computers entire processes are in memory (albeit virtual memory) and the computer switches between executing code in each of them. In other types of systems, such as airline reservation systems, a single application may actually do much of the timesharing between terminals. This way there does not need to be a different running program associated with each terminal.

Many universities and businesses have large numbers of workstations tied together with local-area networks. As PCs gain more sophisticated hardware and software, the line dividing the two categories is blurring.

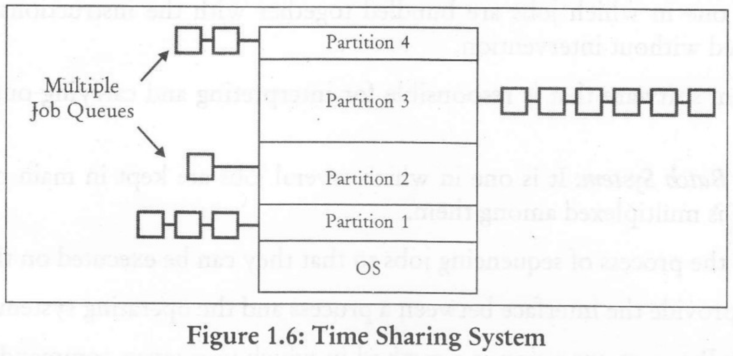

Figure 1.6: Time Sharing System

Check Your Progress

1. What are three two of problems caused by errors in operating system?
2. Name four services of provided by operating system.
3. Explain the categories of system software.
4. True or False:
   (i) Users programme cannot control I/O service.
   (ii) A process needs to communicate only with OS.
   (iii) OS provides service to manage the primary memory only.

1.11 LET US SUM UP

An operating system is a layer of software which takes care of technical aspects of a computer's operation. It shields the user of the machine from the low-level details of the machine's operation and provides frequently needed facilities. There is no universal definition of what an operating system consists of. Modern personal computer systems usually feature a Graphical user interface (GUI) which uses a pointing device such as a mouse or stylus for input in addition to the keyboard. A distributed system is a computer system in which the resources resides in separate units connected by a network, but which presents to the user a uniform computing environment. A real-time operating system (RTOS) is a multitasking operating system intended for real-time applications. Such applications include embedded systems (programmable thermostats, household appliance controllers, and mobile telephones), industrial robots, spacecraft, industrial control and scientific research equipment. A batch system is one in which jobs are bundled together with the instructions necessary to allow them to be processed without intervention. The monitor is system software that is responsible for interpreting and carrying out the instructions in the batch jobs. In multiprogramming batch system several jobs are kept in main memory at the same time, and the CPU is multiplexed among them.

1.12 KEYWORDS

Operating System: An operating system is a layer of software which takes care of technical aspects of a computer's operation.

Batch system: It is one in which jobs are bundled together with the instructions necessary to allow them to be processed without intervention.

Monitor: It is system software that is responsible for interpreting and carrying out the instructions in the batch jobs.

Multiprogramming Batch System: It is one in which several jobs are kept in main memory at the same time, and the CPU is multiplexed among them.

Job scheduling: It is the process of sequencing jobs so that they can be executed on the processor.

System calls: They provide the interface between a process and the operating system.

Program Execution: Program execution is a method in which user given commands call up a processes and pass data to them

I/O Operations: It refers to the communication between an information processing system and the outside world - possibly a human, or another information processing system.

File System Manipulation: Creation, deletion, modification or updation of files is known as File System Manipulation.

Process Communication: A processes need to communicate with other process or with the user to exchange the information, this is known as Process Communication.

Error Detection: This is a process where the operating system constantly monitors the system for detecting the malfunctioning of it.

1.13 QUESTIONS FOR DISCUSSION

1. What is an operating system? Is it a hardware or software?
2. Mention the primary functions of an operating system.
3. What is batch system? What are the shortcomings of early batch systems? Explain it.
4. Briefly explain the evolution of the operating system.
5. What are the key elements of an operating system?
6. Write the differences between the time sharing system and distributed system.
7. What do you understand by the term Computer Generations?
8. What is the difference between system software and application software?
9. Operating system acts as resource manager. What resources does it manage?

Check Your Progress: Model Answers

1. System crashes and instabilities and security flaws
2. I/O Operations, error detection, communication, program execution
3. Operation system and language translator
4. (i) True
   (ii) False
   (iii) False

1.14 SUGGESTED READINGS

Andrew S. Tanenbaum, *Modern Operating System*, Published By Prentice Hall

Silberschatz Galvin, *Operating System Concepts*, Published By Addison Wesley

Andrew M. Lister, *Fundamentals of Operating Systems*, Published By Wiley

Colin Ritchie, *Operating Systems*, Published By BPB Publications

I.a dhotre, *Operating System*, Technical Publications.

LESSON

2

PROCESS DESCRIPTION AND CONTROL

CONTENTS

2.0 AIMS AND OBJECTIVE

After studying this lesson, you should be able to:

- The role of a process the operating systems
- Explain process status, description and control
- Explanation and utilization of the power of threads
- Concept of interprocess communication

2.1 INTRODUCTION

Computers, in the past, allowed only one program to be executed at a time, i.e. they were strictly single tasking systems. The executing program used to have all the resources available to it. This practice caused wastage of valuable CPU cycles. Most of the time, the CPU sat idle without doing anything useful. To harness the CPU capabilities by preventing the wastage of its cycles, multi-tasking systems were developed. Thus, the concept of process management evolved. One definition of a process is that it has an address space and a single thread of execution. Sometimes it would be beneficial if two (or more) processes could share the same address space and run parts of the process in parallel. This is what threads do. Firstly, let us consider why we might need to use threads. Assume we have a server application running. Its purpose is to accept messages and then act upon those messages. Consider the situation where the server receives a message and, in processing that message, it has to issue an I/O request. Whilst waiting for the I/O request to be satisfied it goes to a blocked state. If new messages are received, whilst the process is blocked, they cannot be processed until the process has finished processing the last request. One way we could achieve our objective is to have two processes running. One process deals with incoming messages and another process deals with the requests that are raised. However, this approach gives us two problems

1. We still have the problem, in that either of the processes could still become blocked (although there are way around this by issuing child processes)
2. The two processes will have to up date shared variables. This is far easier if they share the same address space.

2.2 PROCESS DESCRIPTION

An operating system provides an environment, which presents each hardware computing resource in abstract form. This representation hides the unwanted details from the programmers and allows them to view the resources in the form, which is convenient to them. A process is an abstract model of a sequential program in execution. It forms a schedulable unit of work. It is identifiable object in the system having following components:

- The object program (or code) to be executed.
- The data on which the program will execute (obtained from a file or interactively from the user of the process).
- The status of the process execution.

A process may be represented schematically as in Figure 2.1.

In contrast, a program is a passive entity sitting on some secondary storage device. Whereas a process includes, besides instructions to be executed, the temporary data such as subroutine parameters, return addresses and variables (stored on the stack), data section having global variables (if any), program counter value, register values and other associated resources. Although two processes may be associated with the same program, yet they are treated as two separate processes having their respective set of resources.

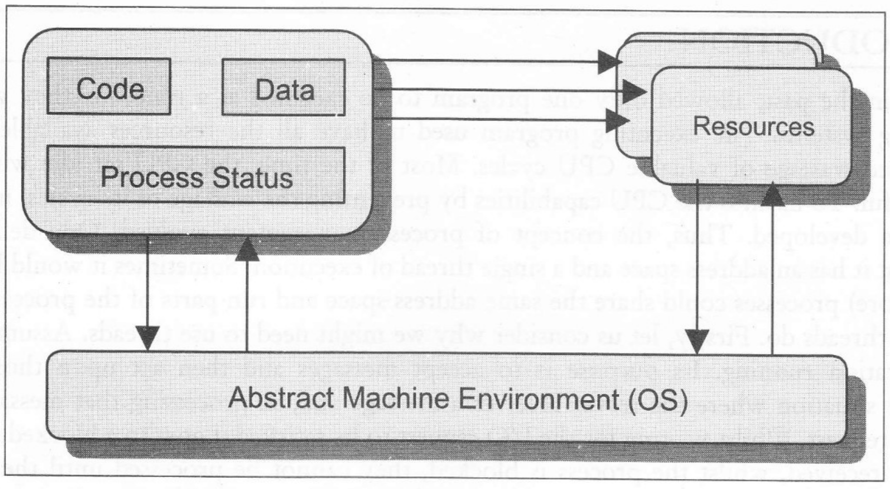

Figure 2.1: Abstract Machine Environment (OS)

2.2.1 Process Hierarchies and Implementation

A process may be created through the create system call. The processes so created are said to be children process of the original process, which in turn is known as parent process. This gives rise to a hierarchical structure of the processes existing in a system. The first process created in the system is the operating system itself.

At any moment the processes for a tree structure of hierarchy, as shown in Figure 2.2.

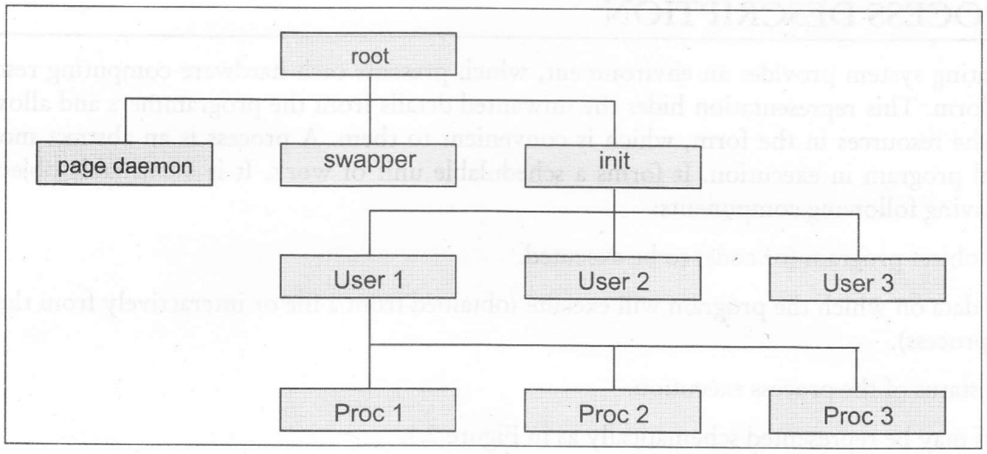

Figure 2.2: Process Hierarchy

In the figure, root is the parent process to page-daemon, swapper and init processes. init process has three children processes while user1 process has three children processes proc1, proc2 and proc3.

2.3 PROCESS CONTROL

Each process is represented in the operating system by a complex data structure called process control block (PCB) - also called a task control block. A typical PCB is shown in Figure 2.3.

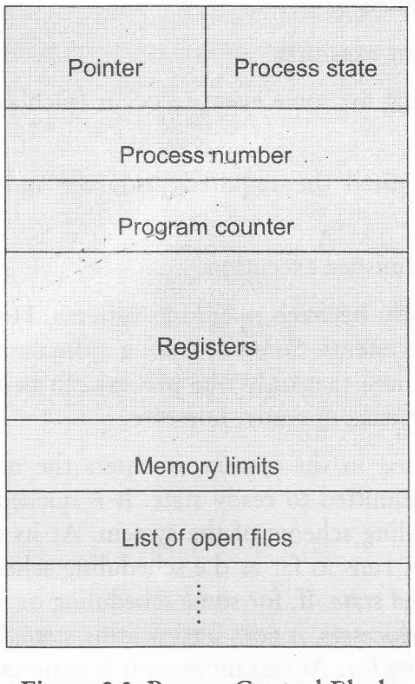

Figure 2.3: Process Control Block

PCB contains much information associated with a specific process, including:

- *Process state:* The state may be new, ready, running, waiting, halted and so on.
- *CPU registers:* The registers vary in number and type, depending on the computer architecture. They include accumulators, index registers, stack pointers, and general-purpose pointers and flags.
- *CPU scheduling*
- *Information:* This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters.

Memory Management

- *Information:* This information may include such information as the base and limit registers, the page tables, or the segment tables depending on the memory system used by the operating system.
- *Accounting*
- *Information:* This information includes the amount of CPU and real time used, time limits, account numbers, job or process numbers, and so on.
- *I/O status*
- *Information:* The information includes the list of I/O devices (such as tape drives) allocated to this process, a list of open files, and so on.

2.3.1 Process States

The current activity of a process is known as its state. As a process executes, its state changes. A process can exist in one of the following states:

- *New:* The process is being created.
- *Running:* Instructions are being executed.
- *Waiting:* The process is waiting for some event to occur (such as an I/O completion or reception of a signal).
- *Ready:* The process has acquired the required resources and is waiting to be assigned to a processor.
- *Terminated:* The process has finished execution.

These names are arbitrary and vary between operating systems. However, the states represented by them are found in all operating systems. Some operating systems may have more finely delineated process states. It is important to note that only one process can be running on any processor at any instant. Many processes may be waiting or ready, however.

Whenever a new process is created in the system, it enters the new state. Soon, it is assigned its required resources, whence it is admitted to ready state. It is queued up in the ready state queue, its priority depending on the scheduling scheme of the system. At its turn, the dispatcher switches the process to running state. Again, it runs as far as the scheduling scheme allows it to run. If it finishes execution, it is sent into terminated state. If, for some scheduling or interrupt reasons, it has to release the resources to other candidate processes, it goes into waiting state. It waits or sleeps until the time it procures the resource it was waiting for. At this moment it is again queued up in the ready queue. The state diagram (Figure 2.4) depicts this scenario.

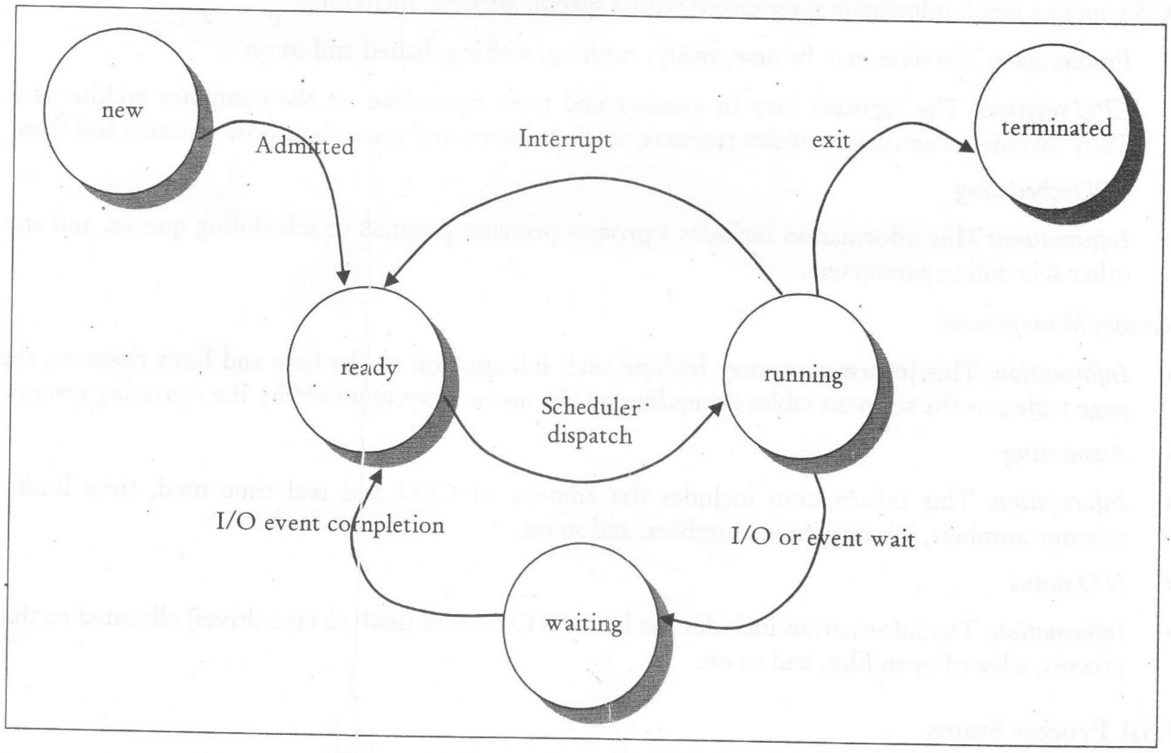

Figure 2.4: Process State Transition Diagram

2.3.2 CPU Scheduling Criteria

On most multitasking systems, only one process can truly be active at a time - the system must therefore share its time between the executions of many processes. This sharing is called scheduling. (Scheduling time management)

Many objectives must be considered in the design of a scheduling discipline. In particular, a scheduler should consider fairness, efficiency, response time, turnaround time, throughput, etc., Some of these goals depends on the system one is using for example batch system, interactive system or real-time system, etc. but there are also some criteria that are desirable in all systems.

- *Fairness:* Fairness is important under all circumstances. A scheduler makes sure that each process gets its fair share of the CPU and no process can suffer indefinite postponement. Note that giving equivalent or equal time is not fair. Think of safety control and payroll at a nuclear plant.
- *Policy Enforcement:* The scheduler has to make sure that system's policy is enforced. For example, if the local policy is safety then the safety control processes must be able to run whenever they want to, even if it means delay in payroll processes.
- *Efficiency:* Scheduler should keep the system (or in particular CPU) busy cent percent of the time when possible. If the CPU and all the Input/Output devices can be kept running all the time, more work gets done per second than if some components are idle.
- *Response Time:* A scheduler should minimize the response time for interactive user.
- *Turnaround:* A scheduler should minimize the time batch users must wait for an output.
- *Throughput:* A scheduler should maximize the number of jobs processed per unit time. A little thought will show that some of these goals are contradictory. It can be shown that any scheduling algorithm that favors some class of jobs hurts another class of jobs. The amount of CPU time available is finite, after all.

CPU Scheduling Algorithms

Different methods of scheduling are appropriate for different kinds of execution. A queue is one form of scheduling in which each program waits its turn and is executed serially. This is not very useful for handling multitasking, but it is necessary for scheduling devices which cannot be shared by nature.

An example of the latter is the printer. Each print job has to be completed before the next one can begin; otherwise all the print jobs would be mixed up and interleaved resulting in nonsense.

We shall make a broad distinction between two types of scheduling, one is nonpreemptive scheduling and another is preemptive scheduling.

2.4 PROCESSES AND THREADS

A thread can be loosely defined as a separate stream of execution that takes place simultaneously with and independently of everything else that might be happening

Threads are typically given a certain priority, meaning some threads take precedence over others. Once the CPU is finished processing one thread, it can run the next thread waiting in line. Threads seldom have to wait more than a few milliseconds before they run.

Computer programs that implement "multi-threading" can execute multiple threads at once. Most modern operating systems support multi-threading at the system level, meaning when one program tries to take up all your CPU resources, you can still switch to other programs and force the CPU-hogging program to share the processor a little bit.

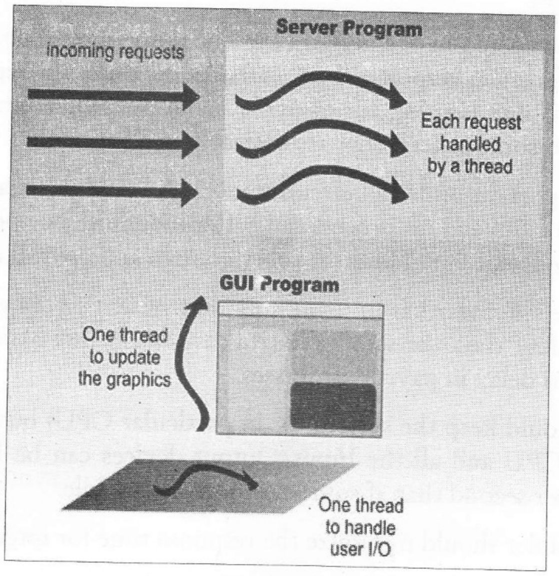
Server Program
incoming requests
Each request handled by a thread
GUI Program
One thread to update the graphics
One thread to handle user I/O

Figure 2.5: Typical Uses of Threads

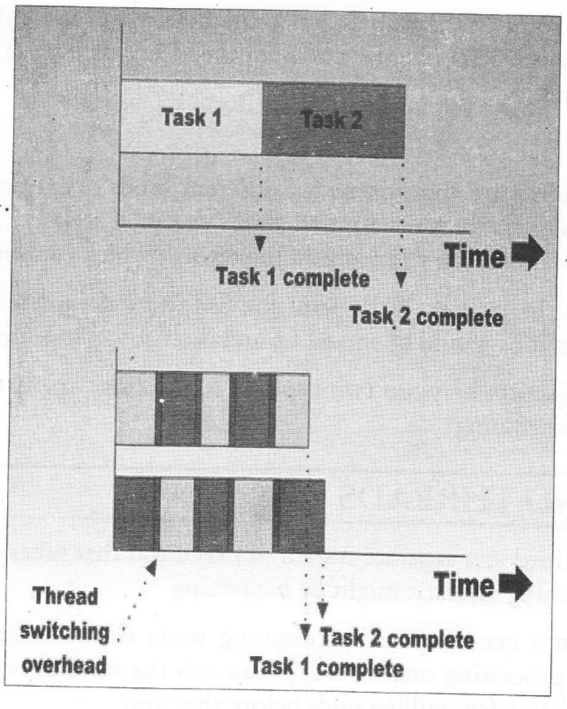

Figure 2.6: Task Switching

Threads are like mini-processes that operate within a single process. Each thread has its own program counter and stack so that it knows where it is. Apart from this they can be considered the same as processes, with the exception that they share the same address space. This means that all threads from the same process have access to the same global variables and the same files.

These tables show you the various items that a process has, compared to the items that each thread has.

Per Thread Items	Per Process Items
• Program Counter	• Address Space
• Stack	• Global Variables
• Register Set	• Open Files
• Child Threads	• Child Processes
• State	• Timers
	• Signals
	• Semaphores
	• Accounting Information

If you have a multi-processor machine, then different threads from the same process can run in parallel.

Multithreaded programming is an essential resource to modern programming. Its use permeates through all types of programming, from embedded systems to the desktop all the way to supercomputing applications. Unfortunately, it's also got a reputation for being hard, which really isn't the case. All that it requires is some careful forethought and a good understanding of what's going on.

A thread is essentially another place where your code is running in the same program. Every program has at least one thread, and some programs have more. They come up all the time and are honestly pretty common practice for most nontrivial programming tasks.

There are many reasons to have multiple threads in the application:

- Support multiple processors
- Long running background computational tasks
- Slow I/O
- Many I/O tasks
- Separate tasks that need to run simultaneously

All of these reasons boil down to the same one of three:

- To use more than one CPU,
- To make the codes much more independent tasks
- To fasten the operation speed

Many of the reasons listed above fall into the last category. Much of a program's time is spent waiting for something to happen. Most user applications sit there and wait for the user to do something, like a mouse moving, or a click. Until then, they block. Blocking is sitting idle while other programs get to use the CPU and you don't until whatever you're blocked waiting for happens.

Example: Here are some examples of popular operating system and their thread support is given below:

- MS-DOS — support a single user process and a single thread
- UNIX — supports multiple user processes but only supports one thread per process
- Solaris — supports multiple threads

2.4.1 Comparison of Multiple-thread and Multiple-process

Comparison of multiple-thread and multiple-process control is given below:

Multiple Process	**Multiple Threads**
has states new, ready, running blocked, terminated	like a process, a thread has states new, ready, running blocked, terminated
processes can create child processes	threads can create child threads
each process operates independently of the others, has its own PC, stack pointer and address space	threads are not independent of each other, threads can read or write over any other's stack

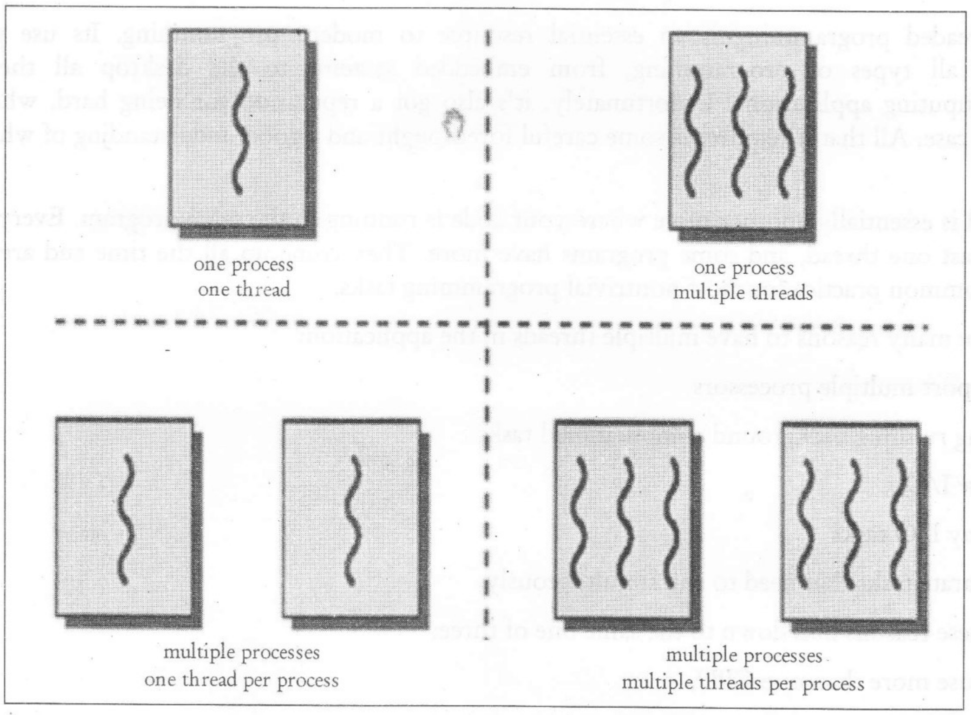

Figure 2.7: Variety of Models for Threads and Processes

2.4.2 Memory Layout for Threading

For a single-threaded program, the memory layout looks like this: An address space and one thread with all its memory mapped to it.

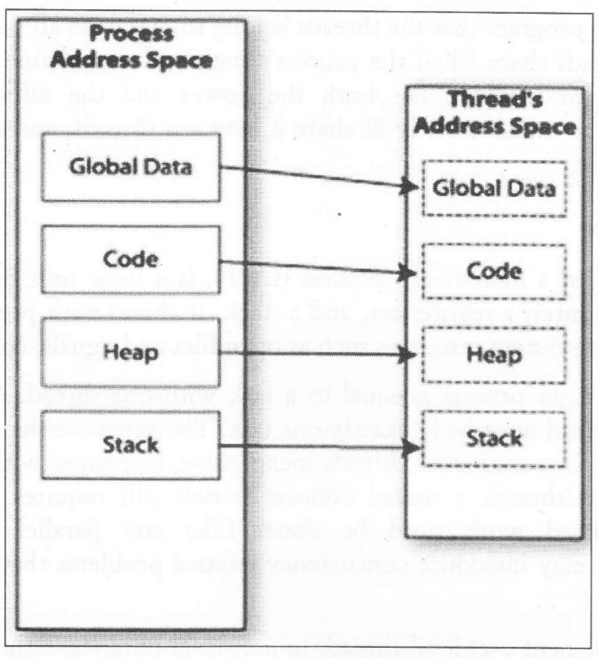

Figure 2.8: Memory Layout for a Single-threaded Process

For a multithreaded program, the memory layout looks like this: An address space and multiple threads sharing it.

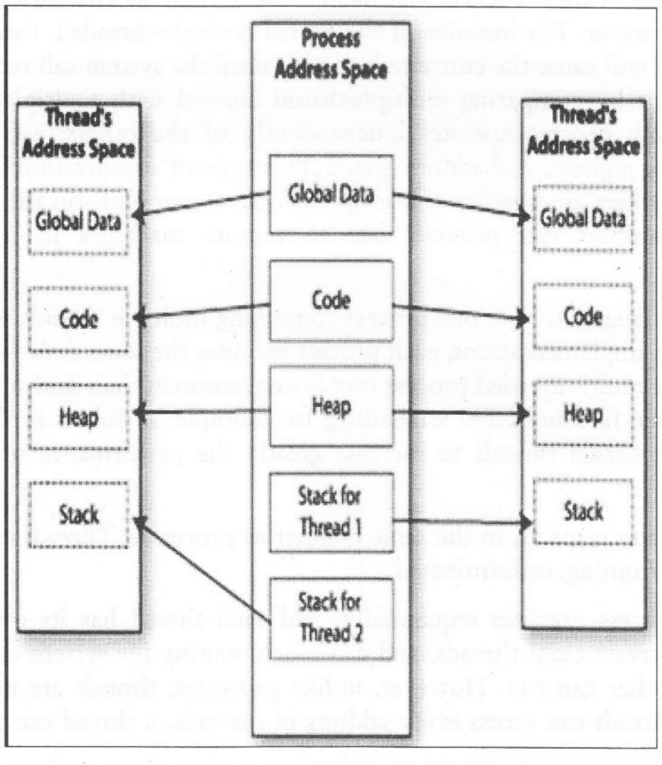

Figure 2.9: Memory Layout for a Multithreaded Process

The process (the running program that the threads belong to) contains all the memory shared between all the threads. The threads share all of the process's memory at the same addresses except for their stacks. Within this shared memory lies both the power and the difficulty with multithreaded programming. Data can be accessed easily & share it between threads, and complete screw up of data structures are also possible.

2.4.3 Thread Structure

A thread, sometimes called a lightweight process (LWP), is a basic unit of resource utilization, and consists of a program counter, a register set, and a stack. It shares with peer threads its code section, data section, and operating-system resources such as open files and signals, collectively known as a task.

A traditional or heavyweight process is equal to a task with one thread. A task does nothing if no threads are in it, and a thread must be in exactly one task. The extensive sharing makes CPU switching among peer threads and the creation of threads inexpensive, compared with context switches among heavyweight processes. Although a thread context switch still requires a register set switch, no memory-management-related work need be done. Like any parallel processing environment, multithreading a process may introduce concurrency control problems that require the use of critical sections or locks.

Also, some systems implement user-level threads in user-level libraries, rather than via system calls, so thread switching does not need to call the operating system, and to cause an interrupt to the kernel. Switching between user-level threads can be done independently of the operating system and, therefore, very quickly. Thus, blocking a thread and switching to another thread is a reasonable solution to the problem of how a server can handle many requests efficiently. User-level threads do have disadvantages, however. For instance, if the kernel is single-threaded, then any user-level thread executing a system call will cause the entire task to wait until the system call returns. We can grasp the functionality of threads by comparing multiple-thread control with multiple-process control. With multiple processes, each process operates independently of the others; each process has its own program counter, stack register, and address space. This type of organization is useful when the jobs performed by the processes are unrelated. Multiple processes can perform the same task as well. For instance, multiple processes can provide data to remote machines in a network file system implementation.

However, it is more efficient to have one process containing multiple threads serve the same purpose. In the multiple process implementations, each process executes the same code but has its own memory and file resources. One multi-threaded process uses fewer resources than multiple redundant processes, including memory, open files and CPU scheduling, for example, as Solaris evolves, network daemons are being rewritten as kernel threads to increase greatly the performance of those network server functions.

Threads operate, in many respects, in the same manner as processes. Threads can be in one of several states: ready, blocked, running, or terminated

A thread within a process executes sequentially, and each thread has its own stack and program counter. Threads can create child threads, and can block waiting for system calls to complete; if one thread is blocked, another can run. However, unlike processes, threads are not independent of one another. Because all threads can access every address in the task, a thread can read or write over any other thread's stacks.

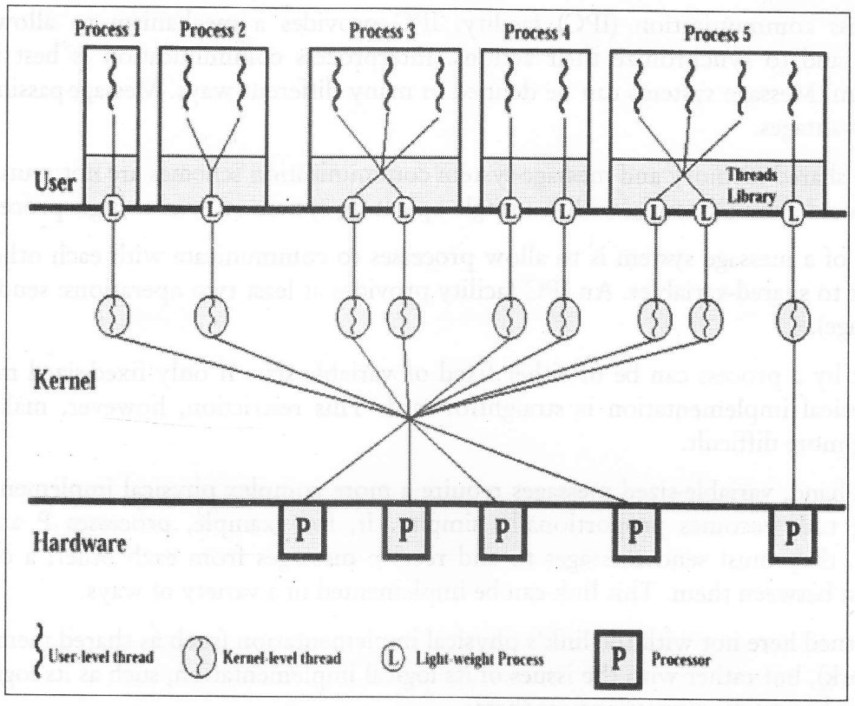

Figure 2.10: Thread Structure

This structure does not provide protection between threads. Such protection, however, should not be necessary. Whereas processes may originate from different users, and may be hostile to one another, only a single user can own an individual task with multiple threads. The threads, in this case, probably would be designed to assist one another, and therefore would not require mutual protection.

Let us return to our example of the blocked file-server process in the single-process model. In this scenario, no other server process can execute until the first process is unblocked. By contrast, in the case of a task that contains multiple threads, while one server thread is blocked and waiting, a second thread in the same task could run. In this application, the cooperation of multiple threads that are part of the same job confers the advantages of higher throughput and improved performance. Other applications, such as the producer-consumer problem, require sharing a common buffer and so also benefit from this feature of thread utilization: The producer and consumer could be threads in a task. Little overhead is needed to switch between them, and, on a multiprocessor system, they could execute in parallel on two processors for maximum efficiency.

2.5 INTER PROCESS COMMUNICATION

In a multiprogramming environment multiple numbers of processes co-exist. A single program may be broken into a number of processes. Those processes that communicate with each other are referred to as cooperating processes.

Cooperating processes can communicate in a shared-memory environment. The scheme requires that these processes share a common buffer pool, and that the code for implementing the buffer be explicitly written by the application programmer. Another way to achieve the same effect is for the operating system to provide the means for cooperating processes to communicate with each other via an interprocess communication (IPC) facility. IPC provides a mechanism to allow processes to communicate and to synchronize their actions. Interprocess communication is best provided by a message system. Message systems can be defined in many different ways. Message-passing systems also have other advantages.

Note that the shared-memory and message-system communication schemes are not mutually exclusive, and could be used simultaneously within a single operating system or even a single process.

The function of a message system is to allow processes to communicate with each other without the need to resort to shared-variables. An IPC facility provides at least two operations: send (message) and receive (message).

Messages sent by a process can be of either fixed or variable size. If only fixed-sized messages can be sent, the physical implementation is straightforward. This restriction, however, makes the task of programming more difficult.

On the other hand, variable-sized messages require a more complex physical implementation, but the programming task becomes proportionally simpler. If, for example, processes P and Q want to communicate, they must send messages to and receive messages from each other; a communication link must exist between them. This link can be implemented in a variety of ways.

We are concerned here not with the link's physical implementation (such as shared memory, hardware bus, or network), but rather with the issues of its logical implementation, such as its logical properties. Some basic implementation questions are these:

1. How are links established?
2. Can a link be associated with more than two processes?
3. How many links can there be between every pair of processes?
4. What is the capacity of a link? That is, does the link have some buffer space? If it does, how much?
5. What is the size of messages? Can the link accommodate variable-sized or only fixed-sized messages?
6. Is a link unidirectional or bidirectional? That is, if a link exists between P and Q, can messages flow in only one direction (such as only from P to Q) or in both directions?

Since a link may be associated with more than two processes. Therefore, links should be defined carefully. Thus, we say that a link is unidirectional only if each process connected to the link can either send or receive, but not both, and each link has at least one receiver process connected to it.

In addition, there are several methods for logically implementing a link and the send/receive operations:

1. Direct or indirect communication
2. Symmetric or asymmetric communication
3. Automatic or explicit buffering
4. Send by copy or send by reference
5. Fixed-sized or variable-sized messages

Inter-process communication (IPC) allows the running of programs concurrently in an operating system.

There are quite a number of methods used in inter-process communications. They are:

- *Pipes:* This allows the flow of data in one direction only. Data from the output is usually buffered until the input process receives it which must have a common origin.
- *Named Pipes:* This is a pipe with a specific name. It can be used in processes that do not have a shared common process origin. Example is FIFO where the data is written to a pipe is first named.
- *Message Queuing:* This allows messages to be passed between messages using either a single queue or several message queues. This is managed by the system kernel. These messages are co-ordinated using an application program interface (API)
- *Semaphores:* This is used in solving problems associated with synchronization and avoiding race conditions. They are integers values which are greater than or equal to zero
- *Shared Memory:* This allows the interchange of data through a defined area of memory. Semaphore value has to be obtained before data can get access to shared memory.
- *Sockets:* This method is mostly used to communicate over a network, between a client and a server. It allows for a standard connection which I computer and operating system independent.

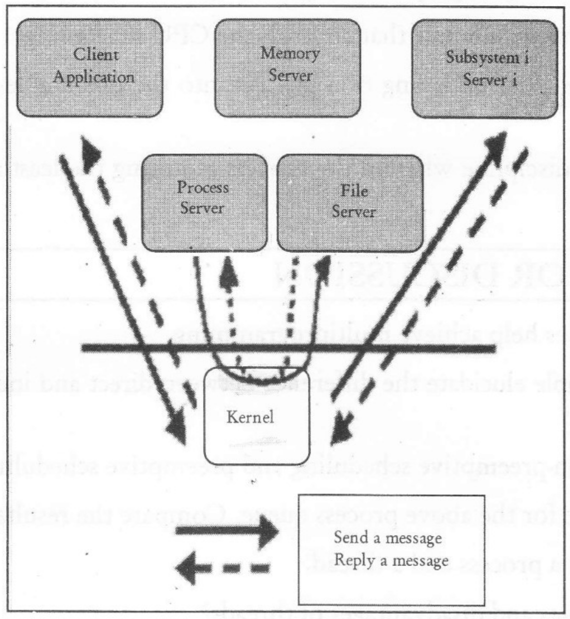

Figure 2.11: Diagram of Interprocess Communication

Check Your Progress

1. Explain five states of process.
2. Name three methods used in interprocess communication.
3. Explain thread structure.

2.6 LET US SUM UP

Process is a unit of program execution that enables the systems to implement multi-tasking behavior. An operating system provides an environment, which presents each hardware resource in its abstract form. A process includes, besides instructions to be executed, the temporary data such as subroutine parameters, return addresses and variables (stored on the stack), data section having global variables (if any), program counter value, register values and other associated resources. A process is represented in the operating system by a complex data structure called process control block. Cooperating processes can communicate in a shared-memory environment.

In the direct-communication discipline, each process that wants to communicate must explicitly name the recipient or sender of the communication. Multitasking and multiprogramming, the two techniques that intend to use the computing resources optimally. A thread is a basic unit of resource utilization, and consists of a program counter, a register set, and a stack.

2.7 KEYWORDS

Process: A unit of program execution inside the operating system's environment.

Interprocess communication: The act of two or more processes sharing information.

CPU Scheduling: The act of assigning a process to the CPU.

Scheduler: An operating system sub unit that controls the CPU scheduling.

Context switch: Loading and/or unloading of a process into the executable state thereby allocating its required resources.

Shortest job first: A queue discipline wherein the process requiring the least amount of time on CPU is executed first.

2.8 QUESTIONS FOR DISCUSSION

1. Describe how processes help achieve multiprogramming.
2. Using a suitable example elucidate the difference between direct and indirect modes of interprocess communication.
3. Compare between non-preemptive scheduling and preemptive scheduling.
4. Obtain a SJF schedule for the above process queue. Compare the results.
5. Differentiate between a process and a thread.
6. What are the advantages and disadvantages of threads?
7. State the problems encountered in interprocess communication. What are the ways of overcoming these limitations?
8. List the services provided by a typical operating system to control process.
9. Why response time is an important criteria in CPU scheduling?

Check Your Progress: Model Answers

1. New, running, waiting, ready, terminated
2. Piped, named pipe, semaphores
3. A thread, sometimes called a lightweight process (LWP), is a basic unit of resource utilization, and consists of a program counter, a register set, and a stack.

2.9 SUGGESTED READINGS

Andrew S. Tanenbaum, *Modern Operating System*,: Prentice Hall

Silberschatz Galvin, *Operating System Concepts*,: Addison Wesley

Andrew M. Lister, *Fundamentals of Operating Systems*,: Wiley

Colin Ritchie, *Operating Systems*,: BPB Publications

UNIT II

LESSON

3

MEMORY MANAGEMENT

CONTENTS

3.0 AIMS AND OBJECTIVES

After studying this lesson, you will be able to:

- Explain Memory Management Requirements
- Differentiate Contiguous and Non Contiguous Allocation
- Elaborate Fixed and variable partitions
- The concept of Paging and Segmentation
- Explain Virtual Memory Management System

3.1 INTRODUCTION

Memory is the electronic holding place for instructions and data that the computer's microprocessor can reach quickly. When the computer is in normal operation, its memory usually contains the main parts of the operating system and some or all of the application programs and related data that are being used. Memory is often used as a shorter synonym for random access memory (RAM). This kind of memory is located on one or more microchips that are physically close to the microprocessor in the computer. Most desktop and notebook computers sold today include at least 16 megabytes of RAM, and are upgradeable to include more. The more RAM you have, the less frequently the computer has to access instructions and data from the more slowly accessed hard disk form of storage.

Memory is sometimes distinguished from storage, or the physical medium that holds the much larger amounts of data that won't fit into RAM and may not be immediately needed there. Storage devices include hard disks, floppy disks, CD-ROM, and tape backup systems. The terms auxiliary storage, auxiliary memory, and secondary memory have also been used for this kind of data repository.

Additional kinds of integrated and quickly accessible memory are read-only memory (ROM), programmable ROM (PROMO), erasable programmable ROM (EPROM). These are used to keep special programs and data, such as the basic input/output system, that need to be in the computer all the time.

The memory is a resource that needs to be managed carefully. Most computers have a memory hierarchy, with a small amount of very fast, expensive, volatile cache memory, some number of megabytes of medium-speed, medium-price, volatile main memory (RAM), and hundreds of thousands of megabytes of slow, cheap, non-volatile disk storage. It is the job of the operating system to coordinate how these memories are used.

3.2 MEMORY MANAGEMENT REQUIREMENTS

In addition to the responsibility of managing processes, the operating system must efficiently manage the primary memory of the computer. The part of the operating system which handles this responsibility is called the memory manager. Since every process must have some amount of primary memory in order to execute, the performance of the memory manager is crucial to the performance of the entire system. The memory manager is responsible for allocating primary memory to processes and for assisting the programmer in loading and storing the contents of the primary memory. Managing the sharing of primary memory and minimizing memory access time are the basic goals of the memory manager.

When an operating system manages the computer's memory, there are two broad tasks to be accomplished:

1. Each process must have enough memory in which to execute, and it can neither run into the memory space of another process nor be run into by another process.
2. The different types of memory in the system must be used properly so that each process can run most effectively.

The first task requires the operating system to set up memory boundaries for types of software and for individual applications.

As an example, let's look at an imaginary small system with 1 megabyte (1,000 kilobytes) of RAM. During the boot process, the operating system of our imaginary computer is designed to go to the top of available memory and then "back up" far enough to meet the needs of the operating system itself. Let's say that the operating system needs 300 kilobytes to run. Now, the operating system goes to the bottom of the pool of RAM and starts building up with the various driver software required to control the hardware subsystems of the computer. In our imaginary computer, the drivers take up 200 kilobytes. So after getting the operating system completely loaded, there are 500 kilobytes remaining for application processes. When applications begin to be loaded into memory, they are loaded in block sizes determined by the operating system. If the block size is 2 kilobytes, then every process that is loaded will be given a chunk of memory that is a multiple of 2 kilobytes in size. Applications will be loaded in these fixed block sizes, with the blocks starting and ending on boundaries established by words of 4 or 8 bytes. These blocks and boundaries help to ensure that applications won't be loaded on top of one another's space by a poorly calculated bit or two. With that ensured, the larger question is what to do when the 500-kilobyte application space is filled.

In most computers, it's possible to add memory beyond the original capacity. For example, you might expand RAM from 1 to 2 megabytes. This works fine, but tends to be relatively expensive. It also ignores a fundamental fact of computing - most of the information that an application stores in memory is not being used at any given moment. A processor can only access memory one location at a time, so the vast majority of RAM is unused at any moment. Since disk space is cheap compared to RAM, then moving information in RAM to hard disk can greatly expand RAM space at no cost. This technique is called virtual memory management.

Disk storage is only one of the memory types that must be managed by the operating system, and is the slowest. Ranked in order of speed, the types of memory in a computer system are:

1. ***High-speed cache:*** This is fast, relatively small amounts of memory that are available to the CPU through the fastest connections. Cache controllers predict which pieces of data the CPU will need next and pull it from main memory into high-speed cache to speed up system performance.
2. ***Main memory:*** This is the RAM that you see measured in megabytes when you buy a computer.
3. ***Secondary memory:*** This is most often some sort of rotating magnetic storage that keeps applications and data available to be used, and serves as virtual RAM under the control of the operating system.

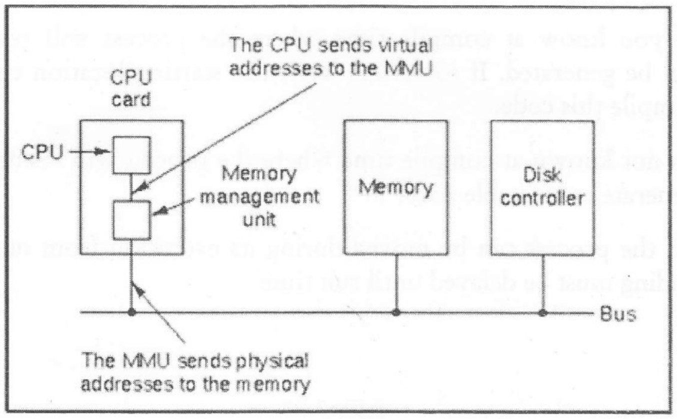
The CPU sends virtual addresses to the MMU
CPU card
CPU
Memory management unit
Memory
Disk controller
Bus
The MMU sends physical addresses to the memory

Figure 3.1: Diagram of Memory Management