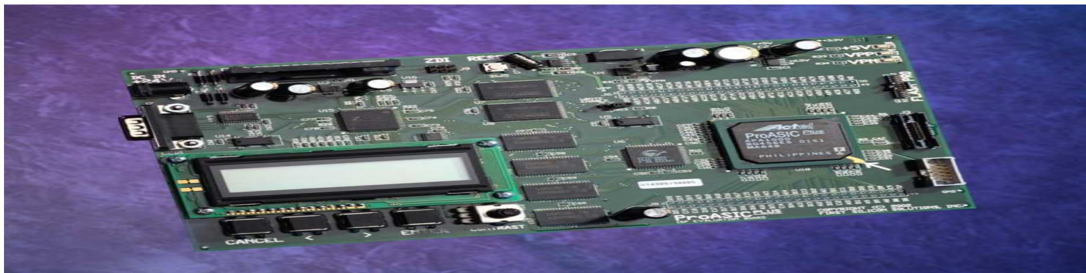
UNESCO-NIGERIA TECHNICAL & VOCATIONAL EDUCATION REVITALISATION PROJECT-PHASE II

NATIONAL DIPLOMA IN COMPUTER TECHNOLOGY



System Programming

COURSE CODE: COM 212

VERSION 1 DECMBER 2008

TABLE OF CONTENTS

WEEK 1 CONCEPTS OF SYSTEM PROGRAMMING

Brief overview of System programming ……………………………………5
Definition of Assembler ……………………………………6
Types of Assemblers and loaders ……………………………………7
Definition and examples of Assemblers ……………………………………8

WEEK 2 Operating System: 1-pass and 2-pass

OS definition ……………………………………10
1-pass assembler ……………………………………11
2-pass assembler ……………………………………..13

WEEK 3: Assembly Functions

Basic Assembler Functions ……………………………………..17
Assembler directives ……………………………………..17

WEEK 4 Basic Elements of and Assembly Program

Basic Elements ………………………………………20
Op code …………………………………….....21
Instruction cycle ………………………………………..22
Types of Instructions ………………………………………...22

WEEK 5 Sample Program Example

Program example and solution ………………………………………..26
Local labels ………………………………………..30
Examples of Local labels ………………………………………..32
Symbol Table ……………………………………… .32

WEEK 6 Assembler Functions

Functions of an assembler ……………………………………… 33
Assembler Modules ……………………………………… 34
Lexical Analysis ……………………………………… 35
Pass-1 module ……………………………………… 35
Pass-2 module ………………………………………..35
Main Program ………………………………………..35
Communicating between modules ………………………………………..36

WEEK 7 Interpretation, Translation, Compilation

Interpretation 38
Byte code Interpreter 38
Compiler 40
Compiler Output 41
Compiler vs Interpreter 41
One-pass vs Multipass compiler 41
Just-in-time compilation 42

WEEK 8 Front End and Back End

Front end 43
Back end 44
Concepts of tokens 45

WEEK 9 Error Checking, Utilities, Libraries

Error Checking 47
Perror 47
Assert 48
Libraries in Computing 49
Types of Libraries 49

WEEK 10 Operating Systems: Importance, Uses, Types

OS in hierarchy 54
Functions of OS 55
OS for PCs .... .56
MS Dos 56
MS Windows 57

WEEK 11 OS Services

OS services 59

WEEK 12 I/O Buffering and Files

I/O Buffering 61
Files devices 61
Spooling 60

WEEK 13 Interrupts

Interrupts 64
Interrupts handler 68

WEEK 14 Multiprogramming, Multitasking, Multiprocessing

Batch processing 70
Time sharing 70
Real time OS 70
Network OS 72
Examples of networks OS 74

WEEK 15 Review of Important Concepts.............................. ..76

WEEK 1

Learning Outcome for this week:
- The concept of system programming - The differences between systems programs and application programs - The difference between Assembler and operating systems. - Brief Review of Program Concepts

Teacher's Activities:

i) Define System Programming, Application Programming
ii) Differentiate between System Programming and Application Programming
iii) Give examples of each
iv) Define Assembler and operating Systems
v) Review what a program is.

DEFINITION

System programming (or **systems programming**) is the activity of programming system software. The primary distinguishing characteristic of systems programming when compared to application programming is that application programming aims at producing software which provides services to the user (e.g. word processor, Spreadsheets, databases, Accounting packages)

Systems programming aims at producing software which provides services to the computer hardware (e.g. disk defragmenter, Operating Systems…). It also requires a greater degree of hardware awareness that is to say it is machine dependent and the programmer needs to know the hardware within which the software will operate

BRIEF OVERVIEW OF SYSTEM PROGRAMMING

In system programming more specifically:

- The programmer will make assumptions about the hardware and other properties of the system that the program runs on, and will often exploit those properties (for example by using an algorithm that is known to be efficient when used with specific hardware)
- Usually a low-level programming language or programming language dialect is used and does the following:
  - Operate in resource-constrained environments
  - Is very efficient and has little runtime overhead
  - Has a small runtime library, or none at all
  - Allows for direct and "raw" control over memory access and control flow
  - Let the programmer write parts of the program directly in assembly language
- Debugging can be difficult if it is not possible to run the program in a debugger due to resource constraints. Running the program in a simulated environment can be used to reduce this problem.
- Note: All underlined words concepts will be discussed as the study progresses.

Systems programming is sufficiently different from application programming that programmers tend to specialize in one or the other.

In system programming, often limited programming facilities are available. The use of automatic garbage collection is not common and debugging is sometimes hard to do. The runtime library, if available at all, is usually far less powerful, and does less error checking. Because of those limitations, monitoring and logging are often used; operating systems may have extremely elaborate logging subsystems.

Implementing certain parts in operating system and networking requires systems programming

For historical reasons, some organizations use the term *systems programmer* to describe a job function which would be more accurately termed systems administrator. This is particularly true

in organizations whose computer resources have historically been dominated by mainframes, although the term is even used to describe job functions which do not involve mainframes.

DEFINITION OF ASSEMBLER

Computer science is not as precise a field as mathematics, so most definitions are not rigorous. An attempt to define Assembler can be formulated as follows: An assembler is a translator that translates source instructions (in symbolic language) into target instructions (in machine language), on a one to one basis.
This means that each source instruction is translated into exactly one target instruction.
This definition has the advantage of clearly describing the translation process of an assembler. It is not a precise definition, however, because an assembler can do (and usually does) much more than just translation. It off.ers a lot of help to the programmer in many aspects of writing the program. The many types of help offered by the assembler are grouped under the general term directives (or pseudo-instructions).
Another good definition of assemblers is can be profered thus: An assembler is a translator that translates a machine-oriented language into machine language.

This definition distinguishes between assemblers and compilers. Compilers being translators of problem-oriented languages or of machine-independent languages.
This definition, however, says nothing about the one-to-one nature of the translation, and thus ignores a most important operating feature of an assembler.
One reason for studying assemblers is that the operation of an assembler reflects the architecture of the computer. The assembler language depends heavily on the internal organization of the computer. Architectural features such as memory word size, number formats, internal character codes, index registers, and general purpose registers, affect the way assembler instructions are written and the way the assembler handles instructions and directives. This fact explains why there is an interest in assemblers today and why a course on assembler language is still required for many, perhaps even most, computer science degrees.
Today, assemblers are translators and they work on one program at a time. The tasks of locating, loading, and linking (as well as many other tasks) are performed by a loader.
A modern assembler has two inputs and two outputs. The first input is short, typically a single line typed at a keyboard. It activates the assembler and specifies the name of a source .le (the .le containing the source code to be assembled). It may contain other information that the assembler should have before it starts. This includes commands and specifications such as:

i) The names of the object file and listing file. Display (or do not display)the listing on the screen while it is being generated.
ii) Display all error messages but do not stop for any error.
iii) Save the listing file and do not print it (see figure below).

This program does not use macros. The symbol table is larger (or smaller) than usual and needs a certain amount of memory. All these terms will be explained elsewhere.

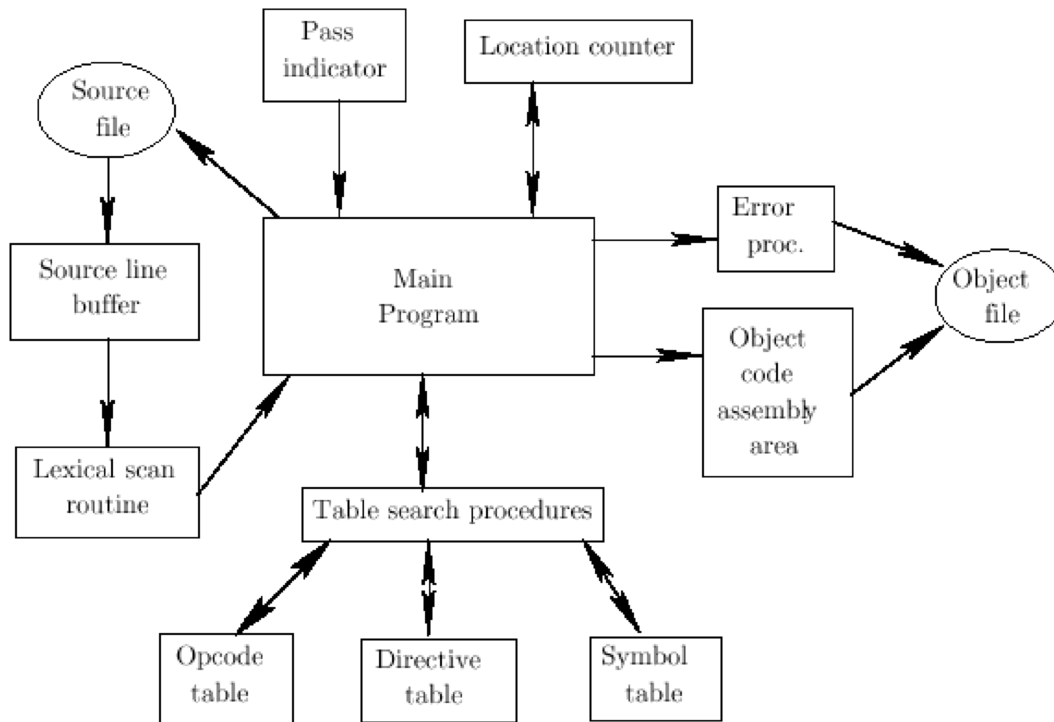


Figure: 1.1 Main Components and Operations of an Assembler

TYPES OF ASSEMBLERS AND LOADERS

1) A One-pass Assembler: One that performs all its functions by reading the source file once.
2) A Two-Pass Assembler: One that reads the source file twice.
3) A Resident Assembler: One that is permanently loaded in memory.
   Typically such an assembler resides in ROM, is very simple (supports only a few directives and no macros), and is a one-pass assembler.
4) A Macro-Assembler: One that supports macros
5) A Cross-Assembler: An assembler that runs on one computer and assembles programs for another. Many cross-assemblers are written in a higher-level language to make them portable. They run on a large machine and produce object code for a small machine.
6) A Meta-Assembler: One that can handle many different instruction sets.
7) A Disassembler: This, in a sense, is the opposite of an assembler. It translates machine code into a source program in assembler language.

We shall later discuss (1) and (2) in this course

WHAT IS A PROGRAM?

Depending on who you talk to, you will get very different answers. In operating systems, when a program is running, it is called a process. The program itself can be in many forms, i.e. machine code, assembly code, C code, C++ or Fortran, Java, etc…

Processors execute machine code ONLY. Machine code is written with binary machine words composed of op codes and operands. The op codes represent different instructions, for instance an arithmetic addition. Operands are operated on by the instruction. A machine word may look like:

```
10001000110101001010101101101011
```

In this case, **10001000** encodes the op code, 110101001010 and 101101101011 are the two operands. Most people don't want to write code in this format. Naturally, our forerunners sought a way out, using mnemonics to represent all those binary streams. Then the line above may become:

```
ADD AX, BX
```

These mnemonics and some other directive commands are collectively called assembly language. The assembly program now looks more readable and maintainable:

```
0000000000000100    MOV BX, VALUE1
0000000000001000    MOV AX, FACTOR
0000000000001100    MUL AX, BX
```

This is how CPU goes about to scale VALUE1 by FACTOR. VALUE1 x FACTOR doesn't make any sense to the processor. 04, 08 and 0C, written in HEX numbers, are memory addresses where corresponding lines in the program are stored.

Things became easier, but not easy enough and one could write a lot of programs using these commands But if one has a life away from a keyboard, there are problems:

1. The mnemonics in assembly language are in one-to-one correspondence with machine codes, which are processor dependent. If you move from Intel x86 to Motorola 68k, a new program has to be written. Here is a real assembly program that runs on 68k:

```
        ORG     $1000
N       EQU     5
CNT1    DC.B    0
CNT2    DC.B    0
ARRAY   DC.B    2,7,1,6,3

        ORG     $1500
MAIN    LEA     ARRAY,A2
        MOVE.B  #N,D1
        CLR     D6
        CLR     D7
```

```
            JSR     SORT
            STOP    #$2700

SORT        MOVE.B #0,D6
            MOVE.B #1,D7
LOOP        MOVE.B $0(A2,D6.W),D2
            MOVE.B $0(A2,D7.W),D3
            CMP.B   D2,D3
            BGT     EXCHANGE
            ADD.B #1,D7
            CMP.B #5,D7
            BLT     LOOP
            JMP     CHECK1
EXCHANGE  MOVE.B D2,$0(A2,D7.W)
           MOVE.B D3,(A2,D6.W)
           ADD.B #1,D7
           CMP.B #5,D7
           BLS    LOOP
CHECK1    ADD.B #1,D6
          MOVE.B D6,D7
          CMP.B #4,D6
          BLT    LOOP
          RTS
          END    $1500
```

Note, although assembly code are very close to what machine code is, they are still different. A n assembler converts assembly code into true machine code as we have seen earlier (the realm of binary!).

2. In the above program, the addresses are physical addresses, which correspond to individual bytes on your memory banks. If you have hard coded these addresses, you better only run one program on your processor at each time!

3. How tedious it is to program in assembly language! It may take 30 lines of code to just put a character on your console, provided that you are not making system calls. As an anecdote, while WordPerfect and Word look very similar, one is written in assembly and the other in C.

WEEK TWO

Learning Outcome for this week:
- An insight into the concept of Operating System - The meaning and work of 1-pass Assembler. - The meaning of a 2-pass Assembler

Teacher's Activities:

Go through the program example and give a line by line explanation of the working of a 1-pass assembler and a 2-pass assembler

ASSEMBLER

We have seen that a typically modern **assembler** creates object code by translating assembly instruction mnemonics into op-codes, and by resolving symbolic names for memory locations and other entities. The use of symbolic references is a key feature of assemblers, saving tedious calculations and manual address updates after program modifications. Most assemblers also include macro facilities for performing textual substitution—e.g., to generate common short sequences of instructions to run inline, instead of in a subroutine.

OPERATING SYSTEM

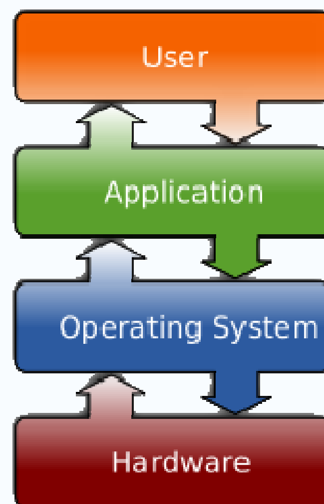


Figure 2.1 Location of Operating System

An **operating system** (commonly abbreviated *OS* and *O/S*) is the software component of a computer system that is responsible for the management and coordination of activities and the sharing of the limited resources of the computer. The operating system acts as a host for applications that are run on the machine. As a host, one of the purposes of an operating system is

to handle the details of the operation of the hardware. This relieves application programs from having to manage these details and makes it easier to write applications. Almost all computers, including handheld computers, desktop computers, supercomputers, and even video game consoles, use an operating system of some type. Some of the oldest models may however use an embedded operating system, that may be contained on a compact disk or other data storage device.

Common contemporary operating systems include Microsoft Windows, Mac OS, Linux and Solaris. Microsoft Windows has a significant majority of market share in the desktop and notebook computer markets, while servers generally run on Linux or other Unix-like systems. Embedded device markets are split amongst several operating systems.[1] [2]

We will now discuss a 1-pass and 2-pass assembler and see how they works,

THE ONE-PASS ASSEMBLER

As its name implies, this assembler reads the source file once. During that single pass, the assembler handles both label definitions and assembly. The only problem is future symbols and, to understand the solution Let 's consider the following example:

```
LC
36 BEQ AB ;BRANCH ON EQUAL
.
.
67 BNE AB ;BRANCH ON NOT EQUAL
.
.
89 JMP AB ;UNCONDITIONALLY
.
.
126 AB  anything
```

Symbol AB is used three times as a future symbol. On the first reference, when The LC happens to stand at 36, the assembler searches the symbol table for AB , does not find it, and therefore assumes that it is a future symbol. It then inserts AB into the symbol table but, since AB has no value yet, it gets a special type. Its type is U (underlined). Even though it is still underlined, it now occupies an entry in the Symbol table, an entry that will be used to keep track of AB as long as it is a future symbol. The next step is to set the 'value 'field of that entry to 36 (the current value of the LC).This means that the symbol table entry for AB is now pointing to the instruction in which AB is needed. The 'value 'field is an ideal place for the pointer since it is the right size, it is currently empty, and it is associated with AB .The BEQ instruction itself is only partly assembled and is stored, incomplete, in memory location 36.The field in the instruction were the value of AB should be stored (the address field),remains empty.

When the assembler gets to the BNE instruction (at which point the LC stands at 67), it searches the symbol table for AB , and fi.nds it. However, AB has a type of U , which means that it is a future symbol and thus its ‘value ’field (=36) is not a value but a pointer .It should be noted that, at this point, a type of U does not necessarily mean an underlined symbol. While the assembler is performing its single pass, any underlined symbols must be considered future symbols. Only at the end of the pass can the assembler identify underlined symbols (see below).The assembler handles the BNE instruction by:

i) Partly assembling it and storing it in memory location 67.

ii) Copying the pointer 36 from the symbol table to the partly assembled instruction in location 67. The instruction has an empty field (where the value of AB should have been), where the pointer is now stored. There may be cases where this field

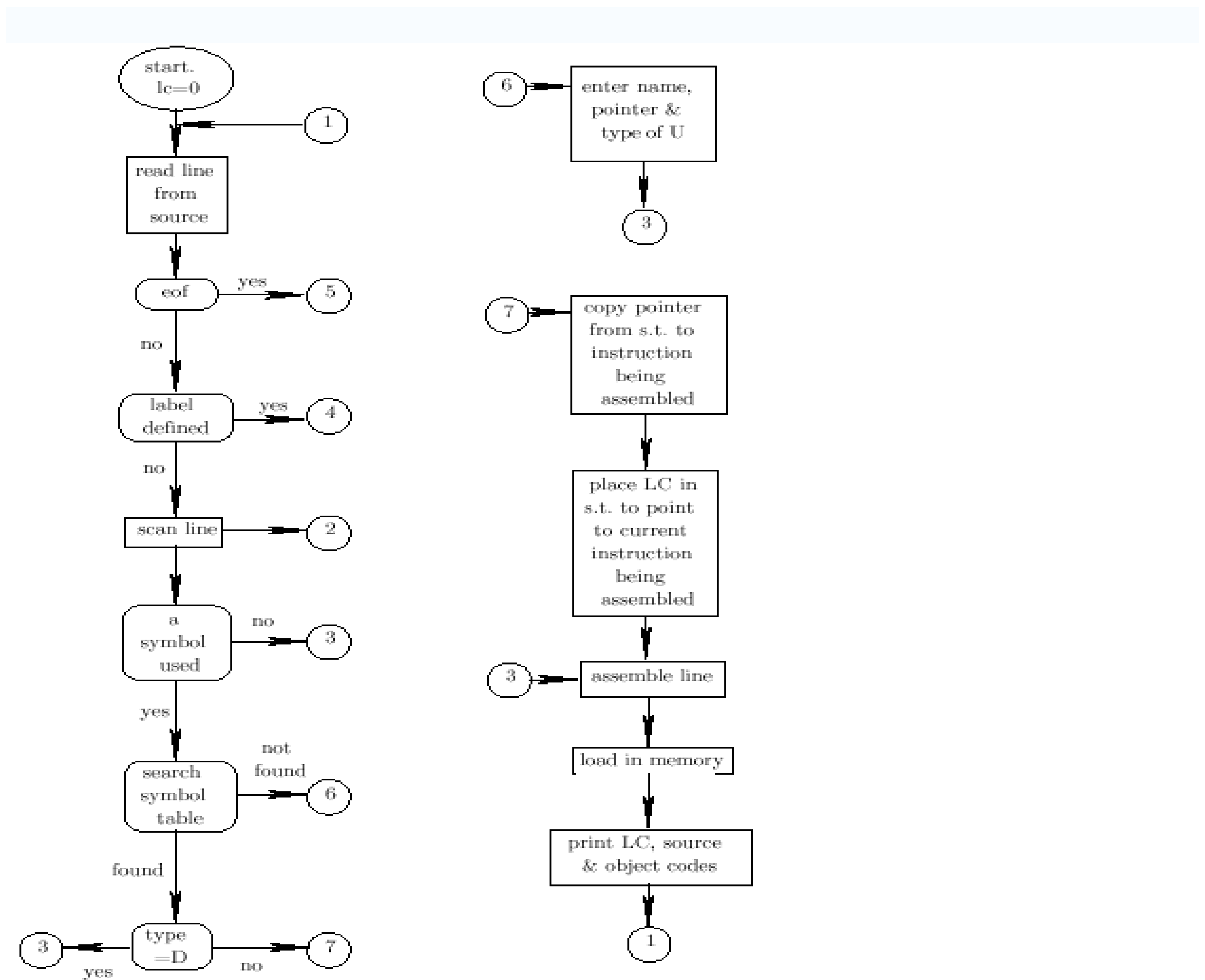


Figure 2.2 A 1-Pass Assembler

TWO 2-PASS ASSEMBLER

A two-pass assembler is easier to understand .Such an assembler performs two passes over the source file. In the fi.rst pass it reads the entire source file, looking only for label definitions. All labels are collected, assigned values, and placed in the symbol table in this pass. No instructions are assembled and, at the end of the pass, the symbol table should contain all the labels defined in the program. In the second pass, the instructions are again read and are assembled, using the symbol table.

Exercise What if a certain symbol is needed in pass 2, to assemble an instruction, and is not found in the symbol table?

To assign values to labels in pass 1, the assembler has to maintain the LC. This in turn means that the assembler has to determine the size of each instruction (in words), even though the instructions themselves are not assembled. In many cases it is easy to figure out the size of an instruction. On the IBM 360 for example, the mnemonic determines the size uniquely. An assembler for this machine keeps the size of each instruction in the Op-Code table together with the mnemonic and the Op-Code . On the DEC PDP-11 the size is determined both by the type of the instruction and by the addressing mode(s) that it uses. Most instructions are one word (16-bits) long. However, if they use either the index or index deferred modes, one more word is added to the instruction. If the instruction has two operands (source and destination) both using those modes, its size will be 3 words. On most modern microprocessors, instructions are between 1 and 4 bytes long and the size is determined by the Op-Code and the specific operands used. This means that, in many cases, the assembler has to work hard in the first pass just to determine the size of an instruction. It has to look at the mnemonic and, sometimes, at the operands and the modes, even though it does not assemble the instruction in the first pass. All the information about the mnemonic and the operand collected by the assembler in the first pass is extremely useful in the second pass, when instructions are assembled. This is why many assemblers save all the information collected during the first pass and transmit it to the second pass through an intermediate file .Each record on the intermediate file contains a copy of a source line plus all the information that has been collected about that line in the first pass. At the end of the fi.rst pass the original source .le is closed and is no longer used. The intermediate .le is reopened and is read by the second pass as its input file.
A record in a typical intermediate file contains:

i) The record type. It can be an instruction, a directive, a comment, or an invalid line.
ii) The LC value for the line.
iii) A pointer to a specific entry in the Op-Code table or the directive table.

The second pass uses this pointer to locate the information necessary to assemble or execute the line.

More sophisticated high-level assemblers provide language abstractions such as:

- Advanced control structures
- High-level procedure/function declarations and invocations
- High-level abstract data types, including structures/records, unions, classes, and sets
- Sophisticated macro processing
- Object-Oriented features such as encapsulation, polymorphism, inheritance, interfaces

Note In normal professional usage, the term **assembler** is often used ambiguously: It is frequently used to refer to an assembly language itself, rather than to the assembler utility. Thus: "CP/CMS was written in S/360 assembler" as opposed to "ASM-H was a widely-used S/370 assembler."

- A copy of the source line.
  Notice that a label, if any, is not used by pass 2 but must be included in the intermediate file since it is needed in the final listing.

There can be two problems with labels in the first pass; multiply-defined labels and invalid labels Before a label is inserted into the symbol table, the table has to be searched for that label. If the label is already in the table, it is doubly (or even multiply-) defined. The assembler should treat this label as an error and the best way of doing this is by inserting a special code in the type .eld in the symbol table.
Thus a situation such as:
AB ADD 5,X
.
.
AB SUB 6,Y
.
.
JMP AB
will generate the entry:
name value type
AB .MTDF
in the symbol table.
Labels normally have a maximum size (typically 6 or 8 characters), must start with a letter, and may only consist of letters, digits, and a few other characters. Labels that do not conform to these rules are invalid labels and are normally considered a fatal error. However, some assemblers will truncate a long label to the maximum size and will issue just a warning, not an error, in such a case.

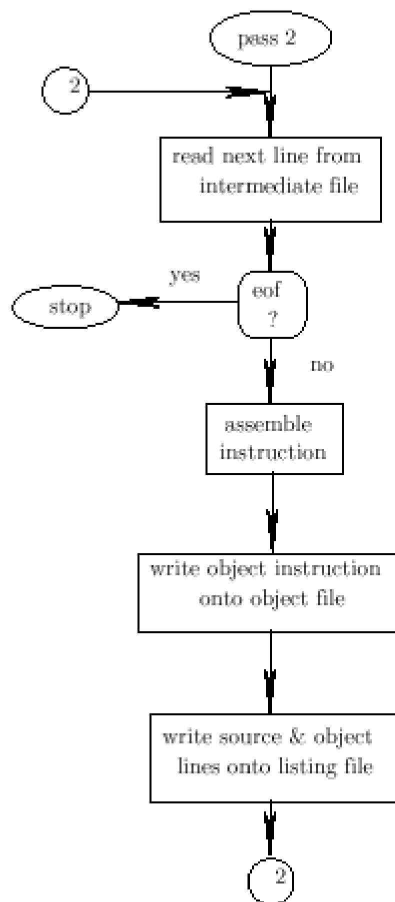
Exercise What is the advantage of allowing characters other than letters and digits in a label?

The only problem with symbols in the second pass is bad symbols .These are either multiply-defined or underlined symbols. When a source line uses a symbol in the operand field, the assembler looks it up in the symbol table. If the symbol is found but has a type of MTDF ,or if the symbol is not found in the symbol table (i.e., it has not been defined), the assembler responds as follows.

- It flags the instruction in the listing .le. It assembles the instruction as far as possible, and writes it on the object file.
- It flags the entire object file. The flag instructs the loader not to start execution of the program. The object .le is still generated and the loader will read and load it, but not start it. Loading such a .le may be useful if the user wants to see a memory map assemblers.

This point is the reason why a one-pass assembler can only produce an absolute object .le (which has only limited use), whereas a two-pass assembler can produce a re-locatable object file, which is much more general.

Exercise What would be good Pascal declarations for such a future symbol list:
a. Using absolute pointers.
b. Housed in an array.



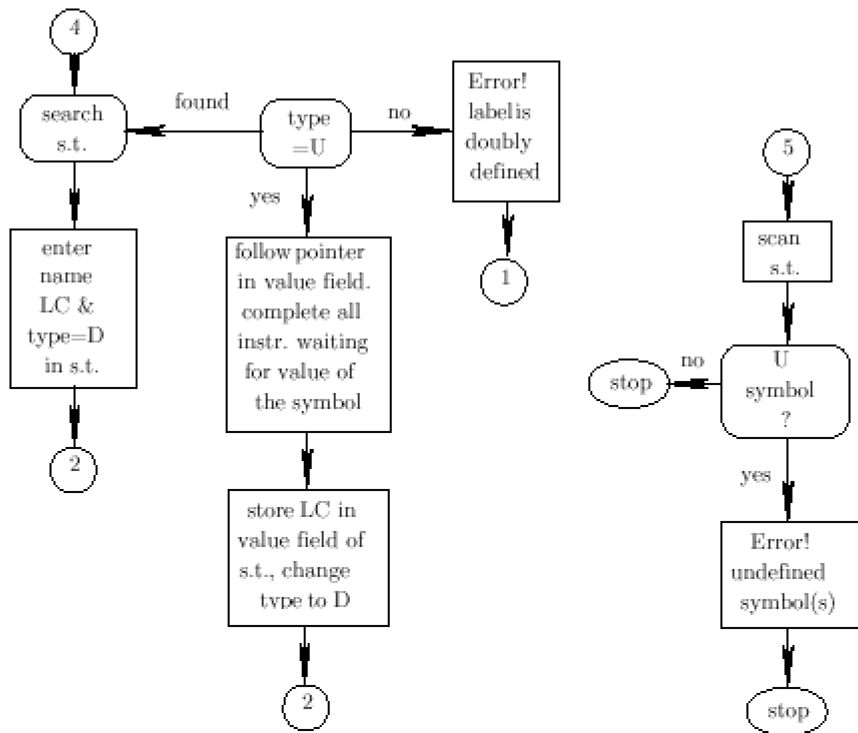


Figure: 2.3 Operations of a 2-pass assembler

WEEK THREE

Learning outcome for this week:

- Review the work of a 2-pass assembler
- Basic Assembly Functions
- Assembler directives
- An Assembler example program

Teacher's Activities:

Outline

i) Basic assembler functions
ii) A simple SIC assembler example

BASIC ASSEMBLER FUNCTIONS

1) Translating mnemonic operation codes to their machine language equivalents
2) Assigning machine addresses to symbolic labels

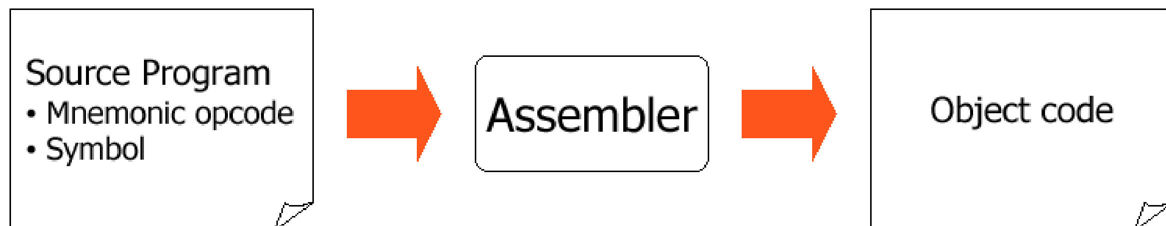


Figure 3.1 Basic Assembler Functions

OTHER FUNCTIONS INCLUDE:

- Converting mnemonic operation codes to their machine language equivalents
- Converting symbolic operands to their equivalent machine addresses
- Deciding the proper instruction format
- Converting the data constants to internal machine representations
- Writing the object program and the assembly listing

ASSEMBLER DIRECTIVES

- Assembler directives are pseudo instructions
- They provide instructions to the assembler itself
- They are not translated into machine operation codes
- Example for SIC assembler directive
- START : specify name & starting address
- END : end of source program, specify the first execution instruction
- BYTE, WORD, RESB, RESW
- End of record : a null char (00)
- End of file : a zero- length record

NOTE: This program example will be theoretically explained in the class and further used in a lab as practical example.

PROGRAM EXAMPLE

```
5      COPY     START   1000       COPY FILE FROM INPUT TO OUTPUT
10     FIRST    STL     RETADR     SAVE RETURN ADDRESS
15     CLOOP    JSUB    RDREC      READ INPUT RECORD
20              LDA     LENGTH     TEST FOR EOF (LENGTH = 0)
25              COMP    ZERO
30              JEQ     ENDFIL     EXIT IF EOF FOUND
35              JSUB    WRREC      WRITE OUTPUT RECORD
40              J       CLOOP      LOOP
45     ENDFIL   LDA     EOF        INSERT END OF FILE MARKER
50              STA     BUFFER
55              LDA     THREE      SET LENGTH = 3
60              STA     LENGTH
65              JSUB    WRREC      WRITE EOF
70              LDL     RETADR     GET RETURN ADDRESS
75              RSUB               RETURN TO CALLER
80     EOF      BYTE    C'EOF'
85     THREE    WORD    3
90     ZERO     WORD    0
95     RETADR   RESW    1
100    LENGTH   RESW    1          LENGTH OF RECORD
105    BUFFER   RESB    4096       4096-BYTE BUFFER AREA
110    .
115    .        SUBROUTINE TO READ RECORD INTO BUFFER
120    .
125    RDREC    LDX     ZERO       CLEAR LOOP COUNTER
130             LDA     ZERO       CLEAR A TO ZERO
135    RLOOP    TD      INPUT      TEST INPUT DEVICE
140             JEQ     RLOOP      LOOP UNTIL READY
145             RD      INPUT      READ CHARACTER INTO REGISTER A
150             COMP    ZERO       TEST FOR END OF RECORD (X'00')
155             JEQ     EXIT       EXIT LOOP IF EOR
160             STCH    BUFFER,X   STORE CHARACTER IN BUFFER
165             TIX     MAXLEN     LOOP UNLESS MAX LENGTH
170             JLT     RLOOP        HAS BEEN REACHED
175    EXIT     STX     LENGTH     SAVE RECORD LENGTH
180             RSUB               RETURN TO CALLER
185    INPUT    BYTE    X'F1'      CODE FOR INPUT DEVICE
190    MAXLEN   WORD    4096
195    .
```

Forward reference

PURPOSE OF EXAMPLE PROGRAM

- Reads records from input device (code F1)
- Copies them to output device (code 05)
- At the end of the file, writes EOF on the output device, then RSUB to the operating system
- Data transfer (RD, WD)
- A buffer is used to store record. Buffering is necessary for different I/ O rates
- The end of each record is marked with a null character (00) 16
- The end of the file is indicated by a zero- length record
- Subroutines (JSUB, RSUB) RDREC, WRREC
- Save link register first before nested jump

WEEK FOUR

Learning outcome for this week
- The general format of an Assembly program statement - The purpose of each field of Assembly language statement - The meaning of symbolic operations, types of operations, program counter - Registers, Instruction cycle - Types of Instructions

TEACHER'S ACTIVITIES

- Describe the general format of an Assembly Language Program Statement (label, op-code, Addresses, operands, pseudo-operations , pseudo-instructions)
- Explain the purpose of each field of assembly language statement
- Example of a list of op-code to write a simple program

BASIC ELEMENTS OF AN ASSEMBLY PROGRAM

Instructions (statements) in assembly language are generally very simple, unlike those in high-level languages. Each instruction typically consists of an *operation* or *op-code* plus zero or more *operands*. Most instructions refer to a single value, or a pair of values.

Generally, an op-code is a symbolic name for a single executable machine language instruction. Operands can be either immediate (typically one byte values, coded in the instruction itself) or the addresses of data located elsewhere in storage. This is determined by the underlying processor architecture: the assembler merely reflects how this architecture works.

Most modern assemblers also support *pseudo-operations*, which are directives obeyed by the assembler at assembly time instead of the CPU at run time. (For example, pseudo-ops would be used to reserve storage areas and optionally set their initial contents.) The names of pseudo-ops often start with a dot to distinguish them from machine instructions.

Some assemblers also support *pseudo-instructions*, which generate two or more machine instructions.

Symbolic assemblers allow programmers to associate arbitrary names (*labels* or *symbols*) with memory locations. Usually, every constant and variable is given a name so instructions can reference those locations by name, thus promoting self-documenting code. In executable code,

the name of each subroutine is associated with its entry point, so any calls to a subroutine can use its name. Inside subroutines, GOTO destinations are given labels. Some assemblers support *local symbols* which are lexically distinct from normal symbols (e.g., the use of "10$" as a GOTO destination).

Most assemblers provide flexible symbol management, allowing programmers to manage different name spaces, automatically calculate offsets within data structures, and assign labels that refer to literal values or the result of simple computations performed by the assembler. Labels can also be used to initialize constants and variables with re-locatable addresses.

Assembly languages, like most other computer languages, allow comments to be added to assembly source code that are ignored by the assembler. Good use of comments is even more important with assembly code than with higher-level languages, as the meaning of a sequence of instructions is harder to decipher from the code itself.

Wise use of these facilities can greatly simplify the problems of coding and maintaining low-level code. *Raw* assembly source code as generated by compilers or dis-assemblers — code without any comments, meaningful symbols, or data definitions — is quite difficult to read when changes must be made.

Op-code

In computer technology, an **op-code** (**op**eration **code**) is the portion of a machine language instruction that specifies the operation to be performed. Their specification and format are laid out in the instruction set architecture of the processor in question (which may be a general CPU or a more specialized processing unit). Apart from the op-code itself, an instruction normally also has one or more specifiers for operands (i.e. data) on which the operation should act, although some operations may have *implicit* operands, or none at all. There are instruction sets with nearly uniform fields for op-code and operand specifiers, as well as others (the x86 architecture for instance) with a more complicated, varied length structure. [1]

Depending on architecture, the **operands** may be register values, values in the stack, other memory values, I/O ports, etc, specified and accessed using more or less complex addressing modes. The types of **operations** include arithmetic, data copying, logical operations, and program control, as well as special instructions (such as CPUID and others).

Assembly language commands have the following syntax: a 1-digit op code followed by a 2-digit memory address or operand. Each command is stored in a separate memory block. For example: 20 0 68 means that in memory block 20, we have a command with the op code 0 which executes on memory block 68.

Finally, the ***program counter*** keeps track of the memory block that contains the next command to be executed. By default the program counter is incremented by 1 after each command execution because we typically store commands in consecutive memory blocks. However, the program counter can jump to any memory block depending on the program's sequence of execution.

- **IR** -- The instruction register. It holds the instruction currently being executed.
- **CSR** -- The control status register. It contains information pertaining to the execution of the current and previous instructions.

THE INSTRUCTION CYCLE

The computer's operation consists of running instructions repetitively. This is known as the instruction cycle. The instruction cycle consists of 4 general phases:

- 1. Decode instruction (in **IR**)
- 2. Execute instruction
- 3. Determine next instruction
- 4. Load next instruction into the **IR**

What is an instruction? Like everything else, it's a sequence of 0's and 1's. Instructions are stored as part of a program's memory, and the instruction that is pointed to by the **pc** register is the one that gets loaded into the **IR** for execution.

In other words, if the **pc** contains the value **0x2040**, then the **IR** is executing the instruction contained in the 4 bytes starting at memory address **0x2040**.

An *assembler* converts assembly code into the proper 0's and 1's that compose the program. If you call **gcc** with the **-S** flag, it will produce a **.s** file containing the assembler for that program in C language for instance. Without the **-S** flag, it produces the instructions directly.

TYPES OF INSTRUCTIONS

1. Memory <-> Register instructions:

ld mem -> %reg   Load the value of the register from memory.

st %reg -> mem   Store the value of the register into memory.

There are a few ways to address memory:

st %r0 -> i   Store the value of register r0 into the memory location of global variable i.

st %r0 -> [r1]   Treat the value of register r1 as a pointer to a memory location, and store the value of r0 in that memory location.

st %r0 -> [fp+4]   Treat the value of the frame pointer as a pointer to a memory location, and store the value of r0 in the memory location 4 bytes after that location. You can use any value, positive or negative, not just 4. However, you cannot use a register (i.e.

you can't do st %r0 -> [fp+r2]).
This only works with the frame pointer. It does not work with any other register.

st %r0 -> [sp]-- Treat the value of register sp as a pointer to a memory location, store the value of r0 into that memory location, and then subtract 4 to the value of sp.

st %r0 -> ++[sp] Treat the value of register sp as a pointer to a memory location. First, add 4 to that value, then store the value of r0 into that memory location.

2. Register <-> Register instructions:

mov %reg -> %reg Copy a register's value to another
mov #val -> %reg register, or set its value to a constant.

All arithmetic goes from register to register:

add %reg1, %reg2 -> %reg3 Add reg1 & reg2 and put the sum in reg3.
sub %reg1, %reg2 -> %reg3 Subtract reg2 from reg1.
mul %reg1, %reg2 -> %reg3 Multiply reg1 & reg2.
idiv %reg1, %reg2 -> %reg3 Do integer division of reg2 into reg1.
imod %reg1, %reg2 -> %reg3 Do reg1 mod reg2.

There are two special instructions that let you perform addition and subtraction on the stack pointer:

push %reg This subtracts the value of stack pointer
push #val by value contained in reg or the constant defined in val.

pop %reg This adds the value of %reg or #val
pop #val to the stack pointer.

3. Control instructions

jsr a Call the subroutine starting at instruction a.
ret Return from a subroutine.

There are also "compare" and "branch" instructions, which is how you implement for and if statements, but I won't go over them yet.

Finally, there are also "directives" which are not really code, but specify that memory must be allocated for variables. For example:

.globl i Allocate 4 bytes in the globals segment for the variable i.

The program counter points to where the instruction register must go to load its value. On normal instructions, the **pc** is incremented by 4 so that the next instruction can be loaded. On control instructions, the **pc** gets a new value, allowing the machine to call subroutines, perform "if-then" statements, etc.

Here is the list of the op codes we will be using to construct assembly language programs.

Op Code	**Function Abbreviation**	**Task**
0	INP	Stores the next value from the input stream into the specified memory block. Every time this op code is called the next value in the list is read and stored. For example, 001 means store the next value in the input stream in memory block 01.
1	CLA	Clears the accumulator (reset content to 0) and adds the content of the specified memory location to the accumulator.
2	ADD	Adds the content of the specified memory location to the accumulator without clearing the accumulator first.
3	TAC	Tests the value in the accumulator, if it is negative it jumps to the specified memory block. For example, 325 means check the content of the accumulator, if it is negative set the program counter to 25, go to memory block 25, and execute the command stored in it.
4	SFT	This op code is followed by two 1-digit operands. The first indicates the number of places to shift the content of the accumulator to the left and the second indicates the number of places to shift the content of the accumulator to the right. The left shift is implemented before the right shift. Note that you do not have to have non-zero shifts in both directions. You could set the value to 0 for the direction in which you do not wish to shift. For example consider the instruction 401 and the value in the accumulator is 2121. We see that the amount of shift to the left is 0 so no shift to the left. The amount of shift to the right is 1 so we move all digits of the accumulator to the right one place dropping the rightmost digit we get 212. Shifting 1 place to the right is analogous to getting the

		quotient of the division of the accumulator's content by 10. Let's try another example, 420. The accumulator again contains 2121 initially. The amount of shift to the left is 2 so we move all digits of the accumulator's content to the left two places and add a zero to the right for every place we shift we get 2100. The first 2 digits are truncated because the accumulator is only four positions in length. Shifting 1 place to the left is like multiplying the accumulator's content by 10.
5	OUT	Outputs the content of the specified memory block. For example, 501 means output the content of memory block 01.
6	STO	Stores the content of the accumulator in the specified memory block. For example, 601 means store the content of the accumulator into memory block 01.
7	SUB	Subtracts the content of the specified memory block from the content of the accumulator. For example, 701 means subtract the content of memory block 01 from the content of the accumulator.
8	JMP	Jumps to the specified memory block unconditionally. For example, 825 means set the program counter to 25, go to memory block 25, and execute the command stored in it.
9	HRS	Halts or ends the program. We always use 900.

Rules & Assumptions:

- You may always assume that memory block 00 contains the value 1 at the start of the program. It remains 1 unless you specifically overwrite it.
- You cannot output directly from the accumulator. You must store the accumulator content first to a memory block and then output from the memory block.
- It is a good rule of thumb to always use op code 1 when you want to add the first number in your program to the accumulator. That way you ensure that you are not computing based on a previous value stored in the accumulator.
- You may start your program in any empty memory block you want. You may also use any empty memory block to store data.

WEEK FIVE

Learning Outcome for this week:
- Write a program with no input but output - Symbol table and Local labels

TEACHER'S ACTIVITIES

- Explain step by step the following program to students
  Explain the meaning and uses of local labels

Note: There will be further practice of this program in a lab class

Sample Program

Write a program that has no input and outputs a continuous string of numbers, starting with 1, and counting by one. In other words, your output should be 1 2 3 4 5 6 ….

Solution

Step 1: 1 00 We clear the accumulator resetting it to zero and then load the content of memory block 00 into the accumulator. The accumulator now contains 001.

Program Counter	25	**Accumulator**	1
Memory Block	**Block Content**	**Memory Block**	**Block Content**
00	1	30	827
01		31	
02		32	
……		33	
25	100	34	
26	500	35	
27	200	36	
28	602	37	
29	502	38	

Output Stream:

Step 2: 5 00 We output the contents of memory block 00. We do this because we need a 1 as the first number in our output stream. Memory block 00 already contains a 1 so we can output directly from it.

Program Counter	26	Accumulator	1
Memory Block	**Block Content**	**Memory Block**	**Block Content**
00	1	30	827
01		31	
02		32	
……		33	
25	100	34	
26	500	35	
27	200	36	
28	602	37	
29	502	38	

Output Stream: 1

Step 3: 2 00 To construct the rest of the output stream we need to increment each output value by 1 to create the next value in the stream. Since we have 1 in the accumulator we need to add another 1 to it to get the next output value, a 2. So in this instruction we add the content of memory block 00, a 1, to the contents of the accumulator without resetting it first. The accumulator content is now 2.

Program Counter	27	Accumulator	2
Memory Block	**Block Content**	**Memory Block**	**Block Content**
00	1	30	827
01		31	
02		32	
……		33	
25	100	34	
26	500	35	
27	200	36	
28	602	37	
29	502	38	

Step 4: 6 02 We want to output the next number of the output stream, 2, which is now in the accumulator. Remember that we cannot output directly from the accumulator so we first need to store its content to a memory block. In this instruction, we select memory block 02 for this purpose. You could have chosen any empty block. So now memory block 02 contains a 2.

Program Counter	28	**Accumulator**	2
Memory Block	**Block Content**	**Memory Block**	**Block Content**
00	1	30	827
01		31	
02	2	32	
……		33	
25	100	34	
26	500	35	
27	200	36	
28	602	37	
29	502	38	

Step 5: 5 02 We can now output the next number of the output stream from memory block 02.

Program Counter	29	**Accumulator**	2
Memory Block	**Block Content**	**Memory Block**	**Block Content**
00	1	30	827
01		31	
02	2	32	
……		33	
25	100	34	
26	500	35	
27	200	36	
28	602	37	
29	502	38	

Output Stream: 1 2

Step 6: 8 27 Since we need to increments again by 1 to get the next output value we can just repeat the increment (block 27)-store (block 28)-output (block 29) sequence of instructions infinitely to construct the rest of the output stream. The first instruction in this sequence, increment by 1, resides in memory block 27. Therefore, that's the location to which we need to jump to repeat the sequence.

Program Counter	30	Accumulator	2
Memory Block	**Block Content**	**Memory Block**	**Block Content**
00	1	30	827
01		31	
02	2	32	
……		33	
25	100	34	
26	500	35	
27	200	36	
28	602	37	
29	502	38	

Follow the changes in the next few steps of the program. Note the pattern that develops. I have only shown the increment and store steps (lumped into one figure) to emphasize how the contents of the accumulator and memory block 02 change for every execution of the increment-store-output sequence.

Steps 7 & 8

Program Counter	27	Accumulator	3
Memory Block	**Block Content**	**Memory Block**	**Block Content**
00	1	30	827
01		31	
02	3	32	
……		33	
25	100	34	
26	500	35	
27	200	36	
28	602	37	
29	502	38	

Output Stream (after step 9): 1 2 3

…… Steps 11 & 12

Program Counter	27	Accumulator	4
Memory Block	**Block Content**	**Memory Block**	**Block Content**
00	1	30	827
01		31	
02	4	32	
……		33	
25	100	34	
26	500	35	
27	200	36	
28	602	37	
29	502	38	

Output Stream (after step 13): 1 2 3 4

…… Steps 15 & 16

Program Counter	27	Accumulator	5
Memory Block	**Block Content**	**Memory Block**	**Block Content**
00	1	30	827
01		31	
02	5	32	
……		33	
25	100	34	
26	500	35	
27	200	36	
28	602	37	
29	502	38	

Output Stream (after step 17): 1 2 3 4 5

The sequence keeps repeating infinitely.

LOCAL LABELS

In principle, a label may have any name that obeys the simple syntax rules of the assembler. In practice, though, label names should be descriptive. Names such as DATE,MORE,LOSS,RED are preferable to A001,A002,...
There are exceptions, however. The use of the non-descriptive label A1 in the following example:
.

JMP A1
DDCT DS 12 reserve 12 locations for array DDCT
A1 .
.
is justified since it is only used to jump over the array DDCT .(Note that the array 's name is descriptive, possibly meaning deductions or double-dictionary) We say that A1 is used only locally, to serve a limited purpose.

As a result, many assemblers support a feature called local labels. .The main idea is that if a label is used locally and does not require a descriptive name, why not give it a name that will signify this fact. Names such as 1H ,2H for the local labels were used. The name of a local label in our examples is a single decimal digit. When such a label is referred to (in the operand field), the digit is followed by either B or F (for Backward or Forward).

LC
.
.
13 1:...
.
.
17 JMP 1F jump to 24
.
.
24 1:LOD R2,1B 1B here means address 13
.
.
31 1:ADD R1,2F 2F is address 102
.
.
102 2:DC 1206,-17
.
.
115 SUB R3,2B-1 102-1=101

EXAMPLE.LOCAL LABELS.

The example shows that local labels is a simple, useful, concept that is easy to implement. In a two-pass assembler, each local label is entered into the symbol table as any other symbol, in pass 1.Thus the symbol table in our example contains:

Symbol Table
n v
113
124
131
2 102

The order of the labels in the symbol table is important. If the symbol table is sorted between the two passes, all occurrences of each octal label should remain sorted by value. In pass 2,when an

instruction uses a octal label such as 1F , the assembler identifies the specific occurrence of label 1 by comparing all local labels 1 to the current value of the LC. The first such instruction in our example is the ‘JMP 1F ’at LC=17.Clearly,the assembler should look for a octal label with the name ‘1 ’and a value .17.The smallest such label has value 24.In the second case, LC=24 and the assembler is looking for a 1B .It needs the label with name ‘1 ’ and a value which is the largest among all values <24.It therefore identifies the label as the ‘1 ’at 13.

EXERCISE

If we modify the instruction at 24 above to read 1:LOD R2,1F would the 1F refer to address 31 or 24?

In a one-pass assembler, again the labels are recognized and put into the symbol table in the single pass. An instruction using a octal label iB is no problem, since is needs the most recent occurrence of the local label ‘1 ’in the table. An instruction using an iF is handled like any other future symbol case. An entry is opened in the symbol table with the name iF ,a type of U ,and a value which is a pointer to the instruction.

In the example above, a snapshot of the symbol table at LC=32 is:

SYMBOL TABLE

n v t
113 D
124 D
1 31 D 31 is the value of the third 1
2 31 U 31 is a pointer to the ADD instruction

An advantage of this feature is that the local labels are easy to identify as such, since their names start with a digit. Most assemblers require regular label names to start with a letter. In modern assemblers, local labels sometimes use a syntax different from the one shown here.

The LC as a local symbol

Virtually all assemblers allow a notation such as ‘BPL *+6 ’where ‘*’stands for the current value of the LC. The operand in this case is located at a point 6 locations following the BPL instruction.

The LC symbol can be part of any address expression and is, of course, re-locatable. Thus *+A is valid if A is absolute, while *-A is always okay (and is absolute if A is relative, relative if A is absolute).This feature is easy to implement. The address expression involving the ‘*’is calculated, using the current value of the LC, and the value is used to assemble the instruction, or execute the directive, on the current source line. Nothing is stored in the symbol table.

Some assemblers use the asterisk for multiplication, and may designate the period ‘.’or the ‘$’for the LC symbol. On the PDP-11 the notation ‘X:.=.+8 ’is used to increment the LC by 8,and thus to reserve eight ocations (compare this to the DS directive).

EXERCISE

What is the meaning of JMP *,JMP *-*?

WEEK SIX

Learning Outcome for this week:
- The Assembler Functions - Assembler modules

ASSEMBLER FUNCTIONS

Two important steps are involved in designing software: dividing the software into smaller, more manageable components or modules, and determining how the modules will communicate. For both of these steps, the designer needs to keep the major kinds of software functionality in mind. The following are the primary functions of an assemblers.

- Generate machine code output

  This is the primary purpose of an assembler. The assembler usually generates a file copy of data and machine instructions that will later be loaded into a computer in preparation for execution. The file copy must also contain a starting address - the address of the first instruction to be executed.

- Provide program error information for the programmer

  Assembly language programming is difficult. Assembly language programmers make mistakes. Some of these mistakes can be caught by the assembler. The ease of assembly language programming is dependent to a large extent on the quality of assembler error messages.

- Provide machine code information for the programmer

  The assembler cannot catch all programmer errors. Some can only be detected by executing the assembled program. Then assembler can, however, provide information about the machine code that aids the programmer in debugging runtime errors. For difficult debugging problems, the programmer may need to know what code was generated and where data and instructions are located in memory.

- Assign memory for data and instructions

  Early assemblers force programmers to assign memory addresses for all data and keep track of addresses assigned for instructions. Modern assembler allow programmers to use symbols (usually statement labels) to represent addresses for data or branch or jump targets. This makes the programmer's life much simpler. However, addresses are required for machine code generation. Thus the assembler must pick up the responsibity of assigning addresses to program symbols. These assignments must be remembered for use when the symbols appear in instruction operands.

DEALING WITH CHARACTERS

The input to an assembler consists of a stream of characters which represent assembly information in several different ways: integers, real numbers, labels, quoted characters and strings, register names, and various kinds of punctuation. If character processing is mixed in with algorithms for building symbol tables or algorithms for code generation, the result is a complex mess that would challenge even the best programmers. The problem is that this organization (or disorganization) forces programmers to deal with two distinct kinds of abstraction simultaneously. The result is difficult algorithm development, a large number of errors, and difficulty in debugging. These problems are magnified if the assembler requires maintenance at a later time.

A good general design principle is to assign responsibility for different kinds of abstractions to different program modules. This principle is crucial for large designs, but is a good practice for smaller designs as well. Dividing responsibilities for different kinds of abstractions into different modules allows programmers to focus their attention on one aspect of the problem at a time.

For an assembler, the implication is that there should be a module dedicated to handling text at the level of characters. This module is called a lexical analyser or scanner.

ASSEMBLER MODULES

Thus an assembler has four main modules,

- A scanner,
- A pass 1 module,
- A pass 2 module, and
- A main program module.

The responsibilities of these modules are described in more detail in the following sections.

LEXICAL ANALYSIS: THE SCANNER

The primary purpose of the scanner module is processing characters into higher level units that are more meaningful for the pass 1 and pass 2 modules. These units are called *tokens*. The process of forming these groups is called *lexical analysis*. The need for a scanner arises in most programs that deal with complex input.

Part of the design of a scanner involves deciding precisely what a token is; that is, deciding the level of the units that the scanner delivers to the pass modules. Some possibilities are discussed in Options for Scanner Interfaces.

BUILDING THE SYMBOL TABLE: PASS 1

During pass 1, the input is read and memory addresses are assigned to program labels. Memory is allocated sequentially so that the pass 1 module can use a location counter. For each input statement, this counter is incremented by the size of memory allocated. Whenever a label is encountered, it is recorded in the symbol table. The address assigned is the value of the location counter at the beginning of the statement.

Most assemblers need to do some processing of assembler directives to determine the size of the data involved. Modern RISC processors have fixed instruction lengths, so machine instructions require very little processing during pass 1.

Some assemblers keep data and instructions in separate regions of memory. If this is done then two separate location counters are used, one for data and one for instructions.

The symbol table could be treated as a sub-module of the pass 1 module. This choice has little if any effect on the complexity of coding, but a separately compiled sub-module does facilitate separate testing.

GENERATING OUTPUT: PASS 2

The primary effort in pass 2 is translating instructions into machine code. If the assembler is mixing data and instructions in the same area of memory then translation of data must be done in pass 2. For assemblers that use separate areas of memory for data and instructions, translation of data could be moved into pass 1. This is somewhat advantageous in that it results in a better balance of the complexities of the two pass modules.

Most of the error reports generated by an assembler are generated during pass 2. These reports can be interleaved with assembler listing output so that the assembly language programmer can readily associate an error report with the code that caused it.

While pass 2 is running, machine code is saved in a byte array (two arrays if data and instructions are kept separate). If there are no errors then at the end of pass 2 the array(s) is written to a file in binary form and it can also be displayed as a hexadecimal dump for the assembly language programmer. The Assembler Output web page describes C programming techniques for saving binary data in an array and writing the array to a file. There is enough complexity involved in handling the binary data arrays that a separate sub-module could be used.

For large instruction sets, a Table-Driven Design is useful. In this approach, instructions are classified according to their operand types. This classification information, along with other coding information, is stored in a table. In a language like C that allows initialization of arrays, the table does not require any runtime code for its construction. It is just an initialized array. The pass 2 module uses the information in the table to determine the kind of information it seeks from the scanner, and the order of that information.

A table driven design could also be used for handling directives. This could be used in pass 1 as well as pass 2. However, if the number of assembler directives is small then it is not as important as for the handling of machine instructions.

THE MAIN PROGRAM

The main program in an assembler is not complex. The primary work is providing file parameters for function calls to the other modules and passing an error boolean from pass 1 to pass 2 so that an executable output is not generated when there is an error in the assembly language source.

COMMUNICATION BETWEEN MODULES

The diagram below indicates the communication pathways between the modules of an assembler. For each arrow in the diagram, the module at the tail of the arrow plays the role of a client and the module at the head of the arrow plays the role of a server. This means that the client calls functions provided by the server.

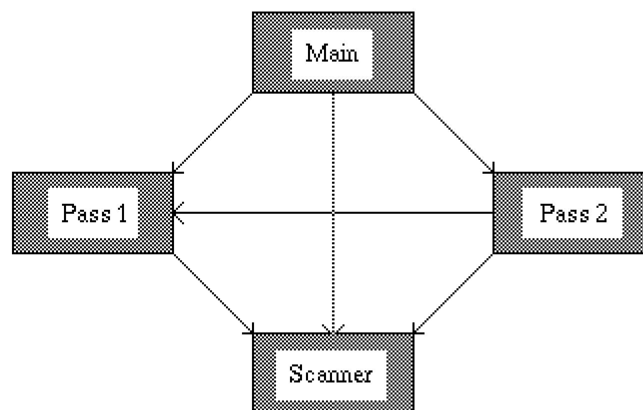


Figure 6.1 Module Communication

The communication between the main program and the two pass modules is quite simple - the main program just calls pass 1 or 2 functions directing them to do their work in the proper order. The functions do not return any data except for possibly an error indication.

The communication between the main program and the scanner is also simple. The name of the file to be assembled is known directly in the main program. The main program either passes the name to the scanner or opens the file and passes it to the scanner. This could be done indirectly through the pass 1 and pass 2 modules.

The communication between pass 1 and pass 2 involves symbol table information. Pass 2 needs to get addresses for labels and values of defined constants from the symbol table. Although there is a fair amount of communication, the interface is simple. It can be a standard table interface.

The scanner is the communication focal point of an assembler. All of the other modules communicate with the scanner. The communication between the pass modules and the scanner is more complex than the communication along other pathways. For this reason, the best place to start working on assembler communication is the scanner client interface. This is an important aspect of assembler design. It cannot be taken lightly.

WEEK SEVEN

Learning Outcome for this week
- The meaning of translation and compilation - The types of compiler - The stages compilation

TEACHER'S ACTIVITIES

- Define interpretation, translation and compilation
- Differentiate between interpretation translation and compilation
- Describe various types of compilers
- Describe types of tables generated in the process of compilation
- Explain code generation and optimization
- Describe error handling

THE INTERPRETER

In computer science, an **interpreter** normally means a computer program that executes, i.e. *performs*, instructions written in a programming language. While interpretation and compilation are the two principal means by which programming languages are implemented, these are not fully distinct categories, one of the reasons being that most interpreting systems also perform some translation work, just like compilers.

An *interpreter* may be a program that either

1. Executes the source code directly
2. Translates source code into some efficient intermediate representation (code) and immediately executes this
3. Explicitly executes stored precompiled code made by a compiler which is part of the interpreter system

Perl, Python, MATLAB, and Ruby are examples of type 2, while UCSD, Pascal and Java are type 3: Source programs are compiled ahead of time and stored as machine independent code, which is then linked at run-time and executed by an interpreter and/or compiler (for JIT systems). Some systems, such as Smalltalk, and others, may also combine 2 and 3.

The terms *interpreted language* or *compiled language* merely mean that the canonical implementation of that language is an interpreter or a compiler; a high level language is basically an abstraction which is (ideally) independent of particular implementations.

EFFICIENCY, ADVANTAGES AND DISADVANTAGES

The main disadvantage of interpreters is that when a program is interpreted, it typically runs slower than if it had been compiled. The difference in speeds could be tiny or great; often an order of magnitude and sometimes more. It generally takes longer to run a program under an interpreter than to run the compiled code but it can take less time to interpret it than the total time required to compile and run it. This is especially important when prototyping and testing code when an edit-interpret-debug cycle can often be much shorter than an edit-compile-run-debug cycle.

Interpreting code is slower than running the compiled code because the interpreter must analyze each statement in the program each time it is executed and then perform the desired action, whereas the compiled code just performs the action within a fixed context determined by the compilation. This run-time analysis is known as "interpretive overhead". Access to variables is also slower in an interpreter because the mapping of identifiers to storage locations must be done repeatedly at run-time rather than at compile time.

There are various compromises between the development speed when using an interpreter and the execution speed when using a compiler. Some systems (e.g., some LISPs) allow interpreted and compiled code to call each other and to share variables. This means that once a routine has been tested and debugged under the interpreter it can be compiled and thus benefit from faster execution while other routines are being developed. Many interpreters do not execute the source code as it stands but convert it into some more compact internal form. For example, some BASIC interpreters replace keywords with single byte tokens which can be used to find the instruction in a jump table. An interpreter might well use the same lexical analyzer and parser as the compiler and then interpret the resulting abstract syntax tree.

BYTECODE INTERPRETERS

There is a spectrum of possibilities between interpreting and compiling, depending on the amount of analysis performed before the program is executed. For example, Emacs Lisp is compiled to byte code, which is a highly compressed and optimized representation of the Lisp source, but is not machine code (and therefore not tied to any particular hardware). This "compiled" code is then interpreted by a byte code interpreter (itself written in C). The compiled code in this case is machine code for a virtual machine, which is implemented not in hardware, but in the byte code interpreter. The same approach is used with the Forth code used in Open Firmware systems: the source language is compiled into "F code" (a byte code), which is then interpreted by a virtual machine.

THE COMPILER

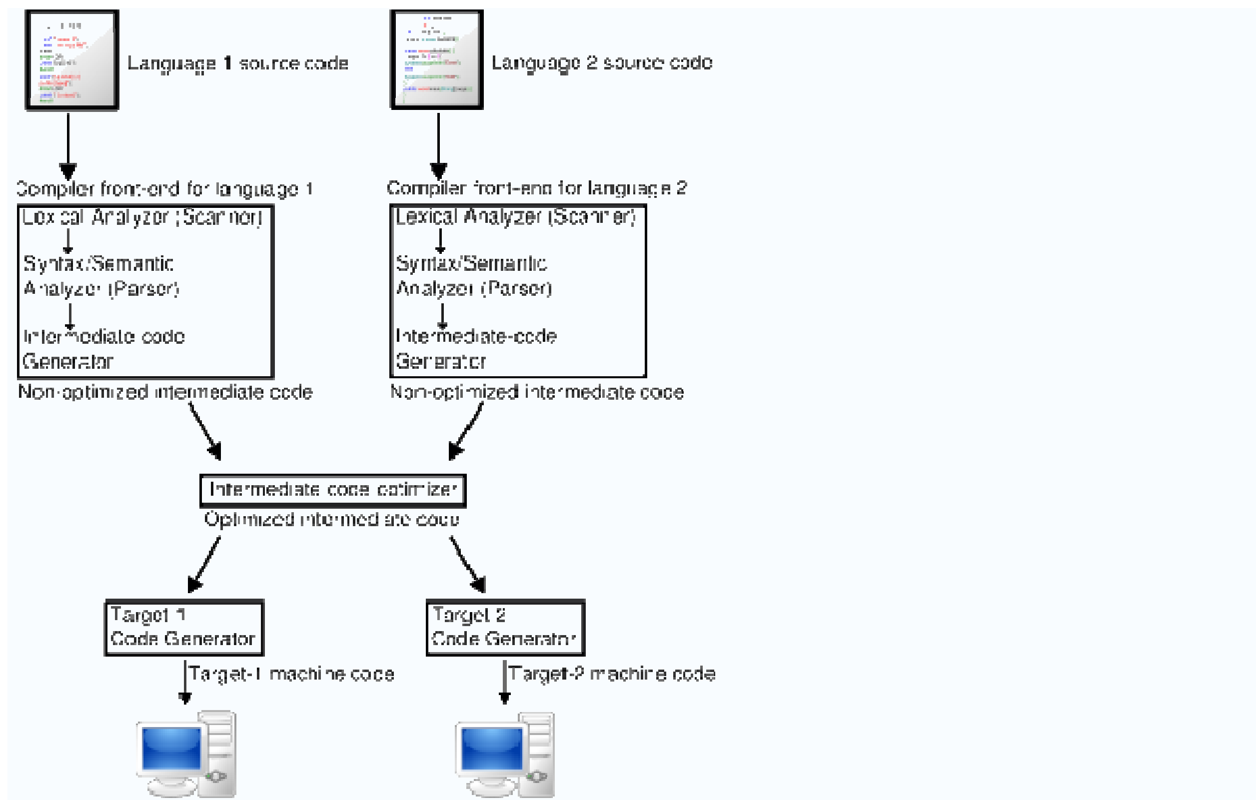


Figure: 7.1 A diagram of the operation of a typical multi-language, multi-target compiler.

A **compiler** is a computer program (or set of programs) that translates text written in a computer language (the *source language*) into another computer language (the *target language*). The original sequence is usually called the *source code* and the output called *object code*. Commonly the output has a form suitable for processing by other programs (e.g., a linker), but it may be a human-readable text file.

The most common reason for wanting to translate source code is to create an executable program. The name "compiler" is primarily used for programs that translate source code from a high-level programming language to a lower level language (e.g., assembly language or machine language). A program that translates from a low level language to a higher level one is a *decompiler*. A program that translates between high-level languages is usually called a *language translator*, *source to source translator*, or *language converter*. A *language rewriter* is usually a program that translates the form of expressions without a change of language.

A compiler is likely to perform many or all of the following operations: lexical analysis, preprocessing, parsing, semantic analysis, code generation, and code optimization.

COMPILER OUTPUT

One classification of compilers is by the platform on which their generated code executes. This is known as the *target platform.*

A *native* or *hosted* compiler is one whose output is intended to directly run on the same type of computer and operating system as the compiler itself runs on. The output of a cross compiler is designed to run on a different platform. Cross compilers are often used when developing software for embedded systems that are not intended to support a software development environment.

The output of a compiler that produces code for a virtual machine (VM) may or may not be executed on the same platform as the compiler that produced it. For this reason such compilers are not usually classified as native or cross compilers.

COMPILED VERSUS INTERPRETED LANGUAGES

Higher-level programming languages are generally divided for convenience into compiled languages and interpreted languages. However, there is rarely anything about a language that *requires* it to be exclusively compiled, or exclusively interpreted. The categorization usually reflects the most popular or widespread implementations of a language — for instance, BASIC is sometimes called an interpreted language, and C a compiled one, despite the existence of BASIC compilers and C interpreters.

In a sense, *all* languages are interpreted, with "execution" being merely a special case of interpretation performed by transistors switching on a CPU. Modern trends toward just-in-time compilation and byte code interpretation also blur the traditional categorizations.

There are exceptions. Some language specifications spell out that implementations *must* include a compilation facility; for example, Common Lisp. Other languages have features that are very easy to implement in an interpreter, but make writing a compiler much harder; for example, APL, SNOBOL4, and many scripting languages allow programs to construct arbitrary source code at runtime with regular string operations, and then execute that code by passing it to a special evaluation function. To implement these features in a compiled language, programs must usually be shipped with a runtime library that includes a version of the compiler itself.

ONE-PASS VERSUS MULTI-PASS COMPILERS

Classifying compilers by number of passes has its background in the hardware resource limitations of computers. Compiling involves performing lots of work and early computers did not have enough memory to contain one program that did all of this work. So compilers were split up into smaller programs which each made a pass over the source (or some representation of it) performing some of the required analysis and translations.

The ability to compile in a single pass is often seen as a benefit because it simplifies the job of writing a compiler and one pass compilers are generally faster than multi-pass compilers. Many languages were designed so that they could be compiled in a single pass (e.g., Pascal).

In some cases the design of a language feature may require a compiler to perform more than one pass over the source. For instance, consider a declaration appearing on line 20 of the source which affects the translation of a statement appearing on line 10. In this case, the first pass needs to gather information about declarations appearing after statements that they affect, with the actual translation happening during a subsequent pass.

The disadvantage of compiling in a single pass is that it is not possible to perform many of the sophisticated optimizations needed to generate high quality code. It can be difficult to count exactly how many passes an optimizing compiler makes. For instance, different phases of optimization may analyse one expression many times but only analyse another expression once.

Splitting a compiler up into small programs is a technique used by researchers interested in producing provably correct compilers. Proving the correctness of a set of small programs often requires less effort than proving the correctness of a larger, single, equivalent program.

While the typical multi-pass compiler outputs machine code from its final pass, there are several other types:

- A "source-to-source compiler" is a type of compiler that takes a high level language as its input and outputs a high level language. For example, an automatic parallelizing compiler will frequently take in a high level language program as an input and then transform the code and annotate it with parallel code annotations (e.g. OpenMP) or language constructs (e.g. Fortran's `DOALL` statements).
- Stage compiler that compiles to assembly language of a theoretical machine, like some Prolog implementations
    - This Prolog machine is also known as the Warren Abstract Machine (or WAM). Byte code compilers for Java, Python, and many more are also a subtype of this.
- Just-in-time compiler, used by Smalltalk and Java systems, and also by Microsoft .Net's Common Intermediate Language (CIL)

JUST-IN-TIME COMPILATION

Further blurring the distinction between interpreters, byte-code interpreters and compilation is just-in-time compilation (or JIT), a technique in which the intermediate representation is compiled to native machine code at runtime. This confers the efficiency of running native code, at the cost of startup time and increased memory use when the byte code or AST is first compiled. Adaptive optimization is a complementary technique in which the interpreter profiles the running program and compiles its most frequently-executed parts into native code. Both techniques are a few decades old, appearing in languages such as Smalltalk in the 1980s.

Just-in-time compilation has gained mainstream attention amongst language implementors in recent years, with Java, Python and the .NET Framework all now including JITs.

WEEK EIGHT

Learning Outcome for this week:
- Front end compilation and its stages - Back end compilation and its stages

FRONT END

The front end analyzes the source code to build an internal representation of the program, called the intermediate representation or *IR*. It also manages the symbol table, a data structure mapping each symbol in the source code to associated information such as location, type and scope. This is done over several phases, which includes some of the following:

1. **Line reconstruction**. Languages which strop their keywords or allow arbitrary spaces within identifiers require a phase before parsing, which converts the input character sequence to a canonical form ready for the parser. The top-down, recursive-descent, table-driven parsers used in the 1960s typically read the source one character at a time and did not require a separate tokenizing phase. Atlas Autocode, and Imp (and some implementations of Algol and Coral66) are examples of stropped languages whose compilers would have a *Line Reconstruction* phase.
2. **Lexical analysis** breaks the source code text into small pieces called *tokens*. Each token is a single atomic unit of the language, for instance a keyword, identifier or symbol name. The token syntax is typically a regular language, so a finite state automaton constructed from a regular expression can be used to recognize it. This phase is also called lexing or scanning, and the software doing lexical analysis is called a lexical analyzer or scanner.
3. **Preprocessing.** Some languages, e.g., C, require a preprocessing phase which supports macro substitution and conditional compilation. Typically the preprocessing phase occurs before syntactic or semantic analysis; e.g. in the case of C, the preprocessor manipulates lexical tokens rather than syntactic forms. However, some languages such as Scheme support macro substitutions based on syntactic forms.
4. **Syntax analysis** involves parsing the token sequence to identify the syntactic structure of the program. This phase typically builds a parse tree, which replaces the linear sequence of tokens with a tree structure built according to the rules of a formal grammar which define the language's syntax. The parse tree is often analyzed, augmented, and transformed by later phases in the compiler.
5. **Semantic analysis** is the phase in which the compiler adds semantic information to the parse tree and builds the symbol table. This phase performs semantic checks such as type checking (checking for type errors), or object binding (associating variable and function references with their definitions), or **definite assignment** (requiring all local variables to be initialized before use), rejecting incorrect programs or issuing warnings. Semantic analysis usually requires a complete parse tree, meaning that this phase logically follows the parsing phase, and logically proceeds the code generation phase, though it is often possible to fold multiple phases into one pass over the code in a compiler implementation.

BACK END

The term *back end* is sometimes confused with *code generator* because of the overlapped functionality of generating assembly code. Some literature uses *middle end* to distinguish the generic analysis and optimization phases in the back end from the machine-dependent code generators.

The main phases of the back end include the following:

1. **Analysis**: This is the gathering of program information from the intermediate representation derived from the input. Typical analyses are data flow analysis to build use-define chains, dependence analysis, alias analysis, pointer analysis, escape analysis etc. Accurate analysis is the basis for any compiler optimization. The call graph and control flow graph are usually also built during the analysis phase.
2. **Optimization:** the intermediate language representation is transformed into functionally equivalent but faster (or smaller) forms. Popular optimizations are inline expansion, dead code elimination, constant propagation, loop transformation, register allocation or even automatic parallelization.
3. Code generation: the transformed intermediate language is translated into the output language, usually the native machine language of the system. This involves resource and storage decisions, such as deciding which variables to fit into registers and memory and the selection and scheduling of appropriate machine instructions along with their associated addressing modes

Compiler analysis is the prerequisite for any compiler optimization, and they tightly work together. For example, dependence analysis is crucial for loop transformation.

In addition, the scope of compiler analysis and optimizations vary greatly, from as small as a basic block to the procedure/function level, or even over the whole program (interprocedural optimization). Obviously, a compiler can potentially do a better job using a broader view. But that broad view is not free: large scope analysis and optimizations are very costly in terms of compilation time and memory space; this is especially true for interprocedural analysis and optimizations.

Due to the extra time and space needed for compiler analysis and optimizations, some compilers skip them by default. Users have to use compilation options to explicitly tell the compiler which optimizations should be enabled.

Lexical analysis

In computer science, **lexical analysis** is the process of converting a sequence of characters into a sequence of tokens. Programs performing lexical analysis are called **lexical analyzers** or **lexers**. A lexer is often organized as separate **scanner** and **tokenizer** functions, though the boundaries may not be clearly defined.

Lexical grammar

The specification of a programming language will include a set of rules, often expressed syntactically, specifying the set of possible character sequences that can form a token or lexeme. The whitespace characters are often ignored during lexical analysis.

Tokens

A token is a categorized block of text. The block of text corresponding to the token is known as a lexeme. A lexical analyzer processes *lexemes* to categorize them according to function, giving them meaning. This assignment of meaning is known as tokenization. A token can look like anything; it just needs to be a useful part of the structured text.

Consider this expression in the C programming language:

```
sum=3+2;
```

Tokenized in the following table:

lexeme	token type
sum	**IDENT**
=	**ASSIGN_OP**
3	**NUMBER**
+	**ADD_OP**
2	**NUMBER**
;	**SEMICOLON**

Tokens are frequently defined by regular expressions, which are understood by a lexical analyzer generator such as lex. The lexical analyzer (either generated automatically by a tool like lex, or hand-crafted) reads in a stream of characters, identifies the lexemes in the stream, and categorizes them into tokens. This is called "tokenizing." If the lexer finds an invalid token, it will report an error.

Following tokenizing is parsing. From there, the interpreted data may be loaded into data structures, for general use, interpretation, or compiling.

Consider a text describing a calculation:

```
46 - number of (cows);
```

The lexemes here might be: "46", "-", "number_of ", "(", "cows", ")" and ";". The lexical analyzer will denote lexemes "46" as 'number', "-" as 'character' and "number_of " as a separate token. Even the lexeme ";" in some languages (such as C) has some special meaning.

The Scanner

The first stage, the **scanner**, is usually based on a finite state machine. It has encoded within it information on the possible sequences of characters that can be contained within any of the tokens it handles (individual instances of these character sequences are known as lexemes). For instance, an *integer* token may contain any sequence of numerical digit characters. In many cases, the first non-whitespace character can be used to deduce the kind of token that follows and subsequent input characters are then processed one at a time until reaching a character that is not in the set of characters acceptable for that token (this is known as the maximal munch rule). In some languages the lexeme creation rules are more complicated and may involve backtracking over previously read characters.

The Tokenizer

Tokenization is the process of demarcating and possibly classifying sections of a string of input characters. The resulting tokens are then passed on to some other form of processing. The process can be considered a sub-task of parsing input.

WEEK NINE

Learning Outcome for this week
- Describe error checking and handling - Explain utilities and give some examples - Discuss the types of libraries

ERROR CHECKING

One of the major differences between systems programming and application programming is that error checking is not something nice to have but so essential that one cannot live without. Very commonly, you can see people do "printf" to output error messages to the console. Although better than not having any error checking, however, this is not enough even when not doing systems programming.

NOTE:

WE SHALL BASED OUR EXAMPLES HERE ON C PROGRAMMING LANGUAGE

When each program runs and becomes a process, there are three files opened for it by default, *stdin*, *stdout* and *stderr*. The first two obviously are the input and output consoles and the third, *stderr*, is where the error messages are supposed to go to. *stdin* and *stdout* may be redirected, while *stderr* cannot. Therefore the error messages are best output using:

```
fprintf(stderr,"your error message");
```

It should also be known to you that *printf* and *fprintf(stdout,"...")* do the same task.

Besides using *fprintf*, there are two other important tools for use in error checking: **perror** and **assert**, both are ANSI C standard and available on all operating systems that claim to support ANSI C'1987.

PERROR

When a computer is turned on, the program that gets executed first is called the ``*operating system*.'' It controls pretty much all activity in the computer. This includes who logs in, how disks are used, how memory is used, how the CPU is used, and how you talk with other computers. In the following discussions, our study Operating System example will be "Unix".

The way that programs talk to the operating system is via ``*system calls*.'' A system call looks like a procedure call (see below), but it's different -- **it is a request to the operating system to perform some activity**.

System calls are expensive. While a procedure call can usually be performed in a few machine instructions, a system call requires the computer to save its state, let the operating system take control of the CPU, have the operating system perform some function, have the operating system save its state, and then have the operating system give control of the CPU back to you.

Usually when an error occurs in a system or library call, a special return value comes back, and a global variable "**errno**" is set to say what the error is. For example, suppose you try to open a file that does not exist:

```
#include < stdio.h >
#include < errno.h >

main()
{
  int i;
  FILE *f;

  f = fopen("~huangj/nonexist", "r");
  if (f == NULL) {
    printf("f = null.  errno = %d\n", errno);
    perror("f1");
  }
}
```

ch1a.c tries to open the file **~huangj/nonexist** for reading. That file doesn't exist. Thus, **fopen** returns **NULL** (read the man page for **fopen**), and sets **errno** to flag the error. When you run the program, you'll see that **errno** was set to 2. To see what that means, you can do one of two things:

- 1. Look up the **errno** value in **/usr/include/errno.h** (You will have to eventually look at /usr/include/sys/errono.h on UNIX flavor machines since on that type of system, the C standard errno.h does have *"#include < sys/errno.h >"* in it.). You'll see the line:
- `#define ENOENT          2               /* No such file or directory */`
- 2. Use the procedure "**perror()**" -- again, read the man page. It prints out what the errno means. Thus, the output of f1 is
- `f = null.  errno = 2`
- `f1: No such file or directory`

This is the standard interface for errors.

ASSERT

Most of the time, there is a need to make assumptions when you write code. Everybody does it. But what if the assumption is wrong? Is there a good way to check it? The answer is to use *assert*.

The most typical use of the *assert* (very likely implemented as a macro on most operating systems you can find) is to identify program errors during development. The argument given to

assert should be chosen so that it holds true only if the program is operating as intended. The macro evaluates the *assert* argument and, if the argument expression is false (0), alerts the user and halts program execution. No action is taken if the argument is true (nonzero).

When an assertion fails, an output message with the following text is generated:

assertion failed in file *name* in line *num*
where *name* is the name of the source file and *num* is the line number of the assertion that failed.

The liberal use of assertions throughout your programs can catch errors during development. A good rule of thumb is that you should write assertions for any assumptions you make. For example, if you assume that an argument is not NULL, use an assertion statement to check for that condition.

```
void checkerror_strcpy(char * src, char *dst)
{
  assert(src!=dst);
  assert(src!=NULL);
  assert(dst!=NULL);
}
```

Here we check that some assumptions we made for *strcpy* are true. After we are sure there are no errors in the software, we can easily disable all assertion checks adding *"#define NDEBUG"* before where *"#include < assert.h >"* appears in the source code.

LIBRARY IN COMPUTING

In computer science, a **library** is a collection of subroutines or classes used to develop software. Libraries contain code and data that provide services to independent programs. This allows code and data to be shared and changed in a modular fashion. Some executables are both standalone programs and libraries, but most libraries are not executables. Executables and libraries make references known as *links* to each other through the process known as *linking*, which is typically done by a linker.

Most modern operating systems (OS) provide libraries that implement the majority of system services. Such libraries have commoditized the services a modern application expects an OS to provide. As such, most code used by modern applications is provided in these libraries.

TYPES OF LIBRARIES

- **STATIC LIBRARIES**

Historically, libraries could only be *static*. A static library, also known as an *archive*, consists of a set of routines which are copied into a target application by the compiler, linker, or binder, producing object files and a stand-alone executable file. This process, and the stand-alone executable file, are known as a static build of the target application.

The linker resolves all of the unresolved addresses into fixed or relocatable addresses (from a common base) by loading all code and libraries into actual runtime memory locations.

A linker may work on specific types of object files, and thus require specific (compatible) types of libraries. The linking process *resolves* references by searching the libraries in the order given. Usually, it is not considered an error if a name can be found multiple times in a given set of libraries.

- **DYNAMIC LINKING**

Dynamic linking means that the subroutines of a library are loaded into an application program at runtime, rather than being linked in at compile time, and remain as separate files on disk. Only a minimum amount of work is done at compile time by the linker; it only records what library routines the program needs and the index names or numbers of the routines in the library. The majority of the work of linking is done at the time the application is loaded (load time) or during execution (runtime). The necessary linking code, called a loader, is actually part of the underlying operating system. At the appropriate time the loader finds the relevant libraries on disk and adds the relevant data from the libraries to the process's memory space.

Some operating systems can only link in a library at load time, before the process starts executing; others may be able to wait until after the process has started to execute and link in the library just when it is actually referenced (i.e., during runtime). The latter is often called "delay loading" or "deferred loading". In either case, such a library is called a **dynamically linked** library.

RELOCATION

One wrinkle that the loader must handle is that the actual location in memory of the library data cannot be known until after the executable and all dynamically linked libraries have been loaded into memory. This is because the memory locations used depend on which specific dynamic libraries have been loaded. It is not possible to depend on the absolute location of the data in the executable, nor even in the library, since conflicts between different libraries would result: if two of them specified the same or overlapping addresses, it would be impossible to use both in the same program.

However, in practice, the shared libraries on most systems do not change often. Therefore, it is possible to compute a likely load address for every shared library on the system before it is needed, and store that information in the libraries and executables. If every shared library that is loaded has undergone this process, then each will load at their predetermined addresses, which speeds up the process of dynamic linking. This optimization is known as pre-binding in Mac OS X and pre-linking in Linux. Disadvantages of this technique include the time required to pre-compute these addresses every time the shared libraries change, the inability to use address space layout randomization, and the requirement of sufficient virtual address space for use (a problem that will be alleviated by the adoption of 64-bit architectures, at least for the time being).

➢ *LOCATING LIBRARIES AT RUNTIME*

Dynamic linkers/loaders vary widely in functionality. Some depend on explicit paths to the libraries being stored in the executable. Any change to the library naming or layout of the file system will cause these systems to fail. More commonly, only the name of the library (and not the path) is stored in the executable, with the operating system supplying a system to find the library on-disk based on some algorithm.

One of the biggest disadvantages of dynamic linking is that the executables depend on the separately stored libraries in order to function properly. If the library is deleted, moved, or renamed, or if an incompatible version of the DLL is copied to a place that is earlier in the search, the executable would fail to load. On Windows this is commonly known as DLL hell.

Unix-like systems

Most Unix-like systems have a "search path" specifying file system directories in which to look for dynamic libraries. On some systems, the default path is specified in a configuration file; in others, it is hard coded into the dynamic loader. Some executable file formats can specify additional directories in which to search for libraries for a particular program.

Microsoft Windows

Microsoft Windows will check the registry to determine the proper place to find an ActiveX DLL, but for other DLLs it will check the directory that the program was loaded from; the current working directory; any directories set by calling the SetDllDirectory() function;

AmigaOS

Under AmigaOS generic system libraries are stored in a directory defined by the *LIBS:* path assignment and application-specific libraries can be stored in the same directory as the application's executable. AmigaOS will search these locations when an executable attempts to launch a shared library. An application may also supply an explicit path when attempting to launch a library.

➢ *SHARED LIBRARIES*

In addition to being loaded statically or dynamically, libraries are also often classified according to how they are shared among programs. Dynamic libraries almost always offer some form of sharing, allowing the same library to be used by multiple programs at the same time. Static libraries, by definition, cannot be shared. The term "linker" comes from the process of copying procedures or subroutines which may come from "relocatable" libraries and adjusting or "linking" the machine address to the final locations of each module.

The **shared library** term is slightly ambiguous, because it covers at least two different concepts. First, it is the sharing of code located on disk by unrelated programs. The second concept is the sharing of code in memory, when programs execute the same physical page of RAM, mapped

into different address spaces. It would seem that the latter would be preferable, and indeed it has a number of advantages. For instance on the OpenStep system, applications were often only a few hundred kilobytes in size and loaded almost instantly; the *vast* majority of their code was located in libraries that had already been loaded for other purposes by the operating system. There is a cost, however; shared code must be specifically written to run in a multitasking environment.

In most modern operating systems, shared libraries can be of the same format as the "regular" executables. This allows two main advantages: first, it requires making only one loader for both of them, rather than two (having the single loader is considered well worth its added complexity). Secondly, it allows the executables also to be used as DLLs, if they have a symbol table.

The term DLL is mostly used on Windows and OS/2 products. On Unix and Unix-like platforms, the term *shared library* or *shared object* is more commonly used; consequently, the most common filename extension for shared library files is .so, usually followed by another dot and a version number. This is technically justified in view of the different semantics.

- ***DYNAMIC LOADING***

Dynamic loading is a subset of dynamic linking where a dynamically linked library loads and unloads at run-time on request. Such a request may be made implicitly at compile-time or explicitly at run-time. Implicit requests are made at compile-time when a linker adds library references that include file paths or simply file names. Explicit requests are made when applications make direct calls to an operating system's API at runtime.

Most operating systems that support dynamically linked libraries also support dynamically loading such libraries via a run-time linker API. For instance, Microsoft Windows uses the API functions LoadLibrary, LoadLibraryEx, FreeLibrary and GetProcAddress with Microsoft Dynamic Link Libraries; POSIX based systems, including most UNIX and UNIX-like systems, use dlopen, dlclose and dlsym. Some development systems automate this process.

- **REMOTE LIBRARIES**

Another solution to the library issue is to use completely separate executables (often in some lightweight form) and call them using a remote procedure call (RPC) over a network to another computer. This approach maximizes operating system re-use: the code needed to support the library is the same code being used to provide application support and security for every other program. Additionally, such systems do not require the library to exist on the same machine, but can forward the requests over the network.

The downside to such an approach is that every library call requires a considerable amount of overhead. RPC calls are much more expensive than calling a shared library which has already been loaded on the same machine..

➢ OBJECT LIBRARIES

Although dynamic linking was originally developed in the 1960s, it did not reach consumer operating systems until the late 1980s; it was generally available in some form in most operating systems by the early 1990s. It was during this same period that object-oriented programming (OOP) was becoming a significant part of the programming landscape. OOP with runtime binding requires additional information that traditional libraries don't supply; in addition to the names and entry points of the code located within, they also require a list of the objects on which they depend. This is a side-effect of one of OOP's main advantages, inheritance, which means that the complete definition of any method may be defined in a number of places. This is more than simply listing that one library requires the services of another; in a true OOP system, the libraries themselves may not be known at compile time, and vary from system to system.

It was not long before the majority of the minicomputer and mainframe vendors were working on projects to combine the two, producing an OOP library format that could be used anywhere. Such systems were known as **object libraries**, or **distributed objects** if they supported remote access (not all did). Microsoft's COM is an example of such a system for local use, DCOM a modified version that support remote access.

WEEK TEN

Learning Outcome for this week
- The historical development of operating system. - The importance and uses of operating system - The system commands of MS-DOS, Unix, Windows operating systems.

OPERATING SYSTEMS

When a brand new computer comes off the factory assembly line, it can do nothing. The hardware needs software to make it work.

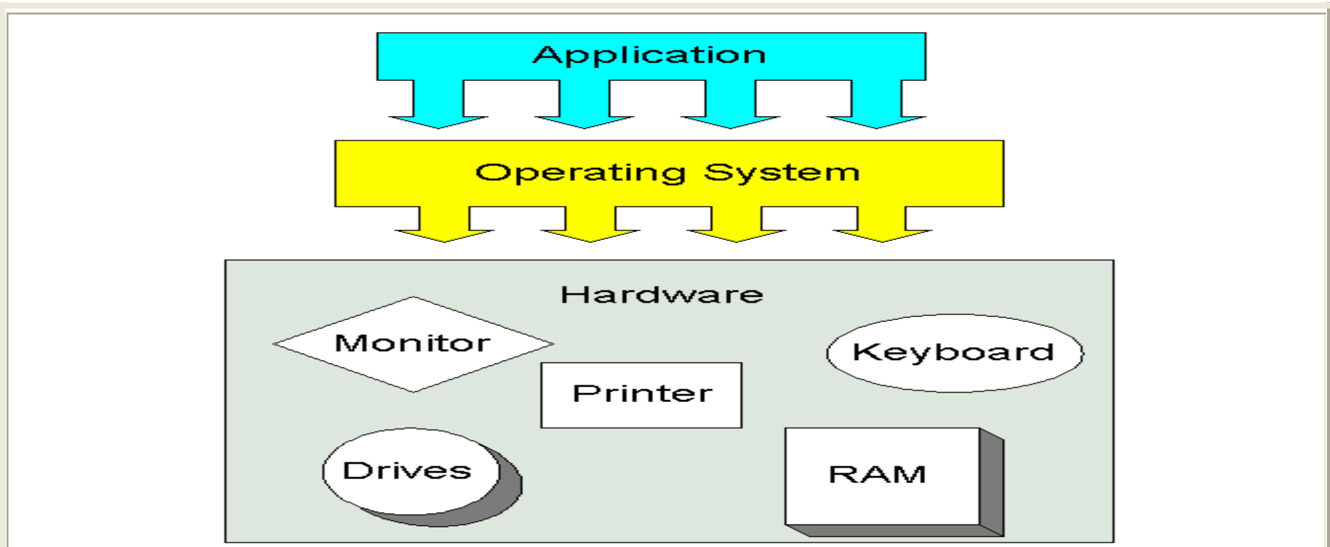


Figure10.1 : The Operating System in a Hierarchy

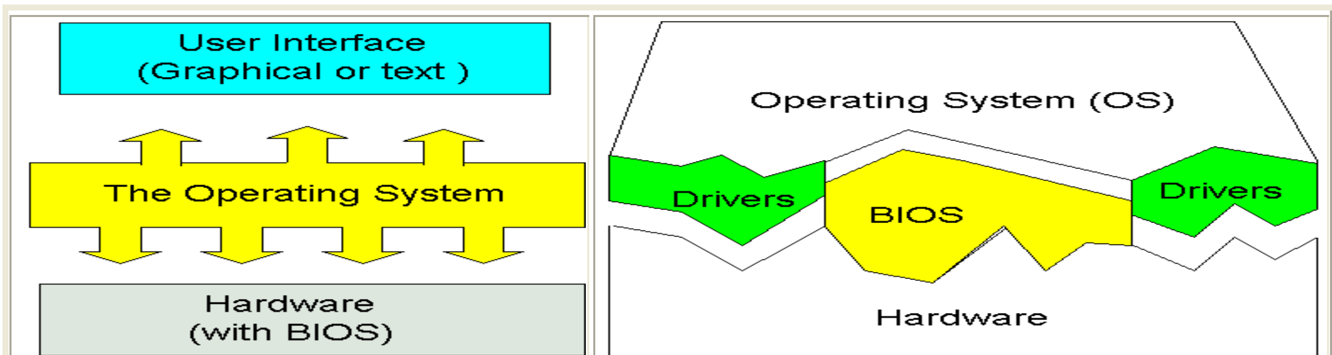


Figure 10.2 : System Software

An application software package does not communicate directly with the hardware. As shown in the Figure above between the applications software and the hardware is a software interface - an *operating system*.

Definition: An operating system is a set of programs that lies between applications software and the computer hardware. Conceptually the operating system software is an intermediary between the hardware and the applications software. Incidentally, the term system software is sometimes used interchangeably with operating system, but system software means all programs related to coordinating computer operations. System software does include the operating system, but it also includes the BIOS software , drivers, and service programs, which we will discuss briefly in this chapter (see Figure above).

- Note that we said that an operating system is a set of programs. The most important program in the operating system, the program that manages the operating system, is the supervisor program, most of which remains in memory and is thus referred to as resident. The supervisor controls the entire operating system and loads into memory other operating system programs (called nonresident) from disk storage only as needed.

FUNCTIONS OF AN OPERATING SYSTEM

An operating system has three main functions:

- To manage the computer's resources, such as the central processing unit, memory, disk drives, and printers,
- To establish a user interface
- To execute and provide services for applications software.

However, that much of the work of an operating system is hidden from the user; many necessary tasks are performed behind the scenes. In particular, the first listed function, managing the computer's resources, is taken care of without the user being aware of the details. Furthermore, all input and output operations, although invoked by an applications program, are actually carried out by the operating system. Although much of the operating system functions are hidden from view, you will know when you are using an applications software package, and this requires that you invoke-call into action-the operating system. Thus you both establish a user interface and execute software.

Operating systems for mainframe and other large computers are even more complex because they must keep track of several programs from several users all running in the same time frame. Although some personal computer operating systems (most often found in business or learning environments) can support multiple programs and users, most are concerned only with a single user. We begin by focusing on the interaction between a single user and a personal computer operating system.

OPERATING SYSTEMS FOR PERSONAL COMPUTERS

If you peruse software sold at a retail store, you will generally find the software grouped according to the computer, probably IBM (that is, IBM compatible) or Macintosh, on which the software can be used. But the distinction is actually finer than the differences among computers: Applications software-word processing, spreadsheets, games, whatever-are really distinguished

by the operating system on which the software can run.

Generally, an application program can run on just one operating system. Just as you cannot place a Nissan engine in a Luxurious bus as we call it . You cannot take a version of WordPerfect designed to run on an IBM machine and run it on an Apple Macintosh. The reason is that IBM personal computers and others like them have Intel-compatible microprocessors and usually use Microsoft's operating system, called MS-DOS (for Microsoft disk operating system) on older computers, and Windows98 , Windows XP … on more modern computers. Computers that have come out since the year 2000 often come with Windows ME (Millennium Edition), or Windows2000. Macintoshes use an entirely different operating system, called the Macintosh operating system, which is produced by Apple. Over 75 percent of personal computers use a versions of Windows as their operating systems. Macintosh comprises about 15 percent of the market, with other operating systems such as Linux comprising the rest.

Users do not set out to buy operating systems; they want computers and the applications software to make them useful. However, since the operating system determines what software is available for a given computer, many users observe the high volume of software available for MS-DOS machines and make their computer purchases accordingly. Others prefer the user-friendly style of the Macintosh operating system and choose Macs for that reason.

Although operating systems differ, many of their basic functions are similar. We will show some of the basic functions of operating systems by examining MS-DOS.

MS-DOS

- Most users today have a computer with a hard disk drive. When the computer is turned on, the operating system will be loaded from the hard drive into the computer's memory, thus making it available for use. The process of loading the operating system into memory is called bootstrapping, or booting the system. The word booting is used because, figuratively speaking, the operating system pulls itself up by its own bootstraps. When the computer is switched on, a small program (in ROM-read-only memory) automatically pulls up the basic components of the operating system from the hard disk. From now on, we will refer to MS-DOS by its commonly used abbreviated name, DOS, pronounced to rhyme with boss.

- The net observable result of booting DOS is that the characters C> (or possibly C:\>) appear on the screen. The C refers to the disk drive; the > is a prompt, a signal that the system is prompting you to do something. At this point you must give some instruction to the computer. Perhaps all you need to do is key certain letters to make the application software take the lead. But it could be more complicated than that because C> is actually a signal for direct communication between the user and the operating system.

- Although the prompt is the only visible result of booting the system, DOS also provides the basic software that coordinates the computer's hardware components and a set of programs that lets you perform the many computer system tasks you need to do. To execute a given DOS program, a user must issue a command, a name that invokes a specific DOS program. Whole books have been written about DOS commands, but we will consider just a few that people use for ordinary activities. Some typical tasks you can do with DOS commands are:
    - To prepare (format) new diskettes for use,
    - list the files on a disk,
    - copy files from one disk to another,
    - erase files from a disk.

- **Microsoft Windows**
  Microsoft Windows started out as a shell. Windows uses a colorful graphics interface that, among other things, eases access to the operating system. The feature that makes Windows so easy to use is a graphical user interface (GUI-pronounced "goo-ee"), in which users work with on-screen pictures called icons and with menus rather than with keyed-in. They are called *pull-down menus* because they appear to pull down like a window shade from the original selection. Some menus, in contrast, called pop-up menus originate from a selection on the bottom of the screen. Furthermore, icons and menus encourage pointing and clicking with a mouse, an approach that can make computer use both fast and easy.

  To enhance ease of use, Windows is usually set up so that the colorful Windows display is the first thing a user sees when the computer is turned on. DOS is still there, under Windows, but a user need never see C> during routine activities. The user points and clicks among a series of narrowing choices until arriving at the desired software.

  Although the screen presentation and user interaction are the most visible evidence of change, Windows offers changes that are even more fundamental. To understand these changes more fully, it is helpful at this point to make a comparison between traditional operating systems for large computers and Windows.

  In addition to adding a friendly GUI, Windows operating systems added another important feature to DOS - *multi-tasking*. Multi-tasking occurs when the computer has several programs executing at one time. PCs that ran under DOS could only run one program at a time. Windows-based computers can have multiple programs (e.g. a browser, a word processor, and several Instant Messaging instances) running at the same time. When programs are executing at the same time, they are said to be executing *concurrently*.

As we learned, personal computers have only one CPU that handles just one instruction at a time. Computers using the MS-DOS operating system without a shell are limited not only to just one user at a time but also to just one program at a time. If, for example, a user were using a word processing program to write a financial report and wanted to access some spreadsheet figures, he or she would have to perform a series of arcane steps: exit the word processing program, enter

and use and then exit the spreadsheet program, and then re-enter the word processing program to complete the report. This is wasteful in two ways:

(1) The CPU is often idle because only one program is executing at a time, and

(2) The user is required to move inconveniently from program to program.

Multi-tasking allows several programs to be active at the same time, although at an instant in time the CPU is doing only one instruction for one of the active programs. The Operating System manages which instructions to send to the CPU. Since computers are so fast, the operating system can switch the program that gets to execute on the CPU so quickly, the user can not tell. This is what allows your computer to be "listening" for incoming instant messages, for instance, while you use a word processor to write a paper.

WEEK ELEVEN

Learning Outcome for this week
Services provided by Operating System

Operating Systems Services

Following are the five services provided by an operating systems to the convenience of the users.

Program Execution

The purpose of a computer systems is to allow the user to execute programs. So the operating systems provides an environment where the user can conveniently run programs. The user does not have to worry about the memory allocation or multitasking or anything. These things are taken care of by the operating systems.

Running a program involves the allocating and de-allocating memory, CPU scheduling in case of multi-process. These functions cannot be given to the user-level programs. So user-level programs cannot help the user to run programs independently without the help from operating systems.

I/O Operations

Each program requires an input and produces output. This involves the use of I/O. The operating systems hides the user the details of underlying hardware for the I/O. All the user sees is that the I/O has been performed without any details. So the operating systems by providing I/O makes it convenient for the users to run programs.

For efficiently and protection users cannot control I/O so this service cannot be provided by user-level programs.

File System Manipulation

The output of a program may need to be written into new files or input taken from some files. The operating systems provides this service. The user does not have to worry about secondary storage management. User gives a command for reading or writing to a file and sees his her task accomplished. Thus operating systems makes it easier for user programs to accomplished their task.

- This service involves secondary storage management. The speed of I/O that depends on secondary storage management is critical to the speed of many programs and hence I think it is best relegated to the operating systems to manage it than giving individual users the control of it. It is not difficult for the user-level programs to Batch processing, multiprogramming, multiprocessing, time-sharing.
- Batch, real-time, timesharing and network operating system.

provide these services but for above mentioned reasons it is best if this service s left with operating system.

Communications

There are instances where processes need to communicate with each other to exchange information. It may be between processes running on the same computer or running on the different computers. By providing this service the operating system relieves the user of the worry of passing messages between processes. In case where the messages need to be passed to processes on the other computers through a network it can be done by the user programs. The user program may be customized to the specifics of the hardware through which the message transits and provides the service interface to the operating system.

Error Detection

An error is one part of the system may cause malfunctioning of the complete system. To avoid such a situation the operating system constantly monitors the system for detecting the errors. This relieves the user of the worry of errors propagating to various part of the system and causing malfunctioning.

This service cannot allowed to be handled by user programs because it involves monitoring and in cases altering area of memory or deallocation of memory for a faulty process. Or may be relinquishing the CPU of a process that goes into an infinite loop. These tasks are too critical to be handed over to the user programs. A user program if given these privileges can interfere with the correct (normal) operation of the operating systems.

WEEK TWELVE

Learning Outcome for this week

- I/O Buffering
- Dealing with files stored in I/O devices
- Spooling: its advantages and disadvantages
-

INPUT/OUTPUT BUFFERING

Often a user process generates requests for output(say) much faster than the device can handle. Instead of having a process waiting for 'request-serviced', introduce a buffer to store all requests, then *process* can go onto do other things. This is buffering. Similarly for input, a buffer can be filled from a device; a user process takes its input from buffer; it is forced to wait only when the buffer becomes empty. When this occurs the operating system refills the buffer and the process continues. Double buffering: is the case when two buffers are used. In a producer/consumer situation, mutual exclusion prevents both processes accessing the buffer at the same time thus, possibly, causing delays. Giving each process its own buffer will reduce the probability of this delay;- transfers between buffers takes place when neither is being accessed by its process. Note: buffering smoothes out the peaks

FILE DEVICES

How do we deal with files stored in I/O devices?

Only some I/O devices can support files (i.e. read/write on particular area of the medium, e.g. disc, magnetic tape, but not printer, keyboard, vdu), these devices are called file devices.

- File: a data area of an arbitrary size which can exist on a medium controlled by the device.
- A file has a unique name which is used by the op sys to find the location of the file on the appropriate medium, . . in a directory of files
- Directing a data stream to/from a file device: associate a stream with a file name, not device name; typical job description: Input1 = 'testdata' i.e. stream 1 data is to come from file 'testdata'

OPENING A FILE

Stream is opened, op sys looks up file name in directory to get device number & file location.

A file descriptor is created to hold info for subsequent accesses to the file to include:

Address of device descriptor

Location of file on that device

Whether read/write

File internal organization

A pointer to the file descriptor is put in stream descriptor.

SPOOLING

Spooling is a higher level buffering to even out demand for unshareable resources: e.g. printers. During periods of high demand several processes are held up waiting for use of scarce resources. During other periods these same devices may be lying unused.
Spool all I/O to these devices, i.e. instead of I/O directly to device, do it on intermediate medium, disc. ‘Spooler’ then moves data between disc and device.

Line printer example: A process wanting to use printer is given disc file to store all its output, i.e. file is virtual line printer. When stream is closed, file is added to queue. Spooler takes files from queue & sends them to printer.

```
repeat indefinitely

begin wait (something to spool);
                  pick file from queue;
                  open file;
                  repeat until end of file;
         begin  DOIO (parameters for disc read);
                wait (disc request serviced);
                DOIO(parameters for line printer output);
                wait (printer request serviced);
         end
end;
```

Notes

1. A buffer is used between disc & printer.
2. Semaphore ‘something to spool’ is signaled (incremented) by any process which closes a line printer stream, i.e. completes a file for output.
3. Output is often dealt with in favour of short files first.

ADVANTAGES OF SPOOLING

1. Evens out pressure on heavily used devices.
2. Reduces possibility of deadlock caused by injudicious peripheral allocation.
3. Easier to produce several copies without re-running jobs.

DISADVANTAGES

1. Need large amount of disc space.
2. Heavy traffic on the disc channel.
3. Not feasible for real-time I/O.

Let us summarize our discussion:

Separating I/O into user process, I/O process and device handler makes it easier to achieve the 3 objectives:

- character code independence
- Device independence
- Uniformity of device treatment

However, Because of their general nature, these routines can sometimes be slower to execute than special pieces of code tailor-made for specific I/O operations and devices. Careful attention must therefore be paid to optimizing the efficiency of these routines. Sometimes, for the sake of efficiency, I/O procedures & device handlers are put together and optimized for specific applications of known operations & devices.

WEEK THIRTEEN

Learning Outcome for this week

- Interrupt handling process
- The concept of interrupts and traps.
- The CPU activity in interrupt mode and pooling and the CPU status.

Interrupts

Interrupt hardware was invented to eliminate the need for explicit calls to a polling procedure from within applications code. Essentially all computers on the market today, from the smallest microcontrollers to the highest performance supercomputers include such hardware. In effect, what the basic interrupt mechanism does is check all of the relevant device status bits just after executing each and every machine instruction, inserting a call to an interrupt handler, analogous to our `poll` routine whenever some device is ready. So long as no devices need service, this allows the computer to execute instructions at full speed.

In general, an *interrupt* can be viewed as a hardware-initiated call to a procedure, the *interrupt handler* or *interrupt service routine*. In effect, the instruction execution loop of the central processor has been made to serve as the main polling loop of our application! Although the abstract description of an interrupt as a hardware initiated procedure call applies to most interrupt hardware, the details vary considerably from machine to machine. The address of the interrupt service routine is frequently stored in a special register or dedicated memory location, called the interrupt vector. On some machines, calls to interrupt service routines parallel procedure calls to the extent that the normal procedure return can be used to return to the code which was running at the time of the interrupt, while on others, special return-from-interrupt instructions must be used.

An assembly language *stub* is a bit of code that masks over the incompatibility between one model of control transfer and another; in this case, allowing the use of an interrupt to call a normal function compiled by a compiler that knew nothing about interrupt service routines.

Once an interrupt service routine has been called, it is essential that the hardware request for that service be disabled or withdrawn. If it were not, an infinite (and possibly recursive) loop would result in which, after executing the first instruction of the interrupt service routine, the hardware would force a control transfer to the start of the same interrupt service routine. On some systems, the interrupt is automatically disabled by the actions of the central processor hardware when it responds to the interrupt, while on others, the first instruction of each interrupt service routine must disable the interrupt.

In the examples presented here, it will be assumed that the hardware automatically disables interrupts as it calls the interrupt service routine.

Even if the hardware can disable interrupts, the software must also be able to enable or disable them. For example, on return from an interrupt service routine, after the software has done whatever the interrupt requested, the software must re-enable the interrupt as it returns to the code which was interrupted. Furthermore, if the output queue is empty, there is no point in responding to an interrupt from the output device when it is ready to transfer more data; a similar argument can be made when the input queue is full. On most machines, there are special instructions to enable and disable interrupts; on some, these instructions apply to all interrupts at the same time, while other machines allow groups of devices to be enabled or disabled as a group.

In addition, it is usual to include, in each device's control register, one or more interrupt enable bits corresponding to each condition the device can sense that might be cause for an interrupt request. If the interrupt enable bit for a condition is set, then when that condition is detected, there will be an interrupt request.

It should be noted that the disabling an output device's ability to request interrupts when the output queue is empty and enabling that device when data is put in the queue is a special purpose solution to a general problem, the producer-consumer problem. The device is the consumer, and the application is the producer, and the general problem is to prevent the consumer from attempting to use data that has not yet been produced. In any producer-consumer system, whether the system is all in hardware, all in software, or mixed between the two, there must be some synchronization mechanism to make the consumer wait until data is available!

Interrupt handlers and the scheduler

Since an interrupt handler blocks the highest priority task from running, and since real time operating systems are designed to keep thread latency to a minimum, interrupt handlers are typically kept as short as possible. The interrupt handler defers all interaction with the hardware as long as possible; typically all that is necessary is to acknowledge or disable the interrupt (so that it won't occur again when the interrupt handler returns). The interrupt handler then queues work to be done at a lower priority level, often by unblocking a driver task (through releasing a semaphore or sending a message). The scheduler often provides the ability to unblock a task from interrupt handler context.

Real-time operating systems (RTOS) LynxOS, Embedded Linux, Prex, Tron, WindowsCE, RTLinux, THEOS, OSE…

WEEK FOURTEEN

Learning Outcome for this week
- Explain batch modes with respect to compilation and library - Batch Processing, Time sharing, Real time, and network operating systems - Multiprogramming, Multitasking and Multiprocessing systems

BATCH PROCESSING

BATCH Processing can be defined as executing a series of non interactive jobs all at one time. The term originated in the days when users entered programs on punch cards. They would give a batch of these programmed cards to the system operator, who would feed them into the computer. Batch jobs can be stored up during working hours and then executed during the evening or whenever the computer is idle. Batch processing is particularly useful for operations that require the computer or a peripheral device for an extended period of time. Once a batch job begins, it continues until it is done or until an error occurs. Note that batch processing implies that there is no interaction with the user while the program is being executed.

An example of batch processing is the way that credit card companies process billing or Power Holding Company processes their bills . The customer does not receive a bill for each separate credit card purchase or meter reading but one monthly bill for all of that month. The bill is created through batch processing, where all of the data are collected and held until the bill is processed as a batch at the end of the billing cycle.

The opposite of batch processing is transaction processing or interactive processing. In interactive processing, the application responds to commands as soon as you enter them.

TIME SHARING

This involves the CPU allocating individual slices of time to a number of users on the computer system. As the number of users increases the response time for each terminal declines. The speed of the CPU compared to that of the VDU and terminal is so much faster that it gives the user the impression that they are the sole user of the system

MULTIPROGRAMMING MULTITASKING AND MULTIPROCESSING SYSTEMS

Multiprograming: In multiprogramming systems, the running task keeps running until it performs an operation that requires waiting for an external event (e.g. reading from a tape) or until the computer's scheduler forcibly swaps the running task out of the CPU. Multiprogramming systems are designed to maximize CPU usage.

Multitasking: In computing, multitasking is a method by which multiple tasks, also known as processes, share common processing resources such as a CPU. In the case of a computer with a single CPU, only one task is said to be running at any point in time, meaning that the CPU is actively executing instructions for that task. Multitasking solves the problem by scheduling which task may be the one running at any given time, and when another waiting task gets a turn. The act of reassigning a CPU from one task to another one is called a context switch.

Multiprocessing: Multiprocessing is a generic term for the use of two or more central processing units (CPUs) within a single computer system. There are many variations on this basic theme, and the definition of multiprocessing can vary with context, mostly as a function of how CPUs are defined (multiple cores on one die, multiple chips in one package, multiple packages in one system unit, etc.).

Multiprocessing sometimes refers to the execution of multiple concurrent software processes in a system as opposed to a single process at any one instant.

REAL TIME OPERATING SYSTEM

Real-Time Operating System (RTOS; generally pronounced as "R-toss") is a multitasking operating system intended for real-time applications. Such applications include embedded systems (programmable thermostats, household appliance controllers), industrial robots, spacecraft, industrial control and scientific research equipment.

A RTOS facilitates the creation of a real-time system, but does not guarantee the final result will be real-time; this requires correct development of the software. An RTOS does not necessarily have high throughput; rather, an RTOS provides facilities which, if used properly, guarantee deadlines can be met generally (soft real-time) or deterministically (hard real-time). An RTOS will typically use specialized scheduling algorithms in order to provide the real-time developer with the tools necessary to produce deterministic behavior in the final system. An RTOS is valued more for how quickly and/or predictably it can respond to a particular event than for the given amount of work it can perform over time. Key factors in an RTOS are therefore a minimal interrupt latency and a minimal thread switching latency.

An early example of a large-scale real-time operating system can be identified in the some Airline reservation operating in Nigeria and overseas such as Transaction Processing Facility developed by American Airlines and IBM for the Sabre Airline Reservations System. Others may be found in some communication companies

.

NETWORK OPERATING SYSTEM

What is a Network Operating System?

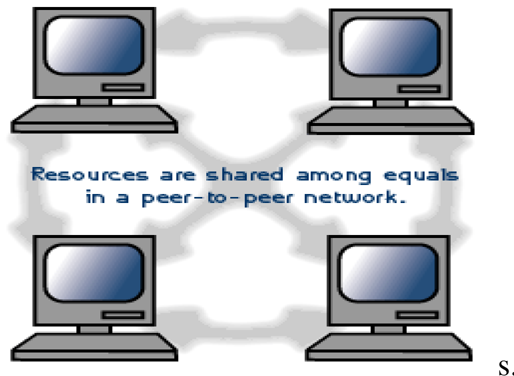
Unlike operating systems, such as DOS and Windows, that are designed for single users to control one computer, network operating systems (NOS) coordinate the activities of multiple computers across a network. The network operating system acts as a director to keep the network running smoothly.

The two major types of network operating systems are:

- Peer-to-Peer
- Client/Server

Peer-to-Peer

Peer-to-peer network operating systems allow users to share resources and files located on their computers and to access shared resources found on other computers. However, they do not have a file server or a centralized management source (See fig. below). In a peer-to-peer network, all computers are considered equal; they all have the same abilities to use the resources available on the network. Peer-to-peer networks are designed primarily for small to medium local area networks. AppleShare and Windows for Workgroups are examples of



s.

Fig 14.1 . Peer-to-peer network

programs that can function as peer-to-peer network operating system

Advantages of a peer-to-peer network:

- Less initial expense - No need for a dedicated server.
- Setup - An operating system (such as Windows XP) already in place may only need to be reconfigured for peer-to-peer operations.

Disadvantages of a peer-to-peer network:

- Decentralized - No central repository for files and applications.
- Security - Does not provide the security available on a client/server network.

CLIENT/SERVER

Client/server network operating systems allow the network to centralize functions and applications in one or more dedicated file servers (See fig. below). The file servers become the heart of the system, providing access to resources and providing security. Individual workstations (clients) have access to the resources available on the file servers. The network operating system provides the mechanism to integrate all the components of the network and allow multiple users to simultaneously share the same resources irrespective of physical location. Novell Netware and Windows 2000 Server are examples of client/server network operating systems.

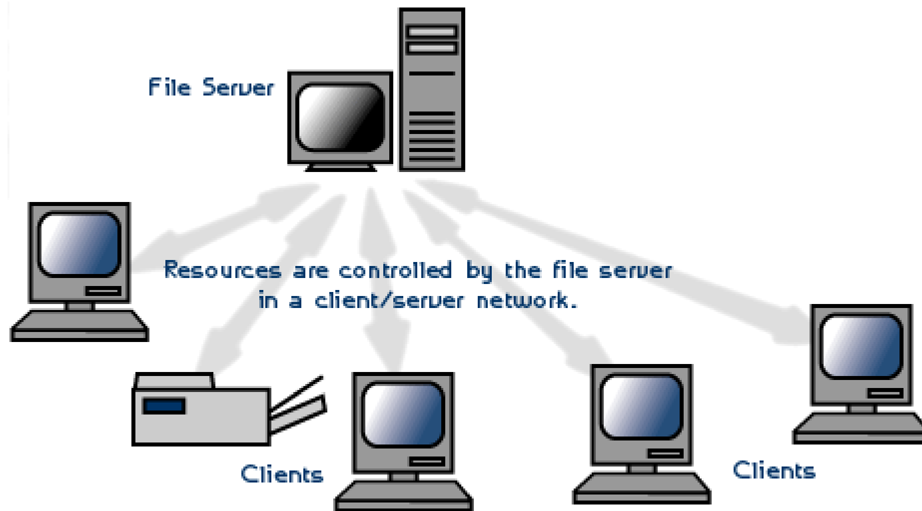


Fig. 14.2 . Client/server network

Advantages of a client/server network:

- Centralized - Resources and data security are controlled through the server.
- Scalability - Any or all elements can be replaced individually as needs increase.
- Flexibility - New technology can be easily integrated into system.
- Interoperability - All components (client/network/server) work together.
- Accessibility - Server can be accessed remotely and across multiple platforms.

Disadvantages of a client/server network:

- Expense - Requires initial investment in dedicated server.
- Maintenance - Large networks will require a staff to ensure efficient operation.
- Dependence - When server goes down, operations will cease across the network.

Examples of network operating systems

The following list includes some of the more popular peer-to-peer and client/server network operating systems.

- AppleShare
- Microsoft Windows Server
- Novell Netware

WEEK FIFTEEN

Learning Outcome this Week: Revision of CONCEPTS LEARNT DURING THE COURSE:

- Basic Assembler 's functions
- Differentiate between Assembler, Interpreter and Compiler
- Make argument on which one is preferable for a given task
- Understand Front end and back end
- Describe tools for error checking
- Discuss the functions of an Operating System
- Identify the features of Ms Dos against Ms Windows
- The meaning and work of 1-pass assembler, 2-pass assembler.

Teacher's Activities:

- The teacher is expected to revise the above areas with students
- Identify areas where difficulties in understanding appear.
- Attempts should be made at assessing knowledge and understanding .
- At least two tests, one hour each , should be conducted as assessment before a final exam.
- Home work should also be given for additional materials that the student may need to increase his understanding of some of the concepts covered.

REVIEW QUESTIONS

1. What is the general format of an assembler instruction? What is the meaning of each field ?
2. What is the difference between a label and a symbol?
3. List some typical zero-operand instructions.
4. In old assemblers the source .le was punched on cards, one line per card. Why was it important to punch a sequence number on each card?
5. Use your knowledge of data structures; what are good data structures for an Op-Code table? The table is static (no insertions or deletions) and is searched very often.

HOME ASSIGNMENT QUESTIONS

1. For each of the questions below, if the project describes a 2-pass assembler, design a format for the intermediate .le. What information should each record contain?

2 Look at several textbooks on assembler language programming for different computers. What are the rules for:

a. Symbol names. b. The syntax of a source line.

8. The asterisk '*'is a favorite character of assembler writers and has been mentioned in this lecture many times, in connection with several different assembler features. What are those features?
9. Compare and contrast literals and the immediate mode. What are the advantages and Disadvantages of each?