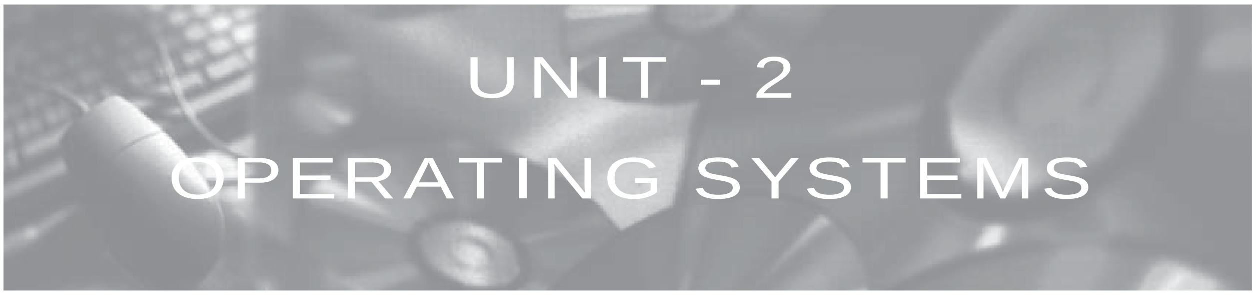

UNIT - 2
OPERATING SYSTEMS

CHAPTER 1

INTRODUCTION TO OPERATING SYSTEMS

LEARNING OBJECTIVES

- To know about Operating Systems and their main functions,
- To have an idea about measuring system performance,
- To understand process management,
- To learn about multiprogramming and its requirements,
- To have an overview of multitasking and multithreading,
- To discuss multiprocessing and its advantages and limitations,
- To know about time-sharing systems and its advantages,
- To discuss various concepts in File Management,
- To understand various features in Operating System Structure and other related concepts, and
- To know about some popular operating systems.

1.1 OPERATING SYSTEM

An Operating System (OS) is a software, consisting of an integrated set of programs that controls computer resources (CPU, memory, I/O devices etc.) and provides common services for efficient execution of various application software with an interface or virtual machine that is more convenient to use. In short, we can say that *operating system is an interface between hardware and user*. For hardware functions such as input and output and memory allocation, the operating system acts as an intermediary between application programs and the computer hardware, although the application code is usually executed directly by the hardware, but will frequently call the OS or be interrupted by it. Operating systems are found on almost any device that contains a computer, from cellular phones and video game consoles to supercomputers and web servers. Examples of popular modern operating systems for personal computers are Microsoft Windows, Mac OS, and Linux. Two major objectives of an operating system are as follows:

- **Making a computer system user-friendly:** A computer system consists of one or more processors, main memory and many types of I/O devices such as disks, tapes, terminals, network interfaces, etc. Writing programs for using these hardware resources correctly and efficiently is an extremely difficult job, requiring in-depth knowledge of the functioning of the resources. Hence, to make computer systems usable by a large number of users, it became clear several years ago that some way is required to shield programmers from the complexity of the hardware resources. The gradually evolved solution to handle this problem is to put a layer of software on top of the bare hardware, to manage all the parts of system, and present the user with an interface or virtual machine that is easier to program and use. This layer of software is called the operating system.

The logical architecture of a computer system is shown in Fig. 1.1.1. As shown in the figure, the hardware resources are surrounded by the operating system layer, which in turn is surrounded by a layer of other system software (such as compilers, editors, command interpreter, utilities, etc.) and a set of application programs (such as commercial data processing applications, scientific and engineering applications, entertainment and educational applications, etc.). Finally, the end users view the computer system in terms of the user interfaces provided by the application programs.

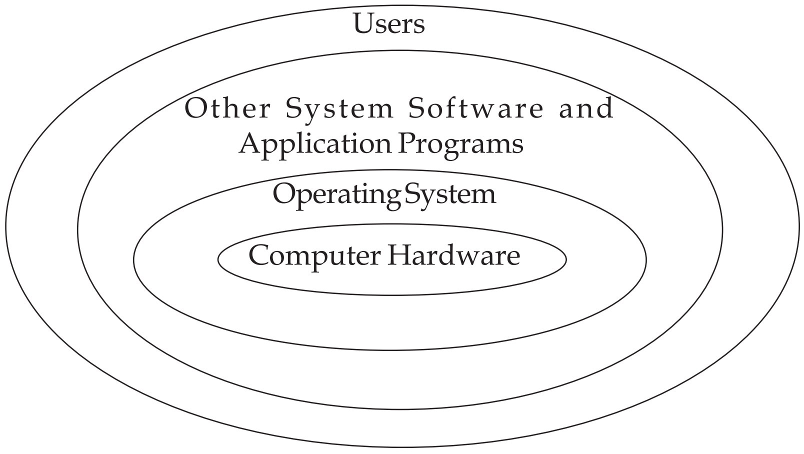

Fig. 1.1.1: Architecture of a computer system

The operating system layers provide various facilities and services that make the use of the hardware resources convenient, efficient, and safe. A programmer makes use of these facilities in developing an application, and the application, while it is running, invokes the required services to perform certain functions. In effect, the operating system hides the details of the hardware from the programmer and provides a convenient interface for using the system. It acts as an intermediary between the hardware and its users, providing a high-level interface to low-level hardware resources and making it easier for the programmer and for application programs to access and use those resources.

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

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

1.2 MAIN FUNCTIONS OF AN OPERATING SYSTEM

It is clearly discussed in the previous section that operating system provides certain services to programs as well as users of those programs. The specific services provided will, of course, differ from one operating system to another, but there are some common types of functions that we can identify. The main functions provided by most of the operating systems are given as follows:

- **Process Management:** A process is a program in execution. The operating system manages many kinds of activities ranging from user programs to system programs like printer spooler, name servers, file server etc. Each of these activities is encapsulated in a process. A process includes the complete execution context (code, data, PC, registers, OS resources in use etc.).

  It is important to note that a process is not a program. A process is only an instant of a program in execution. There are many processes, can be running the same program. The five major activities of an operating system with respect to process management are:

  - Creation and deletion of user and system processes;
  - Suspension and resumption of processes;
  - A mechanism for process synchronization;
  - A mechanism for process communication; and
  - A mechanism for deadlock handling.

- **Memory Management:** To execute a program, it must be loaded, together with the data, it accesses in the main memory (at least partially). To improve CPU utilization and to provide better response time to its users, a computer system normally keeps several programs in main memory. The memory management module of an operating system takes care of the allocation de-allocation of memory space to the various programs in need of this resource. Primary-Memory or Main-Memory is a large array of words or bytes. Each word or byte has its own address. Main-memory provides storage that can be accessed directly by the CPU. That is to say for a program to be executed, it must in the main memory. The major activities of an operating system with reference to memory-management are:

  - To keep track of 'which part of memory are currently being used and by whom';
  - To decide 'which process is loaded into memory when memory space becomes available; and to allocate and de-allocate memory space, as needed.

- **File Management.** A file is a collection of related information defined by its creator. Computer can store files on the disk (secondary storage), which provide long term storage. Some examples of storage media are magnetic tape, magnetic disk and optical disk. Each of these media has its own properties like speed, capacity, data transfer rate and access methods. A file system normally organized into directories to ease their use. These directories may contain files and other directions. The five major activities of an operating system with reference to file management are given as under:

  - The creation and deletion of files;
  - The creation and deletion of directions;
  - The support of primitives for manipulating files and directions;
  - The mapping of files onto secondary storage; and
  - The back up of files on stable storage media.

- **Device Management:** A computer system normally consists of several I/O devices such as terminal, printer, disk, and tape. The device management module of an operating system takes care of controlling all the computer's I/O devices. It keeps track of I/O requests from processes, issues commands to the I/O devices, and ensures correct data transmission to/from an I/O device. It also provides an interface between the devices and the rest of the system that is simple and easy to use. Often, this interface is device independent, that is, the interface is same for all types of I/O devices.

- **Security:** Computer systems often store large amounts of information, some of which is highly sensitive and valuable to their users. Users can trust the system and rely on it only if the various resources and information of a computer system are protected against destruction and unauthorized access. The security module of an operating system ensures this. This module also ensures that when several disjoint processes are being executed simultaneously, one process does not interfere with the others, or with the operating system itself.
- **Command Interpretation:** A command interpreter is an interface of the operating system with the user. The user gives commands with are executed by operating system (usually by turning them into system calls). The main function of a command interpreter is to get and execute the next user specified command. Command-Interpreter is usually not part of the kernel, since multiple command interpreters (shell, in UNIX terminology) may be support by an operating system, and they do not really need to run in kernel mode. There are two main advantages to separat the command interpreter from the kernel:
  - If we want to change the way the command interpreter looks, that means., we want to change the interface of command interpreter, we are able to do that if the command interpreter is separate from the kernel. We cannot change the code of the kernel so we cannot modify the interface.
  - If the command interpreter is a part of the kernel; it is possible for a malicious process to gain access to certain part of the kernel that it showed, to avoid this ugly scenario. It is advantageous to have the command interpreter separate from kernel.

In addition to the above listed major functions, an operating system also performs few other functions such as 'keeping an account of which user (or processes) use how much' and 'what kinds of computer resources, maintenance of log of system usage by all users', and 'maintenance of internal time clock'.

1.3 MEASURING SYSTEM PERFORMANCE

The efficiency of an operating system and the overall performance of a computer system are usually measured in terms of the following aspects:

- **Throughput:** *Throughput* is the amount of work that the system is able to do per unit time. It is measured as the number of processes that are completed by the system per unit time. For example, if *n* processes are completed in an interval of *t* seconds, the throughput is taken as *n/t* processes per second during that interval. Throughput is normally measured in processes/hour. Here, it is noteworthy that the value of throughput does not depend only on the capability of a system, but also on the nature of jobs being processed by the system. For long processes, throughput may be one process/hour; and for short processes, throughput may be 100 processes/hour.
- **Turnaround time:** From the point of view of an individual user, an important criterion is 'how long it takes the system to complete a job submitted by him/her'. Turnaround time is the interval from the time of submission of a job to the system for processing to the time of completion of the job. Although higher throughput is desirable from the point of view of overall system performance, but individual users are more interested in better turnaround time for their jobs.
- **Response time:** Turnaround time is usually not a suitable measure for interactive systems, because in an interactive system, a process can produce some output fairly early during its execution and can continue executing while previous results are being output to the user. Thus,

another measure used in case of interactive systems is response time, which is the interval from the time of submission of a job to the system for processing to the time the first response for the job is produced by the system.

In any computer system, it is desirable to maximize throughput and to minimize turnaround time and response time.

1.4 PROCESS MANAGEMENT

A process is a sequential program in execution. The components of a process are the following:

- The object program to be executed (called the *program text* in UNIX);
- The *data* on which the program will execute (obtained from a file or interactively from the process's user);
- *Resources* required by the program (for example, files containing requisite information); and

  The *status* of the process's execution.

During the lifespan of a process, its execution status may be in one of four states (associated with each state is usually a queue on which the process resides):

- **Executing:** the process is currently running and has control of a CPU;
- **Waiting:** the process is currently able to run, but must wait until a CPU becomes available;
- **Blocked:** the process is currently waiting on I/O, either for input to arrive or output to be sent;
- **Suspended:** the process is currently able to run, but for some reason the OS has not placed the process on the ready queue; and
- **Ready:** the process is in memory, will execute given CPU time.

1.4.1 Process Management in Early Systems

In early computer systems, a job was typically executed in the following manner:

- A programmer would first write the program on paper.
- It was then punched on cards or paper tape along with its data.
- The deck of cards or the paper tape containing the program and data was then submitted at the reception counter of the computer centre.
- An operator would then take the card deck or paper tape and manually load it into the system from card reader or paper tape reader. The operator was also responsible for loading any other software resource (such as a language compiler) or setting hardware devices required for the execution of the job. Before loading of the job, the operator had to use the front panel switches of the computer system to clear the main memory to remove any data remaining from the previous job.
- The operator would then set the appropriate switches in the front panel to run the job.
- The result of execution of the job was then printed on the printer, which was brought by the operator to the reception counter, so that the programmer could collect it later.
- The same process had to be repeated for each and every job to be executed by the computer. This method of job execution was known as the *manual loading mechanism* because the jobs had to be manually loaded one after another by the computer operator in the computer system. Notice that in this method, job-to-job transition was not automatic. The manual transition from one job to another caused lot of computer time to be wasted since the computer remained idle while the operator loaded and unloaded jobs and prepared the system for a new job. In order to reduce this idle time of the computer, a method of automatic job-to-job transition was devised. In this

method, known as *batch processing*, when one job is finished, the system control is automatically transferred back to the operating system which automatically performs the housekeeping jobs (such as clearing the memory to remove any data remaining from the previous job) needed to load and run the next job. In case of batch processing systems, jobs were typically executed in the following manner:

- Programmers would prepare their programs and data on card decks or paper tapes and submitted them at the reception counter of the computer centre.
- The operator could periodically collect all the submitted programs and would batch them together and then load them all into the input device of the system at one time.
- The operator would then give a command to the system to start executing the jobs.
- The jobs were then automatically loaded from the input device and executed by the system one-by one without any operator intervention. That is, the system would read the first job from the input device, execute it, print out its result on the printer, and then repeat these steps for each subsequent job till all the jobs in the submitted batch of jobs were over.
- When all the jobs in the submitted batch were processed, the operator would separate the printed output for each job and keep them at the reception counter so that the programmers could collect them later.

1.5 MULTIPROGRAMMING

In *multiprogramming* systems, the running task keeps running until it performs an operation that requires waiting for an external event (e.g. reading from a tape) or until the computer's scheduler forcibly swaps the running task out of the CPU. Multiprogramming systems are designed to maximize CPU usage. In fact, depending on the CPU utilization during the course of processing, jobs are broadly classified into the following two types:

- **CPU-bound jobs:** These jobs mostly perform numerical calculations, with little I/O operations. They are so called because they heavily utilize the CPU during the course of their processing. Programs used for scientific and engineering computations usually fall in this category of jobs.
- **I/O-bound jobs:** These jobs normally input vast amount of data, perform very little computation, and output large amount of information. This is because during the course of their processing, their CPU utilization is very low and most of the time, they perform I/O operations. Programs used for commercial data processing applications usually, fall in this category of jobs.

1.5.1 Requirements of Multiprogramming Systems

Multiprogramming systems have better throughput than uniprogramming systems because the CPU idle time is drastically reduced. However, multiprogramming systems are fairly sophisticated because they require the following additional hardware and software features:

- **Large memory:** For multiprogramming to work satisfactorily, large main memory is required to accommodate a good number of user programs along with the operating system.
- **Memory protection:** Computers designed for multiprogramming must provide some type of memory protection mechanism to prevent a job in one memory partition from changing information or instruction of a job in another memory partition. For example, in Fig. 1.5.2, we would not want job A to inadvertently destroy something in the completely independent job B or job C. In a multiprogramming system, this is achieved by the memory protection feature; a combination of hardware and software, which prevents one job from addressing beyond the limits to its own allocated memory area.
- **Job status preservation:** In multiprogramming, when a running job gets blocked for I/O

processing, the CPU is taken away from this job and is given to another job that is ready for execution. At a later time the former job will be allocated the CPU to continue its execution. It is note werthy that it requires preserving of the job's complete status information when the CPU is taken away from it and restoring this information back before the CPU is given back to it again. To enable this, the operating system maintains a P*rocess Control Block* (PCB) for each loaded process. A typical process control block is shown in Fig. 1.5.1. With this arrangement, before taking away the CPU from a running process, its status is preserved in its PCB, and before the process resumes execution when the CPU is given back to it at a later time, its status is restored back from its PCB. Thus the process can continue do its execution without any problem.

process identifier
process state
program counter
values of various CPU registers
accounting and scheduling information
I/O status information

Fig. 1.5.1: A typical Process Control Block (PCB)

- **Proper job mix:** A proper mix of I/O-bound and CPU-bound jobs are required to effectively overlap the operations of the CPU and I/O devices. If all the loaded jobs need I/O at the same time, the CPU will again be idle. Hence, the main memory should contain some CPU-bound and some I/O-bound jobs so that at least one job is always ready to utilize the CPU.
- **CPU scheduling:** In a multiprogramming system, often there will be situations in which two or more jobs will be in the ready state waiting for CPU to be allocated for execution. When more than one process is in the ready state and the CPU becomes free, the operating system must decide which of the ready jobs should be allocated the CPU for execution. The part of the operating system concerned with this decision is called the *CPU scheduler*, and the algorithm it uses is called the *CPU scheduling algorithm.*

1.6 MULTITASKING

Multitasking is a method with multiple tasks processes sharing common processing resources such as a CPU. In the case of a computer with a single CPU, only one task is said to be *running* at any point in time, meaning that the CPU is actively executing instructions for that task. Multitasking solves the problem by scheduling which task may be the one running at any given time, and when another waiting task gets a turn. The act of reassigning a CPU from one task to another one is called a context switch. When context switches occur frequently enough the illusion of parallelism is achieved. Even on computers with more than one CPU (called multiprocessor machines), multitasking allows many more tasks to be run than there are CPUs.

Many persons do not distinguish between multiprogramming and multitasking because both the terms refer to the same concept. However, some persons prefer to use the term multiprogramming for multi-user systems (systems that are simultaneously used by many users such as mainframe and server class systems), and multitasking for single-user systems (systems that are used by only one user at a time such as a personal computer or a notebook computer). Note that even in a single-user system, it is not necessary that the system works only on one job at a time. In fact, a user of a single-user system often has multiple tasks concurrently processed by the system. For example, while editing a file in the foreground, a sorting job can be given in the background. Similarly, while compilation of a program is in progress in the background, the user may be reading his/her electronic mails in the foreground. In this manner, a user may concurrently work on many tasks. In such a situation, the status of each of the tasks is normally viewed on the computer's screen by partitioning the screen into a number of windows. The progress of different tasks can be viewed on different windows in a multitasking system.

Hence, for those who like to differentiate between multiprogramming and multitasking, *multiprogramming is the concurrent execution of multiple jobs (of same or different users) in a multi user system, while multitasking is the concurrent execution of multiple jobs (often referred to as tasks of same user) in a single-user system.*

1.7 MULTITHREADING

Threads are a popular way to improve application performance. In traditional operating systems, the basic unit of CPU utilization is a process. Each process has its own program counter, its own register states, its own stack, and its own address space (memory area allocated to it). On the other hand, in operating systems, with threads facility, the basic unit of CPU utilization is a thread. In these operating systems, a process consists of an address space and one or more threads of control as shown in Fig 1.7.1 (a). Each thread of a process has its own program counter, its own register states, and its own stack. But all the threads of a process share the same address space. Hence, they also share the same global variables. In addition, all threads of a process also share the same set of operating system resources, such as open files, signals, accounting information, and so on. Due to the sharing of address space, there is no protection between the threads of a process. However, this is not a problem. Protection between processes is needed because different processes may belong to different users. But a process (and hence, all its threads) is always owned by a single user. Therefore, protection between multiple threads of a process is not necessary. If protection is required between two threads of a process, it is preferable to put them in different processes, instead of putting them in a single process.

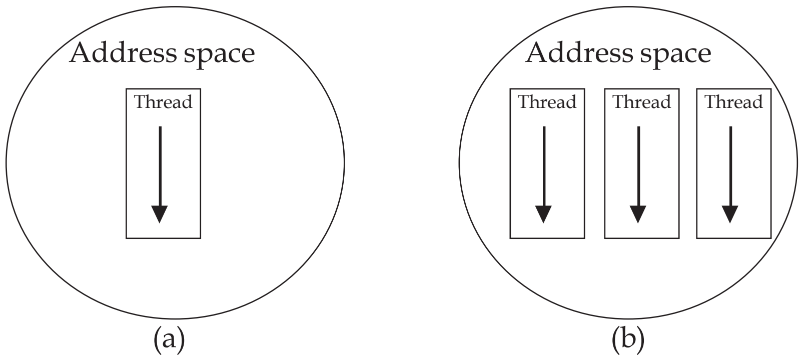

Figure 1.7.1: (a) Single-threaded and (b) multithreaded processes

A single-threaded process corresponds to a process of a traditional operating system. Threads share a CPU in the same way as processes do. At a particular instance of time, a thread can be in anyone of several states namely, running, blocked, ready, or terminated. Due to these similarities, threads are often viewed as miniprocesses. In fact, in operating systems with threads facility, a process having a single thread corresponds to a process of a traditional operating system as shown in Fig.

1.7.1 (b). Threads are often referred to as *lightweight processes* and traditional processes are referred to as *heavyweight processes*.

1.8 MULTIPROCESSING

Multiprocessing is the use of two or more Central Processing Units (CPUs) within a single computer system. The term also refers to the ability of a system to support more than one processor and/or the ability to allocate tasks between them. There are many variations on this basic theme, and the definition of multiprocessing can vary with context, mostly as a function of how CPUs are defined (multiple cores on one die, multiple dies in one package, multiple packages in one system unit, etc.). Multiprocessing sometimes refers to the execution of multiple concurrent software processes in a system as opposed to a single process at any one instant. However, the terms multitasking or multiprogramming are more appropriate to describe this concept, which is implemented mostly in software, whereas multiprocessing is more appropriate to describe the use of multiple hardware CPUs. A system can be both multiprocessing and multiprogramming, only one of the two, or neither of the two.

In a multiprocessing system, all CPUs may be equal, or some may be reserved for special purposes. A combination of hardware and operating-system software design considerations determine the symmetry (or lack thereof) in a given system. For example, hardware or software considerations may require that only one CPU respond to all hardware interrupts, whereas all other work in the system may be distributed equally among CPUs; or execution of kernel-mode code may be restricted to only one processor (either a specific processor, or only one processor at a time), whereas user-mode code may be executed in any combination of processors. Multiprocessing systems are often easier to design if such restrictions are imposed, but they tend to be less efficient than systems in which all CPUs are utilized. Systems that treat all CPUs equally are called Symmetric Multiprocessing (SMP) systems. In systems where all CPUs are not equal, system resources may be divided in a number of ways, including Asymmetric Multiprocessing (ASMP), Non-Uniform Memory Access (NUMA) multiprocessing, and clustered multiprocessing.

Multiprocessing systems are basically of two types namely, tightly-coupled systems and loosely-coupled systems:

- **Tightly and Loosely Coupled Multiprocessing Systems:** Tightly-coupled multiprocessor systems contain multiple CPUs that are connected at the bus level. These CPUs may have access to a central shared memory (SMP or UMA), or may participate in a memory hierarchy with both local and shared memory (NUMA). The IBM p690 Regatta is an example of a high end SMP system. Intel Xeon processors dominated the multiprocessor market for business PCs and were the only x86 option until the release of AMD's Opteron range of processors in 2004. Both ranges of processors had their own onboard cache but provided access to shared memory; the Xeon processors via a common pipe and the Opteron processors via independent pathways to the system RAM. Chip multiprocessors, also known as multi-core computing, involves more than one processor placed on a single chip and can be thought of the most extreme form of tightly-coupled multiprocessing. Mainframe systems with multiple processors are often tightly-coupled.
- **Loosely Coupled Multiprocessing Systems:** Loosely-coupled multiprocessor systems (often referred to as clusters) are based on multiple standalone single or dual processor commodity computers interconnected via a high speed communication system (Gigabit Ethernet is common). A Linux Beowulf cluster is an example of a loosely-coupled system.

Tightly-coupled systems perform better and are physically smaller than loosely-coupled systems, but have historically required greater initial investments and may depreciate rapidly; nodes in a loosely-coupled system are usually inexpensive commodity computers and can be recycled as independent machines upon retirement from the cluster. Power consumption is also a consideration. Tightly-coupled systems tend to be much more energy efficient than clusters. This is because considerable economies can be realized by designing components to work together from the beginning in tightly-coupled systems, whereas loosely-coupled systems use components that were not necessarily intended specifically for use in such systems.

1.8.1 Difference between Multiprogramming and Multiprocessing

Multiprogramming is the interleaved execution of two or more processes by a single-CPU computer system. On the other hand, multiprocessing is the simultaneous execution of two or more processes by a computer system having more than one CPU. To be more specific, multiprogramming involves executing a portion of one program, then a segment of another, etc., in brief consecutive time periods. Multiprocessing, however, makes it possible for the system to simultaneously work on several program segments of one or more programs.

1.8.2 Advantages and Limitations of Multiprocessing

Multiprocessing systems normally have the following advantages:

- **Better Performance:** Due to multiplicity of processors, multiprocessor systems have better performance than single-processor systems. That is, the multiple processors of such a system can be utilized properly for providing shorter response times and higher throughput than a single-processor system. For example, if there are two different programs to be run, two processors are evidently more powerful than one because the programs can be simultaneously run on different processors.
- **Better Reliability:** Due to multiplicity of processors, multiprocessor systems also have better reliability than single-processor systems. In a properly designed multiprocessor system, if one of the processors breaks down, the other processor(s) automatically takes over the system workload until repairs are made. Thus a complete breakdown of such systems can be avoided. For example, if a system has 4 processors and one fails, then the remaining 3 processors can be utilized to process the jobs submitted to the system. Thus, the entire system runs only 25% slower, rather than failing altogether. This ability of a system to continue providing service proportional to the level of non-failed hardware is called *graceful degradation* feature.

Multiprocessing systems, however, require a very sophisticated operating system to schedule, balance, and coordinate the input, output, and processing activities of multiple processors. The design of such an operating system is a complex and time taking job. Moreover, multiprocessing systems are expensive to procure and maintain. In addition to the high charge paid initially, the regular operation and maintenance of these systems is also a costly affair.

1.9 TIME-SHARING

Time-sharing is the sharing of a computing resource among many users by means of multiprogramming and multi-tasking. This concept was introduced in the 1960s, and emerged as the prominent model of computing in the 1970s, represents a major technological shift in the history of computing. By allowing a large number of users to interact concurrently with a single computer, time-sharing dramatically lowered the cost of providing computing capability, made it possible for individuals and organizations to use a computer without owning one, and promoted the interactive use of computers and the development of new interactive applications. Time-sharing is a mechanism to provide simultaneous interactive use of a computer system by many users in such a way that each user is given the impression that he/she has his/her own computer. It uses multiprogramming with a special CPU scheduling algorithm to achieve this.

1.9.1 Requirements of Time-sharing Systems

Time-sharing systems typically require the following additional hardware and software features:

- A number of terminals simultaneously connected to the system so that multiple users can simultaneously use the system in an interactive mode;
- A relatively large memory to support multiprogramming;
- Memory protection mechanism to prevent one job's instructions and data from other jobs in a multiprogramming environment;
- Job status preservation mechanism to preserve a job's complete status information when the CPU is taken away from it and restoring this information back before the CPU is given back to it again;
- A special CPU scheduling algorithm that allocates a very short period of CPU time one-by-one to each user process in a circular fashion; and
- An alarm clock mechanism to send an interrupt signal to the CPU after every time slices.

1.9.2 Advantages of Time-sharing Systems

Although time-sharing systems are complex to design but they provide several advantages to their users. The main advantages of time-sharing systems are given as follows:

- **Reduces CPU idle time:** While a particular user is engaged in thinking or typing his/her input, a time-sharing system can provide to service many other users. In this manner, time-sharing systems help in reducing the CPU idle time to a great extent, increasing the system throughput.
- **Provides advantages of quick response time:** The special CPU scheduling algorithm used in timesharing systems ensures quick response time to all users. This feature allows users to interact with the system more rapidly while working on their problem. For example, a time-sharing system can be effectively used for interactive programming and debugging to improve programmers efficiency. Multiple programmers can simultaneously proceed step-by-step, writing, testing and debugging portions of their programs or trying out various approaches to a problem solution. The greatest benefit of such a system is that errors can be encountered, corrected, and work can continue immediately for all the simultaneous users of the system. This is in contrast to a batch system in which errors are corrected offline and the job is resubmitted for another run. The time delay between job submission and return of the output in a batch system is often measured in hours.
- **Offers good computing facility to small users:** Small users can gain direct access to much more sophisticated hardware and software than they could otherwise justify or afford. In time-sharing systems, they merely pay a fee for resources used and are relieved of the hardware, software, and personnel problems associated with acquiring and maintaining their own installation.

1.10 FILE MANAGEMENT

A file is a collection of related information. Every file has a name, its data, and attributes. The name of a file uniquely identifies it in the system and is used by its users to access it. A file's data is its contents. The contents of a file are a sequence of bits, bytes, lines or records, whose meaning is defined by the file's creator and user. The attribute of a file contains other information about the file such as the date and time of its creation, date and time of last access, date and time of last update, its current size, its protection features etc. the list of attributes mentioned for a file varies considerably from one system to another. The file management module of an operating system takes care of file-related activities such as structuring, accessing, naming, sharing, and protection of files.

1.10.1 File Access Methods

To use the information stored in a file, it must be accessed and read into computer memory. Two commonly supported file access methods at operating system level are sequential and random access. These are briefly discussed given below:

- **Sequential Access:** Sequential access means that a group of elements (e.g. data in a memory array or a disk file or on a tape) is accessed in a predetermined, ordered sequence. Sequential access is sometimes the only way of accessing the data, for example if it is on a tape. It may also be the access method of choice, for example, if we simply want to process a sequence of data elements in order.
- **Random Access:** Random access files consist of records that can be accessed in any sequence. This means the data is stored exactly as it appears in memory, thus saving processing time (because no translation is necessary) both when the file is written and when it is read. Random files are a better solution to database problems than sequential files, although there are a few disadvantages. For one thing, random files are not especially transportable. Unlike sequential files, we cannot peek inside them with an editor, or type them in a meaningful way to the screen.

All operating systems do not support both sequential and random access files. Some of them only support sequential access files, whereas some of them only support random access files, while there are some operating systems, which support both. Those, which support files of both types, normally require that a file be declared as sequential or random, when it is created; such a file can be accessed only in a manner consistent with its declaration. Most of the modern operating systems support only random access files.

1.10.2 File Operations

An operating system provides a set of operations to deal with files and their contents. A typical set of file operation provided by an operating system may be given as follows:

- *Create:* This is used to create a new file.
- *Delete:* This is used to delete an existing file that is no longer needed.
- *Open:* This operation is used to open an existing file when a user wants to start using it.
- *Close:* When a user has finished using a file, the file must be closed using this operation.
- *Read:* This is used to read data stored in a file.
- *Write:* This is used to write new data in a file.
- *Seek:* This operation is used with random access files to first position the read/write pointer to a specific place in the file, so that data can be read from, or written to, that position.
- *Get Attributes:* This is used to access the attributes of a file.
- *Set Attributes:* This is used to change the user-settable attributes such as protection mode, of a file.
- *Rename:* This is used to change the name of an existing file.
- *Copy:* This is used to create a copy of a file, or to copy a file to an I/O device such as a printer or a display.

1.11 OPERATING SYSTEM STRUCTURE

In this section, we will have a look at 'how various components are put together to form an operating system'. These are discussed as follows:

1.11.1 Layered Structure

A layered design of an operating system architecture attempts to achieve robustness by structuring the architecture into layers with different privileges. The most privileged layer would contain code dealing with interrupt handling and context switching, the layers above that would follow with device drivers, memory management, file systems, user interface, and finally the least privileged layer would contain the applications. MULTICS is a prominent example of a layered operating system, designed with eight layers formed into protection rings, whose boundaries could only be crossed using specialized instructions. Contemporary operating systems, however, do not use the layered design, as it is deemed too restrictive and requires specific hardware support.

Most modern operating systems organize their components into a number of layers (levels), each built on top of lower layers. The bottom layer (layer 0) is the hardware, and the highest layer (layer) is the user interface. The number of in-between layers and their contents vary from one operating system to another. The main advantage of the layered approach is modularity. The layers are selected such that each layer uses the functions and services provided by its immediate lower layer. This approach greatly simplifies the design and implementation of the system because each layer is implemented using only those operations provided by its immediate lower level layer.

1.11.2 Kernel

Kernel is the central component of most computer operating systems; it is a bridge between applications and the actual data processing done at the hardware level. The kernel’s responsibilities include managing the system’s resources (the communication between hardware and software components). Usually as a basic component of an operating system, a kernel can provide the lowest-level abstraction layer for the resources (especially processors and I/O devices) that application software must control to perform its function. It typically makes these facilities available to application processes through inter-process communication mechanisms and system calls.

Operating system tasks are done differently by different kernels, depending on their design and implementation. While monolithic kernels execute all the operating system code in the same address space to increase the performance of the system, microkernels run most of the operating system services in user space as servers, aiming to improve maintainability and modularity of the operating system. A range of possibilities exists between these two extremes.

1.11.3 Monolithic Kernel versus Microkernel

The two commonly used models for kernel design in operating systems are the monolithic kernel and the microkernel. In a monolithic kernel, all OS services run along with the main kernel thread, thus also residing in the same memory area. This approach provides rich and powerful hardware access. Some developers, such as UNIX developer Ken Thompson, maintain that it is “easier to implement a monolithic kernel” than microkernels. The main disadvantages of monolithic kernels are the dependencies between system components, a bug in a device driver might crash the entire system, and the fact that large kernels can become very difficult to maintain.

The microkernel approach consists of defining a simple abstraction over the hardware, with a set of primitives or system calls to implement minimal OS services such as memory management, multitasking, and inter-process communication. Other services, including those normally provided by the kernel, such as networking, are implemented in user-space programs, referred to as servers. Microkernels are easier to maintain than monolithic kernels, but the large number of system calls and context switches might slow down the system because they typically generate more overhead than plain function calls.

A microkernel allows the implementation of the remaining part of the operating system as a normal application program written in a high-level language, and the use of different operating systems on top of the same unchanged kernel. It is also possible to dynamically switch among operating systems and to have more than one active simultaneously.

As the computer kernel grows, a number of problems become evident. One of the most obvious is that the memory footprint increases. This is mitigated to some degree by perfecting the virtual memory system, but not all computer architectures have virtual memory support. To reduce the kernel's footprint, extensive editing has to be performed to carefully remove unneeded code, which can be very difficult with non-obvious interdependencies between parts of a kernel with millions of lines of code. By the early 1990s, due to the various shortcomings of monolithic kernels versus microkernels, monolithic kernels were considered obsolete by virtually all operating system researchers. As a result, the design of Linux as a monolithic kernel rather than a microkernel was the topic of a famous debate between famous scientists, Linus Torvalds and Andrew Tanenbaum. There is merit on both sides of the argument presented in the Tanenbaum and Torvalds debate.

1.11.4 Resident and Non-resident Operating System Modules

With all the functionalities of an operating system implemented, it becomes a large software. Obviously, all the functionalities of an operating system are not needed all the time. As the main memory capacity of a system is limited, it is customary to always keep in the system's memory only a very small part of the operating system and to keep its remaining part on an on-line storage device such as hard disk. Those modules of an operating system that are always kept in the system's main memory are called resident modules and those that are kept on hard disk are called non-resident modules. The non-resident modules are loaded into the memory on demand, that is, as and when they are needed for execution.

The system kernel should not be confused with the resident models of the operating system. The two are not necessarily the same. In fact, for most operating systems they are different. The following two criteria normally determine whether a particular operating system module should be resident:

- Its frequency of use, and
- Whether the system can operate at all with out it.

1.12 OTHER RELATED CONCEPTS

Few other important concepts related to operating systems are briefly discussed as follows:

1.12.1 Real-time Operating system

A Real-Time Operating System (RTOS) is an operating system (OS) intended to serve real-time application requests. A key characteristic of an RTOS is the level of its consistency concerning the amount of time it takes to accept and complete an application's task; the variability is jitter. A hard real-time operating system has less jitter than a soft real-time operating system. The chief design goal is not high throughput, but rather a guarantee of a soft or hard performance category. An RTOS that can usually or generally meet a deadline is a soft real-time OS, but if it can meet a deadline deterministically it is a hard real-time OS. A real-time OS has an advanced algorithm for scheduling. Scheduler flexibility enables a wider, computer-system orchestration of process priorities, but a real-time OS is more frequently dedicated to a narrow set of applications. Key factors in a real-time OS are minimal interrupt latency and minimal thread switching latency, but a real-time OS is valued more for how quickly or how predictably it can respond than for the amount of work it can perform in a given period of time. A few examples of such applications are:

- An aircraft must process accelerometer data within a certain period (say every 20 milliseconds) that depends on the specifications of the aircraft. Failure to do so could cause the aircraft to go away from its right course or may even cause it to crash.
- Failure to respond in time to an error condition in a nuclear reactor thermal power plant could result in a melt down.
- Failure to respond in to time to an error conditions in the assembly lime of a automated factory could result in several product units that will have to be ultimately discarded.
- A request for booking a ticket in computerized railway reservation system must be processed within the passengers; perception of a reasonable time.

1.12.2 Distributed Operating Systems

A Distributed operating system is the logical aggregation of operating system software over a collection of independent, networked, communicating, and spatially disseminated computational nodes. Individual system nodes each hold a discrete software subset of the global aggregate operating system. Each node-level software subset is a composition of two distinct provisioners of services. The first is a ubiquitous minimal kernel, or microkernel, situated directly above each node's hardware. The microkernel provides only the necessary mechanisms for a node's functionality. Second is a higher-level collection of system management components, providing all necessary policies for a node's individual and collaborative activities. This collection of management components exists immediately above the microkernel, and below any user applications or APIs that might reside at higher levels.

These two entities, the microkernel and the management components collection, work together. They support the global system's goal of seamlessly integrating all network-connected resources and processing functionality into an efficient, available, and unified system. This seamless integration of individual nodes into a global system is referred to as transparency, or Single system image; describing the illusion provided to users of the global system's appearance as a singular and local computational entity. The operating systems commonly used for distributed computing systems can be broadly classified into two types of network operating systems and distributed operating systems. The three most important features commonly used to differentiate between these two types of operating systems are system image, autonomy, and fault tolerance capability. These features are explained below:

- **System Image:** The most important feature used to differentiate between the two types of operating system is the image of the distributed computing system from the point of view of its users. In case of a network operating system, the users view the distributed computing system as a collection of distinct machines connected by a communication subsystem. That is the users are aware of the fact that multiple computers are being used. On the other hand, a distributed operating system hides the existence of multiple computers and provides a single system image to its users. That is, it makes a collection of networked machines appear to its users as a virtual uniprocessor by providing similar type of user interface as provided by centralized operating system.

- **Autonomy:** In a network operating each computer of the distributed computing system has it own local operating system (the operating systems of different computers may be the same or different), and there is essentially no coordination at all among the computers except for the rule that when two processes of different computers communicate with each other, they must use a mutually agreed on communication protocol. Each computer functions independently of other computers in the sense that each one makes independent decision about the creation and termination of their own processes and management of local resources. It is noteworthy that due to the possibility of difference in local operating systems, the system call from different computers of the same distributed computing system may be different in this case.

  On the other hand, with a distributed operating system, there is a single system- wide operating system and each computer of the distributed computing system runs a part of this global operating system. The distributed operating system tightly interweaves all the computers of the distributed computing system in the sense that they work in close cooperation with each other for the efficient and effective utilization of the various resources of the system. That is processes and several resources are managed globally (some resources are managed locally). Moreover there is a single set of globally valid system calls available on all computers of the distributed computing system.

- **Fault tolerance capability:** A network operating system provides little or no fault tolerance capability in the sense that of 10% of the machines of the entire distributed computing system are down at any moment, at least 10% of the users are unable to continue with their work. On the other hand, with a distributed operating system, most of the users are normally unaffected by the failed machines and can continue to perform their work normally, with only a 10% loss in performance of the entire distributed computing system. Therefore, the fault tolerance capability of distributed operating system is usually very high a compared to that of a network operating system.

In short, both network operating systems and distributed operating system deal with multiple computers interconnected together by a communication network. In case of a network operating system the user view the system as a collection a distinct computers, but in case of distributed operating system the user views the system as a ‘virtual uniprocessor’.