1

AN OVERVIEW OF OPERATING SYSTEMS

This chapter describes the purpose and technological background of operating systems. It stresses the similarities of all operating systems and points out the advantages of special-purpose over general-purpose systems.

1.1. THE PURPOSE OF AN OPERATING SYSTEM

1.1.1. Resource Sharing

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. The sharing of a computer installation is an economic necessity, and the purpose of an operating system is to make the sharing tolerable.

An operating system must have a *policy* for choosing the order in which competing users are served and for resolving conflicts of simultaneous requests for the same resources; it must also have means of *enforcing* this policy in spite of the presence of erroneous or malicious user programs.

Present computer installations can execute several user programs simultaneously and allow users to *retain data* on backing storage for weeks or months. The simultaneous presence of data and programs belonging to different users requires that an operating system *protect* users against each other.

Since users must pay for the cost of computing, an operating system must also perform *accounting* of the usage of resources.

In early computer installations, operators carried out most of these functions. The purpose of present operating systems is to carry out these tasks automatically by means of the computer itself. But when all these aspects of sharing are automated, it becomes quite difficult for the installation management to find out what the computer is actually doing and to modify the rules of sharing to improve performance. A good operating system will assist management in this evaluation by collecting *measurements* on the utilization of the equipment.

Most components of present computer installations are sequential in nature: they can only execute operations or transfer data items one at a time. But it is possible to have activities going on simultaneously in several of these components. This influences the design of operating systems so much that our subject can best be described as the *management of shared multiprogramming systems.*

1.1.2. Virtual Machines

An operating system defines several languages in which the rules of resource sharing and the requests for service can be described. One of these languages is the *job control language*, which enables users to identify themselves and describe the requirements of computational jobs: the types and amounts of resources needed, and the names of programs and data files used.

Another language is the *virtual machine language*: the set of machine operations available to a user during program execution. To maintain control of a computer installation and isolate users from each other, an operating system must prevent user programs from executing certain operations; otherwise, these programs could destroy procedures or data inside the operating system or start input/output on peripheral devices assigned to other users. So the set of machine operations available to users is normally a subset of the original machine language.

But users must have some means of doing input/output. The operating system enables them to do so by calling certain standard procedures that handle the peripherals in a well-defined manner. To the user programs, these standard procedures appear to be extensions of the machine language available to them. The user has the illusion of working on a machine that can execute programs written in this language. Because this machine is

partly simulated by program, it is called a *virtual machine.* So an operating system makes a virtual machine available to each user and prevents these machines from interfering destructively with each other. The simultaneous presence of several users makes the virtual machines much slower than the physical machine.

An operating system can make the programming language of the virtual machine more attractive than that of the original machine. This can be done by relieving the user of the burden of technological details such as the physical identity of peripheral devices and minor differences in their operation. This enables the user to concentrate on logical concepts such as the names of data files and the transfer of data records to and from these files. The virtual machine can also be made more attractive by error correction techniques; these make the virtual machine appear more reliable than the real one (for example, by automatic repetition of unsuccessful input/output operations). In this way an operating system may succeed in making a virtue out of a necessity.

Yet another language is the one used inside the operating system itself to define the policy of sharing, the rules of protection, and so on. A certain amount of bitter experience with present operating systems has clearly shown that an operating system may turn out to be inefficient, unreliable, or built on wrong assumptions just like any other large program. Operating systems should be designed so that they are simple to understand, and easy to use and modify. Even if an operating system works correctly, there is still a need for experimenting with its policy towards users and for adapting it to the requirements of a particular environment, so it is important not only to give users an attractive programming language, but also to design good programming tools to be used inside the operating system itself. But since the operating system is imposed on everyone, it is extremely important that the language used to implement it reflect the underlying machine features in an efficient manner.

1.1.3. Operating Systems and User Programs

Operating systems are *large programs* developed and used by a changing group of people. They are often modified considerably during their lifetimes. Operating systems must necessarily impose certain restrictions on all users. But this should not lead us to regard them as being radically different from other programs—they are just complicated *applications* of general programming techniques.

During the construction of operating systems over the past decade, new methods of multiprogramming and resource sharing were discovered. We now realize that these methods are equally useful in other programming applications. Any large programming effort will be heavily influenced by the characteristics and amounts of physical resources available, by the

possibility of executing smaller tasks simultaneously, and by the need for sharing a set of data among such tasks.

It may be useful to distinguish between operating systems and user computations because the former can *enforce* certain rules of behavior on the latter. But it is important to understand that each level of programming solves some aspect of resource allocation.

Let me give a few examples of the influence of resource sharing on the design of standard programs and user programs.

Store allocation. One of the main reasons for dividing a compiler into smaller parts (called *passes*) is to allocate storage efficiently. During a compilation, the passes can be loaded one at a time from drum or disk into a small internal store where they are executed.

Job scheduling. A data processing application for an industrial plant can involve quite complicated rules for the sequence in which smaller tasks are scheduled for execution. There may be a daily job which records details of production; weekly and monthly jobs which compute wages; a yearly job associated with the fiscal year; and several other jobs. Such long-term scheduling of related jobs which share large data files is quite difficult to control automatically. In contrast, most operating systems only worry about the scheduling of independent jobs over time spans of a few minutes or hours.

Multiprogramming. To control an industrial process, engineers must be able to write programs that can carry out many tasks simultaneously, for example, measure process variables continuously, report alarms to operators, accumulate measurements of production, and print reports to management.

Program protection. The ability to protect smaller components of a large program against each other is essential in real-time applications (such as banking and ticket reservation) where the service of reliable program components must be continued while new components are being tested.

So the problems of resource sharing solved by operating systems repeat themselves in user programs; or, to put it differently, every large application of a computer includes a local operating system that coordinates resource sharing among smaller tasks of that application. What is normally called "the operating system" is just the one that coordinates the sharing of an entire installation among users.

When you realize that resource sharing is not a unique characteristic of operating systems, you may wonder whether the simulation of virtual machines makes operating systems different from other programs. But alas, a closer inspection shows that all programs simulate virtual machines.

Computer programs are designed to solve a *class of problems* such as

the editing of all possible texts, the compilation of all possible Algol programs, the sorting of arbitrary sets of data, the computation of payrolls for a varying number of employees, and so on. The user specifies a particular case of the class of problems by means of a set of data, called the *input*, and the program delivers as its result another set of data, called the *output*.

One way of looking at this flexibility is to say that the input is a sequence of instructions written in a certain language, and the function of the program is to follow these instructions.

From this point of view, an editing program can execute other programs written in an editing language consisting of instructions such as *search*, *delete*, and *insert* textstring. And an Algol compiler can execute programs written in the Algol 60 language. The computer itself can be viewed as a physical implementation of a program called the *instruction execution cycle*. This program can carry out other programs written in a so-called machine language.

If we adopt the view that a computer is a device able to follow and carry out descriptions of processes written in a formal language, then we realize that each of these descriptions (or programs) in turn makes the original computer appear to be another computer which interprets a different language. In other words, an editing program makes the computer behave like an editing machine, and an Algol compiler turns it into an Algol 60 machine. Using slightly different words, we can say that a program executed on a physical machine makes that machine behave like a virtual machine which can interpret a different programming language. And this language is certainly more attractive *for its purpose* than the original machine language; otherwise, there would be no reason to write the program in the first place!

From these considerations it is hard to avoid the conclusion that operating systems must be regarded merely as large application programs. Their purpose is to manage resource sharing, and they are based on general programming methods. The proper aim of education is to identify these methods. But before we do that, I will briefly describe the technological development of operating systems. This will give you a more concrete idea of what typical operating systems do and what they have in common.

1.2. TECHNOLOGICAL BACKGROUND

1.2.1. Computer and Job Profiles

We now go back to the middle of the 1950's to trace the influence of the technological development of computers on the structure of operating systems.

When many users share a computer installation, queues of computa-

tions submitted for execution are normally formed, and a decision has to be made about the order in which they should be executed to obtain acceptable overall service. This decision rule is called a *scheduling algorithm.*

A computation requested by a user is called a *job*; it can involve the execution of several programs in succession, such as editing followed by compilation and execution of a program written in a high-level language. A job can also require simultaneous execution of several programs cooperating on the same task. One program may, for example, control the printing of data, while another program computes more output.

In the following, I justify the need for automatic scheduling of jobs by quite elementary considerations about a *computer installation* with the following characteristics:

instruction execution time	2 μsec
internal store	32 K words
card reader	1,000 cards/min
line printer	1,000 lines/min
magnetic tape stations	80,000 char/sec

($1\ \mu = 10^{-6}$, and 1 K = 1024).

We will consider an environment in which the main problem is to schedule a *large number of small jobs* whose response times are as short as possible. (The *response time* of a job is the interval between the request for its execution and the return of its results.) This assumption is justified for universities and engineering laboratories where program development is the main activity.

A number of people have described the typical *job profile* for this type of environment (Rosin, 1965; Walter, 1967). We will assume that the average job consists of a compilation and execution of a program written in a high-level language. The source text read from cards and listed on a printer, is the major part of the input/output. More precisely, the average job will be characterized by the following figures:

input time (300 cards)	0.3 min
output time (500 lines)	0.5 min
execution time	1 min

1.2.2. Batch-processing Systems

For the moment we will assume that *magnetic tape* is the only form of backing store available. This has a profound influence on the possible forms of scheduling. We also impose the technological restriction on the computer that its mode of operation be *strictly sequential.* This means that: (1) it can

only execute one program at a time; and (2) after the start of an input/output operation, program execution stops until the transfer of data has been completed.

The simplest scheduling rule is the *open shop* where users each sign up for a period of, say, 15 min and operate the machine themselves. For the individual user this is an ideal form of service: it enables him to correct minor programming errors on the spot and experiment with programs during their execution. Unfortunately, such a system leads to prohibitive costs of idle machinery: for an average job, the central processor will only be working for one out of every 15 min; the rest of the time will be spent waiting for the operator. The situation can be characterized by two simple measures of average performance:

processor utilization = execution time/total time

throughput = number of jobs executed per time unit

For the open shop, processor utilization is only about 7 per cent with a throughput of no more than 4 jobs per hour, each requiring only one minute of execution time(!).

Idle processor time caused by manual intervention can be greatly reduced by even the most *primitive* form of *automatic scheduling*. Figure 1.1 illustrates an installation in which users no longer can interact with programs during execution. They submit their jobs to an operator who stacks them in the card reader in their order of arrival. From the card reader, jobs are input directly to the computer, listed on the printer, and executed one by one. This scheduling is done by an operating system which resides permanently in the internal store.

Under this form of scheduling an average job occupies the computer for 1.8 min (the sum of input/output and execution times). This means that processor utilization has been improved to 55 percent with a corresponding throughput of 33 jobs per hour.

But even this simple form of automatic scheduling creates new problems: How do we *protect the operating system* against erroneous user programs? How can we force user programs to *return control* to the operating system when they are finished, or if they fail to terminate after a

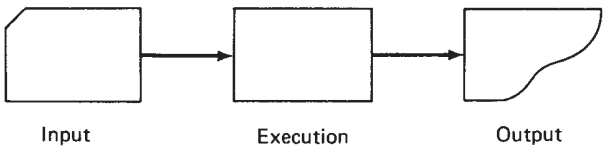

Fig. 1.1 Automatic scheduling of a job queue input directly from a card reader, executed, and output directly on a line printer by a single processor.

period of time defined by the operating system? Early operating systems offered no satisfactory solutions to these problems and were frequently brought down by their jobs. We will ignore this problem at the moment and return to it in the chapters on processor management and resource protection.

This argument in favor of automatic scheduling has ignored the processor time that is lost while the operator handles the peripheral devices: inserting paper in the printer, mounting tapes for larger jobs, and so forth. The argument has also ignored processor time that is wasted when the operator makes a mistake. But these factors are ignored throughout the chain of arguments and do not affect the *trend* towards better utilization of the processor.

The main weakness is that the argument did not include an evaluation of the amount of processor time used by the new component—the operating system. The reason for this omission is that an operating system carries out certain indispensable functions (input/output, scheduling, and accounting) which previously had to be done elsewhere in the installation by operators and users. The relevant factor here—the amount of processor time lost by an inefficient implementation of the operating system—unfortunately cannot be measured. But in any case, the figures given in the following are not my estimates, but measurements of the actual performance of some recent operating systems.

The bottleneck in the previous simple system is the slow input/output devices; they keep the central processor waiting 45 per cent of the time during an average job execution. So the next step is to use the fast tape stations to implement a *batch processing system* as shown in Fig. 1.2. First, a number of jobs are collected from users by an operator and copied from

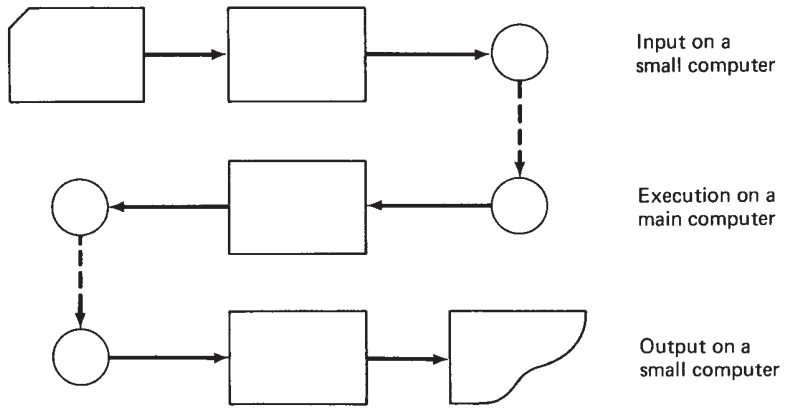

Fig. 1.2 Batch processing of jobs in three phases: input of cards to tape on a small computer; execution with tape input/output on a main computer; and output of tape to printer on a small computer.

cards to magnetic tape on a small, cheap computer. This tape is carried by the operator to the main computer, which executes the batch of jobs one by one, delivering their output to another tape. Finally, this output tape is carried to the small computer and listed on the printer. Notice that although jobs are executed in their order of arrival inside a batch, the printed output of the first job is not available until the entire batch has been executed.

During the execution of a batch on the main computer, the operator uses the small computer to print the output of an earlier batch and input a new batch on tape. In this way the main computer, as well as the card reader and printer, is kept busy all the time. Input/output delays on the main computer are negligible in this system, but another source of idle time has appeared: the mounting and dismounting of tapes. This can only be reduced by batching many jobs together on a single tape. But in doing so we also increase the waiting time of users for the results of their jobs. This dilemma between idle processor time and user response time can be expressed by the following relation:

$$\text{processor utilization} = \frac{\text{batch execution time}}{\text{batch response time}}$$

where

$$\text{batch response time} = \text{batch mounting time} + \text{batch execution time}$$

This can also be rewritten as follows:

$$\text{batch response time} = \frac{\text{batch mounting time}}{1 - \text{processor utilization}}$$

Since there is a limit to the amount of idle processor time management is prepared to accept, the net result is that response time for users is still determined by the manual speed of operators! In the installation considered a *batch cycle* typically proceeds as follows:

Delivery time of 50 jobs	30 min
Conversion of cards to tape	15 min
Mounting of tapes	5 min
Batch execution	50 min
Conversion of tape to printer	25 min
Manual separation of output	15 min
Total batch cycle	140 min

With a tape mounting time of 5 min per batch, utilization of the main processor is now as high as 50/55 = 90 per cent, and throughput has

reached 55 jobs per hour. But at the same time, the shortest response time for any job is 140 min. And this is obtained only if the job joins a batch immediately after submission.

We have also ignored the problem of the large jobs: When jobs requiring hours for execution are included in a batch, the jobs following will experience much longer response times. Most users are only interested in fast response during working hours. So an obvious remedy is to let the operators *sort jobs manually* and schedule the shorter ones during the daytime and the longer ones at night.

If the operator divides the jobs into three groups, the users might typically expect response times of the following order:

1-min jobs:	2–3 hours
5-min jobs:	8–10 hours
other jobs:	1–7 days

We have followed the rationale behind the classical batch-processing system of the late 1950's (Bratman, 1959). The main concern has been to reduce idle processor time, unfortunately with a resultant increase in user response time.

In this type of system the most complicated aspects of sharing are still handled by operators, for example, the scheduling of simultaneous input/output and program execution on two computers, and the assignment of priorities to user jobs. For this reason I have defined an operating system as a set of *manual and automatic procedures* that enable a group of people to *share* a computer installation *efficiently* (Section 1.1.1).

1.2.3. Spooling Systems

It is illuminating to review the technological restrictions that dictated the previous development towards batch processing. The first one was the strict sequential nature of the computer which made it necessary to prevent conversational interaction with running programs; the second limitation was the sequential nature of the backing store (magnetic tapes) which forced us to schedule large batches of jobs strictly in the order in which they were input to the system.

The sequential restrictions on scheduling were made much less severe (but were by no means removed) by technological developments in the early 1960's. The most important improvement was the design of *autonomous peripheral devices* which can carry out input/output operations independently while the central processor continues to execute programs.

The problem of synchronizing the central processor and the peripheral devices after the completion of input/output operations was solved by the

interrupt concept. An *interrupt* is a timing signal set by a peripheral device in a register connected to a central processor. It is examined by the central processor after the execution of each instruction. When an interrupt occurs, the central processor suspends the execution of its current program and starts another program—the operating system. When the operating system has responded properly to the device signal, it can either resume the execution of the interrupted program or start a more urgent program (for example, the one that was waiting for the input/output).

This technique made *concurrent operation* of a central processor and its peripheral devices possible. The programming technique used to control concurrent operation is called *multiprogramming.*

It was soon realized that the same technique could be used to simulate concurrent execution of several user programs on a single processor. Each program is allowed to execute for a certain period of time, say of the order of 0.1–1 sec. At the end of this interval a *timing device* interrupts the program and starts the operating system. This in turn selects another program, which now runs until a timing interrupt makes the system switch to a third program, and so forth.

This form of scheduling, in which a single resource (the central processor) is shared by several users, one at a time in rapid succession, is called *multiplexing.* Further improvements are made possible by enabling a program to ask the operating system to switch to other programs while it waits for input/output.

The possibility of more than one program being in a state of execution at one time has considerable influence on the organization of storage. It is no longer possible to predict in which part of the internal store a program will be placed for execution. So there is no fixed correspondence at compile time between the names used in a program to refer to data and the addresses of their store locations during execution. This problem of *program relocation* was first solved by means of a *loading program*, which examined user programs before execution and modified addresses used in them to correspond to the store locations actually used.

Later, program relocation was included in the logic of the central processor: a *base register* was used to modify instruction addresses automatically during execution by the start address of the storage area assigned to a user. Part of the *protection problem* was solved by extending this scheme with a *limit register* defining the size of the address space available to a user; any attempt to refer to data or programs outside this space would be *trapped* by the central processor and cause the operating system to be activated.

The protection offered by base and limit registers was, of course, illusory as long as user programs could modify these registers. The recognition of this flaw led to the design of central processors with two states of execution: a *privileged state*, in which there are no restrictions on

the operations executed; and a *user state*, in which the execution of operations controlling interruption, input/output, and store allocation is forbidden and will be trapped if attempted. A transition to the privileged state is caused by interrupts from peripherals and by protection violations inside user programs. A transition to the user state is caused by execution of a privileged operation.

It is now recognized that it is desirable to be able to distinguish in a more flexible manner between many levels of protection (and not just two). This early protection system is safe, but it assigns more responsibility to the operating system than necessary. The operating system must, for example, contain code which can start input/output on every type of device because no other program is allowed to do that. But actually, all that is needed in this case is a mechanism which ensures that a job only operate devices assigned to it; whether a job handles *its own devices* correctly or not is irrelevant to the operating system. So a centralized protection scheme tends to increase the complexity of an operating system and make it a bottleneck at run time. Nevertheless, this early protection scheme must be recognized as an invaluable improvement: clearly, the more responsibility management delegates to an operating system, the less they can tolerate that it breaks down.

Another major innovation of this period was the construction of *large backing stores*, disks and drums, which permit fast, *direct access* to data and programs. This, in combination with multiprogramming, makes it possible to build operating systems which handle a continuous stream of input, computation, and output on a single computer. Figure 1.3 shows the organization of such a *spooling system*. The central processor is multiplexed between four programs: one controls input of cards to a queue on the backing store; another selects user jobs from this input queue and

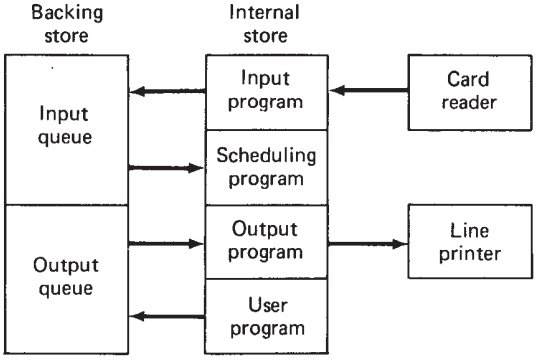

Fig. 1.3 A spooling system controlling continuous buffering of input/output on backing storage and sequential scheduling of user jobs.

starts their execution one at a time; and a third one controls printing of output from the backing store. These three programs form the operating system. The fourth program held in the internal store is the current user program which reads its data from the input queue and writes its results in an output queue on the backing store.

The point of using the backing store as a *buffer* is that the input of a job can be fed into the machine in advance of its execution; and its output can be printed during the execution of later jobs. This eliminates the manual overhead of tape mounting. At the same time, direct access to the backing store makes it possible to schedule jobs in order of *priority* rather than in order of arrival. The spooling technique was pioneered on the *Atlas* computer at Manchester University (Kilburn, 1961).

A very successful operating system with input/output spooling, called *Exec II*, was designed by Computer Sciences Corporation (Lynch, 1967 and 1971). It controlled a *Univac 1107* computer with an instruction execution time of 4 μsec. The backing store consisted of two or more fast drums, each capable of transferring 10,000 characters during a single revolution of 33 msec. The system typically processed 800 jobs per day, each job requiring an average of 1.2 min. It was operated by the users themselves: To run a job, a user simply placed his cards in a reader and pushed a button. As a rule, the system could serve the users faster than they could load cards. So a user could immediately remove his cards from the reader and proceed to a printer where his output would appear shortly.

Less than 5 per cent of the jobs required magnetic tapes. The users who needed tapes had to mount them in advance of program execution.

Fast response to student jobs was achieved by using the scheduling algorithm *shortest job next.* Priorities were based on estimates of execution time supplied by users, but jobs that exceeded their estimated time limits were terminated by force.

Response times were so short that the user could observe an error, repunch a few cards, and resubmit his job immediately. System performance was measured in terms of the *circulation time* of jobs. This was defined as the sum of the response time of a job after its submission and the time required by the user to interpret the results, correct the cards, and resubmit the job; or, to put it more directly, the circulation time was the interval between two successive arrivals of the same job for execution.

About a third of all jobs had a circulation time of less than 5 min, and 90 per cent of all jobs were recirculated ones that had already been run one or more times the same day. This is a remarkable achievement compared to the earlier batch-processing system in which small jobs took a few hours to complete! At the same time, the processor utilization in the *Exec II* system was as high as 90 per cent.

The *Exec II* system has demonstrated that many users do not need a direct, conversational interaction with programs during execution. Users

will often be quite satisfied with a non-interactive system which offers them informal access, fast response, and minimal cost.

Non-interactive scheduling of small jobs with fast response is particularly valuable for *program testing.* Program tests are usually short in duration: After a few seconds the output becomes meaningless due to a programming error. The main thing for the programmer is to get the initial output as soon as possible and to be able to run another test after correcting the errors shown by the previous test.

There are, however, also cases in which the possibility of interacting with running programs is highly desirable. In the following section, I describe operating systems which permit this.

1.2.4. Interactive Systems

To make direct conversation with running programs tolerable to human beings, the computer must respond to requests within a few seconds. As an experiment, try to ask a friend a series of simple questions and tell him to wait ten seconds before answering each of them; I am sure you will agree that this form of communication is not well-suited to the human temperament.

A computer can only respond to many users in a few seconds when the processing time of each request is very small. So the use of multiprogramming for conversation is basically a means of giving fast response to *trivial requests*; for example, in the editing of programs, in ticket reservation systems, in teaching programs, and so forth. These are all situations in which the pace is limited by human thinking. They involve very moderate amounts of input/output data which can be handled by low-speed terminals such as typewriters or displays.

In interactive systems in which the processor time per request is only a few hundred milliseconds, scheduling cannot be based on reliable user estimates of service time. This uncertainty forces the scheduler to allocate processor time in small slices. The simplest rule is *round-robin* scheduling: each job in turn is given a fixed amount of processor time called a *time slice*; if a job is not completed at the end of its time slice, it is interrupted and returned to the end of a queue to wait for another time slice. New jobs are placed at the end of the queue. This policy guarantees fast response to user requests that can be processed within a single time slice.

Conversational access in this sense was first proposed by Strachey (1959). The creative advantages of a closer interaction between man and machine were pointed out a few years later by Licklider and Clark (1962). The earliest operational systems were the *CTSS* system developed at Massachusetts Institute of Technology and the *SDC Q-32* system built by the System Development Corporation. They are described in excellent papers by Corbato (1962) and Schwartz (1964 and 1967).

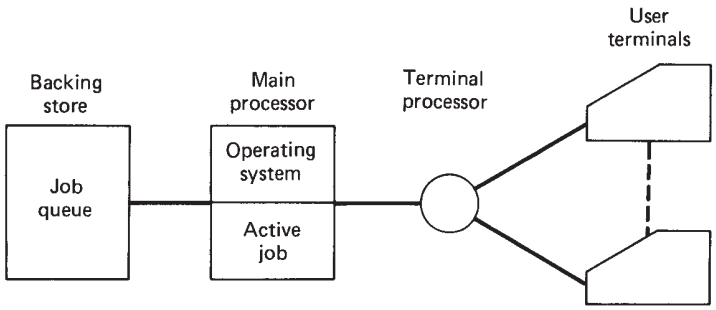

Fig. 1.4 An interactive system with low-speed user terminals and swapping of jobs between internal and backing storage.

Figure 1.4 illustrates the *SDC Q-32* system in a simplified form, ignoring certain details. An internal store of 65 K words is divided between a resident operating system and a single active user job; the rest of the jobs are kept on a drum with a capacity of 400 K words.

The average time slice is about 40 msec. At the end of this interval, the active job is transferred to the drum and another job is loaded into the internal store. This exchange of jobs between two levels of storage is called *swapping*. It takes roughly another 40 msec. During the swapping, the central processor is idle so it is never utilized more than 50 per cent of the time.

During the daytime the system is normally accessed simultaneously by about 25 user terminals. So a user can expect response to a simple request (requiring a single time slice only) in 25*80 msec = 2 sec.

A small computer controls terminal input/output and ensures that users can continue typing requests and receiving replies while their jobs are waiting for more time. A disk of 4000 K words with an average access time of 225 msec is used for semi-permanent storage of data files and programs.

The users communicate with the operating system in a simple job control language with the following instructions:

LOGIN: The user identifies himself and begins using the system.

LOAD: The user requests the transfer of a program from disk to drum.

START: The user starts the execution of a loaded program or resumes the execution of a stopped program.

STOP: The user stops the execution of a program temporarily.

DIAL: The user communicates with other users or with system operators.

LOGOUT: The user terminates his use of the system.

The system has been improved over the years: Processor utilization has been increased from 50 to 80 per cent by the use of a more complicated scheduling algorithm. Nevertheless, it is still true that this system and similar ones are forced to spend processor time on unproductive transfers of jobs between two levels of storage: Interactive systems achieve *guaranteed response* to short requests at the price of decreased processor utilization.

In the *SDC Q-32* system with an average of 25 active users, each user is slowed down by a factor of 50. Consequently, a one-minute job takes about 50 min to complete compared to the few minutes required in the *Exec II* system. So interactive scheduling only makes sense for trivial requests; it is not a realistic method for computational jobs that run for minutes and hours.

Later and more ambitious projects are the *MULTICS* system (Corbato, 1965) also developed at MIT, and the IBM system *TSS-360* (Alexander, 1968). In these systems, the problem of *store multiplexing* among several users is solved by a more refined method: Programs and data are transferred between two levels of storage in smaller units, called *pages*, when they are actually needed during the execution of jobs. The argument is that this is less wasteful in terms of processor time than the crude method of swapping entire programs. But experience has shown that this argument is not always valid because the overhead of starting and completing transfers increases when it is done in smaller portions. We will look at this problem in more detail in the chapter on store management. The paging concept was originally invented for the *Atlas* computer (Kilburn, 1962).

Another significant contribution of these systems is the use of large disks for semi-permanent storage of data and programs. A major problem in such a *filing system* is to ensure the *integrity* of data in spite of occasional hardware failures and *protect* them against unauthorized usage. The integrity problem can be solved by *periodic copying* of data from disk to magnetic tape. These tapes enable an installation to restore data on the disk after a hardware failure. This is a more complicated example of the design goal mentioned in Section 1.1.2: An operating system should try to make the virtual machine appear more reliable than the real one.

The protection problem can be solved by a *password* scheme which enables users to identify themselves and by maintaining as part of the filing system a *directory* describing the authority of users (Fraser, 1971).

Interactive systems can also be designed for *real-time control* of industrial processes. The central problem in a real-time environment is that the computer must be able to receive data as fast as they arrive; otherwise, they will be lost. (In that sense, a conversational system also works under real-time constraints: It must receive input as fast as users type it.)

Usually, a process control system consists of several programs which are executed simultaneously. There may, for example, be a program which is started every minute to measure various temperatures, pressures, and flows

and compare these against preset alarm limits. Each time an alarm condition is detected, another program is started to report it to an operator, and while this is being done, the scan for further alarms continues. Still other programs may at the same time accumulate measurements on the production and consumption of materials and energy. And every few hours these data are probably printed by yet another program as a report to plant management.

This concludes the overview of the technological background of shared computer installations. I have tried through a series of simple arguments to illustrate the influence of *technological constraints* on the service offered by operating systems. Perhaps the most valid conclusion is this: In spite of the ability of present computer installations to perform some operations simultaneously, they remain basically *sequential* in nature; a central processor can only execute one operation at a time, and a drum or disk can only transfer one block of data at a time. There are computers and backing stores which can carry out more than one operation at a time, but never to the extent where the realistic designer of operating systems can afford to forget completely about sequential resource constraints.

1.3. THE SIMILARITIES OF OPERATING SYSTEMS

The previous discussion may have left the impression that there are basic differences between batch processing, spooling, and interactive systems. This is certainly true as long as we are interested mainly in the relation between the user service and the underlying technology. But to gain a deeper insight into the nature of operating systems, we must look for their similarities before we stress their differences.

To mention one example: All shared computer installations must handle concurrent activities at some level. Even if a system only schedules one job at a time, users can still make their requests simultaneously. This is a real-time situation in which data (requests) must be received when they arrive. The problem can, of course, be solved by the users themselves (by forming a waiting line) and by the operators (by writing down requests on paper); but the observation is important since our goal is to handle the problems of sharing automatically.

It is also instructive to compare the batch-processing and spooling systems. Both achieve high efficiency by means of a small number of concurrent activities: In the batch processing system, independent processors work together; in the spooling system, a single processor switches among independent programs. Furthermore, both systems use backing storage (tape and drum) as a buffer to compensate for speed variations of the producers and consumers of data.

As another example, consider real-time systems for process control and conversational programming. In these systems, concurrently executed

programs must be able to exchange data to cooperate on common tasks. But again, this problem exists in all shared computer installations: In a spooling system, user computations exchange data with concurrent input/output processes; and in a batch processing system, another set of concurrent processes exchanges data by means of tapes mounted by operators.

As you see, all operating systems face a common set of problems. To recognize these, we must reject the established classification of operating systems (into batch processing, spooling, and interactive systems) which stresses the dissimilarities of various forms of technology and user service. This does not mean that the problems of adjusting an operating system to the constraints of a particular environment should be ignored. But the designer will solve them much more easily when he fully understands the principles common to all operating systems.

1.4. DESIGN OBJECTIVES

The key to success in programming is to have a realistic, clearly-defined goal and use the simplest possible methods to achieve it. Several operating systems have failed because their designers started with very vague or overambitious goals. In the following discussion, I will describe two quite opposite views on the overall objectives of operating systems.

1.4.1. Special-purpose Systems

The operating systems described so far have one thing in common: Each of them tries to use the available resources in the most simple and efficient manner to give a restricted, but useful form of computational service.

The *Exec II* spooling system, for example, strikes a very careful balance between the requirements of fast response and efficient utilization. The overhead of processor and store multiplexing is kept low by executing jobs one at a time. But with only one job running, processor time is lost when a job waits for input/output. So to reduce input/output delays, data are buffered on a fast drum. But since a drum has a small capacity, it becomes essential to keep the volume of buffered data small. This again depends on the achievement of short response time for the following reason: the faster the jobs are completed, the less time they occupy space within the system.

The keys to the success of the *Exec II* are that the designers: (1) were aware of the *sequential nature* of the processor and the drum, and deliberately kept the degree of multiprogramming low; and (2) took advantage of their knowledge of the *expected workload*—a large number of

small programs executed without conversational interaction. In short, the *Exec II* is a simple, very efficient system that serves a special purpose.

The *SDC Q-32* serves a different special purpose: conversational programming. It sacrifices 20 per cent of its processor time to give response within a few seconds to 25 simultaneous users; its operating system occupies only 16 K words of the internal store. This too is a simple, successful system.

The two systems do not compete with each other. Each gives the users a special service in a very efficient manner. But the spooling system is useless for conversation, and so is the interactive system for serious computation.

On the other hand, neither system would be practical in an environment where many large programs run for hours each and where operators mount a thousand tapes daily. This may be the situation in a large atomic research center where physicists collect and process large volumes of experimental data. An efficient solution to this problem requires yet another operating system.

The design approach described here has been called design according to *performance specifications.* Its success strongly suggests that efficient sharing of a large installation requires a *range of operating systems*, each of which provides a special service in the most efficient and simple manner. Such an installation might, for example, use three different operating systems to offer:

(1) conversational editing and preparation of jobs;

(2) non-interactive scheduling of small jobs with fast response; and

(3) non-interactive scheduling of large jobs.

These services can be offered on different computers or at different times on the same computer. For the users, the main thing is that programs written in high-level languages can be processed directly by all three systems.

1.4.2. General-purpose Systems

An alternative method is to make a single operating system which offers a variety of services on a whole range of computers. This approach has often been taken by computer manufacturers.

The *OS/360* for the *IBM 360* computer family was based on this philosophy. Mealy (1966) described it as follows:

"Because the basic structure of *OS/360* is equally applicable to

batched-job and real-time applications, it may be viewed as one of the first instances of a second-generation operating system. The new objective of such a system is to accommodate an environment of diverse applications and operating modes. Although not to be discounted in importance, various other objectives are not new—they have been recognized to some degree in prior systems. Foremost among these secondary objectives are:

- Increased throughput
- Lowered response time
- Increased programmer productivity
- Adaptability (of programs to changing resources)
- Expandability

"A second-generation operating system must be geared to change and diversity. *System/360* itself can exist in an almost unlimited variety of machine configurations."

Notice that performance (throughput and response) is considered to be of secondary importance to functional scope.

An operating system that tries to be all things to all men naturally becomes very large. *OS/360* is more than an operating system—it is a library of compilers, utility programs, and resource management programs. It contains several million instructions. Nash gave the following figures for the resource management components of *OS/360* in 1966:

Data management	58.6 K statements
Scheduler	45.0
Supervisor	26.0
Utilities	53.0
Linkage editor	12.3
Testran	20.4
System generator	4.4
	219.7 K

(Nato report, 1968, page 67).

Because of its size, the *OS/360* is also quite unreliable. To cite Hopkins: "We face a fantastic problem in big systems. For instance, in *OS/360* we have about 1000 errors each release and this number seems to be reasonably constant" (Nato report, 1969, page 20).

This is actually a very low percentage of errors considering the size of the system.

This method has been called design according to *functional specifications.* The results have been generally disappointing, and the reason is simply this: Resource sharing is the main purpose of an operating system, and resources are shared most efficiently when the designer takes full advantage of his knowledge of the *special characteristics* of the resources and the jobs using them. This advantage is immediately denied him by requiring that an operating system must work in a much more general case.

This concludes the overview of operating systems. In the following chapters, operating systems are studied at a detailed level in an attempt to build a sound theoretical understanding of the general principles of multiprogramming and resource sharing.

1.5. LITERATURE

This chapter owes much to a survey by Rosin (1969) of the technological development of operating systems.

With the background presented, you will easily follow the arguments in the excellent papers on the early operating systems mentioned: *Atlas* (Kilburn, 1961; Morris, 1967), *Exec II* (Lynch, 1967 and 1971), *CTSS* (Corbato, 1962), and *SDC Q-32* (Schwartz, 1964 and 1967). I recommend that you study these papers to become more familiar with the purpose of operating systems before you proceed with the analysis of their fundamentals. The *Atlas*, *Exec II*, and *SDC* systems are of special interest because they were critically reevaluated after several years of actual use.

The paper by Fraser (1971) explains in some detail the practical problems of maintaining the integrity of data in a disk filing system in spite of occasional hardware malfunction and of protecting these data against unauthorized usage.

CORBATO, F. J., MERWIN-DAGGETT, M., and DALEY, R. C., "An experimental time-sharing system," *Proc. AFIPS Fall Joint Computer Conf.*, pp. 335-44, May *1962.*

FRASER, A. G., "The integrity of a disc based file system," *International Seminar on Operating System Techniques*, Belfast, Northern Ireland, Aug.-Sept. *1971.*

KILBURN, T., HOWARTH, D. J., PAYNE, R. B., and SUMNER, F. H., "The Manchester University Atlas operating system. Part I: Internal organization," *Computer Journal 4*, 3, 222-25, Oct. *1961.*

LYNCH, W. C. "Description of a high capacity fast turnaround university computing center," *Proc. ACM National Meeting*, pp. 273-88, Aug. *1967.*

LYNCH, W. C., "An operating system design for the computer utility environment," *International Seminar on Operating System Techniques*, Belfast, Northern Ireland, Aug.-Sept. *1971.*

MORRIS, D., SUMNER, F. H., and WYLD, M. T., "An appraisal of the Atlas supervisor," *Proc. ACM National Meeting*, pp. 67-75, Aug. *1967.*

ROSIN, R. F., "Supervisory and monitor systems," *Computing Surveys 1*, 1, pp. 15-32, March *1969.*

SCHWARTZ, J. I., COFFMAN, E. G., and WEISSMAN, C., "A general purpose time-sharing system," *Proc. AFIPS Spring Joint Computer Conf.*, pp. 397-411, April *1964.*

SCHWARTZ, J. I. and WEISSMAN, C., "The SDC time-sharing system revisited," *Proc. ACM National Meeting*, pp. 263-71, Aug. *1967.*