计算机软件短训班

英语阅读资料

自动化系计算机数学组选编

广 东 工 学 院

一九七六年十二月

SYSTEMS PROGRAMMING

John J. Donovan

Project MAC,
Massachusetts Institute of Technology

1 background

This book has two major objectives: to teach procedures for the design of software systems and to provide a basis for judgement in the design of software. To facilitate our task, we have taken specific examples from systems programs. We discuss the design and implementation of the major system components.

What is systems programming? You may visualize a computer as some sort of beast that obeys all commands. It has been said that computers are basically people made out of metal or, conversely, people are computers made out of flesh and blood. However, once we get close to computers, we see that they are basically machines that follow very specific and primitive instructions.

In the early days of computers, people communicated with them by on and off switches denoting primitive instructions. Soon people wanted to give more complex instructions. For example, they wanted to be able to say X=30 * Y; given that Y=10, what is X? Present day computers cannot understand such language without the aid of systems programs. Systems programs (e.g., compilers, loaders, macro processors, operating systems) were developed to make computers better adapted to the needs of their users. Further, people wanted more assistance in the mechanics of preparing their programs.

Compilers are systems programs that accept people-like languages and translate them into machine language. Loaders are systems programs that prepare machine language programs for execution. Macro processors allow programmers to use abbreviations. Operating systems and file systems allow flexible storing and retrieval of information (Fig. 1.1).

There are over 100,000 computers in use now in virtually every application. The productivity of each computer is heavily dependent upon the effectiveness, efficiency, and sophistication of the systems programs.

In this chapter we introduce some terminology and outline machine structure and the basic tasks of an operating system.

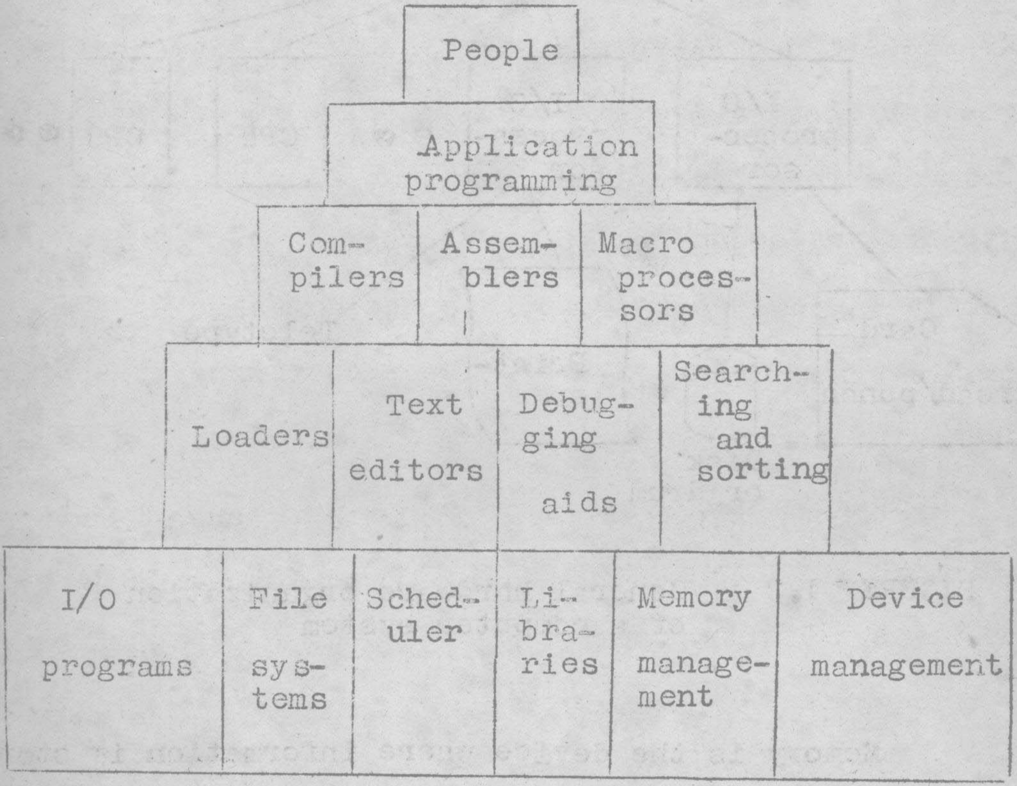

FIGURE 1.1 Foundations of systems programming

1.1 MACHINE STRUCTURE

We begin by sketching the general hardware organization of a computer system (Fig. 1.2).

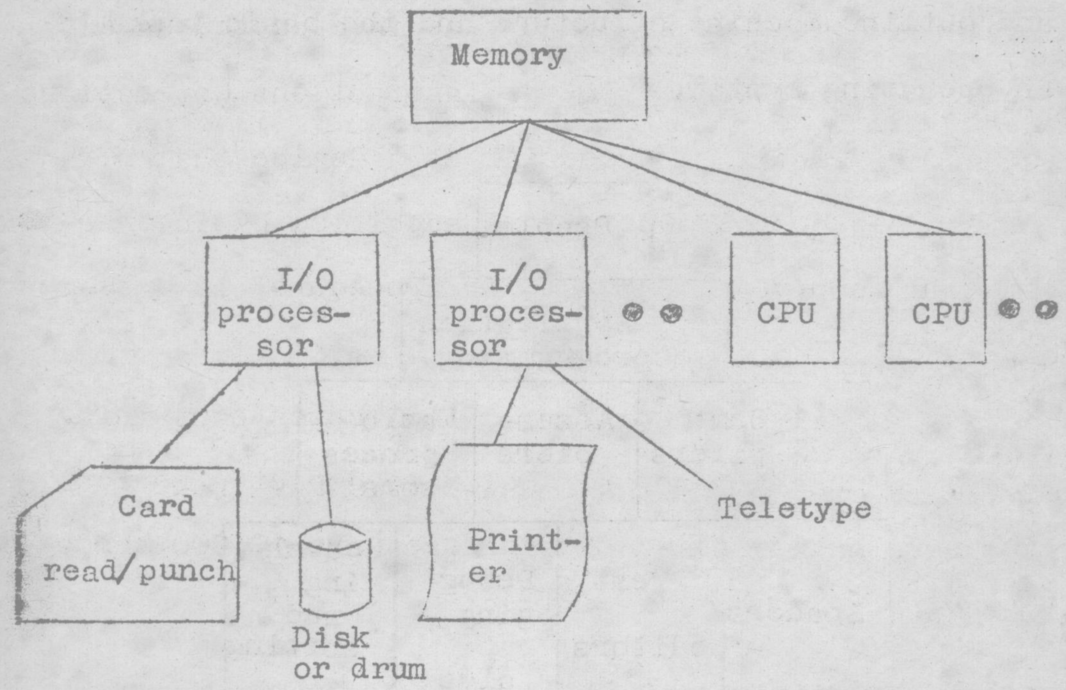

FIGURE 1.2 General hardware organization of a computer system

Memory is the device where information is stored. Processors are the devices that operate on this information. One may view information as being stored in the form of ones and zeros. Each one or zero is a separate binary digit called a bit. Bits are typically grouped in units that are called words, characters, or bytes. Memory locations are specified by addresses, where each address identifies a specific byte, word, or character.

The contents of a word may be interpreted as data (values to be operated on) or instructions (operations to be performed). A processor is a device that performs a sequence of operations specified by instructions in memory. A program (or procedure) is a sequence of instructions.

Memory may be thought of as mailboxes containing groups of ones and zeros. Below we depict a series of memory locations whose addresses are 10,000 through 10,002.

Address	Contents			
10,000	0000	0000	0000	0001
10,001	0011	0000	0000	0000
10,002	0000	0000	0000	0100

An IBM 1130 processor treating location 10,001 as an instruction would interpret its contents as a "halt" instruction. Treating the same location as numerical data, the processor would interpret its contents as the binary number 0011 0000 0000 0000 (decimal 12,288). Thus instructions and data share the same storage medium.

Information in memory is coded into groups of bits that may be interpreted as characters, intruc-tions, or numbers. A code is a set of rules for in-terpreting groups of bits, e.g., codes for represen-tation of decimal digits (BCD), for characters (EBCDIC, or ASCII), or for instructions (specific pro-cessor operation codes). We have depicted two types of processors: Input/Output(I/O) processors and Cen-tral Processing Units (CPUs). The I/O processors are concerned with the transfer of data between memory and peripheral devices such as disks, drums, printers, and typewriters, The CPUs are concerned with manipu-lations of data stored in memory. The I/O processors execute I/O instructions that are stored in memory; they are generally activated by a command from the CPU. Typically, this consists of an "execute I/O" instruction whose argument is the address of the start of the I/O program. The CPU interprets this instruction and passes the argument to the I/O pro-cessor.

The I/O instruction set may be entirely different from that of the CPU and may be executed asynchronous-ly (simultaneously) with CPU operation. Asynchronous operation of I/O channels and CPUs was one of the earliest forms of multiprocessing. Multiprocessing means having more than one processor operating on the same memory simultaneously.

Since instructions, like data, are stored in memory and can be treated as data, by changing the bit configuration of an instruction ——— adding a number to it ——— we may change it to a different instruction. Procedures that modify themselves are called impure procedures. Writing such procedures is poor programming practice. Other programmers find them difficult to read, and moreover they cannot be shared by multiple processors. Each processor executing an impure procedure modifies its contents. Another processor attempting to execute the same procedure may encounter different instructions or data. Thus, impure procedures are not readily reusable. A pure procedure does not modify itself

To ensure that the instructions are the same each time a program is used, pure procedures (re-entrant code) are employed.

1.2 EVOLUTION OF THE COMPONENTS OF A PROGRAMMING SYSTEM

1.2.1 Assemblers

Let us review some aspects of the development of the components of a programming system.

At one time, the computer programmer had at his disposal a basic machine that interpreted, through hardware, certain fundamental instructions. He would program this computer by writing a series of ones and zeros (machine language), place them into the memory of the machine, and press a button, whereupon the computer would start to interpret them as instructions.

Programmers found it difficult to write or read programs in machine language. In their quest for a more convenient language they began to use mnemonic (symbol) for each machine instruction, which they would subsequently translate into machine language. Such a mnemonic machine language is now called an assembly language. Programs known as assemblers were written to automate the translation of assembly language into machine language. The input to an assembler program is called the source program; the output is a machine language translation (object program).

1.2.2 Loaders

Once the assembler produces an object program, that program must be placed into memory and executed. It is the purpose of the loader to assure that object programs are placed in memory in an executable form.

The assembler could place the object program directly in memory and transfer control to it, thereby causing the machine language program to be executed. However, this would waste core[1] by leaving the assembler in memory while the user's program was being executed. Also the programmer would have to retranslate his program with each execution, thus wasting translation time. To overcome the problems of wasted translation time and wasted memory, systems programmers developed another component, called the loader.

A loader is a program that places programs into memory and prepares them for execution. In a simple loading scheme, the assembler outputs the machine language translation of a program on a secondary storage device and a loader is placed in core. The loader places into memory the machine language version of the user's program and transfers control to it. Since the loader program is much smaller than the assembler, this makes more core available to the user's program .

The realization that many users were writually the same programs led to the development of "ready-made" programs (packages were written by the computer manufacturers or the users). As the programmer became more sophisticated, he wanted to mix and combine ready-made programs with his own. In response to this demand, a facility was provided whereby the user could write a main program that used several other programs or subroutines. A subroutine is a body of computer instructions designed to be used by other routines to accomplish a task. There are two types of subroutines: closed and open subroutines. An open subroutine or macro definition is one whose code is inserted into the main program (flow continues). Thus if the same open subroutine were called four times, it would appear in four different places in the calling program. A closed subroutine can be stored outside the main routine, and control transfers to the subroutine. Associated with the closed subroutine are two tasks the main program must perform: transfer of control and transfer of data.

Initially, closed subroutines had to be loaded into memory at a specific address. For example, if a user wished to employ a square root subroutine, he would have to write his main program so that it would transfer to the location assigned to the square root routine (SQRT). His program and the subroutine would be assembled together. If a second user wished to use the same subroutine, he also would assemble it along with his own program, and the complete machine language translation would be loaded into memory. An example of core allocation under this inflexible loading scheme is depicted in Figure 1.3, where core is depicted as a linear array of locations with the program areas shaded.

Note that program 1 has "holes" in core. Program 2 overlays and thereby destroys part of the SQRT subroutine.

Programmers wished to use subroutines that referred to each other symbolically and did not want to be concerned with the address of parts of their

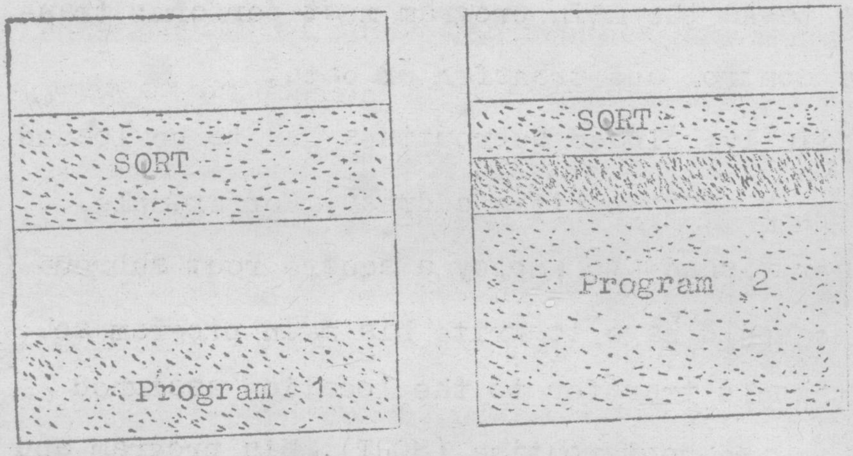

FIGURE 1.3 Example core allocation for absolute loading

programs. They expected the computer system to assign locations to their subroutines and to substitute addresses for their symbolic references.

Systems programmers noted that it would be more efficient if subroutines could be translated into an object form that the loader could "relocate" directly hebind the user's program. The task of adjusting programs so they may be placed in arbitrary core locations is called relocation. Relocating loaders perform four functions:

1. Allocate space in memory for the programs (allocation)

2. Resolve symbolic references between object decks (linking)
3. Adjust all address dependent locations, such as address constants, to correspond to the allocated space (relocation)
4. Physically place the machine instructions and data into memory (loading)

The various types of loaders that we will discuss ("compile-and-go," absolute, relocating , direct-linking, dynamic-loading, and dynamic-linking) differ primarily in the manner in which these four basic functions are accomplished.

The period of execution of a user's program is called execution time. The period of translating a user's source program is called assembly or compile time. Load time refers to the period of loading and preparing an obiect program for execution.

1.2.3. Macros

To relieve programmers of the need to repeat identical parts of their program, operating systems provide a macro processing facility, which permits the programmer to define an abbreviation for a part of his program and to use the abbreviation in his program. The macro processor treats the identical parts of the program defined by the abbreviation as a macro definition and saves the definition. The macro processor substitutes the definition for all occurrences of the abbreviation (macro call) in the program.

In addition to helping programmers abbreviate their programs, macro facilities have been used as general text handlers and for specializing operating systems to individual computer installations. In specializing operating systems (systems generation), the entire operating system is written as a series of macro definitions. To specialize the operating system, a series of macro calls are written . These are processed by the macro processor by substituting the appropriate definitions, thereby producing all the programs for an operating system.

1.2.4. Compilers

As the user's problems became more categorized into areas such as scientific, business, and sta^ti^tical problems, specialized languages (high level languages) were developed that allowed the user to express certain problems concisely and easily. These high level languages—— examples are FORTRAN,COBOL,ALGOL,and

PL/I__ are processed by compilers and interpreters .
A compiler is a program that accepts a program written in a high level language and produces an object program. An interpreter is a program that appears to execute a source program as if it were machine language. The same name (FORTRAN, COBOL, etc.) is often used to designate both a compiler and its associated language.

Modern compliers must be able to provide the complex facilities that programmers are now demanding . The compiler must furnish complex accessing methods for pointer variables and data structures used in languages like PL/I, COBOL, and ALGOL 68. Modern compilers must interact closely with the operating system to handle statements concerning the hardware interrupts of a computer (e.g. conditional statements in PL/I.)

1.2.5 Formal Systems

A formal system is an uninterpreted calculus. It consists of an alphabet, a set of words called axioms, and a finite set of relations called rules of inference. Examples of formal systems are: set theory, boolean algebra, Post systems, and Backus Normal Form. Formal systems are becoming important in the design, implementation, and study of programming languages. Specifically , they can be used to specify the syntax (form) and the semantics (meaning) of programming languages. They have been used in syntax-directed compiler verification, and complexity studies of languages.

1.3. EVOLUTION OF OPERATING SYSTEMS

Just a few years ago a FORTRAN programmer would approach the computer with his source deck in his left hand and a green deck of cards that would be a FORTRAN compiler in his right hand. He would:

1. Place the FORTRAN compiler (green deck) in the card hopper and press the load button. The computer would load the FORTRAN compiler.
2. Place his source language deck into the card hopper. The FORTRAN compiler would proceed to translate it into a machine language deck, which was punched onto red cards.
3. Reach into the card library for a pink deck of cards marked"loader," and place them in the card hopper. The computer would load the loader into its memory.
4. Place his newly translated object deck in the card hopper. The loader would load it