1. Operating System Concepts

1.1 What is an operating system?

Operating systems are an essential part of any computer system. An **operating system** (OS) is software, which acts as an intermediary between the user of a computer system and the computer hardware (Figure 1). The purpose of an operating is to provide an environment in which a user can execute programs in a convenient and efficient manner (Hayhurst 2002).

- It is the first program loaded into memory when the system is booted
- It is a program that manages the computer hardware, and controls interactions between application programs and that hardware.
- Makes the computer more convenient to use
- It allows users to execute application programs in a "higher level" environment without the necessity to communicate directly with hardware devices.
- Helps to ensure the efficient use of the computer resources

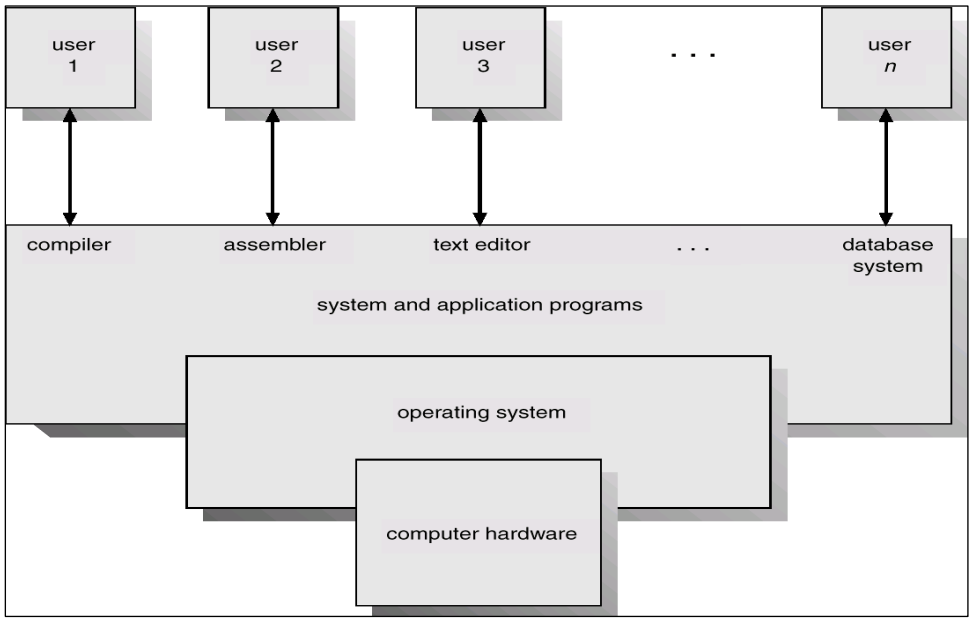

Figure 1. Abstract view of the components of a computer system. (Source: Silberschatz et al. 2002)

A computer system can be divided roughly into four components: (1) the hardware, (2) the operating system, (3) the application programs, and (4) the users.

(1) Hardware: which provides the basic computing resources:

- CPU: Central processing unit; executes commands

- Memory: Contains programs and associated data while they execute. Before anything can be executed it must be loaded into memory
- I/O devices: Printers, disk drives, etc

(2) Operating system: controls and coordinates the use of the hardware among the various application programs for the various users. It is responsible for:

- Provides an environment within which programs can be executed
- Handles communications between application software and I/O devices through the use of device drivers
- Manages memory when multiple programs are executing simultaneously. Loading programs & data into memory. Processing memory needs of programs. Ensuring that programs do not overwrite the memory space "owned" by each other.

(3) Application programs: product or custom written software which defines the ways in which the system resources are used to solve some computing problems: compilers, database systems, word processors). System programs: provide system services to users of the operating system and define a user's view of the operating system.

(4) Users: People, machines, and other computers that interact with the computer system to accomplish some task

1.2 Views of OS

The role of the OS and the services it must provide can be viewed from two different viewpoints (Hayhurst 2002):

(1) The Users Viewpoint

From the Users viewpoint we use a computer system to accomplish some task. The OS is a tool which helps in that task. The OS's role can vary depending on the type of system being used: Some systems such as PC operating systems are designed with ease of use as the primary goal. In a Multi-user system where users share resources of a mainframe or minicomputer the OS should be designed to maximize resource utilization to ensure that: 1) all available CPU time, memory and I/O resources are used efficiently. 2) No individual user takes more than his/her fair share of those resources.

(2) The Systems Viewpoint

From the computer's viewpoint the OS is the program that is the most intimate with the hardware. The OS acts as a:

- Resource allocator - which manages and allocate resources. A computer system has many resources at its' disposal, both hardware and software, which

may be required by an end user application to solve a problem, such as CPU time, memory space, file storage, I/O devices, and so on.

The OS acts as the manager of these resources. As application programs execute, or users issue commands directly to an operating system, it faces numerous possibly conflicting requests for resources. The OS must decide how to allocate these resources as efficiently, and fairly as possible.

- CONTROL PROGRAM: controls the execution of user programs (to prevent errors and improper use of the computer) and operation of I/O devices
- KERNEL: it is the one program running at all times (all else being system/application programs). It incorporates core functionality which must be available at all times, including the resource allocator and control program. The "kernel" is loaded when the computer is "BOOTED". Most computers have a small piece of code stored in ROM, known as the bootstrap program or bootstrap loader. This code is able to locate the kernel, load it into main memory, and start its execution. Surrounding this kernel are typically some type of command interpreters (UNIX shells) or GUI which provide a mechanism for a user to issue a request directly to the OS without writing a program.

1.3 Brief OS History

Internally operating systems are extremely complicated programs, and vary greatly in their makeup and the services which they provide. To some extent, operating systems have evolved as the capabilities of hardware have increased, and the ways in which people have wished to use computers have changed (Figure 2) (Hayhurst 2002).

(1) 1940's – 50's: program = computer

In the early days of computing there was no operating system.

- Early computers were physically enormous, **expensive** machines run from a console.
- 1 User at a time
- 1 program at a time (machine code)
- programmer operates machine himself: programmer would write the program, then operate the program directly from the operators console:
- PROBLEMS:
    - Required the programmer to be a skilled operator knowing the right punched cards/paper tape to load
    - Lots of CPU idle time as information is read from the slower I/O devices
    - Lots of idle time if the programmer failed to notice the program had ended.

- SINGLE USER MACHINE: there was a long wait by other users.

- DRIVING FORCES FOR CHANGE:
  - Computers were VERY expensive: idle CPU time was a waste of money
  - More programmers wanting to use the computer

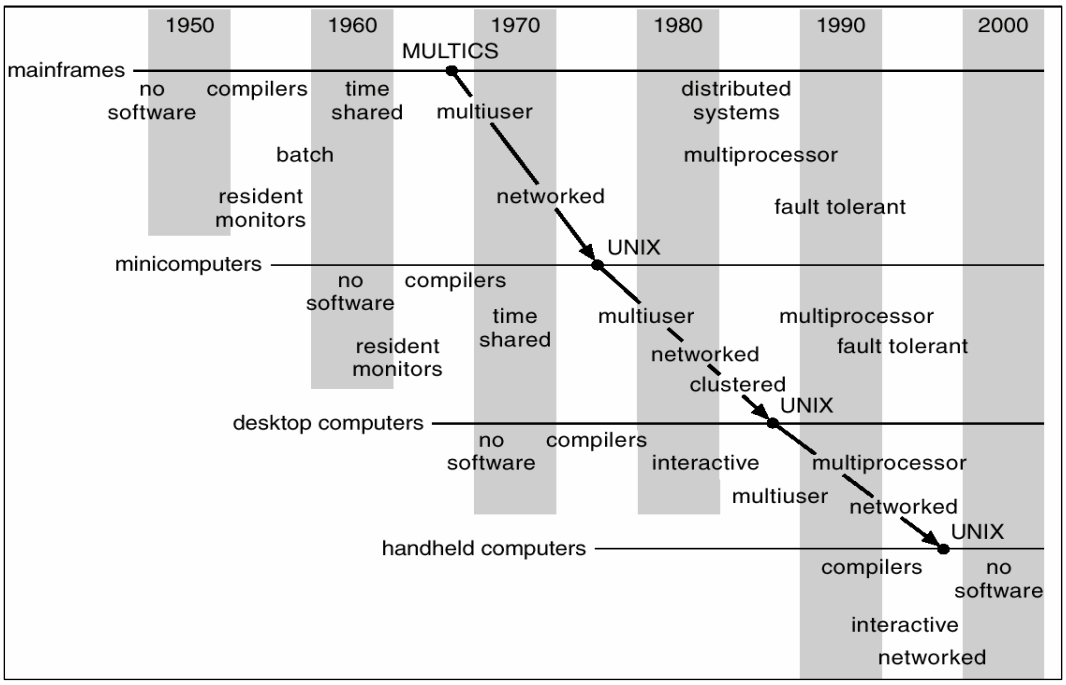

Figure 2. Migration of OS concepts and features.
(Source: Silberschatz et al. 2002)

(2) 1950's – 60's: beginning of Batch operating systems

This is a time period of great change in computing:

- Advances in hardware: magnetic tape, line printers, & magnetic disks
- 1st assemblers were created which allowed programs to be written in something other than machine language. This also included the development of loaders and linkers: to link applications to libraries of common functions.
- Device drivers became available to communicate directly with specific I/O devices. A device driver is a program which is written to communicate with a specific type of I/O device. It knows how the buffers, flags, registers, control bits & status bits for a particular device should be used. Now application programs could link to these drivers from a library rather than directly including the device specific code in the program itself.
- Development of compilers for HLL: FORTRAN, COBOL and others. This allowed programs to be written in a language which more closely resembled natural language. But to prepare a HLL language program for execution it

was necessary to go through several preparation steps (Figure 3): (remember that only machine code can be directly executed by the CPU)

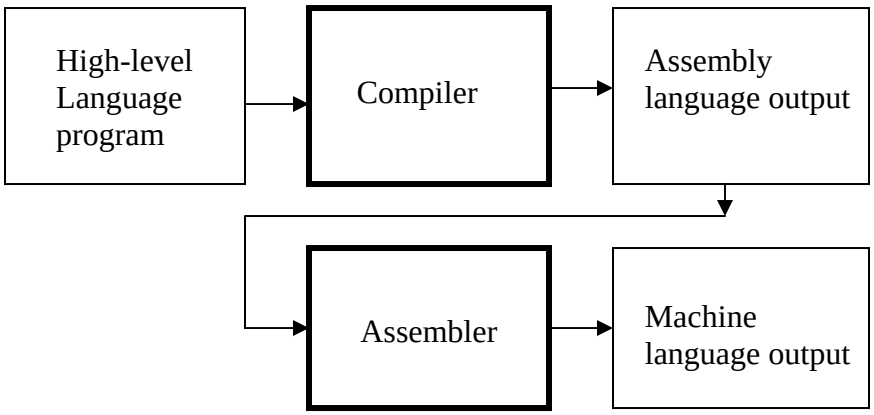

Figure 3. Compilation process.
(Source: Hayhurst 2002)

Another way professional operators could improve the performance of the system was to group/batch together jobs with similar resource needs and run them through the computer as a group to reduce set up time. The operator prepared sequential tapes of these jobs (could read the jobs into the system via punched cards, and write them to a tape.) which could be run through the CPU together in a "batch".

However, there were still problems. If a job stopped the operator would have to notice that it stopped (either normal or abnormal termination), determine why it stopped, dump memory and registers if necessary, load the appropriate device with the next job and restart the computer. If this occurred, the CPU was sitting idle.

To overcome CPU idle time, the **automatic job sequencing** was developed, and thus the 1st rudimentary operating systems were created.

A **Resident Monitor** (RM) is a small program created to transfer control automatically from one job to the next. The resident monitor is ALWAYS in memory. The RM does not decide the order of jobs, but does coordinate their sequencing, automatically starting and terminating jobs

The RM had several identifiable parts:

(1) The **control-card interpreter** which was responsible for reading and carrying out the instructions on the control cards at the point of execution.

(2) The **loader** which is invoked by the control card interpreter to load systems programs and application programs into memory.

(3) **Device drivers** which are used by both the control card interpreter and the loader for the systems I/O devices to perform I/O. Often system and

application programs were linked to these same device drivers providing continuity in their operation, as well as saving memory space and programming time.

These "batch" systems worked fairly well. The RM provided automatic job sequencing and transferred control between programs. Improvement over human operators, but the CPU was still frequently idle. The speed of I/O devices was significantly slower than the slowest CPU's which operated in Microsecond range, while the fastest card readers operated in the 1200 cards/second range.

While this was an improvement, there were still significant problems:
1) Still CPU idle time, particularly when jobs were performing I/O.
2) For programmers: long turn around time for programs, debugging was no longer interactive, and if a program had errors a programmers may have to wait days to retry the job.

The primary force for change was the introduction of Random Access (disk) I/O devices. Disks were an improvement in speed and efficiency over magnetic tapes. Disks were random access devices: it was easy to move the head from one area being used by the card reader to store new cards, to the position needed by the CPU to read the next card. As jobs were read from the card reader they were written to disk, and the location of the card images recorded in a table kept by the OS. When a job is executed the OS reads directly from disk: decreases the time to change from one job to another.

Disks also allowed for jobs to be executed in an order that wasn't sequential. The OS could now implement the human operator's algorithms in deciding the sequence of jobs to run based on the information provided by control cards. At this time we begin to see the process of scheduling of jobs to be totally automated. (1^{st} true OS's)

Problems: I/O devices were still much slower than CPU speed, so CPU was still often idle when I/O is being performed.

Another way to improve performance is to delay the printing of output until the end of a job. To do this when a job requests to print a line to a printer, the line could be copied to a system buffer and retained until the job is finished. Then the output is actually printed: This type of processing is called **SPOOLING** (simultaneous peripheral operation on-line). Spooling in essence uses the disk as a huge buffer for reading as far ahead as possible on the input devices and for storing output files until the output devices are able to accept them. Spooling overlaps the I/O of one job with the computation of other jobs.

Even in simple systems the spooler may be reading the input of one job while printing the output of a different job. During the same time, still another job may be executing reading their cards from disk and printing their output lines onto the disk. This is accomplished through the use of device controllers, which are pieces of hardware that can work in parallel with the CPU to transfer data. In this mode the device controller can

be copying data from disk to the printer while the CPU has begun execution of another job. It also allows the CPU to store output on disk if the printer is full, and MOVE ON.

Spooling has a direct beneficial effect on the performance of the system as it can keep both the CPU and I/O devices working at much higher rates.

However, wait times were still too long. Long jobs (high resource needs) delay everything else. Also scheduling was a delicate task. When we ran high resource jobs, one might take over 1 day.

Spooling naturally leads to Multiprogramming that is the foundation of all modern operating systems.

The most important aspect of job scheduling is the ability to multi-program. A single user cannot, in general, keep either the CPU or the I/O devices busy. **Multiprogramming** increases utilization by organizing jobs so that the CPU always has one to execute:

1) The OS has several jobs in memory simultaneously. These jobs constitute a subset of the total jobs in the job pool. (on disk)
2) Using some algorithm the OS picks and begins to execute one of the jobs in memory.
3) If the executing job must wait for some I/O task, the OS simply switches to and executes another job. And so forth. Eventually, the first job will finish waiting and get the CPU back and continue operation. As long as one job needs to execute, the CPU is never idle.
4) This is the 1st instance where the OS must make decisions for the USERS.
5) Fairly sophisticated: All jobs enter the system and are kept in the job pool: all processes residing on disk awaiting allocation of main memory. If several jobs are ready to be brought into memory and if there is not enough room for all of them then the OS must decide which jobs to choose. This is called Job scheduling.
6) Having several programs in memory requires memory management to allow processes to share memory!!!
7) If several jobs are ready to run at the same time, the OS must choose among them which is CPU scheduling. In a uniprocessor system, only one process may run at a time; any other processes just wait until the CPU is free and can be rescheduled. First come first served, shortest job first, Round Robin…

(3) 1970's – 80's Interactive time sharing (multi-tasking) on mainframes

- Logical extension of multiprogramming.
- Allows many users to share the computer simultaneously.
- CPU executes multiple "jobs" by switching among them, but the switches occur so frequently that the users can interact with each program while it is

running. If time slicing is fast enough users can feel as if they have their own dedicated machine.

- Provides direct communication between the user and the system. The user can give instructions to both the OS and programs using either the keyboard or mouse, and wait for immediate results.
- Time sharing OS's use CPU scheduling and multiprogramming. Each user has at least one separate program in memory. A program loaded into memory and executing is commonly referred to as a process. As processes execute if they need to perform I/O (particularly with a user) the OS can switch to another user's process.
- Time sharing OS's are more complex than multiprogramming systems:
    - In both, several jobs must be kept in memory at the same time; this requires memory management & protection.
    - To obtain a reasonable response time, switching between tasks must occur at a much faster rate, which may require jobs to be swapped in and out of main memory to disk.
    - Time-sharing systems must also provide a file system, disk management facilities, mechanisms for job synchronization and communication, and much more.
    - Although time-sharing systems were demonstrated as early as 1960, they were expensive and difficult to build, and did not become common until the early 1970's

Arguably, one of the most popular and widely used time-sharing systems is UNIX.

1.4 Types of OS Systems

(1) Mainframe systems

Mainframe computer systems were the first computers used to tackle many commercial and scientific applications. These systems evolved from simple batch system, where the computer runs one application, to time-sharing systems, which allow for user interaction with the computer system.

Early computers were physically enormous machines run from a console. The common input devices were card readers and tape devices while the common output devices were line printers, tape drives, and card punches. The user did not interact directly with the computer systems. Rather, the user prepared a job that consisted of the program, the data, and some control information about the nature of the job (control cards) and submitted to the computer operator.

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 OS was always resident in memory.

In order to speed up the processing, operators batched together jobs with similar needs and ran them through the computer as a group, which is called **batch system**. Thus the programmers would leave their program with the operator. The output from each job would be sent back to the appropriate programmers.

In this execution environment, the CPU is often idle, because the speeds of I/O devices are slower than CPU's. This is why **job scheduling** was introduced, which can allow the operating system to use resources and perform tasks efficiently.

The most important aspect of job scheduling is the ability to multiprogram. A single user cannot, in general, keep either the CPU or the I/O devices at all times. **Multiprogramming** increase CPU utilization by organizing jobs so that the CPU always has one to execute.

The OS keeps several jobs in memory simultaneously (Figure 4). This set of jobs is a subset of the jobs kept in the job pool. In a multiprogramming system, the OS simply switches to, and executes, another job. When a job needs to wait, the CPU is switched to another one. Multiprogramming is the first instance where the operating system must make decisions for the users.

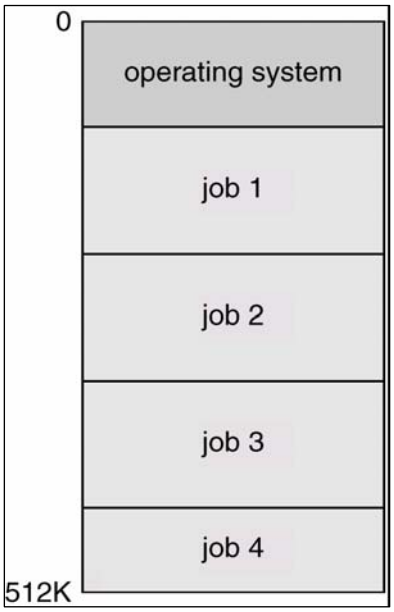

Figure 4. Memory layout for a multiprogramming system.
(Source: Silberschatz et al. 2002)

Multiprogrammed, batched systems provided an environment where the various system resources (for example, CPU, memory, peripheral devices) were utilized effectively, but it did not provide for user interaction with the computer system. Time-sharing (or multitasking) is a logical extension of multiprogramming. The CPU executes multiple jobs by switching among them, but the switches occur so frequently that the users can interact with each program while it is running.

(2) Desktop systems

Personal computers (PCs) appeared in the 1970s. 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 multi-user nor multitasking. These systems include PCs running Microsoft Windows and the Apple Macintosh. The MS-DOS from Microsoft has been superseded by multiple flavors of Microsoft Windows.

(3) Multiprocessor systems

Most systems to date are single-processor systems; that is they have only one main CPU. However, **multiprocessor systems** (also known as parallel systems) are growing in importance. Such systems have more than one processor in close communication, sharing the computer bus, the clock, and sometimes memory and peripheral devices. Multiprocessor systems have three main advantages:

(a) Increased throughput. By increasing the number of processors, we hope to get more work done in less time.

(b) Economy of scale. Multiprocessor systems can save more memory than multiple single-processor systems, because they can share peripherals, mass storage, and power supplies.

(c) Increased reliability. If functions can be distributed properly among several processors, then the failure of one processor will not halt the system, only slow it down.

The most common multi-processor systems now use symmetric multiprocessing (SMP), in which each processor runs an identical copy of the operating system, and these copies communicate with one another as needed. SMP means that all processors are peers; no master-slave relationship exists between processors (Figure 5). 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.

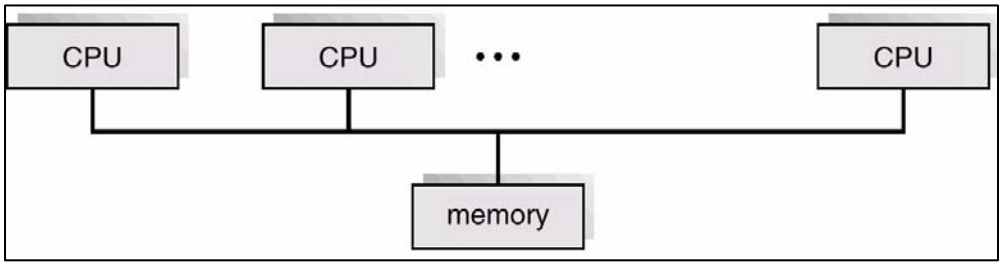

Figure 5. Symmetric multiprocessing architecture.
(Source: Silberschatz et al. 2002)

(4) Distributed systems

A **network**, in the simplest terms, is a communication path between two or more systems. Distributed systems depend on networking for their functionality. By being able to communicate, distributed systems are able to share computational tasks, and provide a rich set of features to users.

Networks vary by the protocols used, the distances between nodes, and the transport media. TCP/IP (transmission control protocol/internet protocol) is the most common network protocol, although other protocols are also used. Most operating systems support TCP/IP, including the Windows and UNIX.

Networks are typecast based on the distances between their nodes. A local-area network (LAN) exists within a room, a floor, or a building. A wide-area network (WAN) usually exists between buildings, cities, or countries.

The growth of computer networks – especially the Internet and World Wide Web (WWW) – has had a profound influence on the recent development of operating systems. When PCs were introduced in 1970s, they were designed for "personal" use and were generally considered standalone computers. With the beginning of widespread public use of the Internet in the 1980s for electronic mail, ftp, and gopher, many PCs became connected to computer networks. With the introduction of the Web in the mid-1990s, network connectivity became an essential component of a computer system. Virtually all modern PCs and workstations are capable of running a web browser for accessing hypertext documents on the Web. The general structure of a client-server system is depicted in Figure 6.

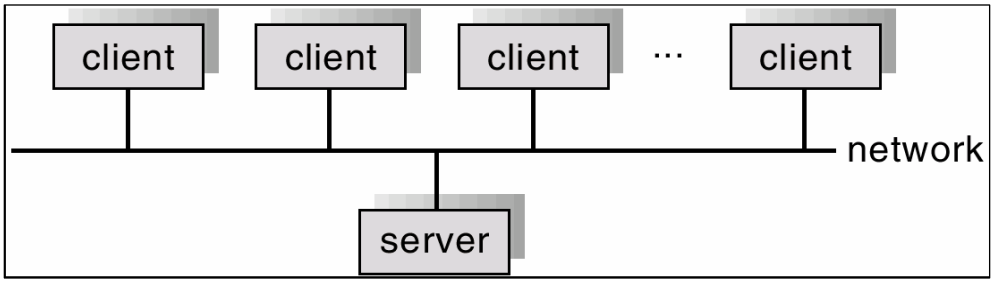

Figure 6. General structure of a client-server system.
(Source: Silberschatz et al. 2002)

(5) Real-time systems

Another form of a special-purpose operating system is the **real-time system**. A real-time system is used when rigid time requirements have been placed on the operation of a processor or the flow of data; thus, it is often used as a control device in a dedicated application. Systems that control scientific experiments, medical imaging systems, industrial control systems, and certain display systems are real-time systems. Some other systems include home-appliance controllers and weapon systems.

(6) Handheld systems

Handheld systems include personal digital assistants (PDAs), such as Palm-Pilots. Due to the limited size, most handheld devices have a small amount of memory, include slow processors, and feature small display screens.

References

Hayhurst, C. 2002. Semantics of Programming Language (Lecture Notes). Lane Department of Computer Science and Electrical Engineering, West Virginia University, Morgantown, West Virginia.

Silberschatz, A., P.B. Galvin, and G. Gagne. 2002. Operating System Concepts (6th Edition). John Wiley & Sons, Inc., New York.