LECTURE-01
INTRODUCTION TO SYSTEM PROGRAMMING

INTRODUCTION

The main aim of system programming is to design of system software and to provide basic for judgment in the design of software. And we will do it through discuss of design and implementation of the major system component.

Computers are basically machines that follow very specific and primitive instructions. Earlier day's people gave instructions to machines through on and off switches. Later people gave instructions like X=3; Y=4; what is X+Y and so on. But computer now a day's cannot understand such languages with the aid of some program which are called as *system programs*. System program (i.e. compiler, loader, macro processors, operating system) were developed to make the computers better adapt to needs of the users. And later on people wanted more assistance in the mechanisms of preparing their programs. This is what gave rise to *system programming*.

Compilers are system programs that accept people-like language and translate them into machine language.

Loaders are system programs that prepare machine language programs for execution.

Macro processors allow programmer s to use abbreviations.

Operating systems and *file systems* allow flexible storing and retrieval of information.

|People|
|Application programming|

|Compilers |Assemblers |Macro processors |

|Loader |Text Editors |Debugging aids |Searching and sorting |

|I/Oprograms|File System|Scheduler|Libraries|Memory management|Device management|

Fig: Foundation of systems programming.

There are number of computers that are in use in different area and the productivity of each computer is heavily dependent upon the effectiveness, efficiency and sophistication of systems programs.

MACHINE STRUCTURE

The general hardware organization of a computer system can be listed as below.

Memory is the device where information is stored.

Bits are the unit of storage that may be either one or zero. Bits are grouped as character, bytes and words. Memory locations are specified by *addresses,* where each address indentifies a specific character, byte and word.

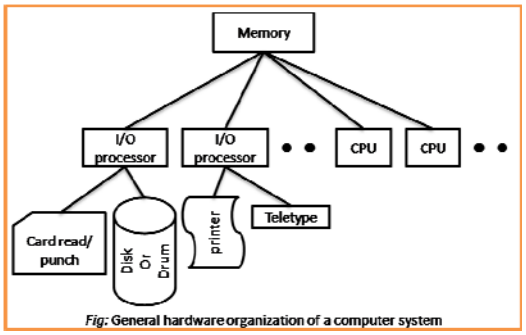

Fig: General hardware organization of a computer system

Data or instructions are the content of words. A processor performs these instructions that are stored in memory. Thus instruction and data share the same storage medium.

Programs are sequence of instructions.

Codes is a set of rules for interpreted groups of bits i.e. codes for representation of decimal digits (BCD), for characters (EBCDIC, or ASCII), or for instructions (specific processor operation codes).

Processors are the devices that operate on this information. There are two types of processors i.e. ***Input/Output*** *(I/O) processor* that are concerned with transfer of data between memory and peripheral devices such as disks, drums, printers and typewriters where as the ***Central processing unit*** *(CPU)* is concerned with the manipulation data stored in the memory. The I/O instructions are stored in the memory that is activated by the command from the CPU. Typically this consists of an "execute I/O" instruction whose argument is the address of an I/O program. The CPU interprets this instruction and passes the argument to the I/O processor (commonly called I/O channels).

The I/O instruction are completely different form the CPU instruction and run *asynchronously* (simultaneously) with the CPU operation. Asynchronously

operation of I/O channels and the CPUs was one of the earliest forms of *multiprocessing* (more than one processor operating simultaneously on the same memory).

Since instruction like data are stored in memory and can be treated like data by changing the bit configuration (possibly by adding a number it may change to a different instruction). Procedures that modify themselves are called as *impure procedures*. Creating impure procedures is a poor programming practice and also cause problem when shared by multiple processor (as each can change and implement a different instruction). A *pure procedure* on the other hand does not modify itself. Programs called as *re-entrant* are employed to see that the instruction remains same always.

EVOLUTION OF THE COMPLEMENTS OF A PROGRAMMING SYSTEM

Assembler: At one time a person used to write the program in machine level codes (interns of 0 and 1) and used to up the same in the memory and press a button to start the program. Programmer found it difficult to write the program in machine language and started to make use of *mnemonic* which they would subsequently translate into machine language. Such a mnemonic machine language is now called as *assembly language*. Programs known as *assemblers* were written to automate the translation of assembly language to machine language. The input to an assembler is called as *source program* and the output is a machine language translation (*object program)*.

Loader: After the program is translated then the object program needs to be loaded on to the memory which is accomplished by another system program called *loader*. The work to a loader is twofold that are as follows:

1. Loader save the core (memory) as in the absence of loader the source program, assembler and the object program needs to be present simultaneously on the core. But with a loader the assembler loads the object program in the secondary memory and a loader is placed in the core that loads the object program in the core and transfer control to it.
2. Loader also save time when the program is to be executed a number of times as each time the object program is placed from secondary memory to core. Hence saving the translation time that is consumed at each time the program is executed.
3. It is also used for *relocation* in case of subroutine. *Subroutines* are set of instructions that are to be used by other routine for a particular task. Basically subroutines are of two types *open subroutine (macro definition)* and *closed definition*. In case of open subroutine the whole code is inserted in the main program i.e. the flow of control is continuous. But that is not the case with the closed subroutine i.e. the control is transferred to the subroutine. The subroutines are first on to the core hence it may happen sometime that two programs make use of the same subroutine. In that case it may happen that a program may overlap the subroutine hence destroying it. This problem can be overcome by translating the subroutine to an object that can be relocated if there is a overlap. The process of relocation can described as follows:

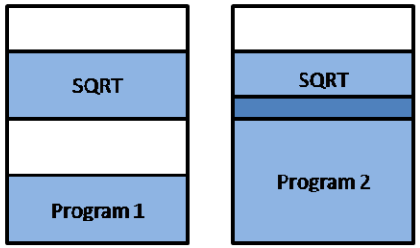

Fig: Core allocation for absolute loading

   a. *Allocation*- allocates space in memory for the program
   b. *Linking*- resolves symbolic references between object decks.
   c. *Relocation*-adjust all address-dependent locations.
   d. *Loading*-physically places the machine instruction and data into core.

The time taken for a user program to execute is called *execution time.* The needed for translation is called *assembly* or *compile time*. The time needed to load a program into core is called as the *load time*.

Macros: In case of lengthy programs the repeated identical parts uses the macro processing facility of operating system. Here the identical part of the program is abbreviated and this is treated as a *macro definition* and saves definition. The macro processor substitutes the definition for all occurrences of the abbreviation (macro call) in the program. Basically it is used in design of operating systems where a number of macro calls are written.

Compiler: A compiler is a program that accepts a program in high level language and produces an object program where an *interpreter* is a program that executes a source program as if it were in machine language. Basically the name of the language and the compiler are same i.e. C, C++, FORTRAN.

Linker: Linker or link editor is a program that takes one or more objects generated by a compiler and combines them into a single executable program.

Computer programs typically comprise several parts or modules; all these parts/modules need not be contained within a single object file, and in such case refer to each other by means of *symbols*. Typically, an object file can contain three kinds of symbols:

- Defined symbols, which allow it to be called by other modules,
- Undefined symbols, which call the other modules where these symbols are defined,
- Local symbols, used internally within the object file to facilitate relocation.

When a program comprises multiple object files, the linker combines these files into a unified executable program, resolving the symbols as it goes along. Linkers can take objects from a collection called a library. Some linkers do not include the whole library in the output; they only include its symbols that are referenced from other object files or libraries. Libraries exist for diverse purposes, and one or more system libraries are usually linked in by default.
The linker also takes care of arranging the objects in a program's address space. This may involve relocating code that assumes a specific base address to another base. Since a compiler seldom knows where an object will reside, it often assumes a fixed base location (for example, zero). Relocating machine code may involve re-targeting of absolute jumps, loads and stores.

The executable output by the linker may need another relocation pass when it is finally loaded into memory (just before execution). This pass is usually omitted on hardware offering virtual memory — every program is put into its own address space, so there is no conflict even if all programs load at the same base address. This pass may also be omitted if the executable is a position independent executable.

Formal system: A formal system is an uninterpreted calculus consisting of alphabet, a set of words called axioms, and a finite set of relations called rules of inference. E.g. set theory, Boolean algebra, Backus Normal Form. Formal systems are basically used to represent the syntax and semantic of a programming language.

EVOLUTION OF OPERATING SYSTEM

Few year back when FORTRAN programming was used, the user goes to the computer with two decks of cards namely source program and FORTRAN compiler (green deck) in his hand and then does the following.

1. Place the FORTRAN compiler on the card hopper and press the load button. This loads the FORTRAN compiler onto the core.
2. Place the source program on the card hopper and the compiler will translate the program into a machine language (object) deck that is punched onto red deck.
3. Place the loader (pink deck) onto the card hopper. This will load the loader onto core.
4. Place the object deck (red) on the card hopper. This will load the object program onto the core.
5. Place the deck of subroutine that may be used by the program. The loader will load the subroutine onto core.
6. Finally the loader will transfer the control to the user program and require some data card as input to give the expected output.

This was the process that was followed whenever a task is given to the computer. In this process the machine stands idle for most of the time. To overcome this wastage, jobs (unit of specified work i.e. a set of programs) of similar type called as *batch* jobs are given to the machine. This type of system is called as *batch operating system*. Batch processing systems performs the task of batching jobs.

With the increase in demand of computer resources (time, memory, devices, files etc), effective management of these resources became difficult. In batch processing system the whole of the memory is dedicated to a single program. Hence if that program is not utilizing the whole of core, then always a part of the core is wasted. To overcome this, the memory is partitioned and allocated to more than one program. This is called as *multiprogramming*. Multiprogramming are of two types i.e. *multiprogramming with fixed task (MFT)* and *multiprogramming with variable task (MVT)*. MFT uses fixed partitions of core whereas MVT uses variable partitions of core.

Even with partitioning we cannot overcome wastage of core as some partitions are too small to be used (*holes)*. These holes or unused portions of memory cause problem called as *fragmentation*. Fragmentation can be overcome by *relocatable partition* and *paging*. Relocatable partitioning allows the hole to condense to a single continuous part of core. Paging is a process of memory allocation where the program is divided into *pages* and the core is divided into *blocks* where the pages are loaded. There are two kinds of paging i.e. *simple* and *demand paging*. In simple all pages are loaded in the core and in demand, pages are fetched form secondary memory whenever it is needed (demanded).

In multiprocessing system where there is more than one processor, *traffic controller* coordinates the processors and processes. The processor time is allocated to the processes by a program called as *scheduler*. *Time sharing system* is one where the processer time is shared by more than one process.

File of information is allocated by *file system*. *File* or *segments* are the group of information that the user wished to treat as an entity. Files are of two types i.e. *directories* and *data or program*. Directories are file that contain location of other files.

Virtual memory consist of addresses that may be generated by a processor during execution of program. Virtual memory is a computer system technique which gives an application program the impression that it has contiguous working memory (an address space), while in fact it may be physically fragmented and may even overflow on to disk storage.

Function of operating system

- Job sequencing, scheduling, traffic controlling application.
- Input/ output programming.
- Protecting itself from uses and protecting the user from other users.
- Secondary storage management.
- Error handling.