Operating Systems: An Overview

The purpose of an operating system is to share computational resources among competing users. Operating system is control program(s) which manages computer resources and dynamically allocate these resources among competing tasks. It provides a base environment for application programs to be executed and acts as an intermediary between the computer user and the computer hardware. We can say that operating system is an interface between computer hardware and users' programs. One of the earliest definitions of operating system of main frame era is still applicable for networked and distributed computing resources era.

> "An *operating system* is a set of manual and automatic procedures that enable a group of people to share a computer installation efficiently. The key word in this definition is *sharing:* it means that people will *compete* for the use of physical resources such as processor time, storage space, and peripheral devices; but it also means that people can *cooperate* by exchanging programs and data on the same installation." [P. B. Hansen 1973]

From the above stated definition, we can easily define main objectives of an operating system are ***convenience*** and ***efficiency***. An operating system makes a computer more convenient to use by providing an environment on which applications programs are developed and executed. An operating system allows the computer system resources to be used in an efficient manner. Due to varying computing resources in a computer system, operating system's design focus varies from convenient for users, to efficient for resources utilization. Even some operating system may have to be convenient and efficient at the same time.

Operating systems allocate and deallocate computing resources (CPU, Memory, File system, I/O devices) to competing tasks in an efficient manner and can be called a resource manager. To manage varying resources, operating systems are designed in a modular way to have different functionalities managed by different modules. Operating system is all the time memory resident and is running most of the time to manage resources to provide an environment for execution of user's programs. Not all programs require all the functionalities of the operating system all the time. To utilize memory in an efficient manner, only those modules of the operating system required all the time are kept in memory. Those modules are called '**kernel**'. Other modules required occasionally are disk resident and are placed in memory when required.

Digital computing has a short history with rapid developments in hardware. Along with developments in hardware resources, operating systems were designed and developed to manage those varying available resources. In 1950's computing resources were relatively scarce. During that era, operating systems were either batch or interactive. Batch operating systems processed jobs in bulk, with predetermined input from files or other data sources. In batch operating systems, only one program was memory resident, and a batch of same types of programs were executed one after the other one. Interactive systems waited for input from users. To optimize the use of the computing resources, multiple users shared time on these systems and operating system was timesharing one. Time-sharing operating systems used a timer and scheduling algorithms to cycle processes rapidly through the CPU, giving each user a share of the resources.

Operating systems has evolved as per evolution of computing resources and usage of computer systems by users. We describe historical development of operating systems.

Operating Systems Development: Historical Perspective

To describe historical overview of operating systems, we must discuss a historical overview of early computer systems. We begin from 1940's when the idea of a stored program computer was being experimented. Such a computer had both a program store and a data store, where the program store provides instructions about what to do to the data.

First working stored-program general-purpose computer was the Manchester Mark 1, which ran successfully in 1949. The first commercial computer- the Ferranti Mark 1, which went on sale in 1951 was an offspring of Manchester Mark 1.

Those early computers were physically enormous machines run from consoles. The programmer, who was also the operator of the computer system, would write a program (in binary form) and then would operate the program directly from the operator's console. First, the program would be loaded manually into memory from the front panel switches (one instruction at a time), from paper tape, or from punched cards. Then the appropriate buttons would be pushed to set the starting address and to start the execution of the program. As the program ran, the programmer/operator could monitor its execution by the display lights on the console. If errors were discovered, the programmer could halt the program, examine the contents of memory and registers, and debug the program directly from the console. Output was printed or was punched onto paper tape or cards for later printing.

Dedicated Computer Systems

Within next couple of years, additional software and hardware were developed to create **dedicated computer systems**. Card readers, line printers, and magnetic tape became commonplace. The routines that performed I/O were especially important. Each new I/O device had its own characteristics, requiring careful programming. A special subroutine - called a **device driver -** was written for each I/O device. A device driver knows how the buffers, flags, registers, control bits, and status bits for a particular device should be used. Each type of device has its own driver. A simple task, such as reading a character from a paper-tape reader, might involve complex sequences of device-specific operations. Rather than writing the necessary code every time, the device driver was simply used from the library. Assemblers, loaders, and linkers were designed to ease the programming (in mnemonic form) task. Libraries of common functions were created. Common functions could then be copied (using same deck of cards, or magnetic tape) into a new program without having to be written again, providing software reusability.

Later, compilers for FORTRAN, COBOL, and other languages appeared, making the programming task much easier but the operations of computer more complex. Compiling at that time was a two-stage process, first to translate high language program into assembly language program and then translate assembly language program into machine code. To prepare a FORTRAN program for execution, for example, the programmer would first need to load the FORTRAN compiler into the computer. The compiler was normally kept on magnetic tape, so the proper tape would need to be mounted on a tape drive. The program would be read through the card reader and written onto another tape. The FORTRAN compiler produced assembly-language output, which then had to be assembled. This procedure required mounting another tape with the assembler. The output of the assembler would need to be linked to supporting library routines. Finally, the binary object form of the program would be ready to execute. It could be loaded into memory and debugged from the console, as before.

A significant amount of setup time could be involved in the running of a job. Each job consisted of many separate steps:

1. Loading the COBOL compiler tape
2. Running the compiler
3. Unloading the compiler tape
4. Loading the assembler tape
5. Running the assembler
6. Unloading the assembler tape
7. Loading the object program
8. Running the object program

If an error occurred during any step, the programmer/operator might have to start over at the beginning. Each job step might involve the loading and unloading of magnetic tapes, paper tapes, and punch cards.

The job setup time was a real problem. While tapes were being mounted or the operator was operating the console, the CPU sat idle. Multi-million dollars computer time was extremely valuable, and owners wanted high utilization of computer system, so concept of **Shared Computer System** was explored.

Resident Monitor and Batches of Jobs

First, a professional computer operator was hired. The programmer no longer operated the machine. As soon as one job was finished, the operator could start the next. Since the operator had more experience with mounting tapes than a programmer, setup time was reduced. The programmer provided whatever cards or tapes were needed, as well as a short description of how the job was to be run. Of course, the operator could not debug an incorrect program at the console since the operator would not understand the program. Therefore, in the case of program error, a dump of memory and registers was taken, and the programmer had to debug from the dump. Dumping the memory and registers allowed the operator to continue immediately with the next job but left the programmer with the more difficult debugging problem.

Second, jobs with similar needs were batched together and run through the computer as a group to reduce setup time. For instance, suppose the operator received one FORTRAN job, one COBOL job, and another FORTRAN job. If he ran them in that order, he would have to set up for FORTRAN (load the compiler tapes and so on), then set up for COBOL, and then set up for FORTRAN again. If he ran two FORTRAN programs as a batch, however, he could setup only once for FORTRAN, saving operator time.

But there were still problems. For example, when a job stopped, the operator would have to notice that it had stopped (by observing the console), determine why it stopped (normal or abnormal termination), dump memory and register (if necessary), load the appropriate device with the next job, and start the processing of next task. During this transition from one job to the next, the CPU sat idle.

Automatic Job Sequencing

To overcome this idle time, operating system designers developed automatic job sequencing. With this technique, the first rudimentary operating system was created. A small program, called a **resident monitor**, was designed to transfer control automatically from one job to the next. The resident monitor is always in memory.

When the computer was turned on, the resident monitor was invoked, and it would transfer control to a program. When the program terminated, it would return control to the resident monitor, which would then go on to the next program. Thus, the resident monitor would automatically sequence from one program to another and from one job to another. Resident monitor must be told which program to execute? Previously, the operator had been given a short description of what programs were to be run on what data. Control cards were introduced to provide this information directly to the monitor. This was the first interface between user program and resident monitor. The idea is simple. In addition to the program or data for a job, the programmer supplied control cards, which contained directives to the resident monitor indicating what program to run. For example, a normal user program might require one of three programs to run: the FORTRAN compiler (FTN), the assembler (ASM), or the user's program (RUN). A separate control card was used for each activity to be performed by the resident monitor.

$FTN - Execute the FORTRAN compiler.
$ASM - Execute the assembler.
$RUN - Execute the user program.

Two additional control cards were used to define the boundaries of each job along with accounting for the machine resources used by the programmer:

$JOB - First card of a job
$END - Final card of a job

Parameters were used to define the job name, account number to be charged, and so on. Other control cards can be defined for other functions, such as asking the operator to load or unload a tape. One problem with control cards was how to distinguish them from data or program cards. The usual solution was to identify them by a special character or pattern on the card. Several systems used the dollar-sign character ($) in the first column to identify a control card. Others used a different code. IBM's Job Control Language (JCL) used slash marks (//) in the first two columns. Figure-1 shows a sample cards setup for a simple batch system.

```
//JOB
//FTN
•
•   FORTRAN instructions
•
//LOAD
//RUN
•
•   Data
•
//END
```

Figure-1: Control cards for a simple program.

These batch systems worked fairly well. The resident monitor provided automatic job sequencing as indicated by the control cards. When a control card indicates that a program is to be run, the monitor loads the program into memory and transfers control to it. When the program completes, it transfers control back to the monitor, which reads the next control card, loads the appropriate program, and so on. This cycle is repeated until all control cards are interpreted for the job. Then the monitor automatically continues with the next job.

Batch systems with automatic job sequencing improved performance of computer systems, since it replaced human operation with operating-system software. However, most computer systems in the late 1950s and early 1960s were batch systems reading from card readers and writing to line printers or card punches. Due to speed disparity between CPU and IO devices, even in batch system CPU was often idle.

IO Operations Overlapping

To overcome speed disparity of CPU and IO devices, concept of buffering was introduced. A small highspeed memory (buffer) is made part of card reader and printer. This buffer is used to read a card before it is required for computation and is available for computation when required without any delay (IO operation and computation is overlapped). Buffering is overlapping of an I/O operation of the job with its computation.

Another solution to overcome speed disparity of CPU and IO devices, was offered by computer manufactures, to replace slow card readers (input devices) and line printers (output devices) with magnetic-tape units. The CPU did not read directly from cards, however; instead, the cards were first copied onto a magnetic tape via a separate device (low-cost special purpose computer). When the tape was sufficiently full, it was taken down and carried over to the computer. When a card was needed for input to a program, the equivalent record was read from the tape. Similarly, output was written to the tape, and the contents of the tape were printed later. The card readers and line printers were operated **off-line**, rather than by the main computer. Figure-2 illustrate off-line and on-line IO operations using IBM-1401 and IBM-7094 computers, where main computer performed IO operation online on magnetic tapes.

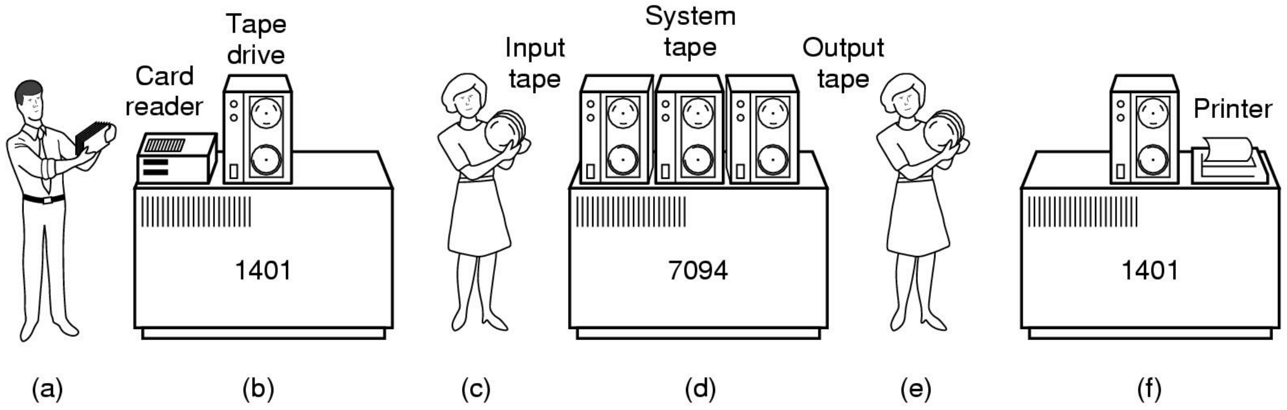

Figure-2: Satellite Computers

With the availability of magnetic disk systems, the off-line preparation of jobs was replaced in most systems. Disk, being direct access storage device, it allowed direct access to any area of disk in contrast to magnetic tape (only sequential access). With the availably of disk, concept of spooling (simultaneous peripheral operation on-line) was introduced where disk is used as a huge buffer for reading as far ahead as possible on input devices and for storing output files until the output devices are ready to accept them as shown in Figure-3. 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 other jobs) may be executed, reading its “cards” from disk and printing” its output lines onto the disk. Spooling has a beneficial

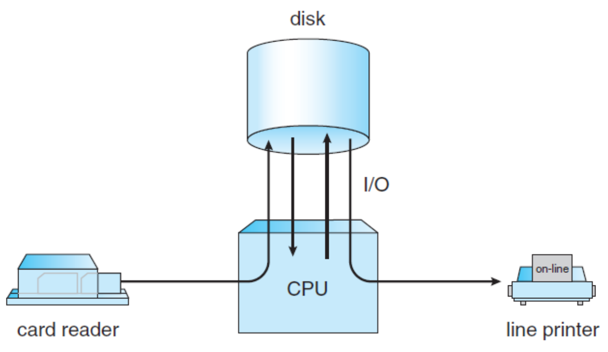

Figure-3: Spooling

effect on the performance of the system. By using some disk space and a few tables, the computation of one job and the I/O of other jobs can take place at the same time.

Remote Site Data Processing (Remote Printer/Card-reader)
Spooling made it possible for processing data at remote sites. The CPU sends the data via communication 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 can keep both the CPU and the I/O devices (even at remote sites) working at much higher rates.

Multiprogramming Operation System
Spooling leads naturally to multiprogramming, which is the foundation of all modern operating systems. You must keep in mind that apart from early computers being operated and managed by the programmer, dedicated and shared computers were managed by a computer operator, who was responsible for directly interacting with operating system on user’s behalf. Initially pure multi programming operating system was designed, where more than one programs were residing in memory at any given time. Programs address space was protected by relocatable addresses and pair of base and limit registers was used to restrict the executing program within its address range. Simple non-preemptive scheduling was used to allocate CPU to one of the memory resident programs. At a given time while one process is using CPU, IO devices were performing IO operations initiated by previously running processes. CPU, memory, and IO devices utilization was improved considerably as compared to batch OS. Since scheduling was voluntary by a process, a CPU bound process can monopolize CPU usage, while IO bound processes are waiting to use CPU, which in most cases, turn into poor utilization of IO devices because IO bound processes get CPU after waiting and are not able to initiate IO operations frequently. To overcome this problem of unfair use of CPU by compute bound processes, preemptive scheduling policy to allocate CPU was implemented. Processes were allocated priorities statically as numeric value. For our description we assume processes are allocated priorities as low and high. When a low priority process is using CPU, and a high priority process arrives, then system will preempt and take CPU from low priority process and allocate CPU to high priority process. This priority-based operating was called Multiprogramming with Priority Operating System.

Time-Sharing Operation System
With computers being made available with terminals, it was possible for users to directly interact with computers. For a user, now it was possible to perform different steps for program development to program execution in an interactive manner. One large step of a computing job is now consisting of multiple steps (editing, compiling, linking, loading, executing). Time-Sharing operating system is basically multiprogramming with interactivity. More than one program is memory resident, using computing resources. Using a preemptive scheduling, CPU is allocated to a process for a fixed time slot. During this time, that process is using all the resources like being used only by one process. CPU time is shared among competing processes, so it is called Time-Sharing. Just like in multiprogramming operating system, along with interactivity and ease of use for users, resources utilization is improved. In time-sharing OS, remote procedure call function also facilitated users at remote site. These users by connecting a computer terminal with a telephone network were able to interact with the system.

General Purpose OS
During 1970's in main frame computers era, most of the computer installations were providing services to different types of applications. To cater these requirements mostly main frames were managed by a general-purpose OS. In such environments, OS is managing some jobs in batch mode, some jobs are executed in multi-programed mode, and others interactive jobs are managed in timesharing mode.

Distributed Computing Resources
With successful usage of remote card reader, remote printer and telephone networks for remote procedure calls, the concept of distributed digital computing resources was explored. In one approach the distributed computing resources were tightly coupled to create parallel processing computer systems. These special purpose computers were used for multi/parallel processing in special fields. Whereas loosely coupled distributed computing resources, called distributed system, was used for general purpose usage of computer systems.
Figure-4 illustrate five computing units being networked by using a bus topology.

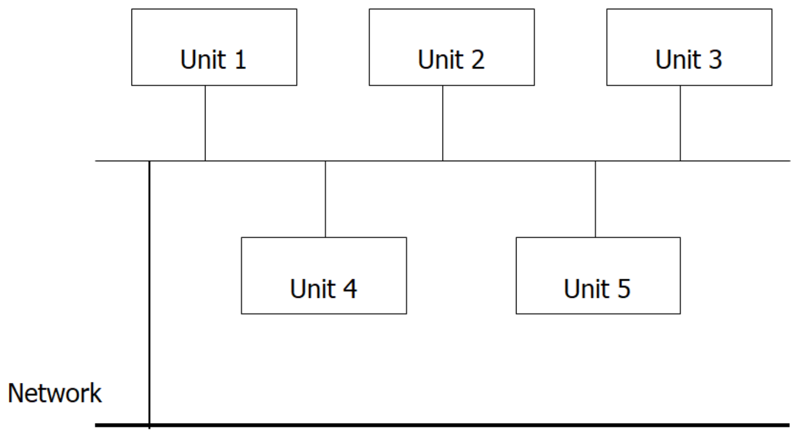

Figure-4

The distributed system, with every unit having its CPU and memory were managed in two ways.

Network Operation System
One through Network Operating System, where every unit is loosely coupled through a network and each unit has a communication module, local OS, and network OS. In this environment every computing resource is visible, and the user uses each resource explicitly through a communication module. Network OS is a small set of procedures, which facilitate data/file transfer, remote usage of resources. Figure-5 show different components of Network OS.

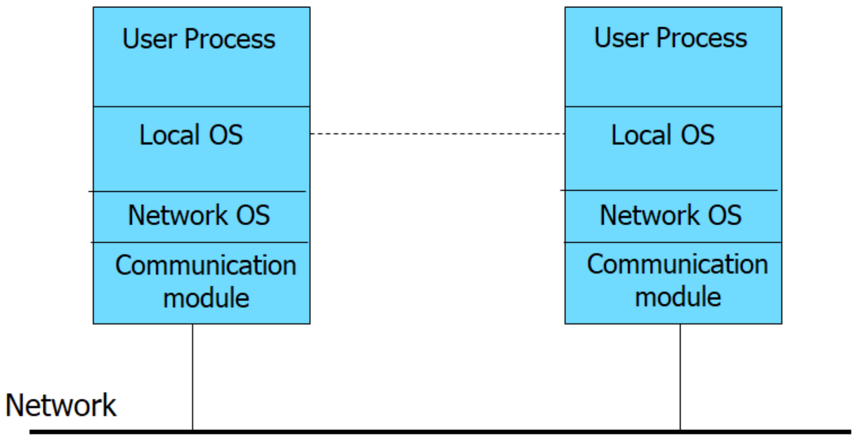

Figure-5: Network OS

Network operating system is useful to share data, files, hardware devices and printer resources among multiple computers to communicate with each other. In a distributed system managed by a Network OS, if each unit has same local OS, it is called homogeneous Network OS. If different units, use different local OS, it called heterogeneous Network OS. Homogeneous Network OS has small number of functions, whereas heterogeneous Network OS is complex due to command interface translation of different operating systems.

Distributed Operating System
In distributed OS, all resources are accessed by the user like a one virtual computer with large memory, high speed processing and a large number of varying IO devices. Operating system hides resources and location. Users interact like with a centralized OS and a central computer, and is not aware of which unit is running user process and which other resources are being used from which location. Figure-6 show different components of Distributed OS.

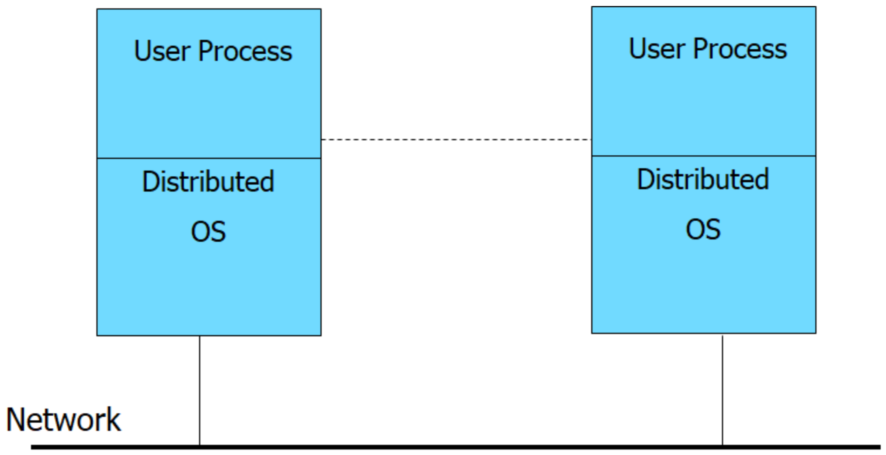

Figure-6

Embedded Operating System
Beginning of new century, witnessed, the computer system's embedded hardware configurations on a very large scale, almost in every field. Embedded operating system is the specific purpose operating system used in the computer system's embedded hardware. These operating systems are designed to work on dedicated devices like automated teller machines (ATMs), airplane systems, digital home assistants, washing machines, microwave and the internet of things (IoT) devices.

For specific computer-based machines, specific operating systems are designed. A few of these are listed below:

- Real-Time OS
- Multiprocessor OS
- PC OS
- Server OS
- Mobile-Device OS

References

[Brinch-Hansen (1973)] P. Brinch-Hansen, *Operating System Principles*, Prentice Hall (1973).