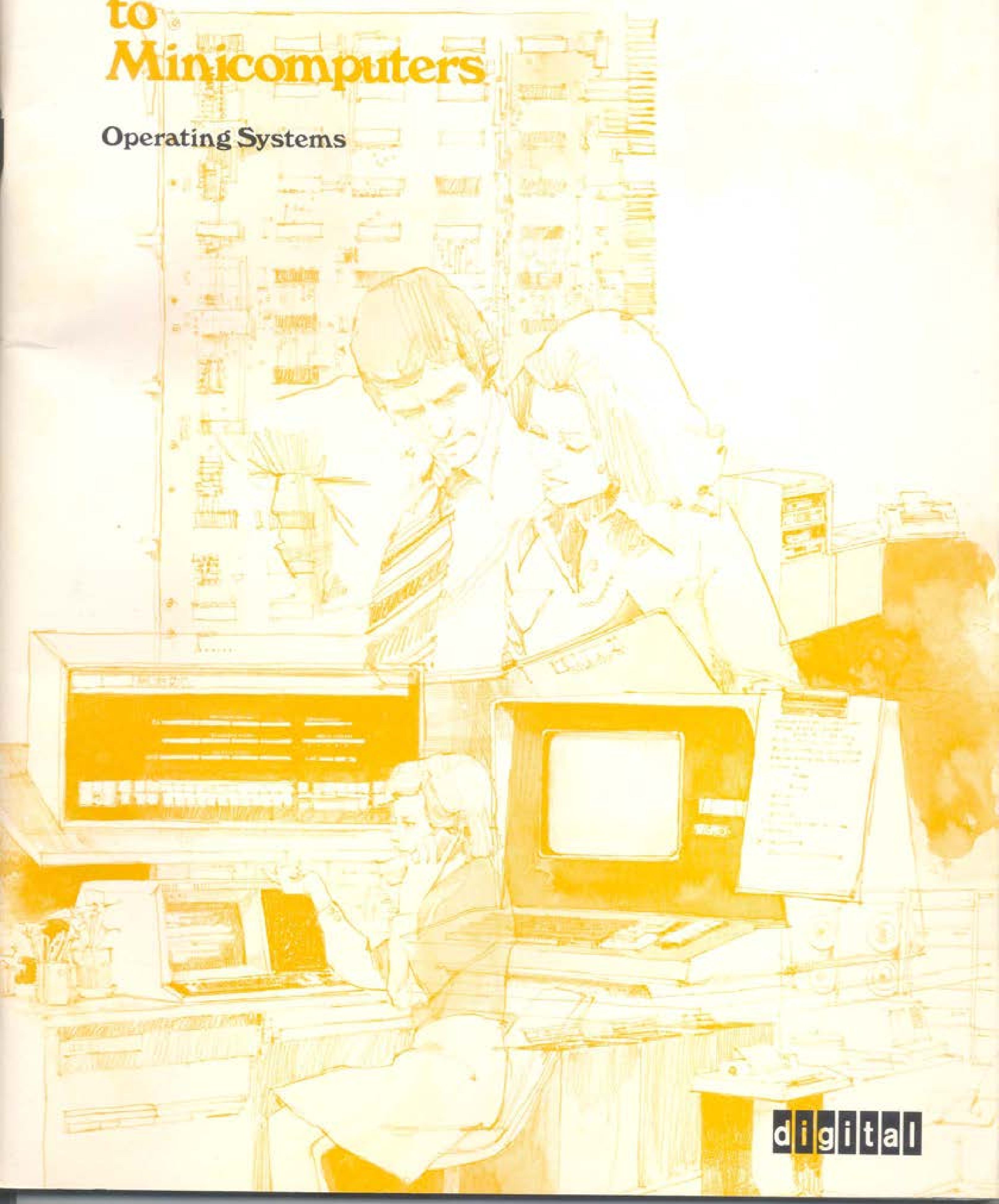
Introduction to Minicomputers

Operating Systems



1st Printing, June 1976
2nd Printing (Rev), October 1977
3rd Printing, August 1979

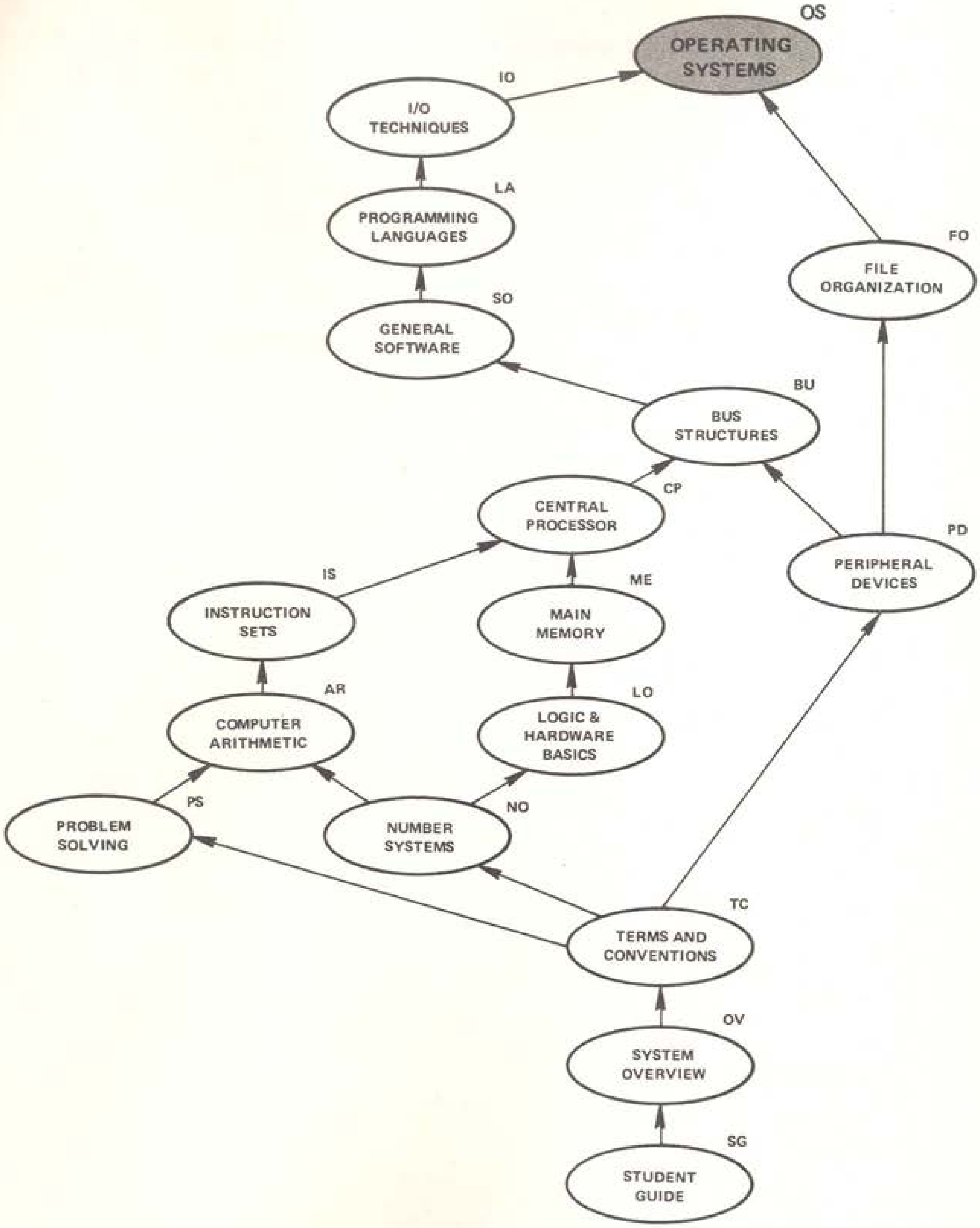
INTRODUCTION TO MINICOMPUTERS

Operating Systems

Student Workbook

Audio-Visual Course by Digital Equipment Corporation

COURSE MAP


OS
OPERATING SYSTEMS
IO
I/O TECHNIQUES
LA
PROGRAMMING LANGUAGES
SO
GENERAL SOFTWARE
FO
FILE ORGANIZATION
BU
BUS STRUCTURES
CP
CENTRAL PROCESSOR
PD
PERIPHERAL DEVICES
IS
INSTRUCTION SETS
ME
MAIN MEMORY
AR
COMPUTER ARITHMETIC
LO
LOGIC & HARDWARE BASICS
PS
PROBLEM SOLVING
NO
NUMBER SYSTEMS
TC
TERMS AND CONVENTIONS
OV
SYSTEM OVERVIEW
SG
STUDENT GUIDE

CONTENTS

Operating Systems

Introduction

Thus far, you have learned about various computer components such as main memory, the central processor, peripheral devices, and bus structures. Together, these components form the computer hardware or architecture.

You have also learned about low- and high-level programming languages, the translators that are required to convert programming languages into machine code, and three I/O techniques that allow programs to initiate and/or to control input/output operations.

As Lesson 1 of this module will show, the presence of both hardware and software does *not* by itself guarantee rapid, easy development and execution of programs. An additional element is needed to make a computer the powerful tool it is. Some means to *manage* or coordinate the many activities and features of the hardware and software is needed and, further, to manage these resources at computer speeds. This management of computer resources is precisely the task of the operating system.

An operating system is a set of programs that collectively automate the management of computer resources to provide efficient computer operation. An operating system is usually tailored for a specific hardware configuration for two reasons:

- The user is ultimately dependent upon the hardware for program execution.
- Different application areas use different peripheral devices and have different operational requirements.

The operating system performs numerous activities such as:

- Allowing user communication and control of the computer system by interpreting and executing keyboard commands.
- Aiding in all phases of program development and execution.
- Making available common software items such as peripheral device handlers, interrupt servicing routines, and language translators.
- Allowing multiple users and concurrent programming.
- Scheduling CPU and peripheral usage for greatest efficiency.

To accomplish these goals, operating systems commonly perform a standard set of general functions. Often, these functions are performed by individual components or routines, each having a specific task. These functions and components are discussed in Lesson 2 of this study unit.

Also, an operating system is frequently tailored for particular areas of applications. Hence, some operating systems are better suited for controlling industrial processes, while others may be better for applications such as airline reservations, program development or processing large amounts of data. Lessons 3 and 4 of this module discuss the major categories of operating systems and the application areas for which they are normally used.

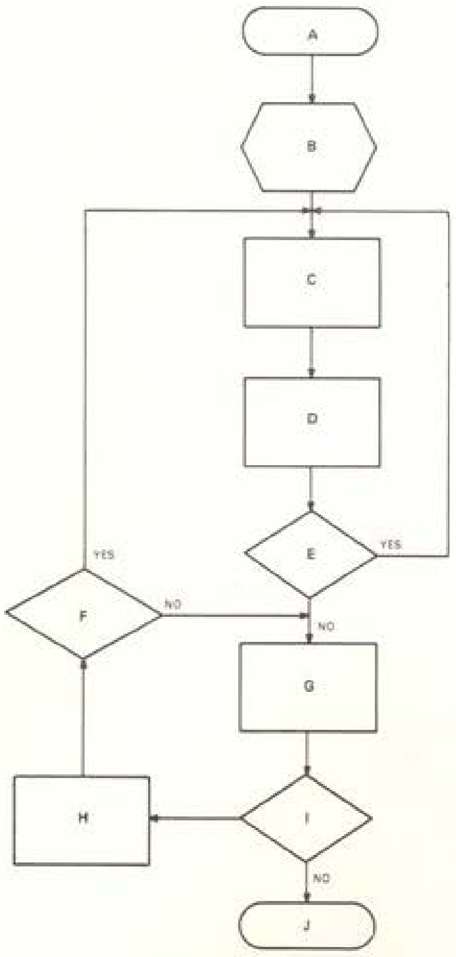
Program Development Without An Operating System

OBJECTIVE

Given a blank flowchart of the program development process for a computer without an operating system and a list of flowchart contents, be able to match each flowchart symbol with its contents.

SAMPLE TEST ITEM

The blank flowchart below represents the program development process for a computer without an operating system. Match each *step* of the process with its corresponding position in the flowchart by writing the correct letter in the space provided.



Step	Position
Start	_____
First Error Check	_____
Assemble Program	_____
Second Error Check	_____
Execute Program	_____
Enter and Edit Program	_____
Debug Program	_____
Document Program	_____
Done	_____
Initialize Computer	_____

Mark your place in this workbook and view Lesson 1 of the A/V program, "Operating Systems."

Program Development

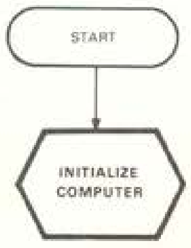
The development of a program is a lengthy process. Without the assistance of an operating system, programmers must perform many of the steps manually. This lesson discusses this process in detail.

Before using the computer, the task the computer is to perform must be analyzed using problem solving procedures. Then, a flowchart of the procedure must be generated, and the flowchart must be translated into a handwritten source program in a programming language. Because this step is performed without the computer's assistance, the existence of an operating system has no effect on this phase. The programmer is generally the only active agent, and the output of this phase is the handwritten source program on a piece of paper.

assembly language program

```
START,  CLA
        ADD NUMBER
        ADD NUMBER
        ADD NUMBER
        ADD NUMBER
        STR SUM
        HLT
```

Figure 1   Source Program



1. Once the source program is written on paper and ready to be further developed using the computer, the programmer must *initialize* the computer for this purpose. The programmer enters the first program *manually*.

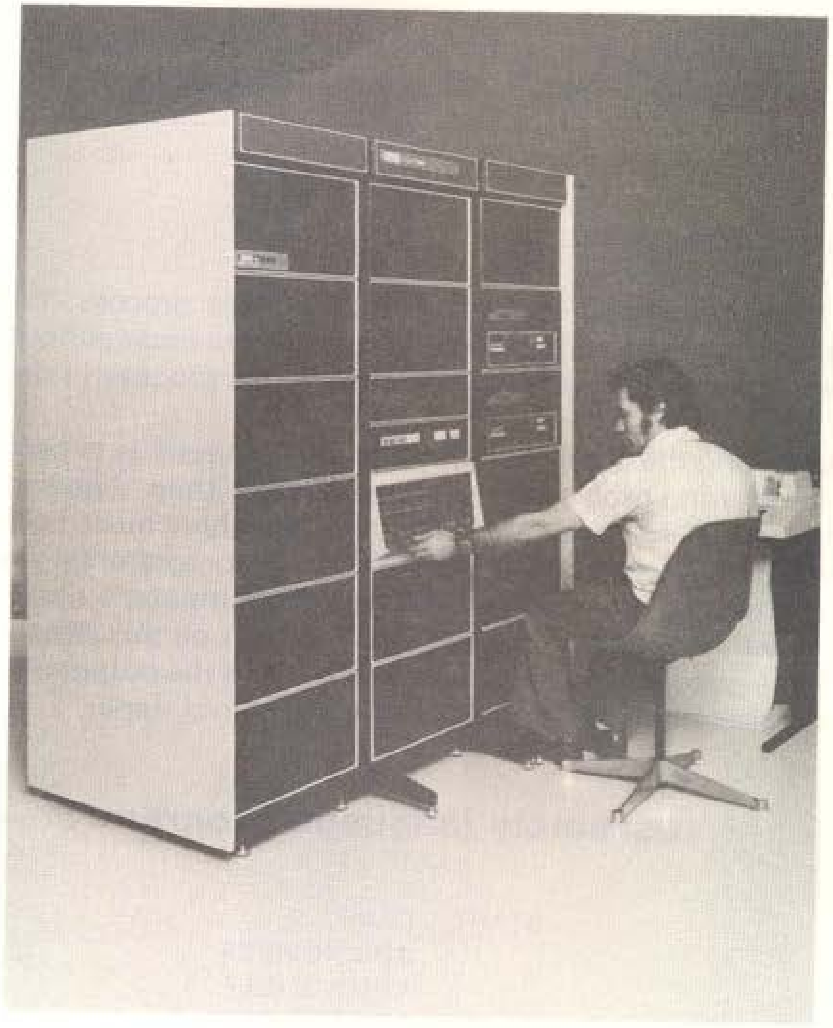


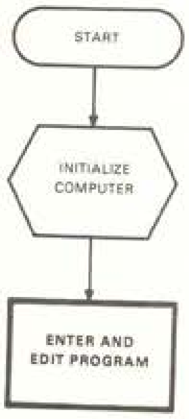
Figure 2 Programmer Enters First Program Manually

The first program loaded is called a *bootstrap*. The bootstrap is a program that is designed to be short to reduce the manual effort of the programmer in entering it, and yet to contain enough instructions to enable the computer to input other programs. A program that instructs the computer to input other programs into memory is called a *loader* – the bootstrap is sometimes called a *bootstrap loader.*

The only function of the bootstrap is to allow the computer to input, or load, the loader program. On small computers that do not have an operating system, the loader program is stored on, and loaded from, *paper tape.*

The programmer places the general-purpose loader machine code paper tape into the paper tape reader, enters the starting address of the bootstrap through the front panel switches into the program counter, and starts the computer. The computer reads the paper tape of the general-purpose loader and deposits it into memory. This completes the initialization of the computer.

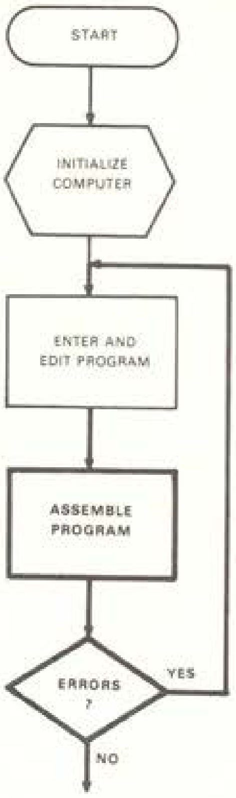
From this point on, any time a program is to be loaded into memory, the programmer places a machine tape of that program in the paper tape reader. Then, the programmer enters the starting address of the general-purpose loader through the front panel switches into the program counter, and starts the computer. The computer halts when the loading process is complete. The program development process requires the loading of many programs.



2. The first purpose of the general-purpose loader is to load an *editor* program into memory. Once this is done, the editor is started by entering its starting address into the program counter through the front panel switches, and starting the computer.

Figure 3 Programmer Entering Source Program

While the editor is executing, the programmer enters the source program through the keyboard. After the program has been entered, the programmer can also correct mistakes using the editor. When the programmer is satisfied with the source program, he/she types a command to the editor, which instructs the computer to punch out the edited source program on paper tape.



3. When the source tape has been punched, the program is ready for translation. The programmer loads the *assembler* (or compiler) using the general-purpose loader, and starts it. The translator then processes the source tape. Note that the source tape must be loaded *once for each pass* of the assembler. The translator punches the machine code tape and prints listings of the symbol table and the program. The work of this phase is divided between the user (loading four paper tapes) and the computer (translating the program). If errors are detected during assembly, the programmer returns to the third phase (Enter and Edit Program) to correct the source program. The editor and source tapes are loaded, and the corrections are typed in. Then, the assembly process is attempted again. When no further errors are detected, the development process continues with the next phase.

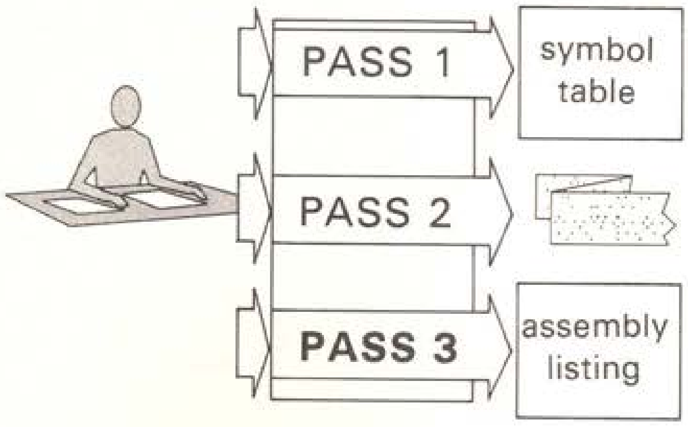
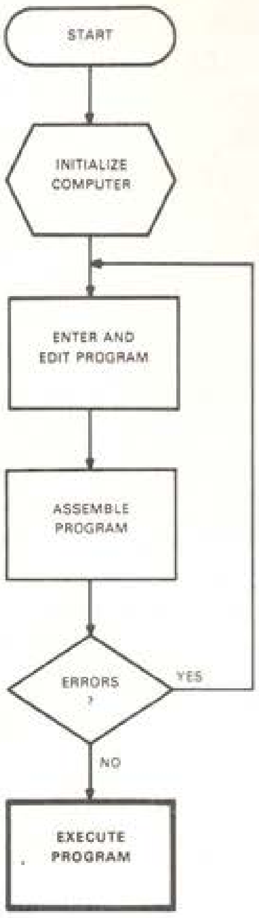
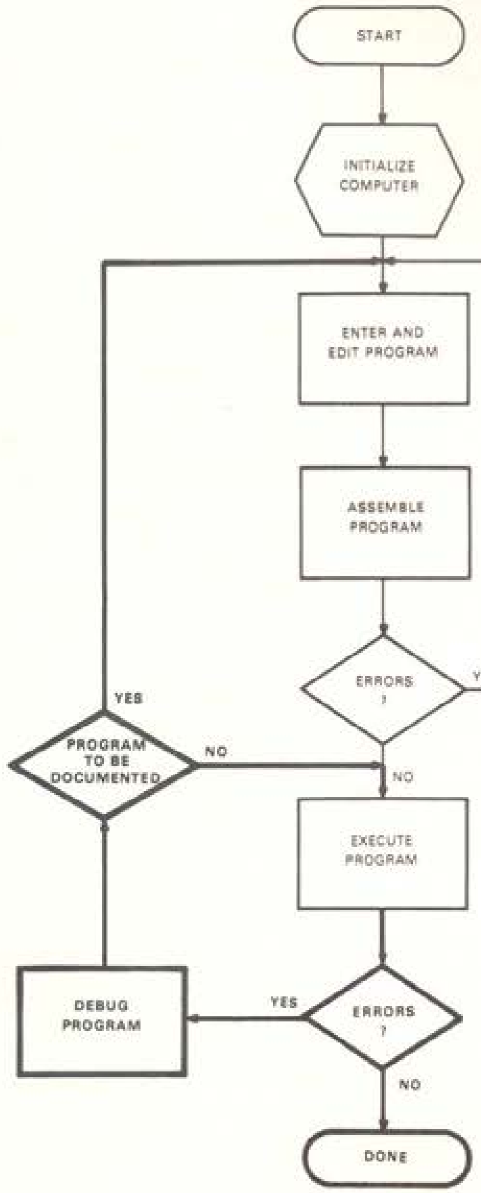


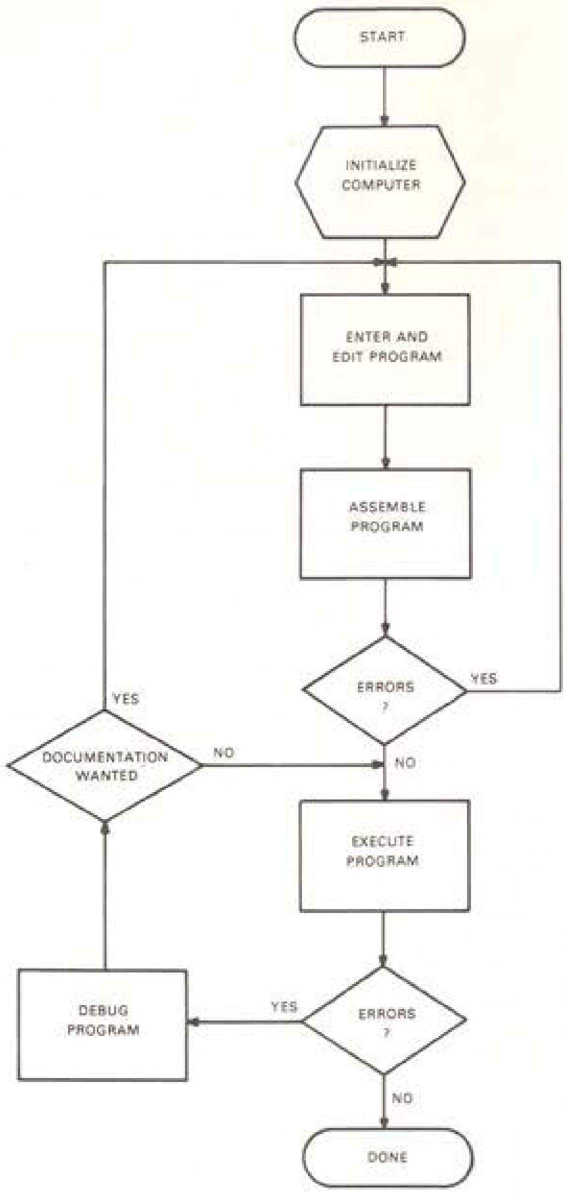
Figure 4 Assembly Phase



4. When no more assembly errors have been detected, the program is ready for execution. Before this can happen, the programmer must load the *machine code tape* into memory. Errors in the program logic are almost certain to appear during the first execution of the program. Program results are also produced until a serious error occurs.



5. Many computers have *debugging programs* to assist in correcting errors within machine code programs. To use this process, the programmer loads the debugging program, and then the machine code tape of the erroneous program. These debugging programs allow a programmer at a terminal to selectively execute parts of the program to check the functioning of each section. When errors are discovered, the debugging program allows the programmer to directly modify the machine code.



Once the corrections to the program have been made, a *new* machine code tape is punched. The execute-debug loop is then repeated until all detected errors have been corrected. Frequently, however, correcting the machine code is not the desired operation. If the program is to be properly documented, the source code of the program should be corrected and a new listing generated. New documentation prevents a discrepancy between what the documentation (flowchart and program listing) says and what the program actually does. Without these discrepancies, other programmers can easily tailor the program to their specific needs. Hence, execution errors or logic errors usually cause the programmer to go back to editing the source program and performing all the subsequent steps again.

As can be seen, the program development process is a complex and lengthy one – regardless of whether an operating system is present. However, when no operating system is available, the high software development cost is greatly increased due to the large amount of manual programmer intervention required.

The following chart summarizes the material of this lesson.

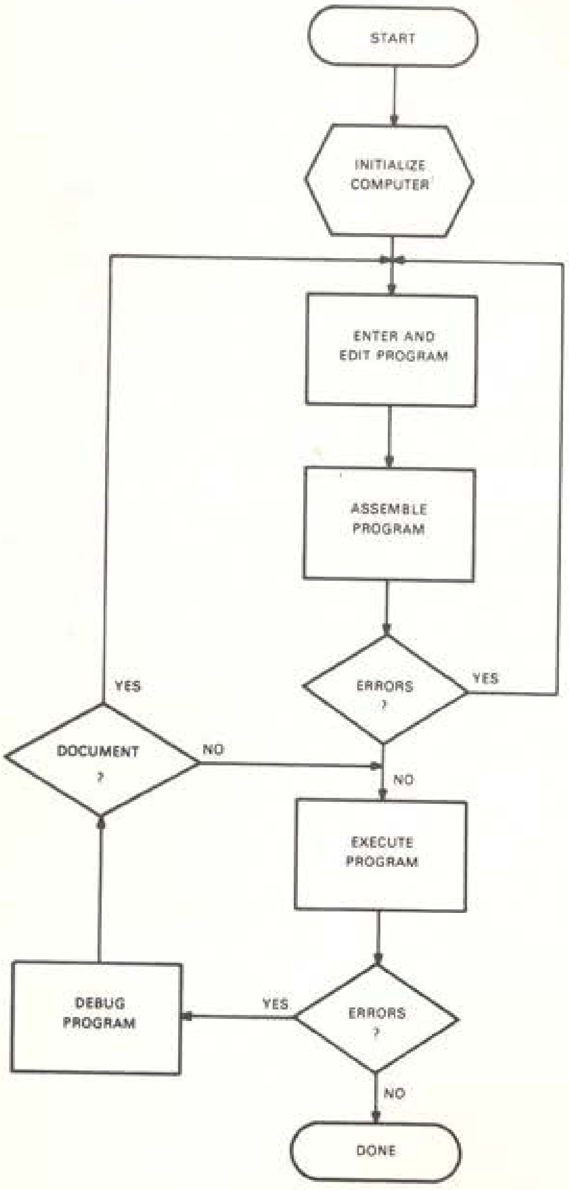
Software Development Summary (Without an Operating System)

Agents Performing This Action	Input	Output
Programmer Only	Problem Specification	Source Program
Programmer Only	"Empty" Machine, Toggle Switches Loader (Paper tape)	"Ready" Machine
Programmer, Loader, Editor	Editor (Paper tape) Source Program (Keyboard)	Source tape (Paper tape)
Programmer, Loader, Assembler	Assembler (Paper tape) Source tape (Paper tape) Source tape (Paper tape) Source tape (Paper tape)*	Symbol table (Printer) Machine code (Paper tape) Program listing (Printer) and assembly errors if any
Programmer, Loader, Program	Machine code (Paper tape)	Results, showing logical errors if any.
Programmer, Loader, Debugger	Debugger (Paper tape) Machine code tape	Corrected Program Machine code tape
Upon completion, the paper tape of the working program is saved with its documentation for later use as needed		

*Some systems with several output devices can output assembly listing and machine code during pass 2. Three passes are typically required when paper tape punch and printer are both part of the same terminal.

EXERCISES

1. The flowchart for the program development process is given below. Explain the operations within the four process blocks (rectangles) in terms of the steps that the programmer must do when no operating system is available.



a) **Enter and Edit Program**

b) **Assemble Program**

c) **Execute Program**

d) **Debug Program**

SOLUTIONS

1. Steps that the programmer must do when *no* operating system is available.

   a) **Enter and Edit Program** – The programmer first loads the paper tape of the machine code of the editor, and the editor is started. Next, the source program is entered using the keyboard. Editing is performed when errors are detected in entering the program. When the programmer is satisfied with the entered source program, a paper tape is punched.

   b) **Assemble Program** – The assembler is first loaded from a paper tape and started. The source paper tape from the edit phase is loaded two or three times during the assembly process. If the assembly is error free, a machine code paper tape is punched.

   c) **Execute Program** – The machine code paper tape is loaded into memory. It is then executed by setting the program counter (PC) to the starting address of the program and pressing START.

   d) **Debug Program** – If execution errors are encountered, the program must be debugged. The debugging program is loaded from a paper tape and started. The machine code program is then loaded. Using the debugger, the programmer can selectively execute parts of the program and examine the behavior of the program. Depending on the level of documentation required, either the machine code or the source code is corrected to remove the errors.

EXERCISES

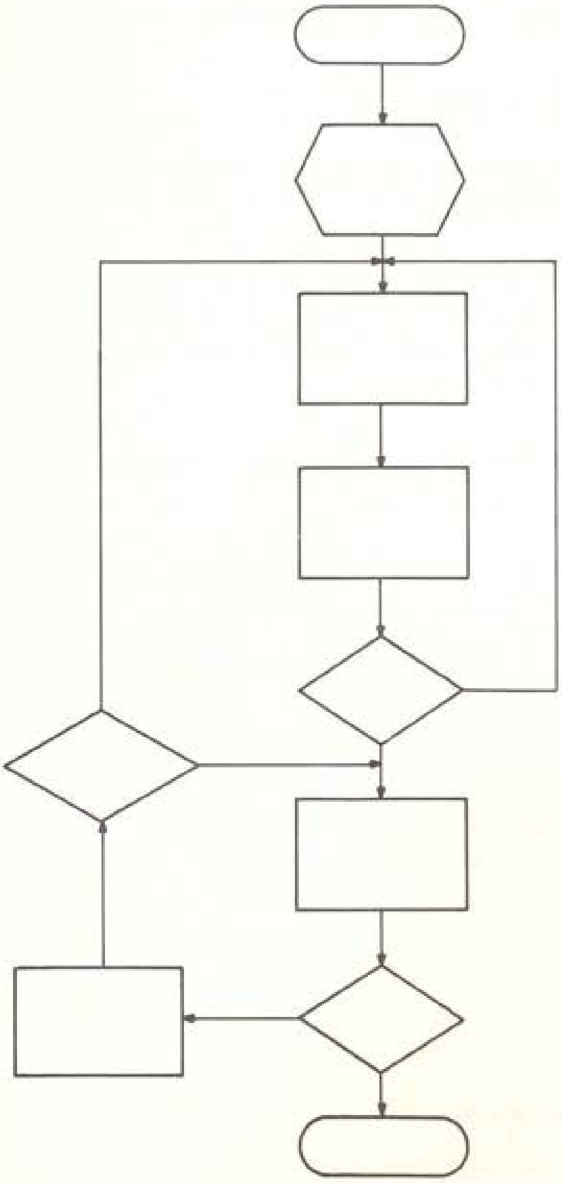
2. Why would a programmer go back and edit the source program after debugging, instead of simply punching a new machine code tape?

2. Why would a programmer go back and edit the source program after debugging, instead of simply punching a new machine code tape?

• A programmer re-edits the source program after debugging to provide up-to-date documentation of the present form of the program. Because the program listing and the symbol table listing are valuable tools for understanding the program, they must reflect the actual version of the program saved as machine code. Otherwise, if a new machine code tape is produced directly after debugging, the changes will not be reflected in the documenation. Thus, the program will not execute as the documentation says it should.

EXERCISES

3. Fill in the following flowchart of the program development process. Next to each process step write the results or output of the step.



Results or Output

– Programmer outputs hand-written source code.

a)

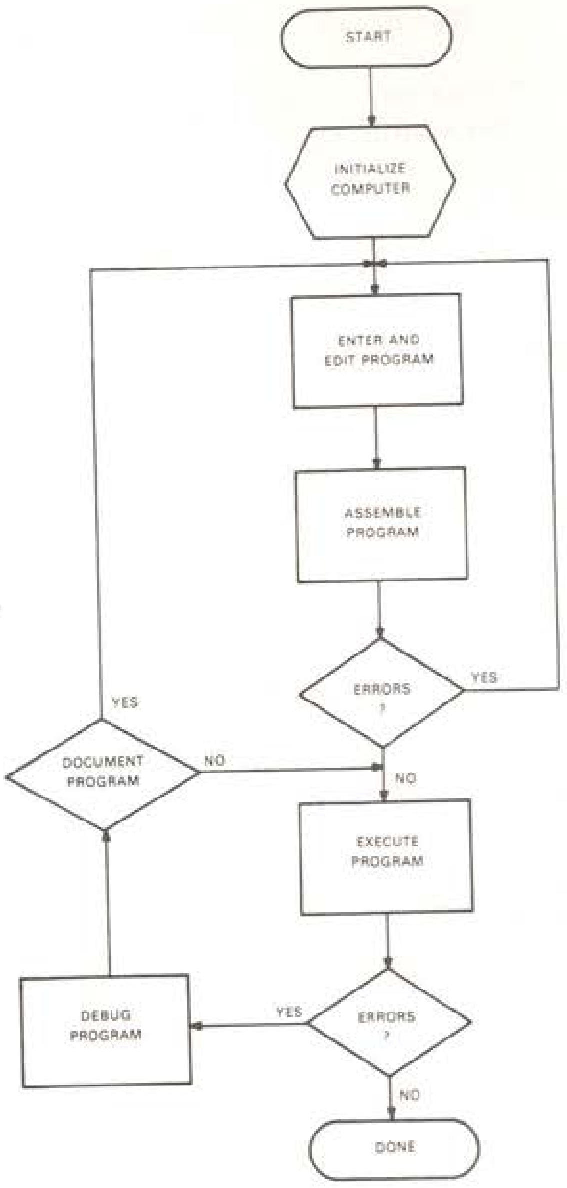
b)

c)

d)

e)

3. Fill in the following flowchart of the program development process. Next to each process step write the results or output of the step.



Results or Output

– Programmer outputs hand written source code.

a) "Ready" machine.

b) Source program on paper tape.

c) Symbol table and program listing, machine code paper tape and assembly errors if any.

d) Program results, logical errors if any.

e) Corrected machine code program on paper tape.

General Functions of An Operating System

OBJECTIVES

1. Given the six major parts of an operating system and six functions, be able to match each part with its function.

2. Given five memory management statements, be able to label those that refer to swapping, overlaying, and queuing, respectively.

SAMPLE TEST ITEMS

1. Match each of these major parts of an operating system with its function.

Part	Function
Interrupt Handler	______
Library Manager	______
•	•
•	•
•	•

Functions

a. Maintains and fetches programs when they are requested.

b. If a peripheral device were to request service, this unit would determine which routine would respond to the request.

•
•
•

SAMPLE TEST ITEMS

2. Indicate whether the following statements refer to swapping (S), overlaying (O), or to queuing (Q) by writing the correct letter in the space provided.

Statement	Refers To
Only parts of a single user program are in memory; remaining parts are fetched from secondary storage as needed.	________
⋮	⋮

General Functions

Operating systems automate the interaction between users and computers. Many operations that require manual intervention on the paper tape computer system are handled automatically on computers with operating systems.

In this lesson, the simplest type of operating system, one designed for a single user, is compared to paper-tape software, which also serves a single user. Also discussed are the general functions of an operating system.

Mark your place in this workbook and view Lesson 2 of the A/V program, "Operating Systems."

In a disk-based computer, the operating system loads programs to be executed, just as the general-purpose loader does in a paper tape-based computer. However, operating systems do much more.

For example, consider the development of an assembly language program on a paper tape computer system and on a computer with a single-user operating system. The general flowchart for the development process (see Figure 5) is the same for both systems.

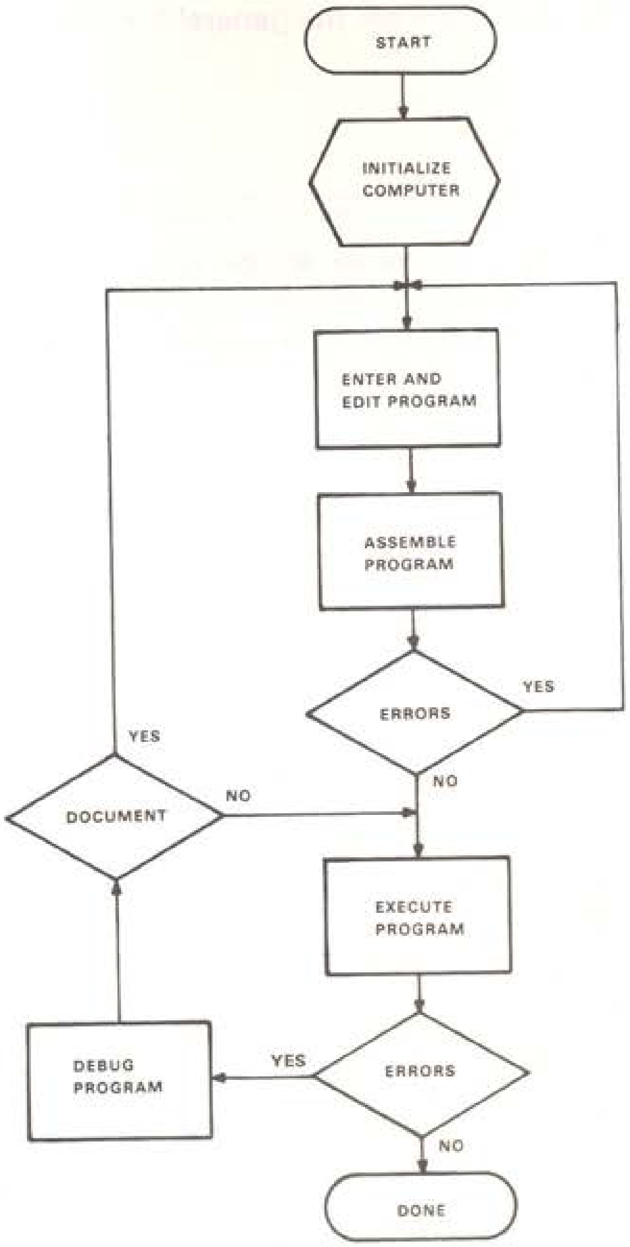


Figure 5 Development of an Assembly Language Program

However, in a paper tape system, each process begins with the programmer manually entering the starting address of the loader into the program counter through the front panel switches, threading a paper tape through the reader, and starting the computer. Then, after the program is loaded, its starting address is entered, and the computer is started again.

With even a simple single-user operating system, programs are started automatically when they are loaded in response to keyboard commands. Further, with an operating system, programs are loaded directly from the disk and are not handled by the programmer.

Major Components

A typical operating system is composed of *seven* major components:

- the *executive* (also called the monitor or supervisor)
- the *scheduler* (also called the dispatcher or queue manager)
- the *interrupt handler* (also called the interrupt service routine)
- the *device handlers* (also called the peripheral drivers)
- the *storage allocator* (also called the memory manager)
- the *library manager*
- the *library of system programs*

Each of the components and its function is discussed in turn in this lesson.

Executive. The *executive* (or supervisor or monitor) is the central component of the operating system. The executive is *always* memory-resident. As its name implies, the executive coordinates and controls all other components of the operating system.

Scheduler. The *scheduler* (or dispatcher) also has a self-explanatory name. The scheduler is often part of the executive and is responsible for two activities:

- scheduling or allocating CPU time to jobs
- establishing and maintaining various queues representing jobs waiting for computer resources

A queue is a storage structure in which the first request is the first to be served. This first-in/first-out (FIFO) characteristic is similar to the "lines" you encounter in everyday life outside theaters and restaurants. In large operating systems, there is a queue of programs waiting for each resource, including the CPU. Additional queues may be used to establish priorities or categories for jobs waiting to execute. Thus, some jobs may be preferentially allocated resources before other jobs. This is similar to restaurant lines in which customers with and without reservations are waiting for the same set of tables. The actual behavior of the scheduler in allocating resources and the number of queues is highly dependent on the fundamental type of the operating system. This topic will be discussed in the next two lessons as the various types of operating systems are described.

Interrupt Handler. The *interrupt handler* (or interrupt service routine) is also frequently part of the executive. The interrupt handler is responsible for handling interrupts arising from conditions such as:

- completion of an I/O process
- a request for service from a peripheral device
- an error occurrence (such an interrupt is also called a *trap*)

The interrupt handler performs *two* operations. When an interrupt is detected, the handler determines which routine is to be used to service the interrupt request. Before transferring control to the selected routine, the interrupt handler also saves the status information of the interrupted process. This action is done to ensure that the process may be resumed (once the interrupt request has been serviced).

Device Handlers. The *device handlers* (or peripheral drivers) are a collection of routines that connect the system and the user program with the peripheral devices. There is one device handler for each type of peripheral device in the hardware configuration supported by the operating system. Individual device handlers are required because each handler performs data transfers between its particular device and main memory. Because each device has unique hardware logic, the device handlers must have sections of their programs tailored to the individual devices' particular needs and features.

Storage Allocator. The *storage allocator* performs *two* major functions. First, the storage allocator is responsible for the allocation of main memory to a program. The allocator maintains a *memory map* that indicates to the scheduler what areas of memory are available for programs and what areas of memory are already allocated to which particular programs. In systems in which more than one program is allowed in memory at one time, the storage allocator makes sure the programs do not use each others' allocated memory. This activity is called *memory protection*. A sample memory map appears in Figure 6.

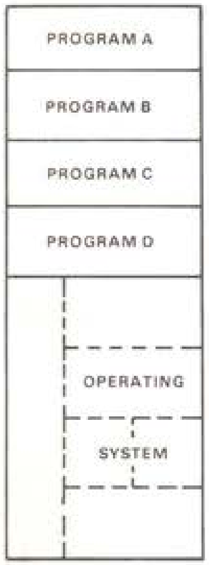


Figure 6 Memory Map

The second major function of the storage allocator is controlling the use of auxiliary storage for the *temporary* storage of all or parts of a program during execution. This function is required particularly for two operations: *swapping* and *overlaying*.

Swapping is a technique by which an operating system permits two or more programs to alternately use the same area of memory. While one program executes in main memory the remaining programs are stored in a suspended state on auxiliary storage. At some point, the program in main memory is suspended, stored on auxiliary storage, and another program is then loaded into main memory for execution. This *exchange of entire programs* is called swapping. If the swapping process occurs regularly and at frequent intervals, the computer can appear as if it were executing many programs at the same time and allowing each program to use the entire system. Thus, swapping is *invisible* to the programmer and is handled by the operating system through the storage allocator.

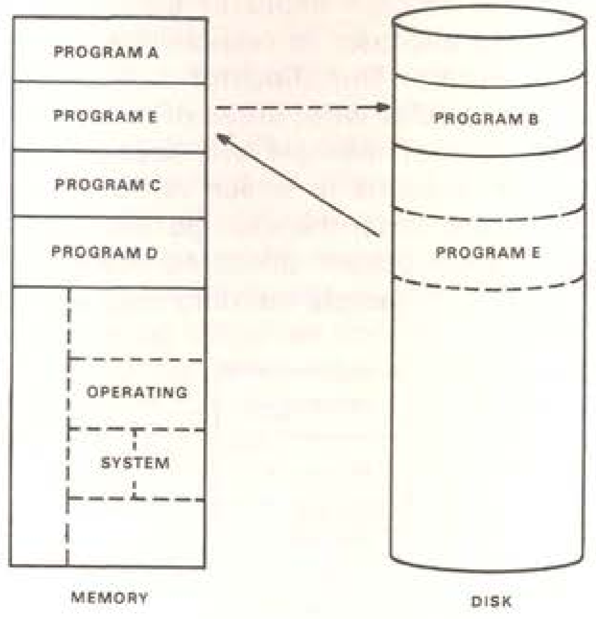


Figure 7 Swapping

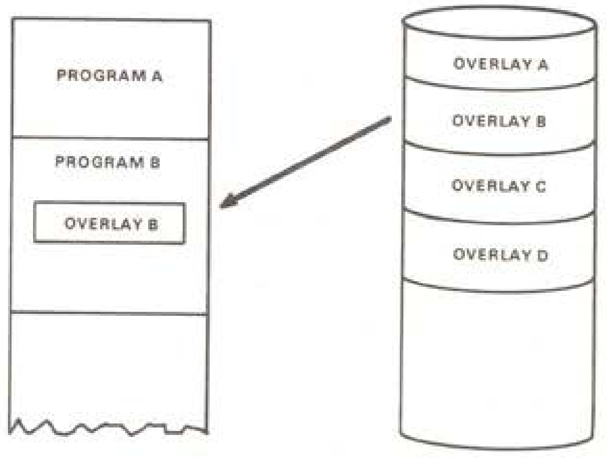


Figure 8 Overlaying with One Overlay Region and Four Overlays

Overlaying is a technique by which certain parts of a *single-user* program may take turns in main memory. This technique is useful for situations in which a program is physically larger than the available memory. When a program has been written with subroutines, overlaying allows the programmer to bring a given subroutine into memory only when it is needed. At other times the memory locations can be used by other subroutines. Figure 9 shows the "calling structure" of a typical user program. Notice the tree-like appearance. The mainline program calls subroutines A and B, while subroutine A in turn calls subroutines C and D. To use overlaying, the programmer specifies which routines are *always* to be memory-resident. These routines are called the *root*.

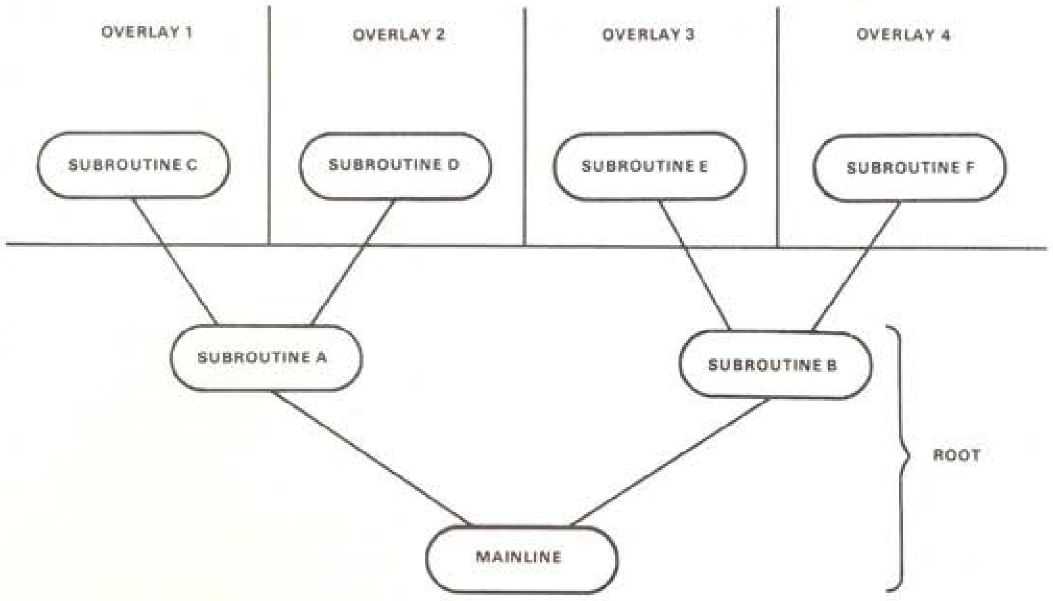


Figure 9 Overlay Structure

Note that in Figure 9, the mainline program (which is *always* in the root) and the subroutines A and B compose the root. The remaining subroutines are then grouped into overlays. In Figure 9, the program will have in memory, at any time, the following routines:

- the mainline, subroutine A *and* subroutine B, and
- one and *only* one of subroutines C, D, E or F in an overlay region.

The overlay region is typically a fixed area in the memory area assigned to the program. When a *new* overlay is required, the contents of the overlay regions are overwritten with the new overlay that is moved from auxiliary storage into the overlay region.

Thus, because only part of the program is present in main memory at any point, the physical size of the program may be considerably larger than the amount of available memory. Note that, unlike swapping, the user can control the behavior of the program by specifying which routines belong to the root and which routines belong to each overlay region. Once the program is executing, the actual overlaying operations are handled automatically by the memory allocator. Table 1 summarizes the differences between swapping and overlaying.

Table 1 Differences Between Swapping and Overlaying

Swapping	Overlaying
• Initiated by Operating System.	• Initiated by Program.
• Two directional — one program goes to the disk and another comes in.	• One directional — all overlays are kept on disk and copied into memory as needed, erasing what was in memory before.
• Entire programs and their data and status are swapped.	• Only subroutines are "overlayed." Data is not.

Libraries and the Library Manager

Libraries are collections of programs that are supplied with the operating system by the computer manufacturer or developed by users. There are three major parts to an operating system's libraries:

- the *system library*, including language translators, frequently used subroutines, and utility routines (such as the debugger, linker, editor, etc.);
- the *user library*, including user-written programs and data files;
- the *directory*, which is an index to where each member (a program file or a data file) has been stored.

The libraries are not strictly part of the operating system but *the program that controls their use* (the *library manager*) is, in fact, a component of the operating system that detects a request for a library member and locates it so that a device handler can be called to read or write it.

In summary, we have seen that the operating system itself is composed of six parts:

- the *executive* (controlling and coordinating all operations),
- the *scheduler* (determining which program gets a resource next)
- the *interrupt handler* (selecting which device is serviced next),
- the *device handlers* (managing data transfers and I/O operations),
- the *storage allocator* (controlling the usage of main memory and auxiliary storage), and
- the *library manager* (controlling use of the system and user libraries)

In the following lessons, you will learn how these components are combined to produce operating systems with very different characteristics from each other.

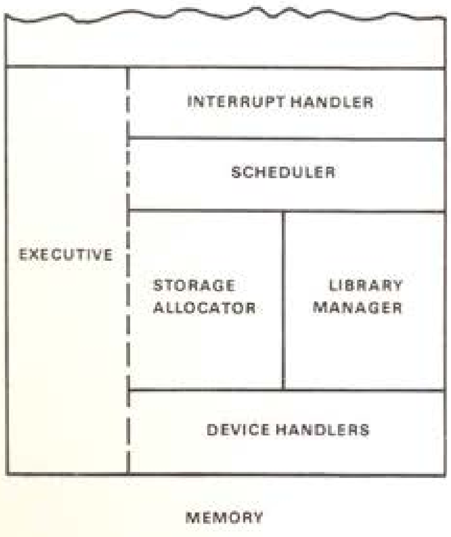


Figure 10 Operating System

EXERCISES

1. List the operations that a programmer performs manually on a paper tape system, but that are done automatically by an operating system.

2. Define:

   a) **Swapping**

   b) **Overlaying**

   c) **Queue**

SOLUTIONS

1. List the operations that a programmer performs manually on a paper tape system, but that are done automatically on an operating system.

   a) *Start Programs*

      – Load starting address into program counter.

      – Start computer.

   b) *Handle paper tapes* containing programs to be executed.

2. Define:

   a) **Swapping** – A memory management technique in which several different user programs and their data and status are alternately executed and exchanged between a common memory area and auxiliary storage.

   b) **Overlaying** – A memory management technique in which only parts of a program are in memory at any point. The remaining parts are stored on auxiliary storage until they are called by the program.

   c) **Queue** – A data structure in which the first item in the line is the first to be serviced (first-in/first-out).

3. List the six major components of an operating system and give one function for each.

Component	Function

3. List the six major parts of an operating system and give one function for each.

Component	Function
Executive	a) Coordinates and controls all other components.
Scheduler	a) Determines which program gets a given resource next.
	b) Maintains and establishes queues for priorities.
Interrupt Handler	a) Selects which device is to be serviced next.
	b) Saves status information of the interrupted program so that it can be resumed.
Device Handler	a) Control data transfers between devices and main memory.
	b) Connects CPU and user program with input/output devices.
Storage Allocator	a) Maintains memory map for scheduler.
	b) Allocates and controls which areas of memory and auxiliary storage are used by which user programs.
Library Manager	a) Controls libraries of system and user programs, and data files.
	b) Fetches a library member when it is requested.

On-Line Operating Systems

OBJECTIVE

Given a table of 5 general functions of operating systems and 3 types of on-line operating systems, and 15 descriptions, be able to match each function/type combination with its description.

SAMPLE TEST ITEM

Three types of on-line operating systems, five functions of operating systems, and fifteen descriptions are given below. Match each combination of type and function with its description.

Function	Time-Sharing	Real-Time	Single User
Executive	______	______	______
Scheduler	______	______	______
Storage Allocator	______	______	______
Library Manager	______	______	______
Application	______	______	______

Functions

Executive
- a. Multiprograms machine tasks.
- b. Multiprograms user tasks.
- c. Loads and starts requested programs.

Scheduler
- d. Time slices all user tasks.
- e. Plans time-critical tasks.
- f. Performed by the user program.

Storage Allocator

•

•

•

Mark your place in this workbook and view Lesson 3 of the A/V program, "Operating Systems."

Single-User Operating Systems

The simplest form of operating system is that which is designed to serve only one user at a time. Because users are not competing for computer resources, each operating system component can be made as simple as possible.

In operation, program development on a single-user system resembles the program development process described in Lesson 1 – with one important exception. Commonly used programs such as the loader, editor, assembler, and debugger are stored on some auxiliary storage medium (usually a disk). Instead of having to load a machine code tape each time the user wishes a new program, the user types a command on the console keyboard, and the operating system loads the desired program without requiring further user action.

In a single-user operating system, the components have the following responsibilities:

- The *executive* coordinates all operating system actions, and loads and starts programs requested by the user through commands typed on the console keyboard.

- The *scheduler* is normally nonexistent in a single-user system because the single user at the terminal makes all the decisions as to what programs to run in which order.

- The *interrupt handler* remains essentially the same size regardless of the number of users as it is responsible for identifying I/O devices that issue interrupts. The demands on the interrupt handler are a function of the hardware configuration and not of the operating system.

- The *device handlers* (one handler for each type of peripheral device) move information between their specific devices and main memory. Device handlers are relatively standard programs that do not vary from one operating system to the next because common I/O techniques are used. When a device handler is started, it is told how many words of information to move, the address of a memory buffer, the location on the peripheral device (if it has locations) and in which direction to move the information. The device handler moves the information and returns control to the executive.

- The *storage allocator* in a single-user system is relatively simple because the memory map will always show just the operating system and one user program. Thus, the user is allocated all of memory not used by the operating system, and the storage allocator only manages one memory area. Overlay capability is included in some systems to handle very large user programs, but swapping capability is generally not implemented.

- The *library manager* on single-user operating systems oversees simple files that contain programs and data and are all available to the user of the system.

NOTE

In many single-user operating systems, the executive can receive commands not only from the user at the terminal, but also from the executing program. This feature allows programmers to use the general functions of the operating system to handle interrupts, handle information movement to and from peripheral devices, and manage the files on the system. Thus, the programmer need not write programs to perform these functions.

The single-user operating system represents the earliest developed form of operating systems. Its primary advantages are its faster and more convenient operation than paper tape systems, and the fact that programs can call the operating systems to perform standard functions. However, single-user operating systems are generally found only on computers with minimal hardware configurations. This is because the major disadvantage of single-user systems is the inefficient use of CPU time compared to larger, more complex operating systems. On a single-user system, while the user program is performing an I/O operation, the CPU is largely idle because CPUs work much faster than peripheral devices.

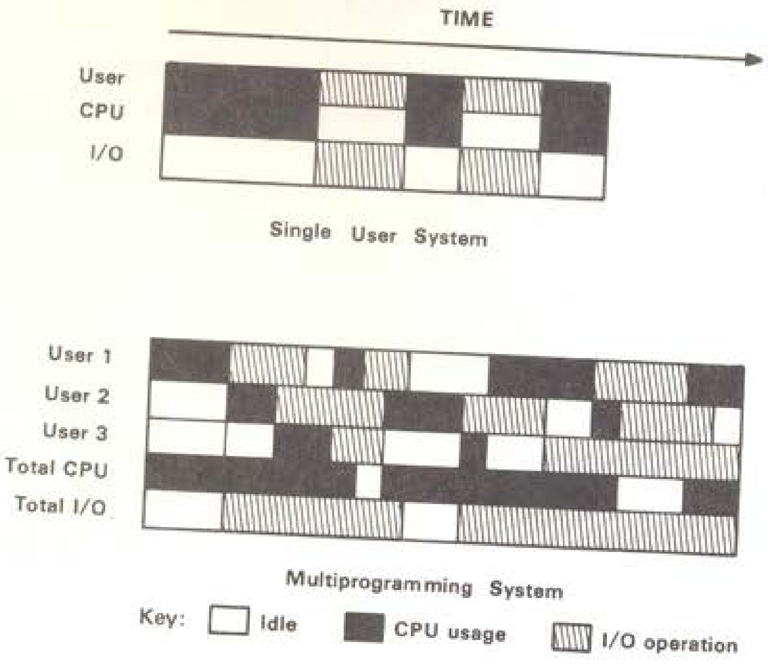


Figure 11 Single User vs Multiprogramming

The idle time of the CPU is reduced in most large operating systems through what is called a *multi-user* or *multiprogramming* capability. Multiprogramming allows two or more programs to be executed by the CPU concurrently. Each program runs in a separate area of memory (called a *partition*) and uses the CPU alternately. Note that only *one program executes* at any given instant because there is still *only one CPU*. But, while one program is waiting for an I/O device, another program can be executed. Figure 11 graphically shows the increased CPU and I/O efficiency of multiprogramming. Notice that the CPU and I/O devices are seldom idle for long, and that frequently both are active at the same time. Hence, multiprogramming provides a much higher utilization of computer resources. On the other hand, an individual user's *program* may occasionally be idle. Thus, the overall system's performance is improved at the expense of slightly slower execution speed for *each* user program.

EXERCISE

For a single-user operating system, describe with a phrase the function of each routine listed below.

Single-User Operating System

a) **executive**

b) **scheduler**

c) **storage allocator**

d) **library manager**

SOLUTION

For a single-user operating system, describe with a phrase the function of each routine listed below.

	Single-User Operating System
a) **executive**	– loads and starts requested programs in response to keyboard commands or calls from executing program
b) **scheduler**	– this function performed by user at terminal, or by executing program
c) **storage allocator**	– manages one memory area
d) **library manager**	– oversees simple files, all available to the user

Timesharing Operating Systems

Timesharing operating systems multiprogram for many on-line users. Computer resources are shared concurrently among the users, permitting each user to interface with the system as if it were a single-user system. Competition for common resources is managed automatically by the operating system.

In a timesharing operating system, the operating system components have the following responsibilities:

- The *executive* is more complex than for a single-user system because of the number of users. The activities of the executive are similar to those of a single-user system, except now there are many users to coordinate. The executive loads and starts programs for each user in response to keyboard commands.

- The *scheduler*, however, is far more complex than for a single-user system. All the user programs, sometimes called user *tasks,* are executed in a sequential or "round-robin" manner. Each program is allowed to execute for a maximum period of time called a "time slice." At the end of the time slice or when an I/O operation is requested, the program is halted and another program executed. The status of the interrupted program is saved to allow it to be resumed when its turn for execution comes again. Determination of the end of a time slice is provided by an internal clock that interrupts the CPU and signals that a specific period of time has elapsed. Some timesharing systems allow program priority levels in order to give some users or some programs preferential access to both CPU time and peripheral devices. Because of these responsibilities, the design of the scheduler is crucial to the performance of a timesharing operating system.

- The *storage allocator* is also very complex. Because there are several users, the memory map is more complex, and users must be protected from each other. Additionally, most timesharing systems allow more user programs than can be stored in memory at one time. Therefore, the technique of *swapping* is heavily used. Hence, a user program may reside in memory only when it is being executed. At all other times, it will be stored on auxiliary storage – waiting for its next turn or time slice. Like the scheduler, the design of the memory allocator is very important to the operating system's performance.

- The *library manager* in a timesharing system must not only provide access to files on auxiliary storage, but must limit access to files to authorized users. This requires users to enter identification numbers and passwords that can be checked against access restrictions for each file. These restrictions are normally recorded in the file directory along with the name of each file and other information.

- The *interrupt handler* and the *device handlers,* are all basically similar to those for a single-user operating system.

Timesharing systems are designed for on-line processing. Programs are generally developed on-line with the assistance of interpreters or compilers of high-level languages. However, most systems also have assemblers for developing programs that must execute quickly. Because timesharing systems allow many users to develop programs concurrently, they are frequently found in academic environments where many students need access to a computer. In a timesharing system, interactive programs can be developed for on-line modification of data files. Typical applications using this feature include inventory control and airline reservations handling programs.

There are several important *advantages* of timesharing systems:

- Each user has the benefit of a dedicated computer at a fraction of the cost.

- If the system's hardware and software are adequate, the user is not aware of other users.

- The CPU and other resources are efficiently used.

- Programs may be interactively developed.

- Data files can be updated on-line.

Counterbalancing these advantages are several important *disadvantages:*

- A significant part of main memory is taken up by the executive and other parts of the operating system.

- There is a larger amount of overhead involved because of swapping and other system operations.

- Individual programs take longer to execute than on a single-user system.

- A much larger capital outlay is required for such things as a fast CPU, multiple terminals for several users, more main memory, and faster and larger auxiliary storage devices.

Hence, timesharing systems are not always the best choice for a particular application.

EXERCISE

For a timesharing operating system, describe with a phrase the function of each routine listed below.

Timesharing Operating System

a) **executive**

b) **scheduler**

c) **storage allocator**

d) **library manager**

SOLUTION

For a timesharing operating system, describe with a phrase the function of each routine listed below.

	Timesharing Operating System
a) **executive**	– multiprograms user tasks, loading and starting each in turn
b) **scheduler**	– time slices all current user tasks
c) **storage allocator**	– swaps user tasks continually
d) **library manager**	– limits access to files to authorized users

Real-Time Systems

Real-time operating systems are designed to interact with machines, which often operate faster than people at terminals. Certain applications, such as industrial process control or intensive care patient monitoring, cannot tolerate the inherent slowness of "round-robin" scheduling. Accordingly, real-time operating systems are designed to provide rapid responses to certain priority programs or devices. The term "real-time" means that the system responds to external events during the time in which they are *actually* (really) occurring. Thus, the computer can process information and respond fast enough to control events while they are happening.

Real-time systems may allow more than one program to operate concurrently. In this case, program priorities are definitely established to indicate which programs receive the most rapid responses.

The responsibilities of the operating system components are similar to those for a time-sharing system, except that the handling of interrupts and the servicing of peripheral devices is far more critical. Therefore, great care is taken to ensure that the interrupt handler and required device handlers are efficient and memory-resident at all times.

Real-time systems are used in applications such as laboratory or medical instrument monitoring and industrial process control. The primary advantage is fast response to external stimuli. The orientation of the operating system toward this special ability makes a real-time system somewhat less flexible for applications not requiring rapid responses. In summary:

- The *executive* in a real-time operating system multiprograms tasks that service other machines.
- The *scheduler* plans execution based upon priorities that have been set according to the time-critical nature of the machine being serviced by each program or task.
- The *storage allocator* swaps tasks that are not time-critical and allocates permanent space to tasks that must be started in less time than it takes to swap them into memory.
- The *library manager* operates quickly to reference files and make information available.

Further abilities that increase a real-time system's flexibility will be discussed in the next lesson under the topic "Foreground and Background Programs."

EXERCISE

For a real-time operating system, briefly describe the function of each routine listed below.

Real-Time Operating System

a) **executive**

b) **scheduler**

c) **storage allocator**

d) **library manager**

SOLUTION

For a real-time operating system, briefly describe the function of each routine listed below.

	Real-Time Operating System
a) **executive**	– multiprograms tasks that interact with machines
b) **scheduler**	– plans execution of time-critical tasks according to priorities based on urgency
c) **storage allocator**	– allocates some memory, permanently, to programs that must be started in a hurry; the rest is allocated on a temporary, swapping basis
d) **library manager**	– accesses and creates files quickly

Summary

Table 2 summarizes the major differences between single-user, timesharing, and real-time operating systems. Look carefully at this table before going on to do the exercises for this lesson.

Table 2 Comparison of On-line Operating Systems

	Single User	Timesharing	Real Time
Executive	Loads and starts requested programs	Multiprograms user tasks	Multiprograms machine tasks
Scheduler	This function performed by the user or program	Time slices all user tasks	Plans time-critical tasks
Storage Allocator	Manages one memory area	Swaps continuously	Allocates memory permanently
Library Manager	Oversees simple files	Limits access to files	References files quickly

Single-user operating systems are used in applications where the capital outlay for a larger system cannot be justified, yet the computer must perform a wide variety of tasks. These applications include small laboratories, small educational institutions, and small private businesses. In these applications the same system is used for program development and the execution of programs for the ongoing operations of the group, department, or business.

Timesharing operating systems are used in applications where many people must interact with the computer to develop and execute programs. Thus, timesharing systems are generally found in the educational environment and in business data processing environments where it is important for people to access data on the system.

Real-time operating systems are used in applications where fast response is necessary to keep automated processes operating properly. Thus, in general, real-time operating systems control machines such as atomic reactors, paper making machines, chemical plants, scientific measuring instruments, and valves in water supply systems and transcontinental pipelines.

EXERCISE

List the general functions that distinguish single-user, timesharing, and real-time operating systems. Describe briefly how each type of operating system performs these functions.

SOLUTION

List the general functions that distinguish single-user, time-sharing, and real-time operating systems. Describe briefly how each type of operating system performs these functions.

	Single User	Timesharing	Real Time
Executive	Loads and starts requested programs	Multiprograms user tasks	Multiprograms machine tasks
Scheduler	This function performed by the user or program	Time slices all user tasks	Plans time-critical tasks
Storage Allocator	Manages one memory area	Swaps continuously	Allocates memory permanently
Library Manager	Oversees simple files	Limits access to files	References files quickly

EXERCISE

For the operating systems below, list applications in which each might be used.

a) **Single-User Operating System**

b) **Timesharing Operating System**

c) **Real-Time Operating System**

SOLUTION

For the operating systems below, list applications in which each might be used.

a) **Single-User Operating System**

- Small groups

  – laboratory, education, business, where both program development and data processing are done.

b) **Timesharing Operating System**

- Education

  – Computer science

  – Executing programs to process data for other disciplines

- Business

  – Where people must access information on computer.

c) **Real-Time Operating System**

- Industrial Control

  – Atomic reactors

  – Paper making machines

  – Chemical production

- Science and Engineering

  – Fast measuring instruments

Off-Line Operating Systems

OBJECTIVES

1. Given five general functions of an off-line operating system and five descriptions, be able to match each function with its description.

2. Given six statements and examples referring to operating systems, be able to label those that refer to foreground programs and those that refer to background programs.

SAMPLE TEST ITEMS

1. Match each of these off-line operating system functions with its description.

Function	Description
Executive	________
Scheduler	________
Storage Allocator	________
Library Manager	________
Application	________

Descriptions

a. Handles multiple directories.
b. Manages space for different sizes of programs.

•
•
•

SAMPLE TEST ITEMS

2. Indicate whether these statements and examples refer to foreground (F) or background (B) programs by writing the correct letter in the space provided.

Statements and Examples	Type of Program
May be a subordinate operating system such as batch.	______
Example: process control.	______
•	•
•	•
•	•

Mark your place in this workbook and view Lesson 4 of the A/V program, "Operating Systems."

Batch Operating Systems

A batch operating system is a system in which programs are handled non-interactively as a single input stream of tasks. The executive identifies each individual program in the input stream. These individual programs are then run consecutively without operator intervention. In larger batch systems allowing multiprogramming, the scheduler may allow priorities to be established for classes of programs.

In contrast to timesharing and most real-time systems, the editing process of program development in a batch system is normally performed off-line, typically using keypunches or similar devices. The program is combined with the accompanying data and submitted as a job. Many jobs are submitted as an input stream, sometimes called a *batch stream.* At some later time, dependent upon the job's priority and its place in the input queue, the job is executed and its results are output.

Batch processing suffers somewhat from the lack of interaction available during program development and execution. Interpreters are rarely found in batch systems; compilers and assemblers are the rule. Further, programs cannot be written to interact with users at keyboard terminals, and all program data must be submitted with the program.

A second disadvantage is the time delay or "turn-around time" existing between the time a program is submitted and the time the results and/or errors are received for analysis. For experienced programmers in a production environment, this feature is actually an advantage as the typing of programs and data onto cards can be performed by less skilled, less expensive personnel. However, with an interactive system, programmers generally must type in their own programs.

The major advantage of batch systems is efficiency. When multiprogramming is implemented, jobs are executed one after another without time lapses in between. Furthermore, when one program is waiting for I/O operations, another can run in its place. There is also less system overhead in a batch system than in a timesharing or real-time system, because the executing program is not being constantly interrupted.

Because of the emphasis on efficiency and the existence of turn-around time, batch system applications tend to be the following:

- programs requiring large amounts of processing time, and often large amounts of stored data

- programs generally of a non-urgent nature, of varying sizes and descriptions

- programs of a routine or periodic nature (e.g., payroll or accounting)

These characteristics describe nearly all commercial and non-time-critical scientific applications with a large amount of data to be processed. Often the data handled by a batch operating system is so large that many auxiliary storage devices, each with its own directory, are needed to hold the information.

In a batch operating system, the general functions are performed by the various operating system routines as follows:

- The *executive* multiprograms non-time-critical programs that have been prepared off-line.

- The *scheduler* uses complex priority systems to queue tasks for execution, and if spooling is available on the system, spooling programs must also be scheduled.

- The *storage allocator* manages space for many different kinds and sizes of programs.

- The *library manager* handles large numbers of program files and data files, often using multiple auxiliary storage devices, each with its own directory.

The following table summarizes the major differences among the three operating systems: timesharing, real-time, and batch.

Table 3 Comparison of Multiprogramming Operating Systems

Criterion	Timesharing	Real-Time	Batch
Primary input medium	Human via terminal	Special purpose hardware device	Cards
Method of service	Round robin	By internal or external event or interrupt	Sequential
Primary advantage	Interactive for many users	Rapid response for time critical tasks	Efficient use of processor and peripherals
Disadvantages	Execution speeds are slow Requires expensive hardware configurations	Less efficient as a multi-user system	Completely non interactive Introduces time delay between submission and results
Typical applications	Education Inventory control Airline reservations	Industrial process control Medical patient monitoring Scientific laboratory monitoring	Many commercial and scientific applications with long jobs and high volumes

Foreground/Background Operating Systems

In some applications, a combination of on-line and off-line processing must be done. This combination occurs most frequently in real-time control applications where inventory and production reports must be generated.

Often, the computer controlling the operation of machinery is not kept busy by this task and spends time waiting for an interrupt from an external event. In these applications, a so-called foreground/background operating system can be used to give the computer other jobs to do while it is waiting for an interrupt.

A foreground/background operating system is basically a real-time operating system that also executes non-time-critical programs such as are normally executed under a batch operating system. In a foreground/background system, memory is typically divided into three areas: the operating system, the current real-time programs, and the current non-critical batch-type programs.

The memory area that holds the real-time programs is called the *foreground;* the area that holds the non-critical batch-type programs is called the *background*.

Thus, foreground programs are time-critical tasks such as patient monitoring or process control. Time-critical tasks are further distinguished by their interaction with devices such as A/D converters and remote sensors. These devices have a "mechanical" rather than "human" orientation.

By contrast, background programs are low-priority tasks that run when the foreground program is not running. Typical background tasks include program development tasks, report generation, mailing list maintenance, payroll, and other tasks that are not time critical. A background task runs until one of the following occurs:

- A hardware interrupt for a foreground task is detected.
- The computer operator requests interruption of the background program.
- The background program calls an operating system routine.
- A scheduled event occurs. This type of interrupt can be triggered by the time of day, an elapsed time counter, or a periodic schedule such as once a minute or every hour.

In summary, foreground/background systems make efficient use of an entire computer system. Real-time control is performed in the foreground. When no time-critical program is running in the foreground, a background program can be executed to produce payrolls, reports, mailing lists, or other batch-type jobs.

EXERCISES

1. For a batch operating system, describe briefly the functions of each routine listed below.

Batch Operating System

a) **executive**

b) **scheduler**

c) **storage allocator**

d) **library manager**

SOLUTIONS

1. For a batch operating system, describe briefly the functions of each routine listed below.

	Batch Operating System
a) **executive**	– multiprograms tasks prepared off-line
b) **scheduler**	– uses complex priorities to queue tasks and spooling programs
c) **storage allocator**	– manages space for different sizes of programs
d) **library manager**	– handles large quantities of data with multiple directories

EXERCISES

2. For each statement, place an X in the box or boxes under the operating system that best fits the statement.

	Timesharing	Real Time	Batch
a) **Primary medium is:**			
• special-purpose hardware	☐	☐	☐
• cards	☐	☐	☐
• humans via terminals	☐	☐	☐
b) **Method of service is:**			
• sequential	☐	☐	☐
• "round robin" basis	☐	☐	☐
• by interrupts	☐	☐	☐
c) **A typical application area is:**			
• education	☐	☐	☐
• medical patient monitoring	☐	☐	☐
• high-volume processing	☐	☐	☐
d) **A disadvantage is:**			
• requires expensive hardware	☐	☐	☐
• introduces "turn around" delay	☐	☐	☐
• less efficient as a multi-user system	☐	☐	☐
• usually completely non-interactive	☐	☐	☐
• execution speeds are slow	☐	☐	☐
e) **The major advantage is:**			
• rapid response for critical tasks	☐	☐	☐
• interactive processing for many users	☐	☐	☐
• very efficient use of resources	☐	☐	☐

SOLUTIONS

2. For each statement, place an X in the box or boxes under the operating system that best fits the statement.

	Timesharing	Real Time	Batch
a) **Primary medium is:**			
• special-purpose hardware	☐	☒	☐
• cards	☐	☐	☒
• humans via terminals	☒	☐	☐
b) **Method of service is:**			
• sequential	☐	☐	☒
• "round robin" basis	☒	☐	☐
• by interrupts	☐	☒	☐
c) **A typical application area is:**			
• education	☒	☐	☐
• medical patient monitoring	☐	☒	☐
• high-volume processing	☐	☐	☒
d) **A disadvantage is:**			
• requires expensive hardware	☒	☐	☐
• introduces "turn around" delay	☐	☐	☒
• less efficient as a multi-user system	☐	☒	☐
• usually completely non-interactive	☐	☐	☒
• execution speeds are slow	☒	☐	☐
e) **The major advantage is:**			
• rapid response for critical tasks	☐	☒	☐
• interactive processing for many users	☒	☐	☐
• very efficient use of resources	☐	☐	☒

3. Explain the difference between foreground and background programs. Give an application of each.

3. Explain the difference between foreground and background programs. Give an application of each.

   - **Foreground programs** – in a real-time operating system, the high-priority programs requiring rapid responses to interrupts to avoid losing highly changeable data. Industrial process control and medical patient monitoring are example applications.

   - **Background programs** – the low-priority programs of a non-time-critical nature. The background task may be a subordinate operating system such as timesharing, single-user, or batch systems. Typical background applications include program development and other non-critical programs such as payroll, etc.

Take the test for this module and evaluate your answers.