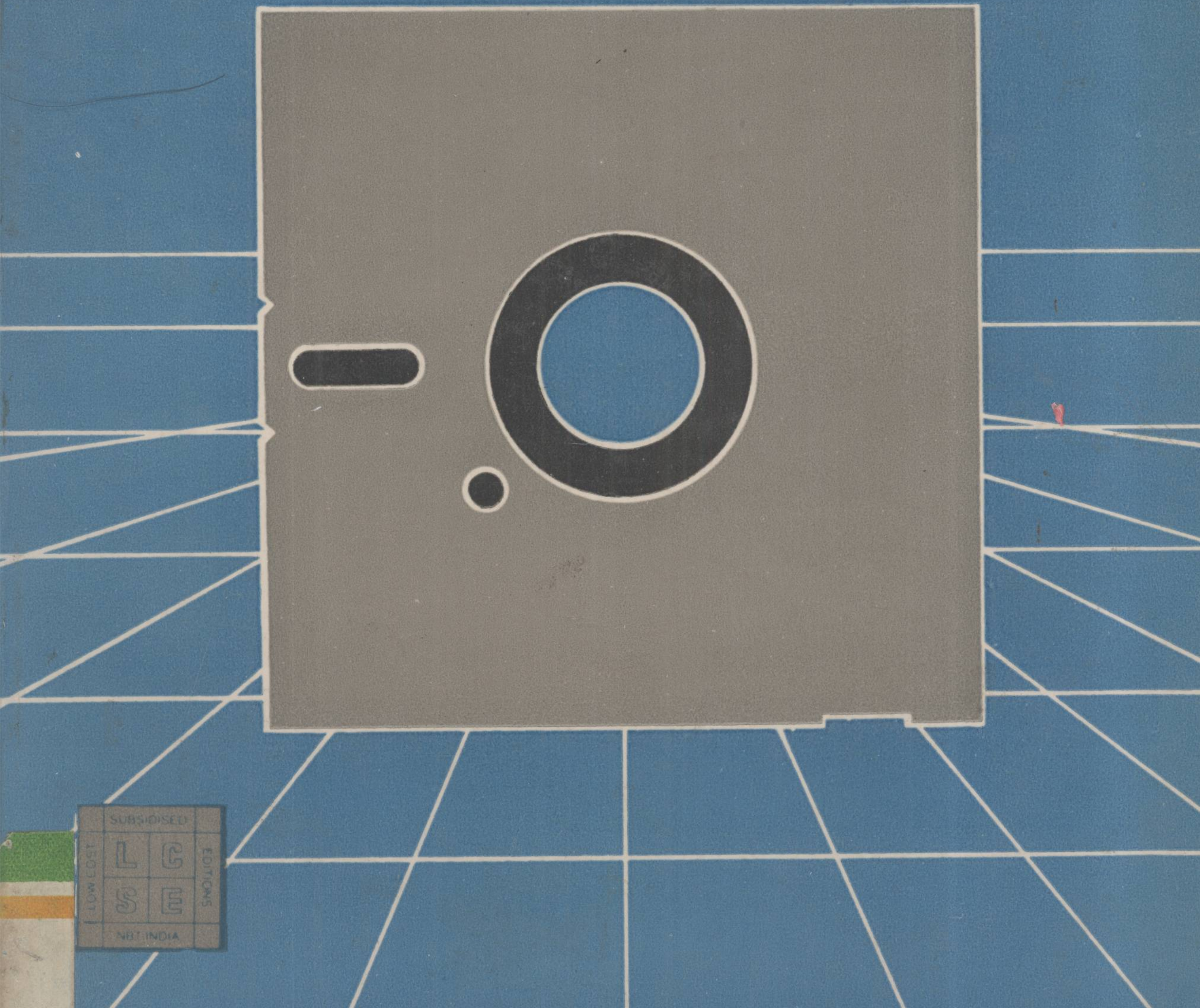
INTRODUCTION TO SYSTEM SOFTWARE

D M DHAMDHERE



Introduction to SYSTEM SOFTWARE

D M DHAMDHERE
Indian Institute of Technology
Bombay

Tata McGraw-Hill Publishing Company Limited
NEW DELHI

Publishing Manager:	RAJIV BERI
Production Editor:	RANJAN KAUL
Production Supervisor:	B L DOGRA
Manuscript Lead:	N SUBRAHMANYAM

This book has been subsidised by the Government of India, through the National Book Trust, India, for the benefit of students.

NBT Code: 45-240/1982
Rs 36.50

Published by Tata McGraw-Hill Publishing Company Limited, 12/4 Asaf Ali Road, New Delhi 110 002, typeset at Composers, E-8 Kalkaji, New Delhi and printed at Rajkamal Electric Press, 4163 Aryapura, Delhi 110 007

Introduction to SYSTEM SOFTWARE

McGraw-Hill Offices

New Delhi
New York
St Louis
San Francisco
Auckland
Bogotá
Guatemala
Hamburg
Lisbon
London
Madrid
Mexico
Montreal
Panama
Paris
San Juan
São Paulo
Singapore
Sydney
Tokyo
Toronto

Preface

Computer science education has undergone an important change in the past few years. With computers making inroads into newer fields, the requirements of the industry have changed from mere programming personnel to professionals capable of a systems approach to the design of computer applications. This requires a computer professional to possess a broad knowledge of computer science, with system software concepts occupying a very important place. The resulting two-tier nature of computer education, aimed at producing professionals and researchers respectively, projects new requirements of teaching and support material.

This book is intended as a text for introductory courses on aspects of system software at the undergraduate and postgraduate levels. The contents are designed to satisfy the requirements of the courses variously described as "Systems programming" (courses I4 of ACM curriculum 68 and CS-11 of AC, curriculum 77) and "Operating systems and computer architecture" (courses CS-6,7 of ACM curriculum 77 and SE-6,7 of IEEE curriculum 77), around which most universities have designed their courses. This book covers all the material considered essential to effectively support these courses, maintaining an optimum balance between the coverage of software processors and software tools on the one hand and computer architecture and operating systems on the other. This book has been motivated by the fact that most existing books fail to cover all aspects of contemporary system software adequately.

The book is organised into three parts with a common introduction. The Introduction creates the necessary background for the rest of the book by describing characteristics of system programs, their evolution and the basic terminology of the field. Part I (Chapters 1–5) of the book is devoted to the study of software processors. After a brief discussion of the basic translator schemes, separate chapters appear on assemblers, compilers, processors for the interactive environment and linkers/loaders. Part II is devoted to an in-depth study of operating systems. Identification of the basic tasks and components of an operating system in Chapter 6 is followed by Chapters 7–9 discussing the processor, storage and information management components of the operating system. Part III (Chapter 10) discusses the variety of software tools used in practice, and their design aspects.

Throughout the text, the approach used is in keeping with the introductory nature of the text. For every topic, preliminary material is introduced in simple words to create the necessary background. This is followed by a clear and detailed discussion of the basic design issues. Some practical case studies are included to give the reader a glimpse of the design considerations used in practice. Exercises included at appropriate points stimulate interest and motivate the reader to an in-depth analysis of practical situations and compromises. The bibliography points to more specialised material on these topics. While using this book as a text, it is expected that the instructor will assign group projects covering various aspects of system software to the participants. This will help consolidate the material covered.

Apart from use as a text, this book can be used in the professional computer environment as a book for ready reference or as a text for enhancement of skills. With the growing use of microcomputers in business and industry many middle and lower management personnel are turning into computer end-users. This book will provide them an opportunity for a comprehensive exposure to the area of system software.

The motivation for this book comes from the teaching of systems programming and operating system courses for a number of years. I thank all my students for contributing to the book in an indirect, but vital way. I also thank the Curriculum Development Cell of IIT Bombay for supporting the manuscript preparation.

D M DHAMDHERE

Contents

Introduction

I.1 What is System Software?

Most programmers using a contemporary computer system know the fundamental distinction between computer software and computer hardware. Many of them might also be aware that computer software can be further divided into 'system software' and 'applications software'. However, a question like "What identifies a program as a part of the system software?" (i.e., "What is a *system program?*" **may not elicit an answer readily.** This is because of the tendency **in manufacturer** supplied literature to describe the conglomerate form of system programs, namely system software, as if it were a single entity performing a collection of functions. Hence a programmer is unaware of the fact that despite their diversity of functions, all system programs do possess a few common characteristics. How can programs that edit a file, perform resource accounting or manage the main storage of a computer system have anything at all in common? To answer this question, we will first consider a simple definition of a system program that we wish to adopt as our working definition throughout this book. Later on we will trace the evolution of individual system programs to examine the motivation for their development and their place in the system software of a contemporary computer system.

A system program (SP) can be defined as a program "which helps an average computer user's program to execute effectively on a computer system." The phrase "average user" has been included in the definition just to exclude special purpose programs which fulfil specialised user needs. Such special programs would be said to constitute the applications software of the computer system. A system program is thus one which is required for the effective execution of a general user program. Note that the term "execution" includes all activities concerned with the initial input of the program text and various stages of its processing by the computer system, namely, editing, storage, translation, relocation, linking and eventual execution.

I.1.1 System Programs and System Programming

System programming can be defined as the activity of designing and implementing SPs. A question that naturally arises is: In what manner does system programming differ from any other kind of programming activity? To answer this question, we first pose a related question: In what manner does an SP differ from any other kind of program, say a scientific program, a data processing program, or an applications program?

In order to answer this question, let us once again look at the definition of an SP. An SP helps to execute a general user program effectively on a computer system. What do we mean by the word 'effectively'? Do we mean 'effectively' in terms of computer time, in terms of the programmer time spent in designing and writing a program, or do we mean something else altogether? Actually we mean "effectiveness" of the entire process of program development and program execution. In other words, this effectiveness is a delicate balance between the (generally conflicting) aspects of (i) effectiveness of utilisation of the computer resources and (ii) the effectiveness of utilisation of the human resources involved in the development of a program. Whatever be the subjective criteria for weighing the human and computer resources against each other, an SP is concerned with the overall optimisation of costs.

Now, would the balance of computer and human costs vary from situation to situation due to the influence of environmental factors like configuration of the computer system in terms of main memory, auxiliary devices, etc.; or due to the nature of the computing environment—whether amateur program development, i.e. student environment, or professional program development, or professional data processing shop environment? If so, then the question naturally arises as to the common characteristic of all SPs. This common characteristic is not in terms of the relative weightages given to computer and human resources, nor in the standardised view of a computer's usage environment. Instead, it is in terms of the primary motivation for the development of a system program in the first place. This primary motivation can be explained as follows:

The writing of a system program is not merely a means to an end, but is instead an end in itself.

Now let us see what this really means. Consider a scientist or engineer writing a program for a specific problem on hand. The program is merely a means to obtain the results for the problem. As long as this is achieved, the nature and characteristics of the program are immaterial to the designer of that program. Thus, efficiency and structure of the program are of no consequence, as also the elegance or clumsiness of algorithms used in it. To the designer of an SP, however, the goal is to design and code the program so that the task is done not only correctly but also effectively. Thus, the subjective balancing of the human and computer costs which form the basis of its design has to be properly manifested in its behaviour while helping to run a user program. Thus, the efficiency of algorithms and the suitability of data structures used are of primary importance. Of equal importance is the prominence given to the human resources required in the development and maintenance of a program. Let us explain this through an example.

Consider a compiler for a language like FORTRAN which is very widely used in installations devoted to scientific computing. If the designer of this compiler, which is an SP, concerns himself mainly with the optimisation of the computer resources, say, CPU-seconds used by a job, he might put the primary aim of the compiler as (i) fast compilation of a FORTRAN program, or (ii) efficient execution of the program after it has been translated into the machine language form. If most jobs in the installation are run only once or twice (a typical characteristic of a student job) and do not execute for long, then fast compilation should be given prominence over efficient execution. On the other hand, if jobs tend to execute for minutes or hours of computer time, then efficient execution should be given a higher priority. However, a compiler designed along these lines may not optimise the sum total of human and computer resources expended in the execution of a job. For example, the compiler might do an indifferent job of indicating all errors existing in a job. As a consequence, the user might have to spend a considerable amount of his time in trying to find *bugs* in his program. What is worse, such painstaking debugging will, in all probability, require extra runs of the program on the computer, leading to consumption of more computer resources as well.

A compiler designed to give higher weightage to the human resources can perform exhaustive checking of the program to detect all possible errors existing in it. This will help the user in obtaining his program in the 'final' form without wasting much time. However, such exhaustive error indication support (known as *diagnostic support*) will slow down the compiler since it would now take more time to process each statement. In this situation programmer time is costlier than computer time, so it would generally turn out that the slower compiler with good diagnostics is more optimal than a fast compiler without adequate diagnostics. The importance attached to the human resources therefore can hardly be underplayed.

The aspect in which an SP differs from any other kind of program is in the importance attached to the environmental factors which, in turn, lead to subjective balancing of the human and computer costs. Note that we keep on using the word "subjective" to underscore the fact that the balancing of costs is performed with a particular perspective of the utilisation of a computer *at a particular point in time*. As perspectives change, the balancing of costs also alters thus completely altering the fundamental considerations in designing a piece of system software. This is evidenced to some extent in the next section wherein we trace the history and evolution of various components of system software. A discussfon of the changing balance of costs and its influence on the strategy for system software development can also be found elsewhere [Dhamdhere, 1980].

1.2 Components of System Software—An Analogy

In order to identify the key components of system software and understand their functions, let us consider the similarities between a programmer *X* approaching a computer system to have his problem solved and an individual *Y* approaching a dress-shop to acquire a dress. While approaching the computer, *X* who is probably a scientist may have an idea about the nature of results he expects ("I expect this to be a beautiful sinusoid," he might say). Similarly, while approaching the dress-shop, *Y* might have a good idea as to what he/she expects to buy. But how do *X* and *Y* obtain what they want? Not merely by dreaming about their ideas. Before they obtain what they want, there is a lot of hard work to be put in.

Let us consider the dress-shop situation first. *Y* has to put in a certain amount of work inside the dress-shop to obtain the dress that he/she has dreamt about. This could be as simple as choosing from a set of readymade dresses. Or, if none suit *Y*'s choice, then *Y* would have to discuss his/her requirements with the master dress-designer, and using his terminology, *Y* should be able to show him how to obtain the desired features by using standard, well established steps in dress design.

After *Y* has specified the requirements, let us see how the dress actually gets made. The master designer to whom *Y* explained his requirements does not actually cut and sew. Instead, using some code or convention, he gives very specific and precise instructions to the cutting-and-tailoring section of the dress-shop, where his instructions are carried out and the dress is actually made. In the cutting-and-tailoring section, virtually hundreds of dress orders and specifications keep pouring in. A section supervisor keeps track of these orders, schedules them for execution and actually gets the work done from the low level cutter-tailor.

The reason why *Y* does not directly interact with the low level cutter-tailor is one of practicability. For one thing, *Y* and the tailor do not speak the same language. While *Y* has ideas as to what he/she wants, *Y* cannot supply the specifics the way the cutter-tailor wants them. For example, *Y* wants bell-bottom trousers which look elegant rather than baggy, while the cutter wants to know whether a 65 cm bottom is okay. This gap has to be bridged by someone who can communicate meaningfully with *Y* and translate ***Y's*** requirements into the form which the cutter-tailor would understand.

A typical modern computer system operates much the same way as the dress-shop. While the actual number-crunching is done in the central processor unit (CPU), it requires its information in a very specific and rigid format. The programmer *X* knows the details only from his own viewpoint but he does not know how to supply them to the CPU. The language in which he can supply the details is the "programming language," while the language in which the CPU wants them is the "machine language". Therefore, the programmer's specifications are presented to a "language translator" or compiler which converts these into a set of instructions in the machine language so that the CPU can understand them.

In order to extract the maximum work out of the CPU, the machine language programs are handed over to the "operating system" which schedules the actual work to be done by the CPU from moment to moment. This is done so as to optimise the performance of the computer system as a whole.

The language translator and the operating system are themselves programs. Their function is to get the user's program, which is written in a programming language, to run on the computer system. We have already seen that all such programs which aid the excution of a user program are called system programs (SPs). The collection of such SPs is the "system software" of a particular computer system.

Now, let us look into the economics of these arrangements from the viewpoint of the customer or the computer user. When a customer goes to a dress-designing outfit, he ends up paying not only for the cutter-tailor who does the actual job but also for the section supervisor and scheduler, as also for the master designer himself. If he were to do the job of the designer and supervisor himself and approach a cutter-tailor on his own, he would have to pay only the tailor who does his job. He could thus save on the overheads of the complete organisation. However, he would have to pay a certain price for these economies. He would have to learn the technical language which the tailor understands and he would also have to acquire the techniques used by the master designer. In the end, he may feel that it would have been cheaper for him to have gone to the master designer **in the first place. Assuming that the organisation is not making inordinate profit, he will** not have to bear the excessive burden of overheads since the supervisor/scheduler would be trying to optimise the performance of the work-force.

An identical argument holds good in the case of a computer system. By writing a program in a higher level programming language, a programmer obviates the need to acquire totally new skills merely in order to run his program. The compiler performs the task of making his program understandable to the CPU. In order to keep the total costs down, the OS optimises the performance of the computer system. Thus, the two fundamental aspects of the task of the system software are: (i) making available new/better facilities and (ii) achieving efficient performance. Various components of system software perform various tasks which contribute towards these fundamental goals. In the next Section, the salient features of these system software components are discussed.

1.3 Evolution of System Software

System programs which are the standard components of the software of most computer systems today have evolved through a series of need-based developmental steps. The two-fold motivation mentioned above arises out of a single primary goal, viz. of making the entire program development process more *effective*.

The two aspects of program development cost, viz. human and computer costs, have to be weighed against each other in order to develop the notion of effectiveness. To be more specific, we can call this the notion of *optimal effectiveness*. Over the last two decades, the balance of human versus computer costs has undergone a sea change. Today, human costs outweigh computer costs almost as drastically as computer costs used to outweigh human costs about 20 years ago. Further, this reversal has been brought about by a continuous and monotonic process. One would therefore expect that the aim of obtaining effective utilisation of the computer system must have played a secondary role throughout the evolution of system software. However the history of system software is replete with quick reversals of priorities given to the two aspects of motivation, viz. introduction of better facilities and effective utilisation of the system. This is so because these two aspects result in contrary pressures on the balance of human and computer costs. The influence of introducing new facilities therefore has to be offset by efforts to enhance effectiveness of computer utilisation so as to restore the balance. We will see many instances of this as we trace the evolution of different components of system software.

1.3.1 Language Translators

Historically, the first important step in the evolution of system software was the development of a language translator. In the early days, programs had to be written in machine language. As we saw in the dress shop analogy of the previous section, this is quite cumbersome from the programmer's viewpoint. Development of language translators permitted the programmer to code his program in a language which was much more convenient to handle than the machine language of a computer. Translators for a low-level programming language were termed *assemblers* and such low-level languages were termed *assembly languages*. The assembly languages were merely humanised machine languages which permitted the use of mnemonic operation codes like LOAD, ADD and symbolic operands like VALUE, RESULT in place of machine instruction codes and machine addresses both of which are merely numbers, i.e. strings of digits. The assembly languages permitted easy writing and modification of programs, but were still highly machine-dependent.

Machine-independent programming languages were the next to evolve. These languages were termed higher level languages (HLLs) because they only required a programmer to specify the logic of solving a problem in the form of an *algorithm,* i.e. a step-by-step procedure which leads to the solution of a problem. Each step of the procedure could represent a significant action in terms of the program logic, like a computation, a decision, input of values, etc., and in no way depended on the computer on which the program was to be executed. The use of HLLs freed the programmer from the need to know intricate details of a computer, also freeing him from mundane, low-level activities like coding his program as a sequence of machine instructions. An HLL program had to be translated into machine language before it could be executed on a machine, and this translation was costlier than the translation of an assembly language program. However, this extra translation cost was more than offset by the reduction in the effort of a programmer in designing, coding and debugging a program. Thus, not only could a program be designed using higher level programming concepts like structuring of data and use of procedure segments which are more natural from the viewpoint of a programmer, but the HLL translator could also assist the programmer by indicating errors of specification at the source-language level during translation of a program. Further, the translator could also make provision for the detection and precise indication of run-time errors while an assembler could not.

Thus, the first two developments in system software were in favour of providing new conveniences to a programmer at the cost of extra computer time requirements. The next developments were concerned more with the efficiency of utilisation of a computer system.

I.3.2 Batch Monitors

Early computer systems were used in a one-program-at-a-time operating mode. Thus, a computer operator would carry out a few actions in order to set up and initiate the processing of a job. These actions could be as simple as flipping a few switches on the console, but very often consisted of manually feeding in a few instructions in the computer memory which, when executed, would initiate a language translator for operation. After this initial sequence, the processing of a job would begin. At the end of the execution of the job, the operator would have to repeat the same steps to initiate the processing of the next job.

This mode of operation made inefficient use of the processing capability of a computer because a lot of time was wasted in the operator's actions. Since human interaction time would be of the order of seconds if not minutes, efficiency of utilisation critically depended on reducing human intervention. This was achieved by designing an SP known as a *batch monitor* which would realise processing of a set of user jobs without the need for operator interaction. The operator was now required to initiate the functioning of the batch monitor. Once this was achieved, the batch monitor would take over control of the computer operations. It would initiate the processing of each job in the batch in such a manner that at the end of its processing, control would return to the batch monitor. It would then initiate the next job in an automatic fashion. After the last job in the batch was processed, the batch monitor would terminate its own operation and control would now come back to the computer operator in order to initiate the next action.

Automatic control of execution of a batch of jobs by the batch monitor improved efficiency of utilisation of the computer system. If a computer system shared by a group of users was used more efficiently, then one would expect all the users to benefit equally. However, in batch processing a curious thing happened. While the efficiency of utilisation improved, a general user had to suffer long *turn-around times*. The turn-around time for a job is defined as the time elapsed from its submission at the computer centre to the time when its results are obtained. In the one-program-at-a-time environment, submission of jobs and release of their results was generally done on an informal basis, so often the turn-around time for a job was only marginally larger than the time taken for the job to be processed on the computer. With the introduction of batch processing, formal procedures had to be introduced to facilitate formation of batches with sufficient number of jobs in them. For example, at some computer centres, all jobs submitted between 9 a.m. and 12 noon might be included in the same batch. Further, in order to obtain still higher efficiency of utilisation, the jobs forming a batch might be recorded on a faster Input Output (IO) medium like a magnetic tape or a disk. The results of processing the batch might similarly be recorded on a magnetic tape or a disk and later printed out on a printer, sorted out and then released to the users.

The turn-around time for a job in the batch processing mode thus depends on (i) the total processing time of all the jobs in a batch and (ii) the batch formation time as well as the printing of output and release time. However, this mode of operation clearly improves the utilisation efficiency and does not require a user to put in any extra effort besides introducing appropriate batch monitor control cards in his program deck.

1.3.3 Multiprogramming Operating Systems

Batch monitor was the first system program devised to improve the utilisation efficiency. However, given the architecture of early second generation machines, the scope of its efforts was restricted simply to automating the transition from the processing of one job in the batch to the next. Around this time the field of computer architecture made one significant advance in terms of the IO (Input Output) organisation of a computer. The concept of an IO channel was introduced in order to free the CPU for more productive tasks.

In the classical computer architecture, the IO instructions were executed the same way as all the other instructions (arithmetic, logical, etc.) by the CPU. Thus, when an IO instruction was decoded, the CPU generated appropriate control signals for the IO devices. The IO devices now got busy and performed their operations and at the end sent an end-of-operation signal to the CPU. The CPU was idle from the moment of IO initiation to IO completion. The channel concept (Fig. I.1) freed the CPU from unnecessary idle times while the IO operation was in progress. The actual IO is now performed as follows: The CPU executes a Start Input-Output instruction (SIO instruction of IBM 360/370), with the address of an IO device as the operand. When this instruction is executed, the device address is passed on to the channel. The channel interrogates the device to see if it is available, and sends an end-of-operation signal with an appropriate condition code (*device available* to perform operation, *device busy, device non-existent,* etc.) to the CPU. If the required device is available, the channel obtains details of the IO operation to be performed from standard storage area and the IO operation is initiated. Since the channel has an independent path to/from the main storage of the computer, the CPU is not required during the IO operation. Thus, after the initial selection and interrogation of the device, the CPU is free to do some useful computations until the IO operation terminates. The end of an IO operation is brought to the notice of the CPU (generally through an *interrupt*) so that it can initiate another IO operation, if necessary.

In spite of the capability of such computer architectures to support computations and IO concurrently, it was soon found that individual programs could not effectively exploit this concurrency. The main reason was the lack of data independence between various parts of the program. For example, consider a program which reads a certain amount of data and then processes it. Now, the CPU cannot simply initiate IO for a data card (let us say the first data card) and continue performing computations, since the values of variables taking part in the computation have to be read in first from cards. Since the processing step requires the same variables (or storage areas) which are being given values through the IO operation, both IO and processing cannot go on simultaneously.

In effect, unless all the required data is read in, no processing can be done. In other words, the CPU would still remain idle during IO.

Since concurrent utilisation of the CPU and the channels requires data independence, and since data independence is difficult to ensure within a program, the stratagem used in practice is to locate two (or more) independent programs within the memory of the computer. While IO is being performed for one program, the CPU can perform useful computations for another program. Thus, two or more programs can be executed in an interleaved fashion so as to keep the CPU (and also possibly the IO subsystem) busy most of the time. Hence the term *multiprogramming*.

In order to obtain the best possible CPU utilisation, it is necessary that a proper mix of programs be multiprogrammed. For example, if all programs being multiprogrammed have a lot of IO requirements and involve very little processing, again the CPU utilisation will not be very high. Apart from the program mix, it is also necessary to switch from the execution of one program to that of another in a systematic and judicious manner. All these aspects of control are exercised by an SP which is permanently resident in the main storage of the computer. We will refer to this control program as a *supervisor*. (It should be noted here that unfortunately there is no standardisation in the terminology used for such control programs. Often the terms *executive* or *monitor* are also used to mean the same thing.) A multiprogramming operating system is a collection of SPs including the multiprogramming supervisor and any other programs required by the supervisor from time to time. The various control functions performed by the supervisor and their implementation details are discussed in Chapter 6.

1.3.4 Time-Sharing Operating Systems

Multiprogramming improves efficiency of system utilisation. However, this does not yield any *direct* benefit to the user population as the turn-around time, which is the primary quantification of user service, does not necessarily improve. For example, in the interests of further improvement in utilisation efficiency, the various programs being multiprogrammed would not be initiated as single jobs by a computer operator. Instead, they would generally be initiated as batches of jobs. Thus, the user would still suffer from long turn-around times inherent in batch processing.

From a user's viewpoint, fast turn-around for his jobs is desirable. This would facilitate faster program development since errors detected in one run of the program can be corrected immediately and the program resubmitted for another test run. As a matter of fact, the best conceivable service that a user would expect from a computer is that of *instant* turn-arounds. This requirement has given rise to the concept of *interactive computing*.

In an interactive computing environment, a user sits at a video-terminal and feeds his input to the computer by keying in a few characters or complete statements. These are displayed on the screen as well as transmitted to the computer. The computer's response is also similarly displayed. Depending on the translating system available at the computer installation, the user may have to feed in his entire program before attempting to run it, or in certain cases he may be able to communicate with the translator on a statement-by-statement basis. In the latter case, the translator would process a statement as soon as it is submitted and indicate errors, if any. The user can now correct an erroneous statement or key in the next statement. This act of keying in of a processing request by the programmer followed by its appropriate fulfilment by the computer system is known as an *interaction* and the time taken by the computer to respond to the processing request is known as the *response time*.

In such an interactive environment, the user would evaluate the computing services in terms of the response times maintained by the system. The easiest way of ensuring good response times seems to be to connect only one terminal to the computer system and run only one programmer's job on the system. But this is obviously inefficient. A better method would be to have a number of terminals connected to the system and to process the requests made by various programmers in such a manner that all of them get reasonably good response times. This is the strategy used in practice, and for this reason operating systems using such techniques are known as *Time-Sharing Operating Systems*. A typical time-sharing operating system would process the requests of all terminals in a circular fashion over and over again so as to give fair response times to all the terminal users.

When we compare the time-sharing operating system with simple batch processing and multiprogramming systems we find that its motivation is the exact opposite of the other two. User service is given a higher weightage than aspects of utilisation efficiency. As a matter of fact, it turns out that time-sharing actually degrades the utilisation efficiency because of the processor time consumed by the OS itself for various housekeeping jobs like switching from the execution of one program to the next, etc. For this reason, in practice, computer systems are rarely operated in the pure time-shared mode. Instead, apart from time-sharing, the OS would also support batch processing, etc. in a multiprogramming environment. In a later section of this Chapter, such an operating system is discussed.

I.3.5 Other System Programs

In this Section we have traced the evolution of some SPs, mainly translators and operating systems, to illustrate how the two conflicting aspects of motivation for the development of SPs are weighed against each other and how the balance of these weights shifts along with a shift in the perspectives of a situation. It was not the intention of this section to exhaustively list out all the SPs that exist in a typical computer system and trace their evolution. Thus SPs like librarians, editors, input-output control systems (IOCS), file systems, etc. have not even been mentioned. Various components of operating systems like storage manager, processor manager, job accounting package, etc. have also not been identified or discussed separately. It is intended to introduce such SPs at appropriate points in the discussion in this and later Chapters.

I.4 The Model of a Computer System

Throughout this book, we will discuss the working principles and design aspects of various SPs. For the sake of generality, we will base these discussions on the model of a typical computer system which exhibits broad architectural features found in most computer systems. The model is defined at two levels—(i) the machine model depicting the hardware features, and (ii) the operating system model depicting the architectural features of its OS. Since the intention in introducing these models is simply to specify the hardware and software features whose conceptual awareness is essential for the discussion in subsequent chapters, particular details of the architecture are being omitted for the present.

I.4.1 The Machine

Figure I.1 shows the machine model, illustrating the fundamental organisational details of the machine, the various component units and their interconnections. No distinction is made between data and control paths. The important components of the model are the storage unit, the CPU, and the IO subsystem consisting of IO processors or IO channels and IO devices, etc. A brief description at the functional level of these components is given below. Detailed description at the functional, design and implementation levels can be found in the texts mentioned in the bibliography.