1 Introduction to Input/Output Systems

The I/O package was the seed pellet injected into the gathering cloud of ingenuity hovering over the computer industry. This was in 1958 or thereabouts. . . . Leo J. Cohen

An I/O (input/output) system is the programmed component of an operating system that stores and retrieves data. I/O systems are the spectacular achievement of a few dozens of people working in groups separated geographically and administratively. Surprisingly, the philosophies and techniques developed by these groups are quite similar, similar enough to constitute an emerging I/O system science. This book is devoted to that new and fascinating science, but before we plunge into that main study, courtesy requires that we give some attention to history. Beyond that, good judgment requires attention to the operating system environment within which an I/O system must perform.

Computer systems are a new phenomenon. The industry points with patronizing pride to Charles Babbage, whose machines and ideas in the nineteenth century were far ahead of his time.[1] But modern computer systems all use the stored program idea, and stored program machines date only from World War II. Many of the outstanding senior people in the computer industry today were already well into their professional careers when the first stored program computers were being built. I/O systems are much newer still, the most ancient of them dating from 1958 or thereabouts.

Before 1950, calculating machines were much less flexible than modern computers. Remington Rand, a parent of the present Sperry Rand Corporation, had a line of punched card handling products whose activities could be controlled somewhat by positioning mechanical levers. IBM had card machines whose actions could be altered by control panel wiring. An important forerunner of the modern computer was the Card Programmed Calculator (CPC), a system composed of several of IBM's card machines interconnected by electrical cables. As the name implies, the CPC was programmed by a deck of cards. Each complete pass of the cards might cause computation of one paycheck or one stage of a numerical approximation process. The instructions in the cards were interpreted by control panel wiring.

In the late 1940s John von Neumann authored several important papers describing a stored program computer, a computer where instructions would

[1] Neither high precision machining, electronics, nor an appreciative society assisted Professor Babbage in his magnificent efforts. The serious student of computer history will be fascinated by Charles' dogged persistence and by the unselfish support of his gifted admirer, the Countess of Lovelace, Lord Byron's daughter. The foreword, preface, and appendix of *Faster than Thought* by B. V. Bowden (1953) are recommended reading.

be stored in the "memory organ" along with other more conventional data. Von Neumann's papers were sometimes authored jointly with two associates, Burke and Goldstein. However, von Neumann was clearly the motivating force, and he is generally acclaimed the father of the stored program computer. Von Neumann's original papers can be found in the *Collected Works of John von Neumann,* volume 5, published after his untimely death.

HOW I/O SYSTEMS BEGAN

During the early 1950s, several stored program computers were introduced commercially. The names of those machines, including the Univac 1101, the IBM Type 650, the Electrodata 204, the Bendix G-15, and the ALWAC-III, may be unfamiliar to some readers. Commercial availability created an immediate demand for generally useful programs so that standard procedures need not be reprogrammed and retested.

The requirement for a generally available I/O program can be illustrated for the IBM defense calculator, later named the Type 701. For its day, the Type 701 was very fast, and like many of the early scientific computers, its I/O capabilities were quite rudimentary. Reading a punched card required execution of 24 COPY instructions, each resulting in the transfer of 36 binary digits of information to main storage. If the card contained decimal information, the newly stored information required sorting. For example, the 12 bits representing any holes punched in card column 1 would be found as the first bit transferred by each of the 1st, 3rd, 5th, . . . , 23rd COPY instructions. All of this had to be accomplished without the aid of index registers.

Beginning with these very first commercially available stored program computers, the manufacturers recognized that a few general utility programs would be required. Some of the first general utility programs furnished were the bootstrap loaders used to load a program and start its execution. Other programs gradually made available could duplicate a card deck, create a magnetic tape copy of a card deck, print a report from magnetic tape, read an input deck converting the data from decimal to binary, and so on. Such programs were either furnished in card deck form or printed in a computer operating manual to be punched by the user. Typically, a small stack of these utility programs sat near the card reader ready for use by one and all. When a card was lost or mutilated, or when decks became hopelessly intermixed, fresh copies were readily available from a filing cabinet close at hand.

One limitation of these early utility programs was that they were intended for independent use only, not for incorporation with the user's payroll or matrix inversion program. By the mid-1950s, general purpose I/O subroutines were available for most computers. These I/O subroutines were typically in the form of card decks that could be added to a user's program deck. The resulting deck, together with a bootstrap loader, was a complete and independent program ready for use whenever the computer was available.

Concurrent with the development of standard utilities and subroutines was the development of interpretive systems. An interpreter is a supervisory program that executes the instructions specified in another program; the latter is never executed directly. An early successful interpretive system was the "Speed Coding System" developed under the direction of John Backus, a famous name in computer science. The "Bell Interpretive System" developed by Bell Telephone Laboratories for the IBM Type 650 was very popular. Interpretive systems were popular for at least two important reasons:

1. They allow the users to specify their programs in a convenient language not closely related to the instruction set of the computer.
2. They retain a great deal of control over the users' programs, thereby simplifying problems such as user program testing.

The principal disadvantages were that interpreters preempted a significant portion of the very limited main storage space and that they tended to be slow. A good discussion can be found in Donald E. Knuth's book *The Art of Computer Programming,* volume 1.

The advent of a successful FORTRAN compiler in the late 1950s redirected operating systems by furnishing an even more convenient language for the user's program and by providing good execution speed for the compiled program without loss of space to the interpreter. One of the earliest noninterpretive systems was developed at the Rocketdyne Division of North American Aviation in California in about 1958. That system was called a "FORTRAN Compile and Go" system. It could accept a number of users' programs in the FORTRAN language, compile them, and execute them without operator intervention.

The Share Operating System (SOS) developed jointly by IBM and the organization of computer users called SHARE included I/O routines that were part of a monitor that remained in the computer. The term *input/output control system* (*IOCS*) was coined about that time and by the early 1960s IOCS was becoming a part of everyone's operating system, in theory if not in fact. At about that same time, the rapid development of a variety of I/O devices, the advent of independent channels, and the popularity of multiprogramming combined to compel the development of the current features of I/O systems.

OPERATING SYSTEMS, WHAT THEY DO AND WHY

An I/O system is a component of an operating system, typically the largest and most fascinating component. Just as a study of the human heart might begin with a survey of the entire circulatory system, our study of the I/O system begins with the entire operating system.

An operating system is a large and complex collection of computer instructions. A system might consist of from 50,000 to 500,000 instructions, and it may be so complex that no one person understands it completely. The largest operating systems are possibly the most complex products ever produced by humans, rivaling such developments as the Apollo missile systems.

Why We Need an Operating System

Nearly all general purpose use of medium and large computers is accomplished under operating system control. The major services furnished by an operating system will be described in just a moment. But first, it is useful to understand why an operating system is necessary. What needs justify a collection of programs so large and complex? When a user has a job for the computer, why doesn't he or she use the computer directly without an operating system? Several factors furnish the answers to these questions.

Most computer systems, as manufactured, presuppose operating systems. As it became clear that programs such as the IOCS mentioned earlier were inevitable, computer designers began to capitalize on their existence. Wherever programs could perform more flexibly or at less cost than electrical circuits, the operating system accepted a new responsibility. Yesterday's luxury has become today's necessity. All of the major manufacturers of computer systems furnish operating systems with their computers.

Operating systems improve operating efficiency in several ways:

- An operating system can overlap the setup time for one job with the execution of other jobs. The setup time required to ready the auxiliary storage units for a particular job often exceeds the execution time for the job. Without this overlap, the routine feat of running several hundred jobs a day on a single computer would be impossible.
- An operating system can arrange concurrent processing. For example, one job might require only one or two magnetic tape units, while another requires little use of direct access storage devices, and yet another requires very little main storage space. An operating system can arrange for several such jobs to be in process concurrently, thereby accomplishing more work in a given interval of time.
- An operating system can reduce the length of time that equipment is required by a job. For example, operating systems frequently use high-performance auxiliary storage devices as substitutes for card readers, printers, and other relatively slow devices. This substitution, accomplished without concern to the user, allows the user's program to perform its function more rapidly, thereby freeing the equipment devoted to that program sooner. At an unrelated time, the operating system will remove the information from its temporary storage and accomplish the printing or punching required by the user.
- An operating system can reduce the elapsed time from receipt of data to printing of results. Before the advent of operating systems, job processing usually included several peripheral operations before and after computer processing. The peripheral processes included activities such as collecting similar jobs onto a single magnetic tape and printing of final results from a computer output tape. Each peripheral operation involved clerical handling of data related to the service request. Not only were requests occasionally processed incorrectly, but also the peripheral machines were often backlogged for days at a time. By integrating the peripheral operations into the computing process, an operating system reduces the elapsed time for the combined operations.

Operating systems afford a useful combination of flexibility and standardization:

- The standardization of data formats imposed by operating systems provides greater interchangeability of data between computer systems and wider usefulness of programs.
- The standardization of procedures imposed by operating systems reduces operator errors.
- An operating system separates the user's program from the computer system in such a way that the latter can be expanded or contracted by addition or deletion of devices without modification to the user's program.
- An operating system separates the user's programs from the computer system in such a way that new, improved devices frequently can be substituted for older, less effective devices without modification to the user's programs.

An operating system can adapt a general purpose computer to any of several operating requirements. For example, an operating system can provide the responsiveness necessary for a communications based system, the reliability required for a missile guidance system, the efficiency of a conventional commercial data processor, or a combination of these attributes.

In summary, an operating system is used because it is profitable to use it, profitable because of the efficiency it produces, the combination of flexibility and standardization it imposes, and its ability to adapt a computer to special operating requirements such as responsiveness.

When one first encounters an operating system, one may feel uneasy because the system is not of real substance; it consists of computer instructions subject to change without notice. The physical devices that compose a computer system appear more solid and durable. Operating systems are, in fact, very real and, though they are frequently extended to make them more useful, their advertised functions are seldom altered or deleted. The reasons for the stability are primarily economic. A major operating system represents a direct investment of millions of dollars, even tens of millions. If one adds the investment in education of users and in development of users' programs that depend on the environment created by an operating system, the total investment for a single operating system may exceed a billion dollars. An investment of such proportions has such a stabilizing influence on an operating system that several major operating systems in use today have outlived the physical devices for which they were originally designed.

Major Activities of an Operating System

The general purpose operating systems of today bear great resemblance to one another. Most of them perform the following major functions:

Scheduling Jobs An operating system accepts jobs and schedules the system resources to satisfy these jobs. Some systems simply schedule jobs in the order of their receipt, that is, first in, first out (FIFO). Others recognize a priority code furnished by the user as an expression of the urgency of the job. Job scheduling involves not only recognition of priorities, but also availability of necessary resources. The resources required for a particular job may not be immediately available because they are being used for another job, or because they are undergoing maintenance. A comprehensive operating system will defer running of jobs when possible within priority constraints to effect efficient use of resources.

Allocating Resources The resources of a computing system include main storage space, I/O devices, and files of data. An operating system controls the use of all of these resources. For example, an operating system allocates a particular direct access storage device (DASD) to be used in processing a particular storage volume. Both the DASD and the files recorded on the storage volume are considered system resources.

Dispatching Programs One very special resource that an operating system controls is central processing unit (CPU)[2] time. At the beginning of an interval of time to be used by a particular program, the operating system takes the necessary steps to start or restart CPU activity for that program. The process of preparing the system for executing a particular program and transferring control of the CPU to that program is called dispatching. The system dispatches programs in the same sense that a clerk might dispatch errand boys.

The list of operating system functions is not yet complete, but we should stop for a moment to reflect on the activities discussed so far: scheduling, allocating, and dispatching. The reader may feel that the distinction among these three requires hairsplitting because all three combined constitute the apparently artless reaction to a request for service. In a simple operating system, the three combined activities do constitute a modest process. For example, completion of one job might trigger the operating system to prepare for the next job by reading one or more cards from a particular card reader. The user's program might identify its I/O devices explicitly, thereby eliminating device allocation activity. And, if the user's program is simply to be started and allowed to run to completion, dispatching is trivial. But unlike a simple system, a comprehensive system may be inspecting a hundred or more service requests at any one time, analyzing priorities, allocating devices so that several programs can be executed simultaneously, and dispatching the programs for intervals of a few milliseconds at a time to achieve the best possible service. In such a system, each of the individual functions of scheduling, allocating, and dispatching is a major activity.

Resuming the list of operating system functions:

Communicating with the Operator Philosophically, the computer-system operator should serve only one purpose: to be the hands for the computer, doing those things that require the mobility and dexterity of a human. This goal was suggested by Doctor Frederick Brooks in about 1968. Practically, operating systems have been unable to achieve the desired level of autonomy, so operators still serve as overall supervisors. In this role, they cancel improper service requests, reassign priorities, or even stop the system entirely when it appears that the operating

[2] The terms *CPU* and *arithmetic unit* are nearly synonymous. The term *CPU* is used extensively in this book, but *arithmetic unit* is used where appropriate.

system has lost control. Examples of communications from the system to the operator include requesting the operator to mount or demount storage volumes, notifying the operator of start and completion of each job, and apprising the operator of any unusual conditions, such as a high frequency of I/O device errors.

Recovering from Incidents Computer systems, and particularly the I/O devices included within systems, are subject to a variety of unexpected (but not unanticipated) conditions. A comprehensive operating system must be prepared to deal with unexpected conditions at all times. Typical unexpected conditions include intermittent I/O device failure, permanent I/O device failure, operator error, improper action by a user's program, and intermittent main storage or CPU failure. The finesse with which an operating system deals with unexpected conditions is one very important measure of its value. A good system can diagnose many situations and recover with modest loss of work in process.

Recording of Statistics The recording of operating statistics is unproductive and time-consuming, but it is essential for distributing costs to users and for analyzing system performance. In comprehensive systems, noteworthy events occur at such a rate that even simple counting of events in main storage tables with occasional copying of the tables to auxiliary storage may require as much as 1 or 2 percent of all available CPU time.

Storing and Retrieving Data The I/O system, a major component of an operating system, is responsible for data storage and retrieval. Because the main body of this book dwells on I/O activity, no more will be said about it in this preliminary section.

Operating Modes

Four very important terms describe the general modes of operation of systems: (1) batch processing, (2) multiprogramming, (3) time sharing, and (4) multiprocessing. These terms will come up frequently during our discussion of I/O systems, so it is important to know what they mean.

Batch Processing. The term *batch processing* refers to that very general working method in which a collection (batch) of related transactions is processed through the computer system as a group. For example, in a parts inventory application, the receipts and withdrawals from inventory might be accumulated manually and, at the end of each day, all transactions that had been accumulated might be submitted as a batch for processing. As illustrated in Fig. 1-1, a batch processing program typically accepts a batch of transactions and a master file as input creating printed reports and a new master file as results. This kind of processing is widely used in processing business data.

Batch processing has several important advantages:

1. Batch processing allows flexibility in scheduling of work, because the submitter usually is not expecting results while he or she waits. Typically, batches may require service within an hour, within one-half day, or overnight.

2. Batch processing allows significant efficiencies in the use of system resources. For example, a batch of transactions can be presorted into an efficient processing order, and the master files stored on removable storage volumes can be used heavily for processing those transactions and then removed from the system.
3. A batch is a convenient unit for accounting for costs and for rerunning of work when necessary.

The roots of batch processing are deep in data processing history. Punched card data processing was first used on a large scale during the 1890 United States Census. The data processing machines developed and manufactured from that time until World War II were batch processing machines. They required manual setup; they accumulated batch totals; they sorted batches of cards; and they performed simple extension and cross-footing. (These last two terms may be unfamiliar to some readers, but they were not strange to the industry in 1940. For the sake of history, extending is the process of multiplying unit price by quantity, and cross-footing is the process of summing the totals that exist at the feet of a set of columns.)

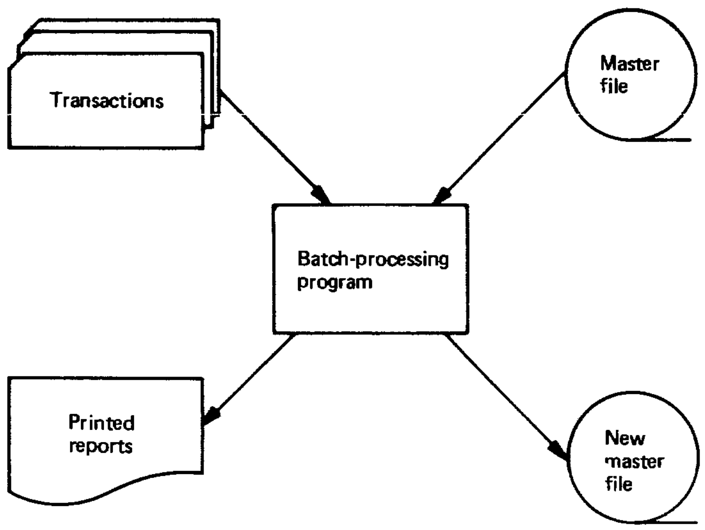

Fig. 1-1 Batch processing.

Much of the experimental work and much of the current literature about data processing concern individual transaction processing. But don't be confused by the difference between industry conversation and computer room practice. Most data processing accomplished by computers today is done in a batch processing mode. Using a simple operating system, operators can run batches one at a time. More comprehensive systems provide for batches to be multiprogrammed and/or multiprocessed as described below.

Multiprogramming. Multiprogramming is the executing of two or more programs concurrently, using a computer system with a single CPU. The critical word in the definition is the word "concurrently": "executing two or more programs *concurrently,* using a computing system with a single CPU." The CPU of a system performs arithmetic and logical instructions, and it is capable of executing only one instruction at a time. In multiprogramming, the CPU is controlled to interleave the processing of several programs.

It is sometimes remarked that multiprogramming does not really accomplish concurrent processing, but, rather, gives the impression of concurrent processing. The remark is superficial; the system may, in fact, be performing I/O activities for many programs simultaneously and may even be performing more than one such operation for each of them.

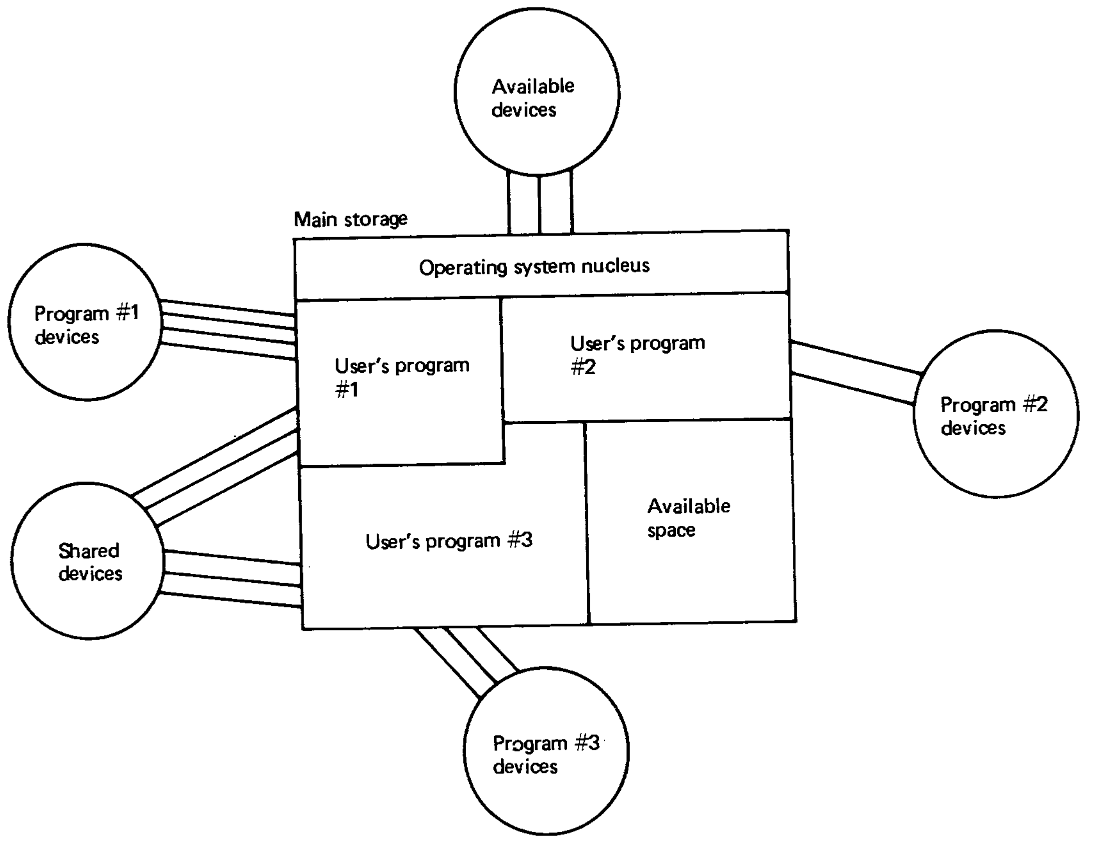

Fig. 1-2 Multiprogramming.

Figure 1-2 illustrates how computer facilities might be allocated to three programs for multiprogramming. In the figure, the main storage of the system is occupied by:

- The operating system nucleus which is that portion of the operating system that must be in main storage at all times to control moment-by-moment operations.
- The three programs that are to be executed concurrently.
- Available space (space not currently in use).

The devices in Fig. 1-2 have been allocated individually to the three users' programs except that some devices are shared by two or more programs and some devices are not in use.

The major justification for multiprogramming is efficiency. The opportunity for efficiency can be illustrated by two programs, A and B. Program A performs a great deal of computation using very little data, while program B performs minor computation affecting many records of a file. Program A might be selecting an optimum firing angle for a ballistic missile by repeatedly solving a set of simultaneous differential equations. Program B might be copying a file of parts inventory records. In many such cases, simultaneous execution of programs is quite efficient. Programs A and B might be executed simultaneously in a little more time than either program by itself. Contemporary multiprogrammed systems accomplish a less than optimum matching of simultaneous programs, yet they may achieve 35 percent or more improvement in total running time as compared to non-multiprogrammed systems.[3] (This improvement may be offset to some degree by the tendency for multiprogrammed systems to include more equipment than non-multiprogrammed systems. One has to pay something to reduce costs!)

Multiprogramming is a complex phenomenon. In 1955, Nathaniel Rochester described a trick called multiprogramming that was being used to increase computer productivity.[4] However, neither the required computer features nor the control-program design for generalized multiprogramming was understood until about 1960. The STRETCH computer, designed for the Los Alamos Laboratory of the Atomic Energy Commission and delivered in 1960, was probably the first computer designed for multiprogramming. The operating system for that computer included a limited form of multiprogramming. In 1962, Dr. Edgar F. Codd published the findings of his team concerning the practicality of multiprogramming.[5] He concluded correctly that multiprogramming was practical, and he predicted its widespread use. Today, a multiprogrammed operating system is available for every medium or large general purpose computer system. Some of the problems identified by these early researchers include efficient allocation of main storage, recoverability of damaged work, recreation of failure situations for testing purposes, and accountability adequate for customer billing. These four problems persist as today's major multiprogramming challenges.

Multiprocessing. Multiprocessing is the use of a computer system containing more than one CPU to satisfy a collection of service requests. Figures 1-3 and 1-4 illustrate the two principal variations, symmetric and asymmetric multiprocessing. In symmetric multiprocessing, the CPUs are used symmetrically and interchangeably to perform any type of processing required. In the asymmetric case, the processing burden is divided by type of activity, and each CPU is assigned one type of activity. Figure 1-4 illustrates one CPU performing all I/O activity while another executes all user's programs.

Multiprocessing might be used for any of several reasons:

- To improve system reliability by addition of a redundant CPU (symmetric case). In a properly designed symmetric system, complete

[3] A paper by Tom Steel entitled "Multiprogramming—Promise, Performance, and Prospect," *Proceedings of AFIPS,* 1968 Fall Joint Computer Conference, cites average improvements of 30 to 70 percent.

[4] Nathaniel Rochester described multiprogramming using a magnetic tape unit controller available for the IBM Type 705. His paper is entitled "The Computer and Its Peripheral Equipment," published in the *Proceedings of the Eastern Joint Computer Conference,* 1955.

[5] Dr. Codd's comprehensive article, published in *Advances in Computers,* volume 3, 1962, is the last of several related reports beginning in 1959.

failure of a CPU simply reduces total system capacity without affecting the kinds of services rendered.

- To segregate system functions, thereby simplifying design and development of the system (asymmetric case).
- To increase the total CPU power in a system (both cases).

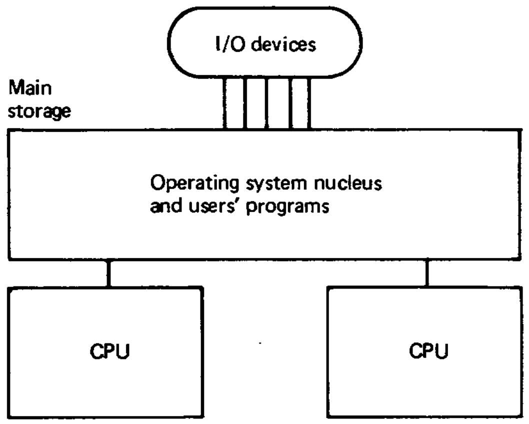

Fig. 1-3 Symmetric multiprocessing.

The asymmetric case has a load balancing problem; one CPU may be overburdened, while another has no work to perform. The specialization of function, though, does actually simplify the design of the operating system, and simplification tends to improve both performance and reliability. The CDC SCOPE system is a good example of asymmetric multiprocessing. In SCOPE, one CPU executes all of the users' programs, and several "peripheral processing units" perform I/O activities.

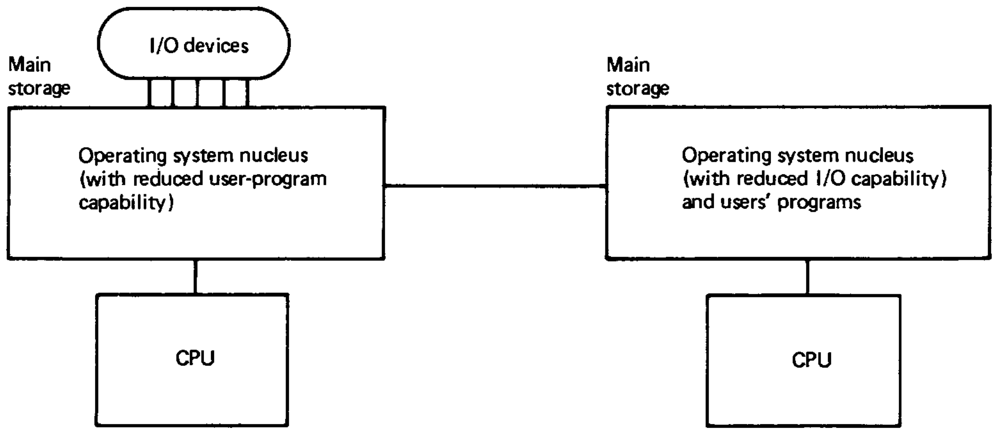
I/O devices
Main storage
Operating system nucleus (with reduced user-program capability)
CPU
Main storage
Operating system nucleus (with reduced I/O capability) and users' programs
CPU

Fig. 1-4 Asymmetric multiprocessing.

Multiprocessing systems existed in the late 1950s. The earliest systems included two essentially separate computer systems with one in a "standby" rather than an active status. Such systems had significant switch-over problems in the event of failure because the failing system was required to detect its own failure and participate in an orderly transfer of work. Today more than half of the major operating systems include multiprocessing capability. Because redundancy can provide a system that is rarely out of service, organizations such as airlines, the military establishment, and the National Aeronautics and Space Administration are very interested in multiprocessing.

Time Sharing. A time sharing mode of operation presupposes that each of a number of system users, usually from 10 to 100 or more, requires response within a few seconds to a series of simple service requests. Typically, time sharing users communicate with the computer system through keyboard terminals attached to the computer by telephone lines. Users may request services such as syntactic analysis of a programming statement, retrieval of a record from a file of data, execution of a mathematical procedure for finding a square root, or any of a variety of other services that individually make modest demands on system resources.

The primary objective in time sharing is to make the power of a centralized computing system available to several users simultaneously and to allow a cooperative relationship between user and system rather than the more conventional submit-a-request-and-wait-for-the-results relationship. Time sharing systems tend to accomplish less processing than other systems, but, in theory, their ready availability and their cooperative approach to problem solution improve the effectiveness of the users they serve.

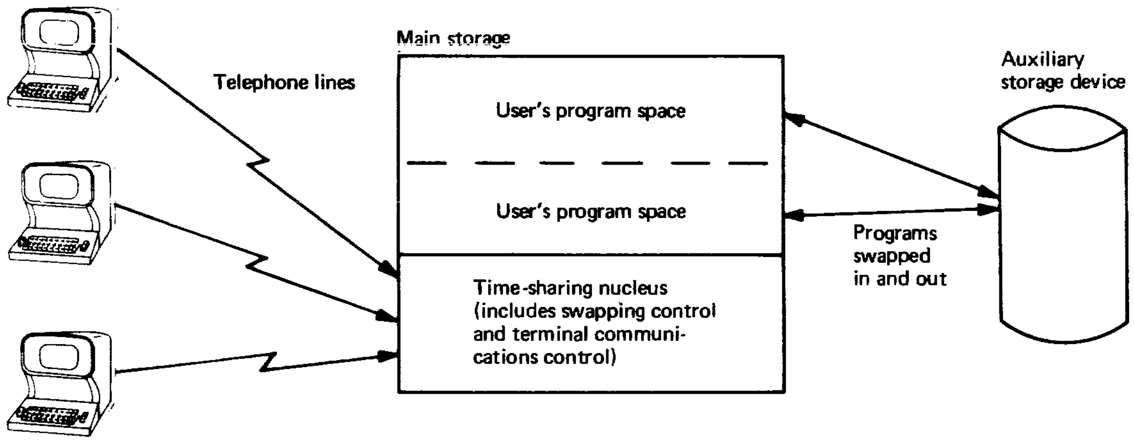
Main storage
Telephone lines
User's program space
User's program space
Time-sharing nucleus (includes swapping control and terminal communications control)
Auxiliary storage device
Programs swapped in and out

Fig. 1-5 A time sharing system.

In its simplest form, time sharing can be accomplished by a single program that performs a limited class of actions for several terminals. There are many such systems furnishing extended desk-calculator kinds of services. The more general time sharing systems use special operating system features such as a time slicing dispatcher and a swapping or paging mechanism. A time slicing dispatcher rotates use of the CPU to programs for fixed intervals of a few milliseconds each. A swapping or paging mechanism moves users' programs or pieces of users' programs called pages between main and auxiliary storage either on a preplanned or on a demand basis. Figure 1-5 illustrates a general time sharing system.

Time sharing was experimentally practiced in about 1961. A commercially usable system that offered several users time shared execution of FORTRAN-like requests was produced in 1964 by IBM under the direction of John Morrissey. In 1965, the General Electric Company produced a time