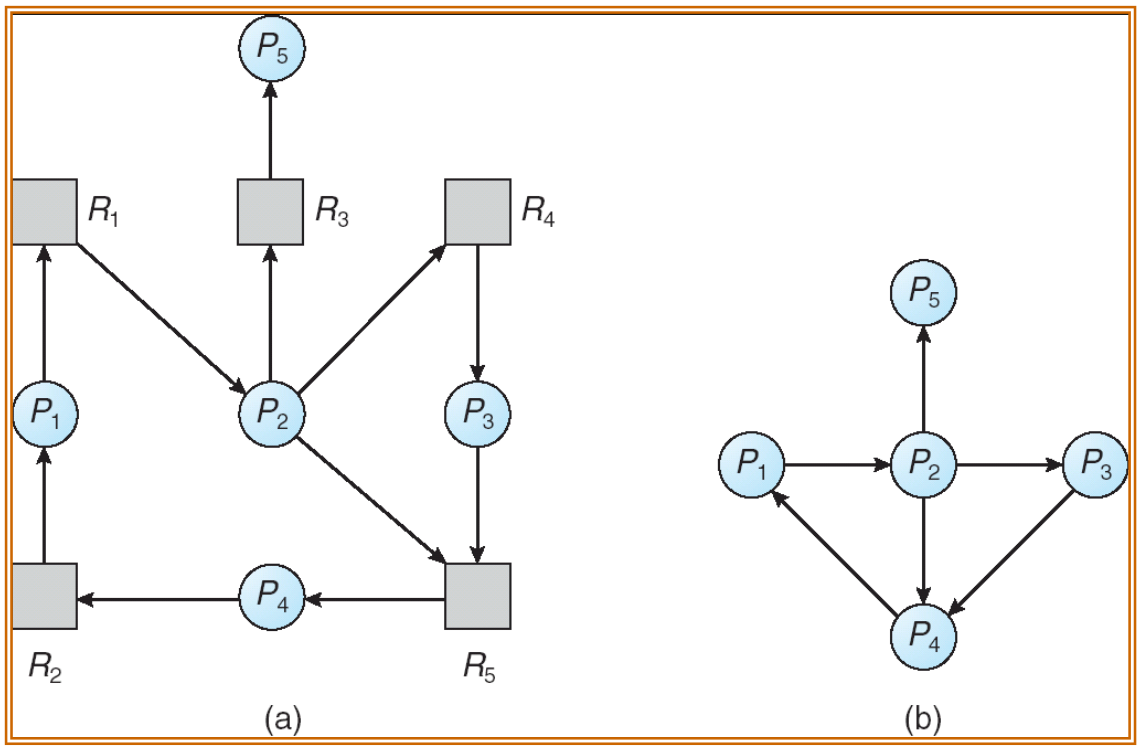


Figure 1.5.5: Wait-for-graph

What can you say about the system whose wait-for-graph is shown in Figure 1.5.5. Is the system dead locked?

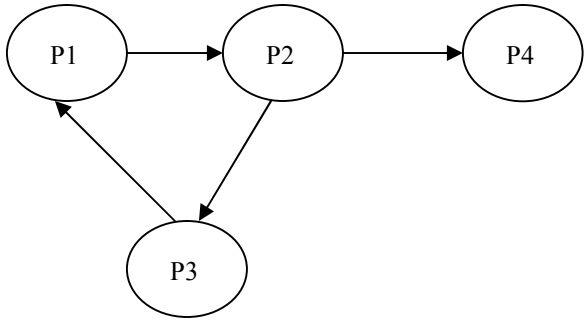


Figure 1.5.6 : Wait-for-graph of a system

Yes, the system is in deadlock state as it contains a cycle between P_1, P_2 and P_3. The processes involved in the deadlock are P_1, P_2 and P_3.

1.5.6.1.2 Deadlock detection algorithm for a multiple instances of each resource type

This algorithm is almost the same as safety part of the Banker's algorithm with few semantic differences. It uses almost the same data structures as Banker's algorithm. As mentioned in the previous section, Avail, Alloc, Finish and Request are the data structures used in Banker's algorithm and here also m is the number of resources and n is the number of processes in the system. Let us go through the steps of this algorithm.

Step 1 If Alloc (i, j) ≠ 0, then
Set Finish (i) = false;
else
Set Finish (i) = True;
for all processes i = 1 to n
and all resources j = 1 to m.

Step 2. Find a process i such that both the conditions
Finish (i) = false
and Request (i, j) ≤ Avail (j)

(This means that request of process i for all resources j should be less than the available amount of resources j)

if no such i exists, go to step 4.

Step 3. Set Avail (j) = Avail(j) + Alloc (i, j)
Set Finish (i) = True;
Go to step 2.

Step 4 If Finish (i) = False, for some process, then the system is in deadlock state. The process i for which Finish (i) = False, is deadlocked.

This algorithm eliminates the consideration of those processes (which do not hold a resource) as participants in the deadlock state of the system. This implies that if a process i do not hold any resource at this time, it will not contribute towards deadlocking the system. Therefore, it is assumed in the step 1 of the algorithm and such processes are not taken into consideration any further.

Example

Process	Allocation (Alloc)			Request			Available (Avail)		
	TD	DD	PR	TD	DD	PR	TD	DD	PR
P_0	0	1	0	0	0	0	0	0	0
P_1	2	0	0	2	0	2			
P_2	3	0	3	0	0	0			

Now, we can see that in this instance the system is not deadlocked. By our algorithm, safe sequence for this set of processes can be < P_0, P_2, P_1 >. Now suppose that process P_2 makes an additional request for printer. If the Request matrix now becomes:

Process	Allocation (Alloc)		
	TD	DD	PR
P_0	0	0	0
P_1	2	0	2
P_2	0	0	1

Now, let us execute our algorithm again. Process P_0 whose request is less than equal to the available vector, avail, can be allowed to finish first. After P_0 executes, avail vector will be (0, 1, 0). Now neither P_1, nor P_2 can be allowed to finish, as its request is not less than equal to available. Thus for P_1 and P_2, finish is still false, and therefore, P_1 and P_2 are deadlocked.

How often should detection - algorithm be used?

- In the extreme case, this algorithm can be executed every time, a new request (for a resource) arrives in the system. This frequency of invoking the algorithm will enhance the overall computation time.
- Another possible way is to invoke the algorithm periodically after some period of time. This period can be short enough to detect which particular process's new request has caused deadlock. This information (that which process caused deadlock) can be helpful for recovering the system from deadlock. Another possibility is that this period can be long (so that cost-factor becomes less), but once deadlock is detected after a long time, it will be difficult to know the exact cause of this deadlock and many processes may have to be restarted altogether.

1.5.6.2 Recovery from deadlock

Once the deadlock has been detected in the system, the system should now be recovered from this state. This is must because it will continuously degrade the

system's performance and if no correction mechanism is applied, there is a possibility of total system failure, some time later. Let us look into the options of recovering the system from deadlock state. One option is process termination and other is resource preemption.

1.5.6.2.1 Process termination

One way is to abort all the dead locked processes. This technique is fast but a lot of the work done by all the deadlocked processes will be lost.

Another way is to abort one deadlocked process at a time and check for deadlock's presence again. This technique involves more work to resolve a deadlock. Because, choosing the order in which processes should be aborted is a very complicated job. Also, after aborting a process, detection algorithm will have to be invoked again and so on.

The advantage of this method is that there is a possibility that not all the deadlocked processes will have to be aborted and thus, not the work done by all these processes will be lost.

1.5.6.2.2 Resource preemption

This technique favours the preemption of resources from the processes and allocation of these resources to other processes till the deadlock cycle is broken. There are three issues pertaining to resource preemption: -

(i) Which resources and which process should become the target of preemption? If a process has completed almost 95 percent of its execution, then it should not be our choice for preemption. Other factor can be the number of resources held by a process.

(ii) What should be done with the preempted process? One possibility is to roll back this process to some earlier safe state and restart its execution from that point. This again involves lots of effort on the part of the system, as it may have to keep more information about the state of each process.

(iii) How to ensure that starvation will not occur in the system? This means that a mechanism must be there which looks into the fact that a particular process should not be deprived of resources again and again i.e. it should not always become the target of resource preemption. A counter can be associated with each process to count the number of times its resources have been taken away. This information should be used by the deadlock recovery mechanism so that count of preemption also becomes a factor in deciding which process should become the next target of preemption. An approach can be to select a process having this counter's value 'least'.

1.5.7 Keywords

Deadlock- it is a situation in which two processes sharing the same set of resources are effectively preventing each other from accessing the resource, resulting in both the

programs to stop proceeding any further. More than two processes can be deadlocked in system.

Hold and wait- it is a situation in which a process holds one resource and waits for acquisition of another resource.

Deadlock prevention- Deadlock can be prevented, if one of the necessary conditions for occurrence of deadlock is prevented from occurring.

Deadlock avoidance- it is to follow certain rules in resource allocation so that deadlock cannot occur.

Deadlock detection- it is to detect the presence of a deadlock in the system.

Deadlock recovery- Let the deadlock occur in the system and then attempt to recover the system from deadlock

Circular wait- existence of a set $\{P_0, P_1, ..., P_n\}$ of waiting processes such that P_0 is waiting for a resource that is held by P_1. P_1 is waiting for a resource held by P_2 ... and so on ... and finally P_n is waiting for a resource held by P_0.It should be avoided to prevent deadlock.

Resource allocation graph- it is a tool for recognizing deadlock. A cycle in this graph indicates the presence of deadlock if there is single instance of each resource type.

Wait-for-graph- Nodes in this graph represent processes and if a process i is waiting for a resource held by another process j, then there is an edge from i to j. It is also used for deadlock detection.

Mutual Exclusion- a resource can be used by only one process at a time.

Unsafe state- If the current state of the system may lead to deadlock, it is said to be in unsafe state**.**

1.5.8 Summary

Deadlock creates a situation in which one or more processes will never run to completion without recovery. It can be prevented by designing resource managers so that they violate at least one of the four necessary conditions for deadlock: mutual exclusion, hold and wait, circular wait and no preemption. Deadlock can be detected by resource allocation graph if there is single instance of each resource type. Safety algorithm can be used to detect deadlock in a system having multiple instances of each resource type.

Prevention can be effective on batch systems but it is usually not practical in timesharing or other interactive systems. The process-resource state model provides a framework for defining deadlock independent of the strategy chosen to address deadlock. Deadlock can be avoided by using additional information, such as each process's maximum claim on each resource type, so as to not put the system into an unsafe state**.** The Banker's algorithm is the classic avoidance algorithm. It is also used to determine if an allocation operation will lead to a state in which the resource manager cannot guarantee that deadlock will not occur. Deadlock recovery options are process termination and resource preemption.

1.5.9 Short Answer Type Questions

Q.1. What are the necessary conditions for the occurrence of a deadlock?

Q.2. What does a cycle in the Resource Allocation Graph (RAG) indicate:

a) When there is a single instance of each resource type

b) When there are multiple instances of each resource type

Q.3. Is there a possibility of deadlock in a system with only one process?

Q.4. Prove that the safety algorithm part of the Banker's algorithm requires operations of the order mxn^2.

Q.5. How can starvation of a particular process be detected by a system?

Q.6. List the methods of deadlock recovery.

Q.7. When is a system in safe state?

1.5.10 Long Answer Type Questions

Q.1 Consider a system consisting of m resource types being shared by n processes. Resources can be requested and released by processes only one at a time. Show that the system is deadlock free if one the following condition holds: -

1. The maximum need of each process is between 1 and m resources.
2. The sum of all maximum needs is less than m + n.

Q.2. Consider the following snapshot of a system.

Process	**Allocation**			**Maximum Requirement**		
	A	B	C	A	B	C
P_0	0	1	0	7	5	3
P_1	2	0	0	3	2	2
P_2	3	0	2	9	0	2
P_3	2	1	1	2	2	2
P_4	0	0	2	4	3	3

Let the available number of resources be given by avail vector as (3, 3, 2).

Use the Banker's algorithm to answer the following

1. Find out the contents of the matrix 'Need'?
2. Is the system in a safe state?
3. If a request from process P_4 for (3, 3, 0) arrives, can it be granted immediately?

Q.3. Explain with the help of an example, how can you prevent deadlock by preventing circular-wait condition.

Q.4. Explain how a deadlock can be represented graphically for two processes and two resources.

Q.1.5. Discuss the merits/demerits of two ways in which the operating system can recover from a deadlock.

Q.6. How often should deadlock detection - algorithm be used?

1.5.11 Suggested Readings

1. Nutt Gary, "Operating Systems" Addison Wesley Publication, 2000.
2. Silberschatz and Galvin, "Operating System Concepts" Sixth Edition, Addison Wesley Publishing Co., 1999.
3. Ekta Walia, "Operating System Concepts" , Khanna Publishing Co., 2002
4. William Stallings, Operating Systems, Internals & Design Principles, 4th edition, Prentice-Hall, 2001.
5. Deitel H.M., "Operating Systems", 3rd Edition, Prentice Hall of India
6. Dhamdhare D.M., "Systems Programming and Operating Systems", Tata McGraw Hill, Second Edition,1999.
7. Shaw, "Logical Design of Operating Systems", Prentice Hall of India, 1978.
8. Andrew S. Tannenbaum, "Modern Operating Systems", Pearson Education Asia, Second Edition, 2001

Web Resources :

www.personal.kent.edu/~rmuhamma/opsystems/os.html
www.wiley.com/college/silberschatz6e/0471417432/slides/slides.html

Print Setting by Department of Distance Education
Punjabi University, Patiala