Chapter 1
INTRODUCTION

An ***operating system*** is a program that acts as an intermediary between a user of a computer and the computer hardware. The purpose of an operating system is to provide an environment in which a user can execute programs in a ***convenient*** and ***eficient*** manner. The operating system provides certain services to programs and to the users of those programs in order to make the programming task easier.

1. 1 Operating System Definition :

An operating system is an important part of almost every computer system. A computer system can be divided roughly into four components: the *hardware,* the *operating system,* the *applications programs,* and the *users.*

The **hardware** - the central processing unit (CPU), the memory, and the input/output (I/0) devices - provides the basic computing resources. The **applications programs** - such as compilers, database systems, games, and business programs - define the ways in which these resources are used to solve the computing problems of the users. There may be many different users (people, machines, other computers) trying to solve different problems.

Accordingly, there may be many different applications programs.

1. The operating system controls and coordinates the use of the hardware among the various applications programs for the various users.

2. An operating system is similar to a *government.* The components of a computer system are its hardware, software, and data. The operating system provides the means for the proper use of these resources in the operation of the computer system. Like a government, the operating system performs no useful function by itself. It simply provides an *environment* within which other programs can do useful work.

3. We can view an operating system as a *resource allocator.* A computer system has many resources (hardware and software) that may be required to solve a problem: CPU time, memory space, file storage space, I/O devices, and so on.

4. The operating system acts as the manager of these resources and allocates

them to specific programs and users as necessary for tasks. Since there may be many-possibly conflicting-requests for resources, the operating system must decide which requests are allocated resources to operate the computer system efficiently and fairly.

5. An Operating systems exist because they are a reasonable way to solve the problem of creating a usable computing system.
6. The fundamental goal of computer systems is to execute user programs and to make solving user problems easier. Toward this goal, computer hardware is constructed. Since bare hardware alone is not particularly easy to use, applications programs are developed. These various programs require certain common operations, such as those controlling the I/O devices. The common functions of controlling and allocating resources are then brought together into one piece of software: the operating system.

1.1 Definition of the operating system:

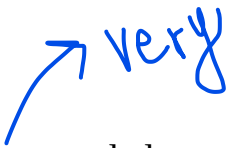
OS is the one program running at all times on the computer (usually called the kernel), with all else being applications programs. It is easier to define operating systems by what they do than by what they are.

The goal of Operating system

1. The primary goal of an operating system is convenience for the user. Operating systems exist because they are supposed to make it easier to compute with them than without them. This view is particularly clear when you look at operating systems for small personal computers.
2. A secondary goal is efficient operation of the computer system. This goal is particularly important for large, shared multiuser systems. These systems are typically expensive, so it is desirable to make them as efficient as possible.

These two goals--convenience and efficiency-are sometimes contradictory. In the past, efficiency considerations were often more important than convenience. Thus, much of operating-system theory concentrates on optimal use of computing resources.

1.2 Simple Batch Systems



Early computers were (physically) enormously large machines run from a console.

1. **Input devices :** card readers and tape drives.
2. **Output devices:** line printers, tape drives, and card punches.
3. The users of such systems did not interact directly with the computer systems.
4. Rather, the user prepared a job-which consisted of the program, the data, and some control information about the nature of the job (control cards)-and submited it to the computer operator.
5. The job would usually be in the form of punch cards. At some later time (perhaps minutes, hours, or days), the output appeared
6. The output consisted of the result of the program, as well as a dump of memory and registers in case of program error.

The operating system in these early computers was fairly simple. Its major task was to transfer control automatically from one job to the next. The operating system was always (resident) in memory (Figure 1.2).

To speed up processing, jobs with similar needs were ***batched*** together and were run through the computer as a group. Thus, the programmers would leave their programs with the operator. The operator would sort programs into batches with similar requirements and, as the computer became available, would run each batch. The output from each job would be sent back to the appropriate programmer.

A batch operating system, thus, normally reads a stream of separate jobs (from a card reader, for example), each with its own control cards that predefine what the job does.

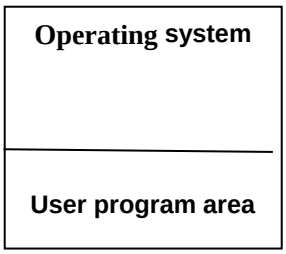


Figure 1.2 Memory layout for a simple batch system.

When the job is complete, its output is usually printed (on a line printer, for example).

The definitive feature of a batch system is the *lack* of interaction between the user and the job while that job is executing. The job is prepared and submitted, and at some later time, the output appears.

The delay between job submission and job completion (called *turnaround* time) may result from the amount of computing needed , or from delays before the operating system starts to process the job.

This execution environment, the CPU is often idle. This idleness occurs because the speeds of the mechanical I/O devices are intrinsically slower than those of electronic devices.

Spooling, in essence, uses the disk as a huge buffer, for reading as far ahead as possible on input devices and for storing output files until the output devices are able to accept them.

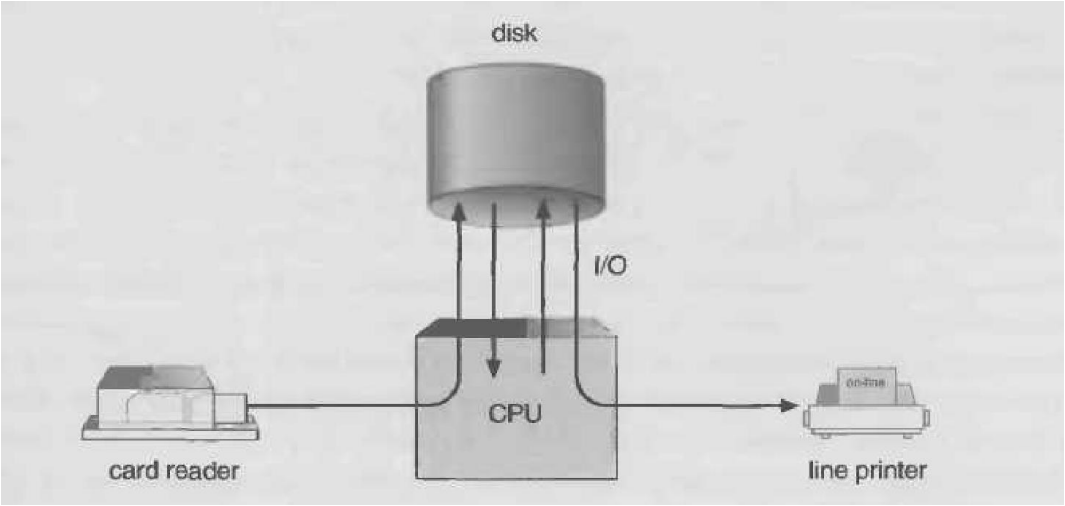


Figure 1.3 Spooling

Spooling is also used for processing data at remote sites. The CPU sends the data via communications paths to a remote printer (or accepts an entire input job from a remote card reader). The remote processing is done at its own speed, with no CPU intervention. The CPU just needs to be notified when the processing is completed, so that it can spool the next batch of data.

Spooling overlaps the I/O of one job with the computation of other jobs. Even in a simple system, the spooler may be reading the input of one job while printing the output of a different job. During this time, still another job (or jobs) may be executed, reading their "cards" from disk and "printing" their output lines onto the disk.

Spooling has a direct beneficial effect on the performance of the system. For the cost of some disk space and a few tables, the computation of one job can overlap with the I/O of other jobs. Thus, spooling can keep both the CPU and the I/O devices working at much higher rates.

1.3 Multiprogrammed Batched Systems

Spooling provides an important data structure: a ***job pool.*** Spooling will generally result in several jobs that have already been read waiting on disk, ready to run. A pool of jobs on disk allows the operating system to select which job to run next, to increase CPU utilization. When jobs come in directly on cards or even on magnetic tape, it is not possible to run jobs in a different order.

Jobs must be run sequentially, on a first-come, first-served basis. However, when several jobs are on a direct-access device, such as a disk, ***job scheduling*** becomes possible.

A few important aspects are covered here. The most important aspect of job scheduling is the ability to ***multiprogram.*** Off-line operation and spooling for overlapped I/O have their limitations. A single user can not, in general, keep either the CPU or the I/O devices busy at all times. Multiprogramming increases CPU utilization by organizing jobs such that the CPU always has one to execute.

The idea is as follows.

The operating system keeps several jobs in memory at a time (Figure 1.4). This set of jobs is a subset of the jobs kept in the job pool (since the number of jobs that can be kept simultaneously in memory is usually much smaller than the number of jobs that can be in the job pool.) The operating system picks and begins to execute one of the jobs in the memory.

In a multiprogramming system, the operating system simply switches to and executes another job. When ***that*** job needs to wait, the CPU is switched to another job, and so on. Eventually, the first job finishes waiting and gets the CPU back. As long as there is always some job to execute, the CPU will never be Idle.

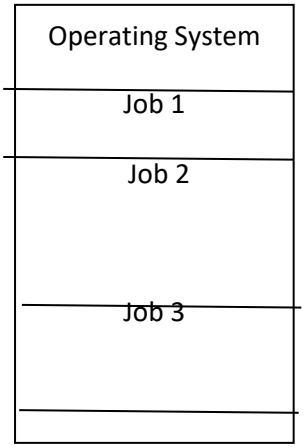


Figure 1.4 Memory layout for a multiprogramming system.

Multiprogramming is the first instance where the operating system must make decisions for the users. Multiprogrammed operating systems are therefore fairly sophisticated. All the jobs that enter the system are kept in the job pool. This pool consists of all processes residing on mass storage awaiting allocation of main memory. If several jobs are ready to be brought into memory, and there is not enough room for all of them, then the system must choose among them. Making this decision is job ***scheduling.***

When the operating system selects a job from the job pool, it loads that job into memory for execution. Having several programs in memory at the same time requires having some form of memory management.

In addition, if several jobs are ready to run at the same time, the system must choose among them. Making this decision is ***CPUscheduling.***

1.4 Time-sharing Systems

Multiprogrammed batched systems provide an environment where the various system resources (for example, **CPU,** memory, peripheral devices) are utilized effectively. There are some difficulties with a batch system from the point of view

of the user, however. Since the user cannot interact with the job when it is executing, the user must set up the control cards to handle all possible outcomes.

In a multistep job, subsequent steps may depend on the result of earlier ones. The running of a program, for example, may depend on successful compilation. It can be difficult to define completely what to do in all cases. Another difficulty is that programs must be debugged statically, from snapshot dumps. A programmer cannot modify a program as it executes to study its behaviour. A long turnaround time inhibits experimentation with a program. (Conversely, this situation may instill a certain amount of discipline into the writing and testing of programs.)

Time sharing, or multitasking, is a logical extension of multiprogramming. Multiple jobs are executed by the CPU switching between them, but the switches occur so frequently that the users may interact with each program while it is running.

An interactive, or hands-on, computer system provides on-line communication between the user and the system. The user gives instructions to the operating system or to a program directly, and receives an immediate response. Usually, a keyboard is used to provide input, and a display screen (such as a cathode-ray tube (CRT), or monitor) is used to provide output. When the operating system finishes the execution of one command, it seeks the next "control statement" not from a card reader, but rather from the user's keyboard. The user gives a command, waits for the response, and decides on the next command, based on the result of the previous one. The user can easily experiment, and can see results immediately. Most systems have an interactive text editor for entering programs, and an interactive debugger for assisting in debugging programs.

If users are to be able to access both data and code conveniently, an on-line file system must be available.

A file is a collection of related information defined by its creator. Commonly, files represent programs (both source and object forms) and data.

Data files may be numeric, alphabetic, or alphanumeric. Files may be free-form, such as text files, or may be rigidly formatted. In general, a file is a sequence of bits, bytes, lines, or records whose meaning is defined by its creator and user.

The operating system implements the abstract concept of a file by managing mass-storage devices, such as tapes and disks. Files are normally organized into logical clusters, or directories, which make them easier to locate and access. Since multiple users have access to files, it is desirable to control by whom and in what ways files may be accessed.

Batch systems are appropriate for executing large jobs that need little interaction. The user can submit jobs and return later for the results; it is not necessary for the user to wait while the job is processed. Interactive jobs tend to be composed of many short actions, where the results of the next command may be unpredictable. The user submits the command and then waits for the results.

Accordingly, the response time should be short--on the order of seconds Early computers with a single user were interactive systems. That is, the entire system was at the immediate disposal of the programmer/operator. This situation allowed the programmer great flexibility and freedom in program testing and development. But, as we saw, this arrangement resulted in substantial idle time while the CPU waited for some action to be taken by the programmer/operator. Because of the high cost of these early computers, idle CPU time was undesirable. Batch operating systems were developed to avoid this problem. Batch systems improved system utilization for the owners of the computer systems.

Time-sharing systems were developed to provide interactive use of a computer system at a reasonable cost.

1. A time-shared operating system uses CPU scheduling and multiprogramming to provide each user with a small portion of a time-shared computer.

2. Each user has at least one separate program in memory. A program that is loaded into memory and is executing is commonly referred to as a ***process.***

3. When a process executes, it typically executes for only a short time before it either finishes or needs to perform I/O.

I/O may be interactive; that is, output is to a display for the user and input is from a user keyboard. Since interactive I/O typically runs at people speeds, it may take a long time to complete. Input, for example, may be bounded by the user's typing speed; five characters per second is fairly fast for people, but is incredibly slow

for computers. Rather than let the CPU sit idle when this interactive input takes place, the operating system will rapidly switch the CPU to the program of some other user.

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

1.5 Personal-Computer Systems

As hardware costs have decreased, it has once again become feasible to have a computer system dedicated to a single user. These types of computer systems are usually referred to as personal computers (PCs). The I/O devices have certainly changed, with panels of switches and card readers replaced with typewriter like keyboards and mice. Line printers and card punches have succumbed to display screens and to small, fast printers.

Personal computers appeared in the 1970s. They are microcomputers that are considerably smaller and less expensive than mainframe systems. During their first decade, the CPUs in PCs lacked the features needed to protect an operating system from user programs. PC operating systems therefore were neither multiuser nor multitasking. However, the goals of these operating systems have changed with time; instead of maximizing CPU and peripheral utilization, the systems opt for maximizing user convenience and responsiveness. These systems include PCs running Microsoft Windows, and the Apple Macintosh.

The MS-DOS operating system from Microsoft has been superseded by multiple flavors of Microsoft Windows, and IBM has upgraded MS-DOS to the OS/2 multitasking system. The Apple Macintosh operating system has been ported to more advanced hardware, and now includes new features such as virtual memory.

Operating systems for these computers have benefited from the development of operating systems for mainframes in several ways. Microcomputers were immediately able to adopt the technology developed for larger operating systems. On the other hand, the hardware costs for microcomputers are sufficiently low that individuals have sole use of the computer, and CPU utilization is no longer a

prime concern. Thus, some of the design decisions that are made in operating systems for mainframes may not be appropriate for smaller systems. For example, file protection may not seem necessary on a personal machine.

The decrease in hardware costs will allow relatively sophisticated operating-system concepts (such as time sharing and virtual memory) to be implemented on an even greater number of systems. Thus, the decrease in the cost of computer hardware, such as of microprocessors, will increase our need to understand the concepts of operating systems.

1.6 Parallel Systems

Most systems to date are single-processor systems; that is, they have only one main CPU. However, there is a trend toward *multiprocessor systems*. Such systems have more than one processor in close communication, sharing the computer bus, the clock, and sometimes memory and peripheral devices. These systems are referred to as *tightly coupled* systems.

There are several reasons for building such systems. One advantage is increased *throughput*. By increasing the number of processors, we hope to get more work done in a shorter period of time. The speed-up ratio with *n* processors is not *n*, however, but rather is less than *n*. When multiple processors cooperate on a task, a certain amount of overhead is incurred in keeping all the parts working correctly. This overhead, plus contention for shared resources, lowers the expected gain from additional processors. Similarly, a group of *n* programmers working closely together does not result in *n* times the amount of work being accomplished. Multiprocessors can also save money compared to multiple single systems because the processors can share peripherals, cabinets, and power supplies. If several programs are to operate on the same set of data, it is cheaper to store those data on one disk and to have all the processors share them, rather than to have many computers with local disks and many copies of the data.

Another reason for multiprocessor systems is that they increase reliability. If functions can be distributed properly among several processors, then the failure of one processor will not halt the system, but rather will only slow it down.

If we have 10 processors and one fails, then each of the remaining nine processors must pick up a share of the work of the failed processor. Thus, the entire system

runs only 10 percent slower, rather than failing altogether. This ability to continue providing service proportional to the level of surviving hardware is called ***graceful degradation.*** Systems that are designed for graceful degradation are also called ***fault-tolerant.***

Continued operation in the presence of failures requires a mechanism to allow the failure to be detected, diagnosed, and corrected (if possible). The Tandem system uses both hardware and software duplication to ensure continued operation despite faults. The system consists of two identical processors, each with its own local memory. The processors are connected by a bus. One processor is the primary, and the other is the backup. Two copies are kept of each process; one on the primary machine and the other on the backup. At fixed checkpoints in the execution of the system, the state information of each job (including a copy of the memory image) is copied from the primary machine to the backup.

If a failure is detected, the backup copy is activated, and is restarted from the most recent checkpoint. This solution is obviously an expensive one, since there is considerable hardware duplication.The most common multiple-processor systems now use the s***ymmetricmultiprocessing*** model, in which each processor runs an identical copy of the operating system, and these copies communicate with one another as needed. Some systems use ***asymmetric multiprocessing,*** in which each processor is assigned a specific task. A master processor controls the system; the other processors either look to the master for instruction or have predefined tasks. This scheme defines a master-slave relationship. The master processor schedules and allocates work to the slave processors.

An example of the symmetric multiprocessing system is Encore's version of UNIX for the Multimax computer. This computer can be configured to employ dozens of processors, all running a copy of UNIX. The benefit of this model is that many processes can run at once (N processes if there are N CPUs) without causing a deterioration of performance. However, we must carefully control I/O to ensure that data reach the appropriate processor. Also, since the CPUs are separate, one may be sitting idle while another is overloaded, resulting in inefficiencies. To avoid these inefficiencies, the processors can share certain data structures.

A multiprocessor system of this form will allow jobs and resources to be shared dynamically among the various processors, and can lower the variance among the systems.

Asymmetric multiprocessing is more common in extremely large systems , here one of the most time-consuming activities is simply processing I/O. In older batch systems, small processors, located at some distance from the main CPU, were used to run card readers and line printers and to transfer these jobs to and from the main computer. These locations are called ***remote-job-entry*** *(RJE)* sites.

In a time-sharing system, a main I/0 activity is processing the I/O of characters between the terminals and the computer. If the main CPU must be interrupted for every character for every terminal, it may spend all its time simply processing characters. So that this situation is avoided, most systems have a separate front-end processor that handles all the terminal I/O.

1.7 Real-Time Systems

Another form of a special-purpose operating system is the real-time system. A real-time system is used when there are rigid time requirements on the operation of a processor or the flow of data, and thus is often used as a control device in a dedicated application.

Sensors bring data to the computer. The computer must analyze the data and possibly adjust controls to modify the sensor inputs. Systems that control scientific experiments, medical imaging systems, industrial control systems, and some display systems are real-time systems. Also included are some automobile-engine fuel-injection systems, home-appliance controllers, and weapon systems.

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

There are two flavours of real-time systems. A hard real-time system guarantees that critical tasks complete on time. This goal requires that all delays in the system be bounded, from the retrieval of stored data to the time that it takes the operating system to finish any request made of it. Such time constraints dictate the facilities that are available in hard real-time systems. Secondary storage of any sort is usually limited or missing, with data instead being stored in short term memory, or in read-only memory (ROM). ROM is located on non-volatile storage devices that retain their contents even in the case of electric outage; most other types of memory are volatile. Most advanced operating-system features are absent too, since they tend to separate the user further from the hardware, and that separation results in uncertainty about the amount of time an operation will take.

Soft real time is an achievable goal that is amenable to mixing with other types of systems. Soft real-time systems, however, have more limited utility than do hard real-time systems. Given their lack of deadline support, they are risky to use for industrial control and robotics. There are several areas in which they are useful, however, including multimedia, virtual reality, and advanced scientific projects such as undersea exploration and planetary rovers. These systems need advanced operating-system features that cannot be supported by hard real-time systems. Because of the expanded uses for soft real-time functionality, it is finding its way into most current operating systems, including major versions of UNIX.

1.8 System Calls

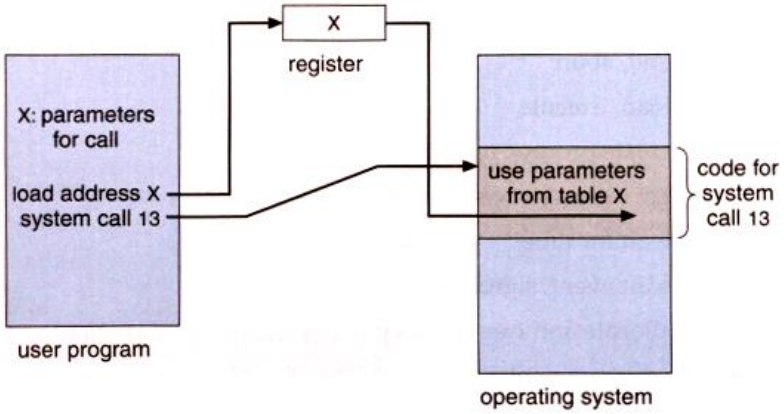
The system call provides an interface to the operating system services.
Application developers often do not have direct access to the system calls, but can access them through an application programming interface (API). The functions that are included in the API invoke the actual system calls. By using the API, certain benefits can be gained:

- Portability: as long a system supports an API, any program using that API can compile and run.
- Ease of Use: using the API can be significantly easier then using the actual system call.

System Call Parameters

Three general methods exist for passing parameters to the OS:

1. Parameters can be passed in registers.
2. When there are more parameters than registers, parameters can be stored in a block and the block address can be passed as a parameter to a register.
3. Parameters can also be pushed on or popped off the stack by the operating system.



Types of System Calls

There are 5 different categories of system calls:

process control, file manipulation, device manipulation, information maintenance and communication.

Process Control

A running program needs to be able to stop execution either normally or abnormally. When execution is stopped abnormally, often a dump of memory is taken and can be examined with a debugger.

File Management

Some common system calls are create, delete, read, write, reposition, or close. Also, there is a need to determine the file attributes – get and set file attribute. Many times the OS provides an API to make these system calls.

Device Management

Process usually require several resources to execute, if these resources are available, they will be granted and control returned to the user process. These resources are also thought of as devices. Some are physical, such as a video card, and others are abstract, such as a file.

User programs request the device, and when finished they release the device. Similar to files, we can read, write, and reposition the device.

Information Management

Some system calls exist purely for transferring information between the user program and the operating system. An example of this is time, or date.

The OS also keeps information about all its processes and provides system calls to report this information.

Communication

There are two models of interprocess communication, the message-passing model and the shared memory model.

Message-passing uses a common mailbox to pass messages between processes.

Shared memory use certain system calls to create and gain access to create and gain access to regions of memory owned by other processes. The two processes exchange information by reading and writing in the shared data.