UNIT – 1

SYLLABUS

Introduction –System Software and machine architecture. Loader and Linkers: Basic Loader Functions - Machine dependent loader features –Machine independent loader features - Loader design options.

Introduction

The subject introduces the design and implementation of system software. Software is set of instructions or programs written to carry out certain task on digital computers. It is classified into system software and application software. System software consists of a variety of programs that support the operation of a computer. Application software focuses on an application or problem to be solved. System software consists of a variety of programs that support the operation of a computer. Examples for system software are Operating system, compiler, assembler, macro processor, loader or linker, debugger, text editor, database management systems (some of them) and, software engineering tools. These software's make it possible for the user to focus on an application or other problem to be solved, without needing to know the details of how the machine works internally.

System Software and Machine Architecture

One characteristic in which most system software differs from application software is machine dependency.

System software – support operation and use of computer. Application software - solution to a problem. Assembler translates mnemonic instructions into machine code. The instruction formats, addressing modes etc., are of direct concern in assembler design. Similarly, Compilers must generate machine language code, taking into account such hardware characteristics as the number and type of registers and the machine instructions available. Operating systems are directly concerned with the management of nearly all of the resources of a computing system.

There are aspects of system software that do not directly depend upon the type of computing system, general design and logic of an assembler, general design and logic of a compiler and, code optimization techniques, which are independent of target machines. Likewise, the process of linking together independently assembled subprograms does not usually depend on the computer being used.

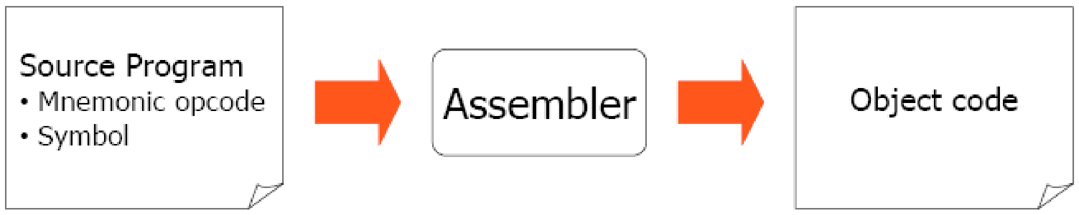
Assembler Design

Assembler is system software which is used to convert an assembly language program to its equivalent object code. The input to the assembler is a source code written in assembly language (using mnemonics) and the output is the object code. The design of an assembler depends upon the machine architecture as the language used is mnemonic language.

Basic Assembler Functions:

The basic assembler functions are:

- Translating mnemonic language code to its equivalent object code.
- Assigning machine addresses to symbolic labels.



- The design of assembler can be to perform the following:
  - Scanning (tokenizing)
  - Parsing (validating the instructions)
  - Creating the symbol table
  - Resolving the forward references
  - Converting into the machine language
- The design of assembler in other words:
  - Convert mnemonic operation codes to their machine language equivalents
  - Convert symbolic operands to their equivalent machine addresses
  - Decide the proper instruction format Convert the data constants to internal machine representations
  - Write the object program and the assembly listing

So for the design of the assembler we need to concentrate on the machine architecture of the SIC/XE machine. We need to identify the algorithms and the various data structures to be used. According to the above required steps for assembling the assembler also has to handle *assembler directives*, these do not generate the object code but directs the assembler to perform certain operation. These directives are:

- SIC Assembler Directive:
  - START: Specify name & starting address.
  - END: End of the program, specify the first execution instruction.
  - BYTE, WORD, RESB, RESW
  - End of record: a null char(00)

    End of file: a zero length record

The assembler design can be done:

- Single pass assembler
- Multi-pass assembler

Single-pass Assembler:

In this case the whole process of scanning, parsing, and object code conversion is done in single pass. The only problem with this method is resolving forward reference. This is shown with an example below:

```
10      1000            FIRST           STL     RETADR                  141033
--
--
--
--
95      1033            RETADR          RESW            1
```

In the above example in line number 10 the instruction STL will store the linkage register with the contents of RETADR. But during the processing of this instruction the value of this symbol is not known as it is defined at the line number 95. Since I single-pass assembler the scanning, parsing and object code conversion happens simultaneously. The instruction is fetched; it is scanned for tokens, parsed for syntax and semantic validity. If it valid then it has to be converted to its equivalent object code. For this the object code is generated for the opcode STL and the value for the symbol RETADR need to be added, which is not available.

Due to this reason usually the design is done in two passes. So a multi-pass assembler resolves the forward references and then converts into the object code. Hence the process of the multi-pass assembler can be as follows:

Pass-1

- Assign addresses to all the statements
- Save the addresses assigned to all labels to be used in *Pass-2*
- Perform some processing of assembler directives such as RESW, RESB to find the length of data areas for assigning the address values.
- Defines the symbols in the symbol table(generate the symbol table)

Pass-2

- Assemble the instructions (translating operation codes and looking up addresses).
- Generate data values defined by BYTE, WORD etc.
- Perform the processing of the assembler directives not done during *pass-1*.
- Write the object program and assembler listing.

Assembler Design:

The most important things which need to be concentrated is the generation of Symbol table and resolving *forward references*.

- Symbol Table:
  - This is created during pass 1
  - All the labels of the instructions are symbols
  - Table has entry for symbol name, address value.
- Forward reference:
  - Symbols that are defined in the later part of the program are called forward referencing.
  - There will not be any address value for such symbols in the symbol table in pass 1.

Example Program:

The example program considered here has a main module, two subroutines

- 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

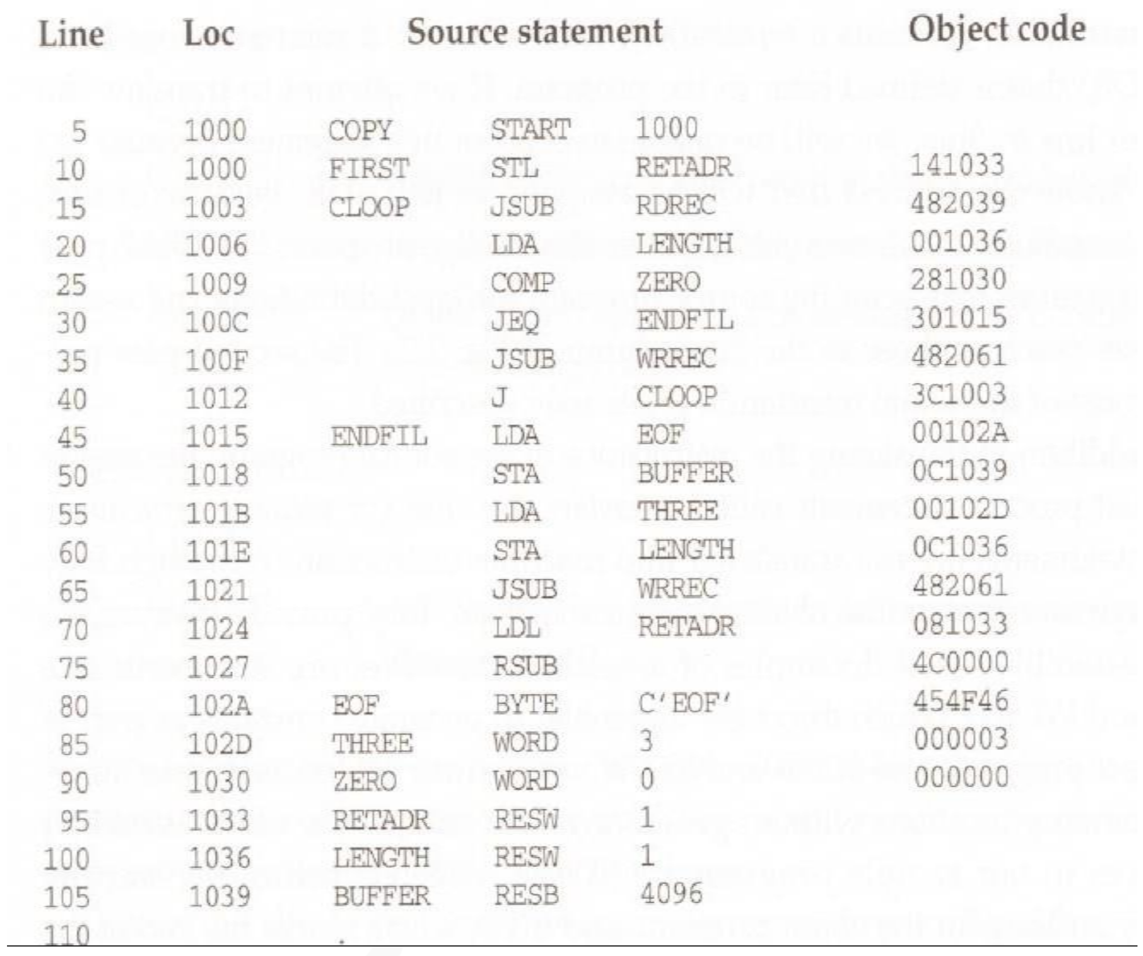
  -The end of each record is marked with a null character (00)16

  -The end of the file is indicated by a zero-length record

- Subroutines (JSUB, RSUB)

  -RDREC, WRREC

  -Save link register first before nested jump



Line	Loc	Source statement			Object code
5	1000	COPY	START	1000	
10	1000	FIRST	STL	RETADR	141033
15	1003	CLOOP	JSUB	RDREC	482039
20	1006		LDA	LENGTH	001036
25	1009		COMP	ZERO	281030
30	100C		JEQ	ENDFIL	301015
35	100F		JSUB	WRREC	482061
40	1012		J	CLOOP	3C1003
45	1015	ENDFIL	LDA	EOF	00102A
50	1018		STA	BUFFER	0C1039
55	101B		LDA	THREE	00102D
60	101E		STA	LENGTH	0C1036
65	1021		JSUB	WRREC	482061
70	1024		LDL	RETADR	081033
75	1027		RSUB		4C0000
80	102A	EOF	BYTE	C'EOF'	454F46
85	102D	THREE	WORD	3	000003
90	1030	ZERO	WORD	0	000000
95	1033	RETADR	RESW	1	
100	1036	LENGTH	RESW	1	
105	1039	BUFFER	RESB	4096	
110		.			

The first column shows the line number for that instruction, second column shows the addresses allocated to each instruction. The third column indicates the labels given to the statement, and is followed by the instruction consisting of opcode and operand. The last column gives the equivalent object code.

The *object code* later will be loaded into memory for execution. The simple object program we use contains three types of records:

• Header record

- Col. 1 H
- Col. 2~7 Program name
- Col. 8~13 Starting address of object program (hex)
- Col. 14~19 Length of object program in bytes (hex)

• Text record

- Col. 1 T
- Col. 2~7 Starting address for object code in this record (hex)
- Col. 8~9 Length of object code in this record in bytes (hex)
- Col. 10~69 Object code, represented in hex (2 col. per byte)

• End record

- Col.1 E
- Col.2~7 Address of first executable instruction in object program (hex) "^" is only for separation only

Object code for the example program:

Some of the features in the program depend on the architecture of the machine. If the program is for SIC machine, then we have only limited instruction formats and hence limited addressing modes. We have only single operand instructions. The operand is always a memory reference. Anything to be fetched from memory requires more time. Hence the improved version of SIC/XE machine provides more instruction formats and hence more addressing modes. The moment we change the machine architecture the availability of number of instruction formats and the addressing modes changes. Therefore the design usually requires considering two things: Machine-dependent features and Machine-independent features.

Machine-Dependent Features:

- Instruction formats and addressing modes
- Program relocation

Instruction formats and Addressing Modes

The instruction formats depend on the memory organization and the size of the memory. In SIC machine the memory is byte addressable. Word size is 3 bytes. So the size of the memory is 2^{12} bytes. Accordingly it supports only one instruction format. It has only two registers: register A and Index register. Therefore the addressing modes supported by this architecture are direct, indirect, and indexed. Whereas the memory of a SIC/XE machine is 2^{20} bytes (1 MB). This supports four different types of instruction types, they are:

- 1 byte instruction
- 2 byte instruction
- 3 byte instruction
- 4 byte instruction
- Instructions can be:
  - Instructions involving register to register
  - Instructions with one operand in memory, the other in Accumulator (Single operand instruction)
  - Extended instruction format
- Addressing Modes are:
  - Index Addressing(SIC): Opcode m, x
  - Indirect Addressing: Opcode @m
  - PC-relative: Opcode m
  - Base relative: Opcode m
  - Immediate addressing: Opcode #c

1. Translations for the Instruction involving Register-Register addressing mode:

During pass 1 the registers can be entered as part of the symbol table itself. The value for these registers is their equivalent numeric codes. **During pass 2,** these values are assembled along with the mnemonics object code. If required a separate table can be created with the register names and their equivalent numeric values.

2. Translation involving Register-Memory instructions:

In SIC/XE machine there are four instruction formats and five addressing modes. For formats and addressing modes refer chapter 1.

Among the instruction formats, format -3 and format-4 instructions are Register-Memory type of instruction. One of the operand is always in a register and the other operand is in the memory. The addressing mode tells us the way in which the operand from the memory is to be fetched.

There are two ways: *Program-counter relative and Base-relative.* This addressing mode can be represented by either using format-3 type or format-4 type of instruction format. In format-3, the instruction has the opcode followed by a 12-bit displacement value in the address field. Where as in format-4 the instruction contains the mnemonic code followed by a 20-bit displacement value in the address field.

2. Program-Counter Relative: In this usually format-3 instruction format is used. The instruction contains the opcode followed by a 12-bit displacement value. The range of displacement values are from 0 -2048. This displacement (should be small enough to fit in a 12-bit field) value is added to the current contents of the program counter to get the target address of the operand required by the instruction. This is relative way of calculating the address of the operand relative to the program counter. Hence the displacement of the operand is relative to the current program counter value. The following example shows how the address is calculated:

3. Base-Relative Addressing Mode: in this mode the base register is used to mention the displacement value. Therefore the target address is

TA = (base) + displacement value

This addressing mode is used when the range of displacement value is not sufficient. Hence the operand is not relative to the instruction as in PC-relative addressing mode. Whenever this mode is used it is indicated by using a directive BASE. The moment the assembler encounters this directive the next instruction uses base-relative addressing mode to calculate the target address of the operand.

When NOBASE directive is used then it indicates the base register is no more used to calculate the target address of the operand. Assembler first chooses PC-relative, when the displacement field is not enough it uses Base-relative.

LDB #LENGTH (*instruction*) BASE

LENGTH (*directive*)

: NOBASE

For example:

```
12    0003   LDB       #LENGTH              69202D
13           BASE      LENGTH
: :
100   0033   LENGTH    RESW       1
105   0036   BUFFER    RESB       4096
: :
160   104E   STCH       BUFFER,   X         57C003
165   1051   TIXR      T          B850
```

In the above example the use of directive BASE indicates that Base-relative addressing mode is to be used to calculate the target address. PC-relative is no longer used. The value of the LENGTH is stored in the base register. If PC-relative is used then the target address calculated is:

The LDB instruction loads the value of length in the base register which 0033. BASE directive explicitly tells the assembler that it has the value of LENGTH.

BUFFER is at location $(0036)_{16}$

$(B) = (0033)_{16}$

disp = 0036 – 0033 = $(0003)_{16}$

```
  op      n i  x b p e      disp
010101    1 1  1 1 0 0      003    ⇒  57C003
```

```
20    000A          LDA       LENGTH          032026
```

```
: :
175     1056    EXIT           STX          LENGTH              134000
```

Consider Line 175. If we use PC-relative

Disp = TA – (PC) = 0033 –1059 = EFDA

PC relative is no longer applicable, so we try to use BASE relative addressing mode.

4. Immediate Addressing Mode

In this mode no memory reference is involved. If immediate mode is used the target address is the operand itself.

If the symbol is referred in the instruction as the immediate operand then it is immediate with PC-relative mode as shown in the example below:

***5. Indirect and PC-relative mode*:**

In this type of instruction the symbol used in the instruction is the address of the location which contains the address of the operand. The address of this is found using PC-relative addressing mode.

The instruction jumps the control to the address location RETADR which in turn has the address of the operand. If address of RETADR is 0030, the target address is then 0003 as calculated above.

3.2 Program Relocation

Sometimes it is required to load and run several programs at the same time. The system must be able to load these programs wherever there is place in the memory. Therefore the exact starting is not known until the load time.

Absolute Program

In this the address is mentioned during assembling itself. This is called *Absolute Assembly.*

Consider the instruction:

```
55      101B                    LDA             THREE           00102D
```

This statement says that the register A is loaded with the value stored at location 102D. Suppose it is decided to load and execute the program at location 2000 instead of location 1000. Then at address 102D the required value which needs to be loaded in the register A is no more available. The address also gets changed relative to the displacement of the program. Hence we need to make some changes in the address portion of the instruction so that we can load and execute the program at location 2000. Apart from the instruction which will undergo a change in their operand address value as the program load address changes. There exist some parts in the program which will remain same regardless of where the program is being loaded.

Since assembler will not know actual location where the program will get loaded, it cannot make the necessary changes in the addresses used in the program. However, the assembler identifies for the loader those parts of the program which need modification. An object program that has the information necessary to perform this kind of modification is called the relocatable program.

3.2.5 Control Sections:

A *control section* is a part of the program that maintains its identity after assembly; each control section can be loaded and relocated independently of the others. Different control sections are most often used for subroutines or other logical subdivisions. The programmer can assemble, load, and manipulate each of these control sections separately.

Because of this, there should be some means for linking control sections together. For example, instructions in one control section may refer to the data or instructions of other control sections. Since control sections are independently loaded and relocated, the assembler is unable to process these references in the usual way. Such references between different control sections are called *external references.*

The assembler generates the information about each of the external references that will allow the loader to perform the required linking. When a program is written using multiple control sections, the beginning of each of the control section is indicated by an assembler directive

– assembler directive: **CSECT**

The syntax

secname CSECT

– separate location counter for each control section

Control sections differ from program blocks in that they are handled separately by the assembler. Symbols that are defined in one control section may not be used directly another control section; they must be identified as external reference for the loader to handle. The external references are indicated by two assembler directives:

EXTDEF (external Definition):

It is the statement in a control section, names symbols that are defined in this section but may be used by other control sections. Control section names do not need to be named in the EXTREF as they are automatically considered as external symbols.

EXTREF (external Reference):

It names symbols that are used in this section but are defined in some other control section.

The order in which these symbols are listed is not significant. The assembler must include proper information about the external references in the object program that will cause the loader to insert the proper value where they are required.

Handling External Reference

Case 1

```
15      0003        CLOOP      +JSUB       RDREC              4B100000
```

- The operand RDREC is an external reference.
    - The assembler has no idea where RDREC is
    - inserts an address of zero
    - can only use extended format to provide enough room (that is, relative addressing for external reference is invalid)

- The assembler generates information for each external reference that will allow the loader to perform the required linking.

Case 2

```
190     0028    MAXLEN      WORD        BUFEND-BUFFER                   000000
```

- There are two external references in the expression, BUFEND and BUFFER.
- The assembler inserts a value of zero
- passes information to the loader
- Add to this data area the address of BUFEND
- Subtract from this data area the address of BUFFER

Case 3

On line 107, BUFEND and BUFFER are defined in the same control section and the expression can be calculated immediately.

```
107      1000   MAXLEN      EQU         BUFEND-BUFFER
```

Object Code for the example program:

The assembler must also include information in the object program that will cause the loader to insert the proper value where they are required. The assembler maintains two new record in the object code and a changed version of modification record.

Define record (EXTDEF)

• Col. 1	D
• Col. 2-7	Name of external symbol defined in this control section
• Col. 8-13	Relative address within this control section (hexadecimal)
• Col.14-73	Repeat information in Col. 2-13 for other external symbols

Refer record (EXTREF)

• Col. 1	R
• Col. 2-7	Name of external symbol referred to in this control section
• Col. 8-73	Name of other external reference symbols

Modification record

• Col. 1	M
• Col. 2-7	Starting address of the field to be modified (hexadecimal)

- Col. 8-9 Length of the field to be modified, in half-bytes (hexadecimal)
- Col.11-16 External symbol whose value is to be added to or subtracted from the indicated field

A define record gives information about the external symbols that are defined in this control section, i.e., symbols named by EXTDEF.

A refer record lists the symbols that are used as external references by the control section, i.e., symbols named by EXTREF.

The new items in the modification record specify the modification to be performed: adding or subtracting the value of some external symbol. The symbol used for modification my be defined either in this control section or in another section.

The object program is shown below. There is a separate object program for each of the control sections. In the *Define Record* and *refer record* the symbols named in EXTDEF and EXTREF are included.

In the case of *Define*, the record also indicates the relative address of each external symbol within the control section.

For EXTREF symbols, no address information is available. These symbols are simply named in the *Refer record.*

Handling Expressions in Multiple Control Sections:

The existence of multiple control sections that can be relocated independently of one another makes the handling of expressions complicated. It is required that in an expression that all the relative terms be paired (for absolute expression), or that all except one be paired (for relative expressions).

When it comes in a program having multiple control sections then we have an extended restriction that:

- Both terms in each pair of an expression must be within the same control section
    - If two terms represent relative locations within the same control section , their difference is an absolute value (regardless of where the control section is located.

        - **Legal:** BUFEND-BUFFER (both are in the same control section)

    - If the terms are located in different control sections, their difference has a value that is unpredictable.

        - **Illegal:** RDREC-COPY (both are of different control section) it is the difference in the load addresses of the two control sections. This value depends on the way run-time storage is allocated; it is unlikely to be of any use.

- **How to enforce this restriction**
    - When an expression involves external references, the assembler cannot determine whether or not the expression is legal.
    - The assembler evaluates all of the terms it can, combines these to form an initial expression value, and generates Modification records.
    - The loader checks the expression for errors and finishes the evaluation.

ASSEMBLER DESIGN

Here we are discussing

- The structure and logic of one-pass assembler. These assemblers are used when it is necessary or desirable to avoid a second pass over the source program.
- Notion of a multi-pass assembler, an extension of two-pass assembler that allows an assembler to handle forward references during symbol definition.

One-Pass Assembler

The main problem in designing the assembler using single pass was to resolve forward references. We can avoid to some extent the forward references by:

- Eliminating forward reference to data items, by defining all the storage reservation statements at the beginning of the program rather at the end.
- Unfortunately, forward reference to labels on the instructions cannot be avoided.

(forward jumping)

- To provide some provision for handling forward references by prohibiting forward references to data items.

There are two types of one-pass assemblers:

- One that produces object code directly in memory for immediate execution (Load-and-go assemblers).
- The other type produces the usual kind of object code for later execution.

Load-and-Go Assembler

- Load-and-go assembler generates their object code in memory for immediate execution.
- No object program is written out, no loader is needed.
- It is useful in a system with frequent program development and testing
    - The efficiency of the assembly process is an important consideration.
- Programs are re-assembled nearly every time they are run; efficiency of the assembly process is an important consideration.

Forward Reference in One-Pass Assemblers: In load-and-Go assemblers when a forward reference is encountered :

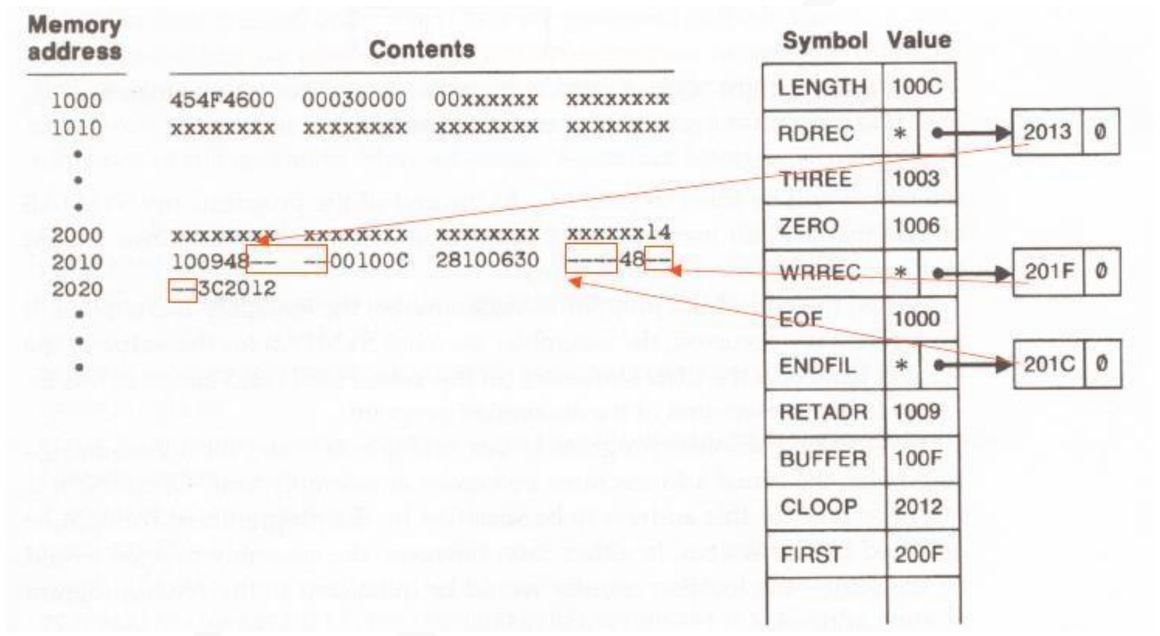
- Omits the operand address if the symbol has not yet been defined
- Enters this undefined symbol into SYMTAB and indicates that it is undefined
- Adds the address of this operand address to a list of forward references associated with the SYMTAB entry
- When the definition for the symbol is encountered, scans the reference list and inserts the address.
- At the end of the program, reports the error if there are still SYMTAB entries indicated undefined symbols.
- For Load-and-Go assembler

- Search SYMTAB for the symbol named in the END statement and jumps to this location to begin execution if there is no error

After Scanning line 40 of the program:

40 2021 J` CLOOP 302012

The status is that upto this point the symbol RREC is referred once at location 2013, ENDFIL at 201F and WRREC at location 201C. None of these symbols are defined. The figure shows that how the pending definitions along with their addresses are included in the symbol table.



The status after scanning line 160, which has encountered the definition of RDREC and ENDFIL is as given below:

If One-Pass needs to generate object code:

- If the operand contains an undefined symbol, use 0 as the address and write the Text record to the object program.
- Forward references are entered into lists as in the load-and-go assembler.
- When the definition of a symbol is encountered, the assembler generates another Text record with the correct operand address of each entry in the reference list.
- When loaded, the incorrect address 0 will be updated by the latter Text record containing the symbol definition.

Multi_Pass Assembler:

- For a two pass assembler, forward references in symbol definition are not allowed:

```
ALPHA       EQU   BETA
BETA        EQU   DELTA
DELTA       RESW 1
```

  - Symbol definition must be completed in pass 1.
- Prohibiting forward references in symbol definition is not a serious inconvenience.
  - Forward references tend to create difficulty for a person reading the program.

Implementation Issues for Modified Two-Pass Assembler:

Implementation Isuues when forward referencing is encountered in *Symbol Defining statements* :

- For a forward reference in symbol definition, we store in the SYMTAB:
  - The symbol name
  - The defining expression
  - The number of undefined symbols in the defining expression
- The undefined symbol (marked with a flag *) associated with a list of symbols depend on this undefined symbol.

Loaders and Linkers

This Chapter gives you…

- Basic Loader Functions
- Machine-Dependent Loader Features
- Machine-Independent Loader Features
- Loader Design Options
- Implementation Examples
- Introduction

The Source Program written in assembly language or high level language will be converted to object program, which is in the machine language form for execution. This conversion either from assembler or from compiler, contains translated instructions and data values from the source program, or specifies addresses in primary memory where these items are to be loaded for execution.

This contains the following three processes, and they are,

Loading - which allocates memory location and brings the object program into memory for execution - (Loader)

Linking- which combines two or more separate object programs and supplies the information needed to allow references between them - (Linker)

Relocation - which modifies the object program so that it can be loaded at an address different from the location originally specified - (Linking Loader)

Basic Loader Functions

A loader is a system program that performs the loading function. It brings object program into memory and starts its execution. The role of loader is as shown in the figure 3.1. In figure 3.1 translator may be assembler/complier, which generates the object program and later loaded to the memory by the loader for execution. In figure 3.2 the translator is specifically an assembler, which generates the object loaded, which becomes input to the loader. The figure 3.3 shows the role of both loader and linker.

Type of Loaders

The different types of loaders are, absolute loader, bootstrap loader, relocating loader (relative loader), and, direct linking loader. The following sections discuss the functions and design of all these types of loaders.

Absolute Loader

The operation of absolute loader is very simple. The object code is loaded to specified locations in the memory. At the end the loader jumps to the specified address to begin execution of the loaded program. The role of absolute loader is as shown in the figure 3.3.1. The advantage of absolute loader is simple and efficient. But the disadvantages are, the need for programmer to specify the actual address, and, difficult to use subroutine libraries.

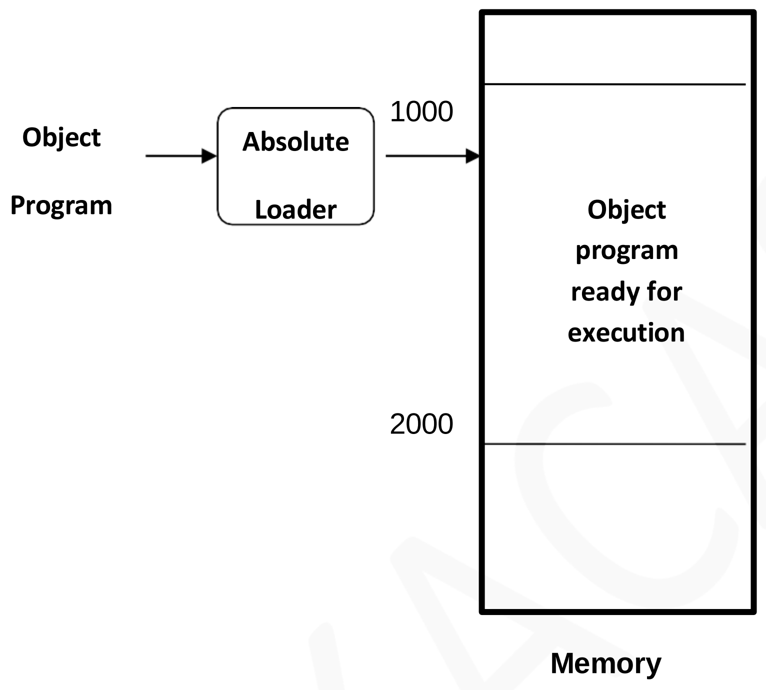


Figure 3.3.1: The Role of Absolute Loader

The algorithm for this type of loader is given here. The object program and, the object program loaded into memory by the absolute loader are also shown. Each byte of assembled code is given using its hexadecimal representation in character form. Easy to read by human beings. Each byte of object code is stored as a single byte. Most machine store object programs in a binary form, and we must be sure that our file and device conventions do not cause some of the program bytes to be interpreted as control characters.

```
Begin
read Header record
verify program name and length
read first Text record
```

```
while record type is <> 'E' do
        begin
        {if object code is in character form, convert into internal representation}
        move object code to specified location in memory
        read next object program record
        end
jump to address specified in End record
end
```

A Simple Bootstrap Loader

When a computer is first turned on or restarted, a special type of absolute loader, called bootstrap loader is executed. This bootstrap loads the first program to be run by the computer -- usually an operating system. The bootstrap itself begins at address 0. It loads the OS starting address 0x80. No header record or control information, the object code is consecutive bytes of memory.

The algorithm for the bootstrap loader is as follows

```
Begin
X=0x80 (the address of the next memory location to be loaded
Loop
        A←GETC (and convert it from the ASCII character
       code to the value of the hexadecimal digit)
        save the value in the high-order 4 bits of S
        A←GETC
        combine the value to form one byte A← (A+S)
        store the value (in A) to the address in register X
        X←X+1
End
```

It uses a subroutine GETC, which is

```
GETC    A←read one character
        if A=0x04 then jump to 0x80
        if A<48 then GETC
        A ← A-48 (0x30)
        if A<10 then return
        A ← A-7
        return
```

Machine-Dependent Loader Features

Absolute loader is simple and efficient, but the scheme has potential disadvantages One of the most disadvantage is the programmer has to specify the actual starting address, from where the program to be loaded. This does not create difficulty, if one program to run, but not for several programs. Further it is difficult to use subroutine libraries efficiently.

This needs the design and implementation of a more complex loader. The loader must provide program relocation and linking, as well as simple loading functions.

Relocation

The concept of program relocation is, the execution of the object program using any part of the available and sufficient memory. The object program is loaded into memory wherever there is room for it. The actual starting address of the object program is not known until load time. Relocation provides the efficient sharing of the machine with larger memory and when several independent programs are to be run together. It also supports the use of subroutine libraries efficiently. Loaders that allow for program relocation are called relocating loaders or relative loaders.

Methods for specifying relocation

Use of modification record and, use of relocation bit, are the methods available for specifying relocation. In the case of modification record, a modification record M is used in the object program to specify any relocation. In the case of use of relocation bit, each instruction is associated with one relocation bit and, these relocation bits in a Text record is gathered into bit masks.

Modification records are used in complex machines and is also called Relocation and Linkage Directory (RLD) specification. The format of the modification record (M) is as follows. The object program with relocation by Modification records is also shown here.

Modification record

col 1:	M
col 2-7:	relocation address
col 8-9:	length (halfbyte)
col 10:	flag (+/-)
col 11-17:	segment name

```
HʌCOPY ʌ000000 001077
Tʌ000000 ʌ1DΛ17202DΛ69202Dʌ48101036ʌ…ʌ4B105Dʌ3F2FECʌ032010
Tʌ00001Dʌ13ʌ0F2016ʌ010003ʌ0F200Dʌ4B10105Dʌ3E2003ʌ454F46
Tʌ001035 ʌ1DʌB410ʌB400ʌB440ʌ75101000ʌ…ʌ332008ʌ57C003ʌB850
Tʌ001053ʌ1Dʌ3B2FEAʌ134000ʌ4F0000ʌF1ʌ..ʌ53C003ʌDF2008ʌB850
Tʌ00070ʌ07ʌ3B2FEFʌ4F0000ʌ05
Mʌ000007ʌ05+COPY
Mʌ000014ʌ05+COPY
Mʌ000027ʌ05+COPY
Eʌ000000
```

The relocation bit method is used for simple machines. Relocation bit is 0: no modification is necessary, and is 1: modification is needed. This is specified in the columns 10-12 of text record (T), the format of text record, along with relocation bits is as follows.

Text record

col 1: T

col 2-7: starting address

col 8-9: length (byte)

col 10-12: relocation bits

col 13-72: object code

Twelve-bit mask is used in each Text record (col:10-12 – relocation bits), since each text record contains less than 12 words, unused words are set to 0, and, any value that is to be modified during relocation must coincide with one of these 3-byte segments. For absolute loader, there are no relocation bits column 10-69 contains object code. The object program with relocation by bit mask is as shown below. Observe FFC - means all ten words are to be modified and, E00 - means first three records are to be modified.

```
H^COPY ^000000 00107A
T^000000^1E^FFC^140033^481039^000036^280030^300015^…^3C0003 ^ …
T^00001E^15^E00^0C0036^481061^080033^4C0000^…^000003^000000
T^001039^1E^FFC^040030^000030^…^30103F^D8105D^280030^...
T^001057^0A^800^100036^4C0000^F1^001000
T^001061^19^FE0^040030^E01079^…^508039^DC1079^2C0036^...
E^000000
```

Program Linking

The Goal of program linking is to resolve the problems with external references (EXTREF) and external definitions (EXTDEF) from different control sections.

EXTDEF (external definition) - The EXTDEF statement in a control section names symbols, called external symbols, that are defined in this (present) control section and may be used by other sections.

ex: EXTDEF BUFFER, BUFFEND, LENGTH

EXTDEF LISTA, ENDA

EXTREF (external reference) - The EXTREF statement names symbols used in this (present) control section and are defined elsewhere.

ex: EXTREF RDREC, WRREC

EXTREF LISTB, ENDB, LISTC, ENDC

How to implement EXTDEF and EXTREF

The assembler must include information in the object program that will cause the loader to insert proper values where they are required – in the form of Define record (D) and, Refer record(R).

Define record

The format of the Define record (D) along with examples is as shown here**.**

Col. 1	D
Col. 2-7	Name of external symbol defined in this control section
Col. 8-13	Relative address within this control section (hexadecimal)
Col.14-73	Repeat information in Col. 2-13 for other external symbols

Example records

D LISTA 000040 ENDA 000054

D LISTB 000060 ENDB 000070

Refer record

The format of the Refer record (R) along with examples is as shown here**.**

Col. 1	R
Col. 2-7	Name of external symbol referred to in this control section

Col. 8-73   Name of other external reference symbols

Example records

R LISTB   ENDB   LISTC   ENDC

R LISTA   ENDA   LISTC   ENDC

R LISTA   ENDA   LISTB   ENDB

Here are the three programs named as PROGA, PROGB and PROGC, which are separately assembled and each of which consists of a single control section. LISTA, ENDA in PROGA, LISTB, ENDB in PROGB and LISTC, ENDC in PROGC are external definitions in each of the control sections. Similarly LISTB, ENDB, LISTC, ENDC in PROGA, LISTA, ENDA, LISTC, ENDC in PROGB, and LISTA, ENDA, LISTB, ENDB in PROGC, are external references. These sample programs given here are used to illustrate linking and relocation. The following figures give the sample programs and their corresponding object programs. Observe the object programs, which contain D and R records along with other records.

```
0000  PROGA    START      0
               EXTDEF   LISTA, ENDA
               EXTREF   LISTB, ENDB, LISTC, ENDC
               ………..
               ……….
0020  REF1     LDA      LISTA                        03201D
0023  REF2     +LDT     LISTB+4                    77100004
0027  REF3     LDX      #ENDA-LISTA                  050014
                     .
                     .
0040  LISTA    EQU      *

0054  ENDA     EQU      *
0054  REF4     WORD         ENDA-LISTA+LISTC         000014
0057  REF5     WORD         ENDC-LISTC-10            FFFFF6
```

```
005A  REF6    WORD        ENDC-LISTC+LISTA-1          00003F
005D  REF7    WORD        ENDA-LISTA-(ENDB-LISTB)     000014
0060  REF8    WORD        LISTB-LISTA                 FFFFC0
               END    REF1

0000  PROGB   START       0
              EXTDEF  LISTB, ENDB
              EXTREF  LISTA, ENDA, LISTC, ENDC
              ………..
              ……….
0036  REF1    +LDA    LISTA                          03100000
003A  REF2    LDT     LISTB+4                          772027
003D  REF3    +LDX    #ENDA-LISTA                     05100000
                     .
                     .
0060  LISTB   EQU     *
0070  ENDB    EQU     *
0070  REF4    WORD        ENDA-LISTA+LISTC            000000
0073  REF5    WORD        ENDC-LISTC-10               FFFFF6
0076  REF6    WORD        ENDC-LISTC+LISTA-1          FFFFFF
0079  REF7    WORD        ENDA-LISTA-(ENDB-LISTB)     FFFFF0
007C  REF8    WORD        LISTB-LISTA                 000060
              END
0000  PROGC   START       0
              EXTDEF  LISTC, ENDC
              EXTREF  LISTA, ENDA, LISTB, ENDB
              ………..
```

```
                    ………..
0018   REF1         +LDA       LISTA                               03100000
001C   REF2         +LDT       LISTB+4                             77100004
0020   REF3         +LDX       #ENDA-LISTA                         05100000
                          .                     .
0030   LISTC        EQU        *


0042   ENDC         EQU        *
0042   REF4         WORD            ENDA-LISTA+LISTC               000030
0045   REF5         WORD            ENDC-LISTC-10                  000008
0045   REF6         WORD            ENDC-LISTC+LISTA-1             000011
004B   REF7         WORD            ENDA-LISTA-(ENDB-LISTB)         000000
004E   REF8         WORD            LISTB-LISTA                    000000 END
```

H **PROGA** 000000 000063

D LISTA   000040 ENDA   000054

R LISTB   ENDB   LISTC   ENDC

.

.

T 000020 0A 03201D 77100004 050014

.

.

T 000054 0F 000014 FFFF6 00003F 000014 FFFFC0

M000024 05+LISTB

M000054 06+LISTC

M000057 06+ENDC

M000057 06 -LISTC

M00005A06+ENDC

M00005A06 -LISTC

M00005A06+PROGA

M00005D06-ENDB

M00005D06+LISTB

M00006006+LISTB

M00006006-PROGA

E000020

H **PROGB** 000000 00007F

D LISTB   000060 ENDB   000070

R LISTA   ENDA   LISTC   ENDC

.

T 000036 0B 03100000 772027 05100000

.

T 000007 0F 000000 FFFFF6 FFFFFF FFFFF0 000060

M000037 05+LISTA

M00003E 06+ENDA

M00003E 06 -LISTA

M000070 06 +ENDA

M000070 06 -LISTA

M000070 06 +LISTC

M000073 06 +ENDC

M000073 06 -LISTC

M000073 06 +ENDC

M000076 06 -LISTC

M000076 06+LISTA

M000079 06+ENDA

M000079 06 -LISTA

M00007C 06+PROGB

M00007C 06-LISTA

E

H **PROGC** 000000 000051

D LISTC 000030 ENDC 000042

R LISTA ENDA LISTB ENDB

.

T 000018 0C 03100000 77100004 05100000

.

T 000042 0F 000030 000008 000011 000000 000000

M000019 05+LISTA

M00001D 06+LISTB

M000021 06+ENDA

M000021 06 -LISTA

M000042 06+ENDA

M000042 06 -LISTA

M000042 06+PROGC

M000048 06+LISTA

M00004B 06+ENDA

M00004B 006-LISTA

M00004B 06-ENDB

M00004B 06+LISTB

M00004E 06+LISTB

M00004E 06-LISTA

E

The following figure shows these three programs as they might appear in memory after loading and linking. PROGA has been loaded starting at address 4000, with PROGB and PROGC immediately following.

For example, the value for REF4 in PROGA is located at address 4054 (the beginning address of PROGA plus 0054, the relative address of REF4 within PROGA). The following figure shows the details of how this value is computed.

The initial value from the Text record

T0000540F000014FFFFF600003F000014FFFFC0 is 000014. To this is added the address assigned to LISTC, which is 4112 (the beginning address of PROGC plus 30). The result is 004126.

That is REF4 in PROGA is ENDA-LISTA+LISTC=4054-4040+4112=4126.

Similarly the load address for symbols LISTA: PROGA+0040=4040, LISTB: PROGB+0060=40C3 and LISTC: PROGC+0030=4112

Keeping these details work through the details of other references and values of these references are the same in each of the three programs.

Algorithm and Data structures for a Linking Loader

The algorithm for a linking loader is considerably more complicated than the absolute loader program, which is already given. The concept given in the program linking section is used for developing the algorithm for linking loader. The modification records are used for relocation so that the linking and relocation functions are performed using the same mechanism.

Linking Loader uses two-passes logic. ESTAB (external symbol table) is the main data structure for a linking loader.

Pass 1: Assign addresses to all external symbols

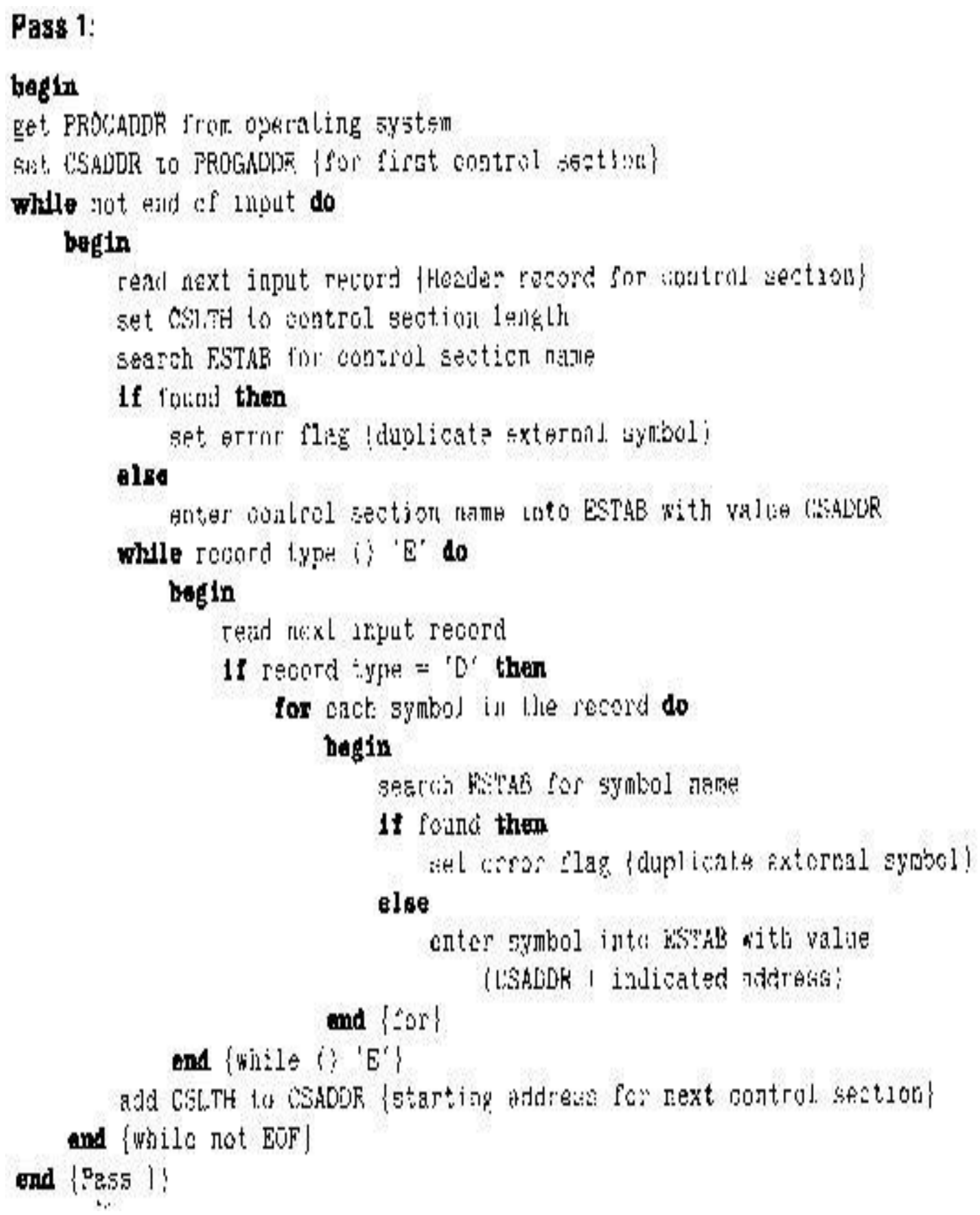
Pass 2: Perform the actual loading, relocation, and linking

ESTAB - ESTAB for the example (refer three programs PROGA PROGB and PROGC) given is as shown below. The ESTAB has four entries in it; they are name of the control section, the symbol appearing in the control section, its address and length of the control section.

Control section	Symbol	Address	Length
PROGA		4000	63
	LISTA	4040	
	ENDA	4054	
PROGB		4063	7F
	LISTB	40C3	
	ENDB	40D3	
PROGC		40E2	51
	LISTC	4112	
	ENDC	4124	

Program Logic for Pass 1

Pass 1 assign addresses to all external symbols. The variables & Data structures used during pass 1 are, PROGADDR (program load address) from OS, CSADDR (control section address), CSLTH (control section length) and ESTAB. The pass 1 processes the Define Record. The algorithm for Pass 1 of Linking Loader is given below.

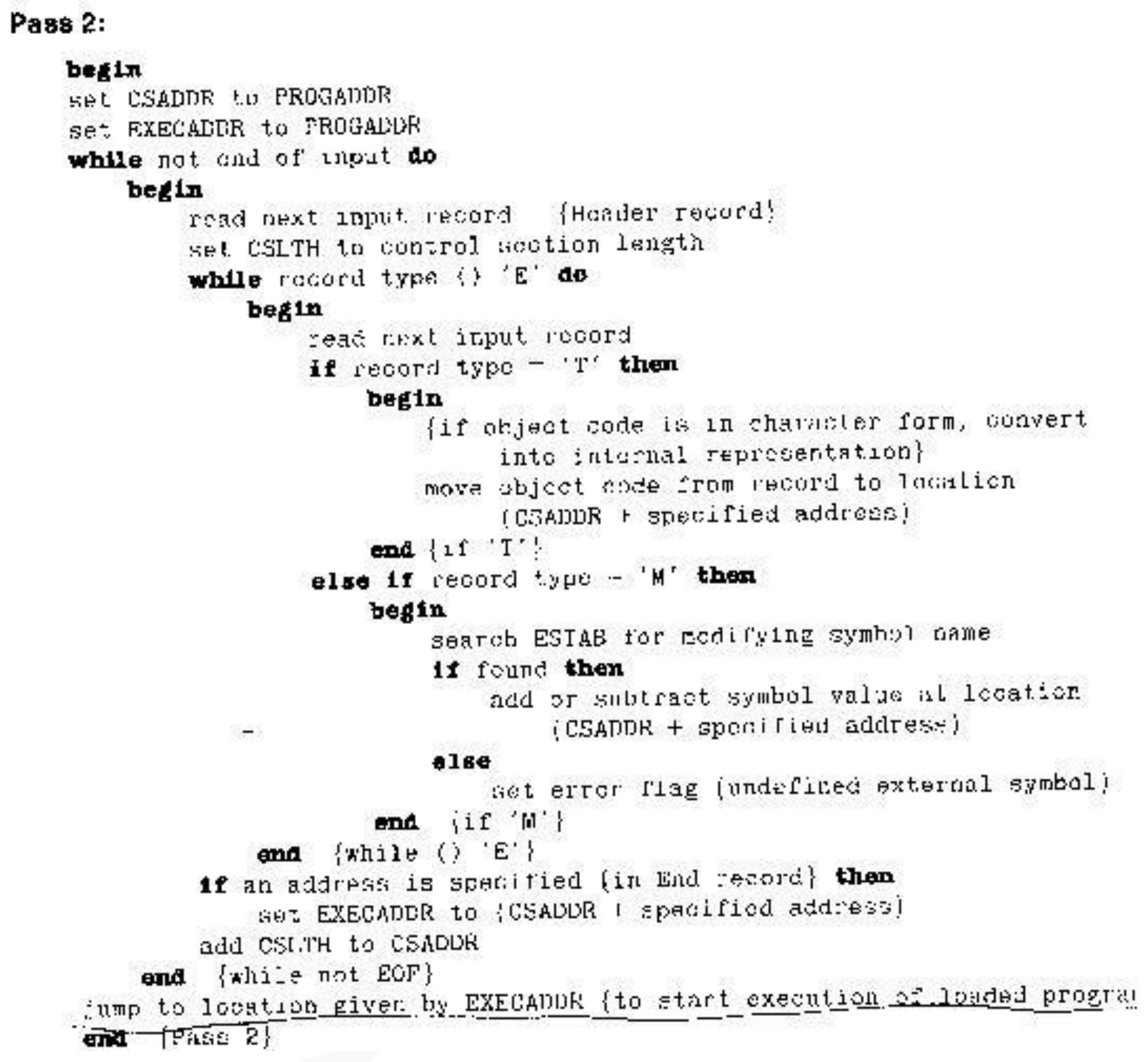


```
Pass 1:

begin
get PROGADDR from operating system
set CSADDR to PROGADDR {for first control section}
while not end of input do
    begin
        read next input record {Header record for control section}
        set CSLTH to control section length
        search ESTAB for control section name
        if found then
            set error flag {duplicate external symbol}
        else
            enter control section name into ESTAB with value CSADDR
        while record type () 'E' do
            begin
                read next input record
                if record type = 'D' then
                    for each symbol in the record do
                        begin
                            search ESTAB for symbol name
                            if found then
                                set error flag {duplicate external symbol}
                            else
                                enter symbol into ESTAB with value
                                    (CSADDR + indicated address)
                        end {for}
            end {while () 'E'}
        add CSLTH to CSADDR {starting address for next control section}
    end {while not EOF}
end {Pass 1}
```

Program Logic for Pass 2

Pass 2 of linking loader perform the actual loading, relocation, and linking. It uses modification record and lookup the symbol in ESTAB to obtain its addres. Finally it uses end record of a main program to obtain transfer address, which is a starting address needed for the execution of the program. The pass 2 process Text record and Modification record of the object programs. The algorithm for Pass 2 of Linking Loader is given below.



Pass 2:

```
begin
set CSADDR to PROGADDR
set EXECADDR to PROGADDR
while not end of input do
    begin
        read next input record    {Header record}
        set CSLTH to control section length
        while record type <> 'E' do
            begin
                read next input record
                if record type = 'T' then
                    begin
                        {if object code is in character form, convert
                             into internal representation}
                        move object code from record to location
                             (CSADDR + specified address)
                    end {if 'T'}
                else if record type = 'M' then
                    begin
                        search ESTAB for modifying symbol name
                        if found then
                            add or subtract symbol value at location
                                 (CSADDR + specified address)
                        else
                            set error flag (undefined external symbol)
                    end  {if 'M'}
            end  {while <> 'E'}
        if an address is specified {in End record} then
            set EXECADDR to (CSADDR + specified address)
        add CSLTH to CSADDR
    end  {while not EOF}
jump to location given by EXECADDR {to start execution of loaded progra
end  {Pass 2}
```

SYSTEM SOFTWARE AND OPERATING SYSTEMS
UNIT-2

- -

UNIT II: Machine dependent compiler features - Intermediate form of the program - Machine dependent code optimization - Machine independent compiler features - Compiler designoptions - Division into passes – Interpreters – p-code compilers - Compiler-compilers.

- -

MACHINE DEPENDANT COMPILER FEATURES:

Synopsis

1. Introduction
2. Definition of compiler
3. Basic compiler function
   - Grammars
   - Lexical analysis
   - Syntactical analysis
   - Code generation
4. Basis step to be followed in the compilation process
5. Machine dependent compiler features
   - Intermediate form of the program
   - Machine dependent code optimization
6. Intermediate form of the program
7. Quartruples
8. Form of Quartruples
9. Machine dependent code optimization
10. Basic block
11. Rearrangement of the Quartruples for code optimizationand figure

Introduction:

➢ In this chapter we discuss the design and operations and compiler for high level programming language.

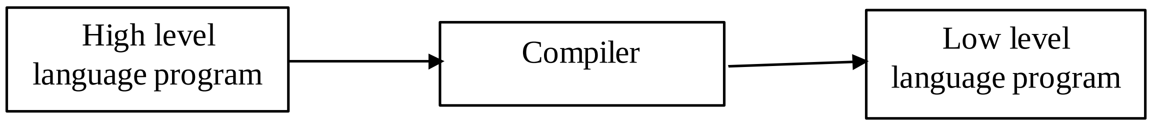
➢ Here also we presents the basic functions of simple one pas compiler which illustrate the operations of the compile.

➢ Here we also discuss above machine dependent extension which is mainly used for,

i. Code generation
ii. Interpreters
iii. P-code
iv. Compiler-compiler

Definition for compiler:

- It is a language program that translates program written in high level language to machine level language.
- For example,***c, c++ compilers, Pascal***



Basic compiler function:

- Here we introduction the fundamental operation that are necessary in compile in the typical high level program.
- The basis compiler function
  - i. Grammar
  - ii. Lexical analysis
  - iii. Syntactical analysis
  - iv. Code generation

Grammar:

- It specify the form or syntax legal statements the language.
- A grammar for a programming language is a format description of a syntax or formof program an integer statement and written in the language.
- It does not describe the symaticsof meaning of various statement.

Lexical analysis:

- It is a process of scanning the source statement, recognizing and classifying the various tokens is called the lexical analysis.
- It is a part of compiler that performs analytic function is known as scanner.

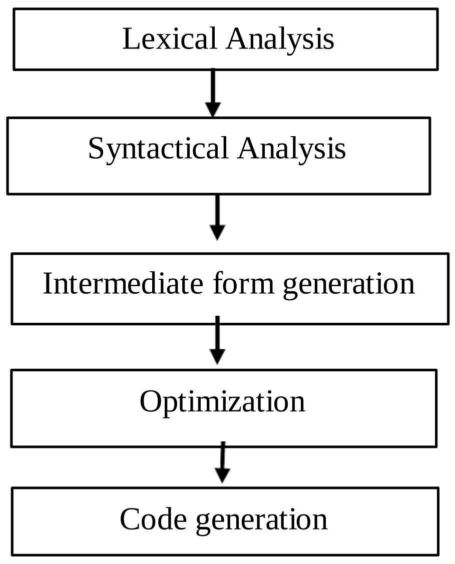
Syntactical analysis (or) parsing:

- After the tokens scan process each statement in the process must be recognized some language construct some as,
  - ❖ Declaration
  - ❖ Assignment statement
  - ❖ Which is described by grammar
- This process is called syntactic analysis (or) parsing
- It is performed by a part of the compiler which is known as parser.

Code generation:

- It is the process of generating the object code.
- It is the code generation technique that creates the object code per each part of the program once syntax has to be recognized.

Basis steps to be followed in the compilation process:



Machine dependent compiler features:

- The purpose of compiler is to translate the programs written in high level program into machine level language.
- Most of the high levels programming languages are designed to the relatively independentof the machine.
- It means the process of analyzing the syntax of program written in this language should be relatively machine independent.
- It is mainly consists of 2 steps,
    - Intermediate form of the program
    - Machine dependent code optimization

Intermediate form of the program:

- In the process of analyzing the syntax and semantics of source program or statement
- Here the translation process (into machine codes) is not yet been performed.
- There are many possible ways of representing in an intermediate form of,
    - Code analysis
    - Optimization

Quadruples:

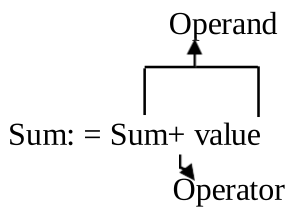
- It is the process of rearrange to eliminate redundant load and store or operation.
- Intermediate form of the program represents the executable instruction of the programwith sequence of Quartruples.

Form of Quadruples:

$\boxed{\text{Operation, op1, op2, result}}$

- In the above form operation: Is some function to be performed by object code.
- Op1and op2: are the operand for this operation
- Result: where the result value to be placed.

For Example: 1



Example: 2

Variance: = sum SQ div 100 – mean * mean

Machine dependent code optimization:

- Here we describe several different possibilities performing the machine dependent code optimization.
- Code optimization is a process of optimizing the code which is used for translating the high level language.
- On many computers there are many numbers of general purpose registers.
- It may be used,
    i. Hold constants
    ii. Values of variables
    iii. Intermediate result etc….
- Some registers can be often used for addressing
- Machine instruction used registers as operands are usually faster than the corresponding instruction that refer to locations in memory.
- Each time a value is fetch from the memory or calculated as intermediate result and it can be assign to some registers
- The value will be available for later use without requiring a memory reference.
- This approach also avoids unnecessary movements of value between memory and registers which takes time but doesn't advance in computation.
- Here we using the divider concepts or basic block concepts to deal the problem.

Basic blocks:

- It is a sequence Quartruples with one entry point, which is at the beginning of the block. One exist point, which is at the ending of the block and no jumps within the blocks.

Rearrangement of Quadruples for code optimization:

- It is the possibility for code optimization before machine code is generated.
- It takes the advantage of specific characteristics of and instructions of the target machines.

- For example: that may be special loop control instruction or addressing modes that can be used to creak more efficient object mode.
- On some computer there are high level machine instructions that can perform completed function such as calling procedure and manipulated operating data structure in a single operation.

Figure: rearrangement for Quadruples for code optimization

a)

```
DIV        SUMSQ        #100        i1
*          MEAN         MEAN        i2
-          i1           i2          i3
=          i3                       VARIANCE
                   ↓
           LDA          SUMSQ
           DIV          #100
           STA          T1
           LDA          MEAN
           MUL          MEAN
           STA          T2
           LDA          T1
           SUB          T2
           STA          VARIANCE
```

b)

```
*          MEAN        MEAN        i2
DIV        SUMSQ       #100        i1
-          i1          i2          i3
: =        i3                      VARIENCE
                 ↓
        LDA       MEAN
        MUL       MEAN
        STA       T1
        LDA       SUMSQ
        DIV       # 100
        SUB       T1
        STA       VARIANCE
```

MACHINE INDEPENDENT FEATURES

Synopsis:

1. Introduction
2. Definition for compiler
3. Basis compiler function
    - Grammar
    - Lexical analysis
    - Syntactical analysis
    - Code generation
4. Basic step for followed in the compilation process
5. Machine dependent compiler features

Machine independent compiler features:

- It describes some common compiler features that are largely independent of the particular machine be use.
- The basic step which is available in the machine independent compiler features.
    - Structure variable
    - Machine independent code optimization
    - Storage allocation
    - Block structured languages

Structure variable:

- During the compilation of program, we use structure variables such as arrays,records, strings, and sets.
- We are primary concerned with the allocation on storage for such variables with the generation of code to referred them.
- Consider first Pascal array declaration.

Example: A: ARRAY [1…...........10] of integer

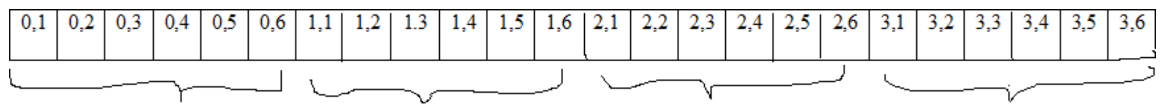
- If each integer variable occupies one words of memory, then we must clearly allocate 10 words to store this array.
- Allocation for a multidimensional array is not much more difficult.
- Consider the following example is a 2 dimensional array.

    B: ARRAY [0…….3, 1……6] of integer

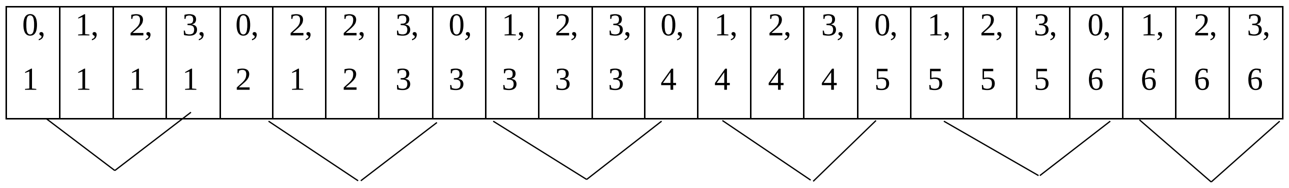
- When we consider the generation of code for array reference, it becomes important to known which array element corresponds to each word for allocated storage.
- The following figure shows 2 possible of storing the previously define array B:
- In figure (A); all array elements that have the same value of subscribe at stored in contiguous location this is called **row major order**.
- If figure (B), all elements that have the same value second subscribe are storedto gather this is called **column major order.**

Storage of B: ARRAY [0…….3, 1…..6]

(A) in row major order



B) In column major order:



The order in which the values 10 are stored in a table (row order, column order)

	1	2	3	4	5	6
0	0,1	0,2	0,3	0,4	0,5	0,6
1	1,1	1,2	1,3	1,4	1,5	1,6
2	2,1	2,2	2,3	2,4	2,5	2,6
3	3,1	3,2	3,3	3,4	3,5	3,6

- In row major order, the right most subscribe various most rapidly.
- In column major order the left most subscribe various most rapidly.
- This concept can be generalized easily to arrays with more than 2 subscript
- For multidimensional array, the generation of code depends on whether row major or column major order is used to store the array.

i. Machine independent code optimization:

- Important source of code optimization is the elimination of common sub expression.
- The sub expression that appeared at more than one point in the program and compute the same value.
- Consider the statement which is given below here the term 2*j is a common sub expression.
- An optimizing compiler should generate code so that multiplication is performed only once and the result is used in the both places.

For example: code optimization by elimination of common sub expression and removal of loop in variance.

```
x, y: ARRAY [1….10, 1… .....10] of integer.
.
.
.
.
.
.
For I:= 1 To 10 Do
x [I, 2* J-1]:=y[I, 2 *J]
```

(A)

```
 1) : =       #1      -       I {loop initialization}
 2) JGT       I       #10     (20)
 3) -         I       #1      T1 {subscript calculation for x}
 4) *         i1      #10     i2
 5) *         #2      J       i3
 6) -         i3      #1      i4
 7) -         i4      #1      i5
 8) +         i2      i5      i6
 9) *         i6      #3      i7
10) -         I       #1      i8
11) *         i8      #10     i9
12) *         #2      J       i10
13) -         i10     #       i11
14) +         i9      i11     i12
15) *         i12     #3      i13
16)  :=       y[i13]          [i17]{assignment operation}
17) +         #1      I       i14 {end of loop}
18)  :=       i14             I
19) J                         next statement}
20)
```

(B)

```
 1): =        #1      -       I {loop initialization}
 2) JGT       I       #10     (20) {subscript calculation for x}
 3) -         I       #1      T1
 4) *         i1      #10     i2
 5) *         #2      J       i3
 6) -         i3      #1      i4
 7) -         i4      i5      i5
 8) +         i2      #3      i6
9) *          i6      i4      i7
```

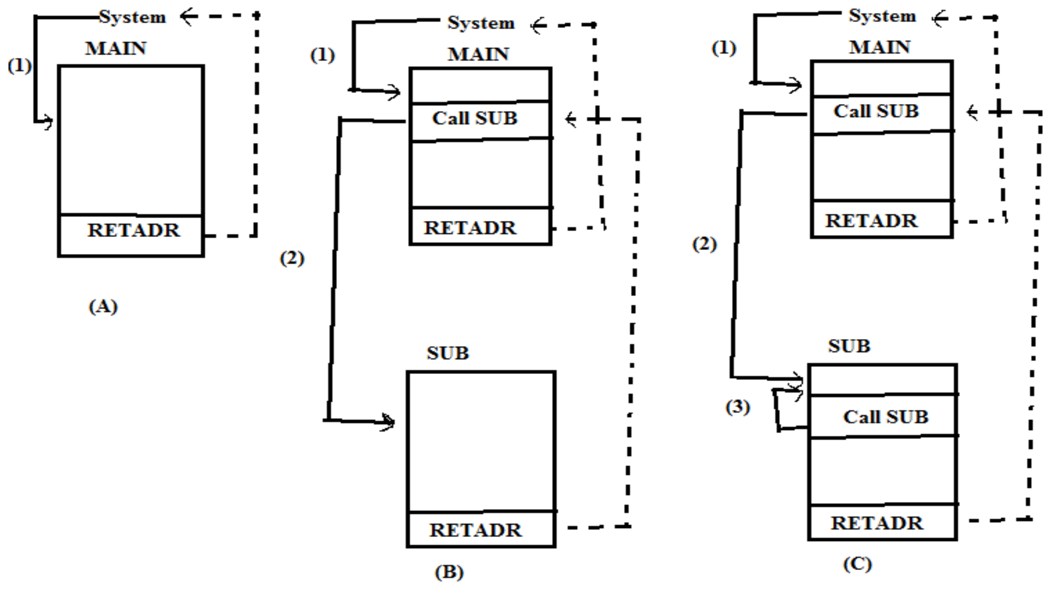
```
10) +        i2        #3        i12 {subscript calculation for V}
11) *        i12       i13
12) :=       y[i13]    I         x[i7] {assignment operation}
13) +        #1                  i14 {end of loop}
14) :=       i14                 I
15) J                            (2)
16)                              {next statement}
```

(C)

- Common sub expression is usually dictated through the analysis of an intermediate form of the program. It is shown in the program (B)
- Another common source of code optimization is the removal of loop invariance.
- The last source of code optimization is the substitution of a more efficient operation forless efficient one.
- The process of transforming the cost or an operation is called reduction is strength of an operation.
- The computation who's operand values are known at compilation time that can beperformed by the compiler this optimization is known as folding.

ii. Storage allocation:

- All programmers define variables were assign storage allocation within the object programas their declarations were processed.
- Temporary variables including the one use to save the written address there also assignfined address within the program.
- This simple type of storage assignment is called static allocation.
- It is often used for languages that do not allow the recursive use of procedure are subroutine and do not provide for the dynamic allocation of storage during execution.
- The following figure illustrates the **recursive invocation of a procedure usingstatic storage allocation.**



Block structured languages:

- In some languages a program can be divided into units called blocks. A block is a portion of a program that has the ability to declare its own identifiers
- This definition of a block is also met by units such as procedures and functions in Pascal
- Each procedure correspond to a block in the following example we shows the outline of a block structured program in a Pascal like languages
- Here we use a terms
    - Procedure
    - block interchangeably
- Note that blocks may be nested with in other block.
- Example: in the following example the procedure(b)&(d) are nested with in procedure (a),(c) is nested with in procedure (b)
- Each block will contain declaration of variables.
- A block may also refer that are defined in any block that contain is, provided the same name. Names are not refined inner block.

Example: nested of blocks in a source program

```
1)  Procedure A;
2)  VAR x, y, z : INTEGER
3)  PROCEDURE B;
4)  VAR w, x, y, z: REAL
5)  PROCEDURE C;
6)  VAR v, w: INTEGER
7)  END {C};
8)  END {B};
9)  PROCEDURE D;
10)  VAR X,Z: CHAR;
11) END {D};
12) END {A};
```

FIGURE (A)

Block Name	Block number	Block level	Surrounding block
A	1	1	-
B	2	2	1
C	3	3	2
D	4	2	1

FIGURE (B)

- In figure (A) as the beginning of each new block is recognized it is assigned the block number in sequence.

- The compiler can construct a table that describe the block structure
- In fig: (B) the block level entry gives the hosting depth for each block.
- The outer most block has 2 level number of 1and each other block has 2 level number that is 1 greater than that surrounding block.
- The main problem which is arriased in block structured language is declaration problems.
- Here most of the variable will be repeatedly arriased in each block
- One common method for providing accepts a variables in surrounding block users called display

COMPILER DESIGN OPTIONS

Synopsis
1. Introduction 2. division into passes 3.interpretors 4.P-code compiler 5.compiler - compilers

Introduction:

- Here we considered some of the possible alternation for the design and construction ofa compiler.
- It was simple one pass design which describes many features that usually required morethan one pass to implement.
- Here we briefly discuss the general questions of dividing the compiler into andthe advantages of 1- pass and multi pass design.
- The compiler consists of following design options.
    - ❖ Division into passes
    - ❖ Interpreters
    - ❖ P- code compiler
    - ❖ Compiler-compilers

Division into passes:

- Here we present a simple 1- pass compilation scheme for a subset of the Pascal language.
- In this design the compiler was driven by parsing process.
- The lexical scanner called when the Pascal needed another input token code generation routine was involve as each language construct was recognized by parser
- The compilation process itself, which required only one pass over the program and no in term ediate code generation steps was quit efficient.
- Not all the languages can be translated by sub a one pass compiler.
- Here the speed of compilation process is important for that one pass design might be referred
- If programs are executed many times for each compilation or if the process large amount of data, speed of execution becomes more important than speed of compilation

- Multi pass compilers are also used when the amount of memory or other system resources is severely limited
- The requirement of each pass can be kept smaller if the work of compilation is divided into several passes
- Other factors may include the design of the compilers
- If a compiler is divided into several passes, each pass become
    - Simpler
    - Easy understand
    - Easy to write
    - Easy to test
- Different passes can be assign to different programmers and can be writtened and tested in parallel which shortest the overall time require for compiler construction.

Interpreters:

- An interpreter process a source program written in a high level language compiler disk
- The main difference in an interpreter a source program directly instead of translating to machine code
- It usually perform lexical and syntactical analysis functions and then translates thesource program into an internal form
- After translating the source program into an internal form the interpreter executethe operation specified by the program
- During this phase the interpreter can be viewed as set of subroutines
- The execution of sub routine is driven by the internal form of the program
- The process of translating a source program into some internal form is simpler and faster the compiling it into machine code
- The execution of translated program by an interpreter is much slower than the execution of the machine code produced by the compiler
- If speed of translation is of primary concerned and execution of translated program will be short, than interpreter may be good choice

Advantages interpreters

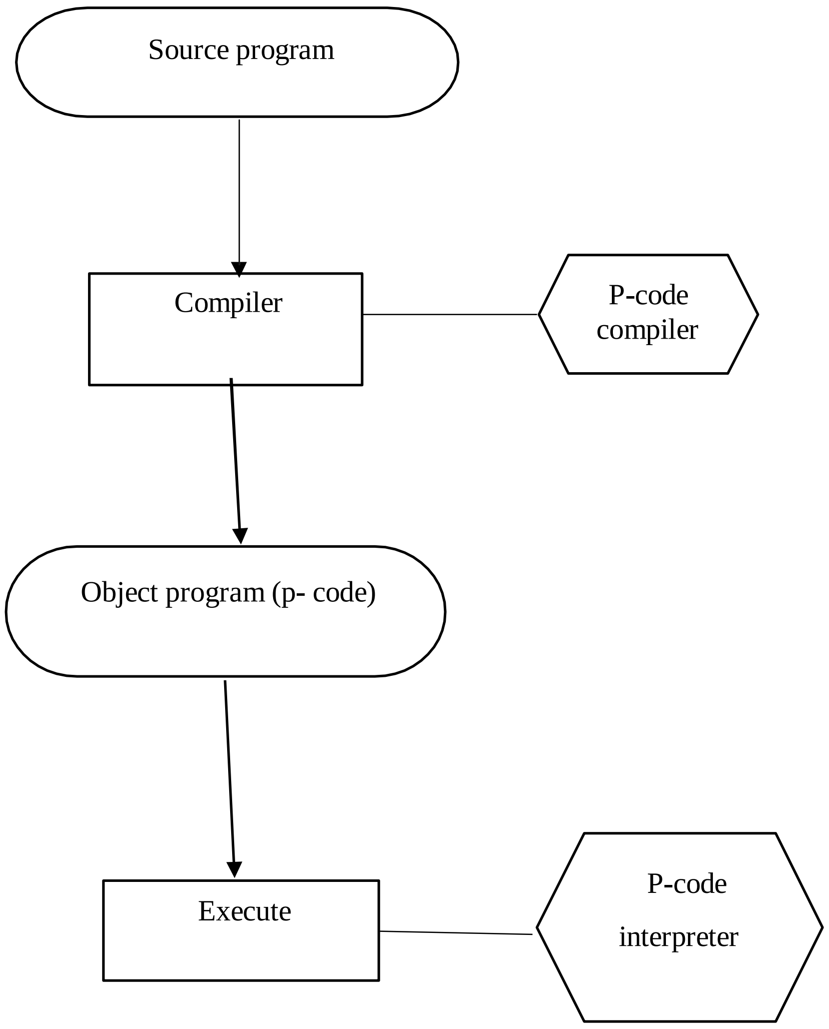
- Debugging facilities can be easy provided
- Symbol table, source line numbers and other information from the source program are usually written by the interpreters
- The interpreters is attractive in educational environment for learning and program testing
- It have a high speed of transaction
- Execution time is less
- It have more additional features

P-code compilers:[byte code compiler]:

- It is very similar in concept to interpreters
- Here the source program is analysis and converted into intermediate form which is then execution interpretively

- With the p- code compiler this intermediated form is the machine language for a hypothetical computer of an code pseudo-machine (or) p- machine

Translation and execution using a p- code compiler:



- The above figure: the source program is compiled with help of p-code compiledand procedure the object program as a result
- Then the p-code program is read and executed and control of p-code interpreter

Advantages of p-code compiler:

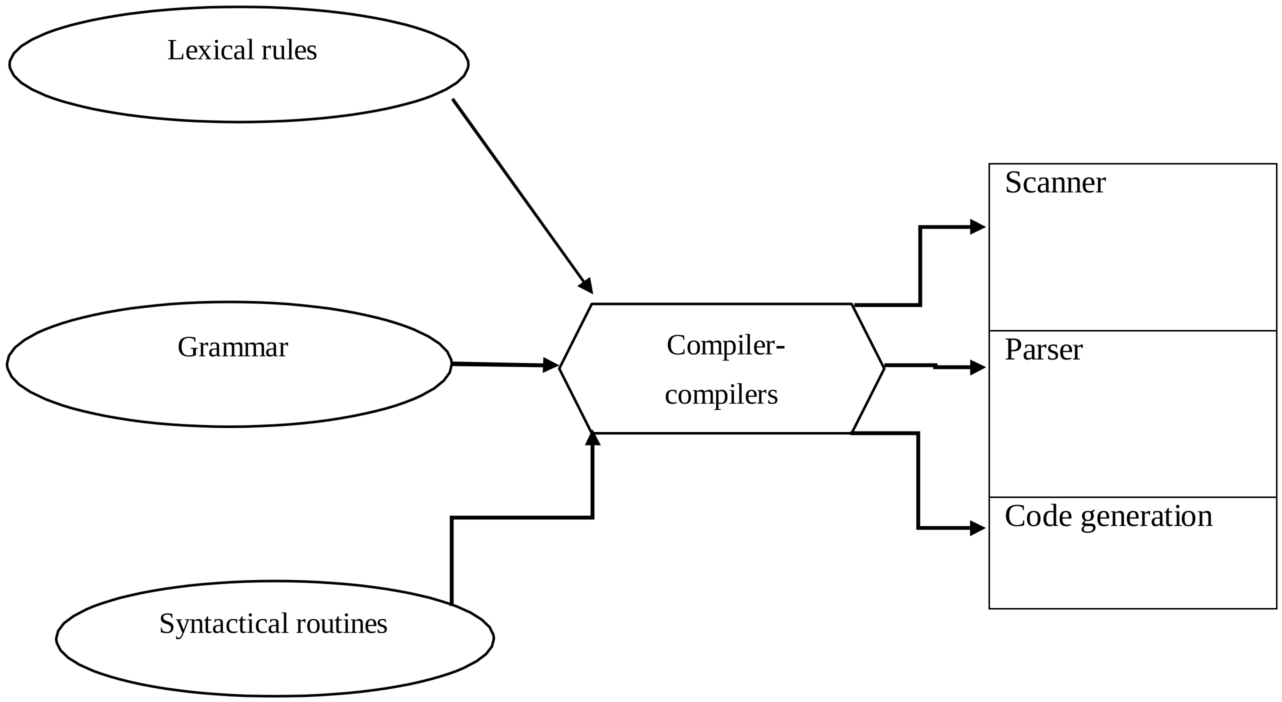
- Portability of software
- P-code object program can be executed on any machine that has p-code interpreter

Design of pseudo-machine (or) p-code machine

- It is related to the requirement of language being compile. For example: P-code for a Pascal compiler might include single P-instruction that performing ,
  - Array calculation
  - Handle of procedure entry and exit
  - Perform elementary operations on set
- P- code compiler are designed for a single user running on a dedicated microcomputer system
- Single user running on a dedicated micro computer system
- Here the execution p is relatively in significant because the system performance may differs and the responds time for the requirest will be verify.
- For the execution speed, the p-code compiler support the use of machine languagesub routine

Compiler-compilers

- It is a software tool that can be used to help in the task of a compiler construction
- Such tools are known as compiler generators (or) translator writing system
- For example: automated compiler construction using a compiler – compilers



- In the above figure we have illustrated the process of using the compiler- compilers
- The user (compiler writer) provides a description of a language to be translated
- The description may construct set of lexical rule for defining tokens and the grammar for source language
- Compiler-compilers use this information to generate a scanner and a parser directly

- Others create tables for use of standard table driven scanning and passing routines that are supplied by the compiler-compilers

Advantages of compiler – compilers

- Easy of compiler construction and testing
- The amount of work required from the user various from one compiler –compiler to another
- It provides special languages
- Provide notations, data structure and other facilities that can be used in writing of symatics routines.

UNIT-III

<u>SYLLABUS</u>

What is an Operating System? - Process Concepts: Definition of Process - Process States - Process States Transition - Interrupt Processing - Interrupt Classes - Storage Management: Real Storage: Real Storage Management Strategies – Contiguous versus Non-contiguous storage allocation – Single User Contiguous Storage allocation- Fixed partition multiprogramming – Variable partition multiprogramming.

OPERATING SYSTEM

An operating system (OS) is system software. It manages computer hardware and software resources. It provides common services for computer programs. Time-sharing operating systems schedule tasks for efficient use of the system. It may also include accounting software for cost allocation of processor time, mass storage, printing, and other resources.

The operating system acts as an intermediary between programs and the computer hardware. The application code is usually executed directly by the hardware. It is frequently making system calls to an OS function or is interrupted by it. Operating systems are found on many devices that contain a computer – from cellular phones and video game consoles to web servers and supercomputers. For hardware functions such as input and output and memory allocation.

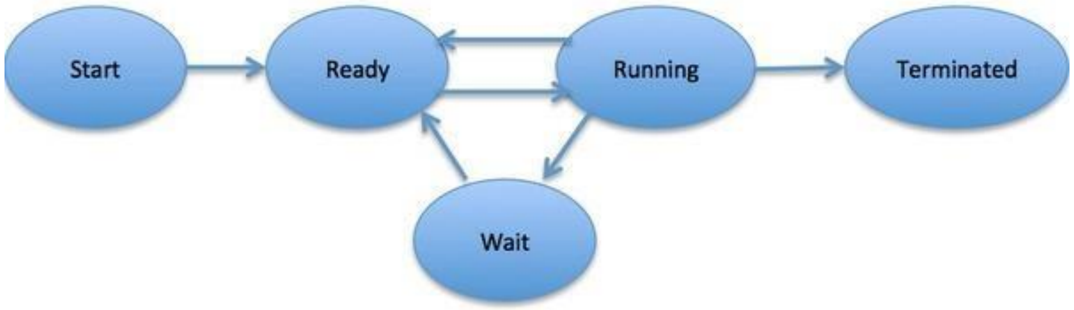
PROCESS- DEFINITIONS OF PROCESS

1. A program in execution
2. An asynchronous activity
3. The “animated sprit” of a procedure
4. The “locus of control” of a procedure in execution

5. That entity to which processors are assigned
6. The “dispatchable” unit

A process is a program at the time of execution. The process is more than the program code. It includes the program counter, the process stack, and the content of the process register, etc. The purpose of the process stack is to store temporary data, such as subroutine parameters, return address and temporary variables.

An instance of a program running on a computer. The entity that can be assigned to and executed on a processor. A unit of activity characterized by the execution of a sequence of instructions, a current state, and an associated set of system resources. An instance of a program running on a computer. The entity that can be assigned to and executed on a processor. A unit of activity characterized by the execution of a sequence of instructions, a current state, and an associated set of system resources.



PROCESS STATES

Start: The process is being created.

Running: The process is being executed.

Waiting: The process is waiting for some event to occur.

Ready: The process is waiting to be assigned to a processor.

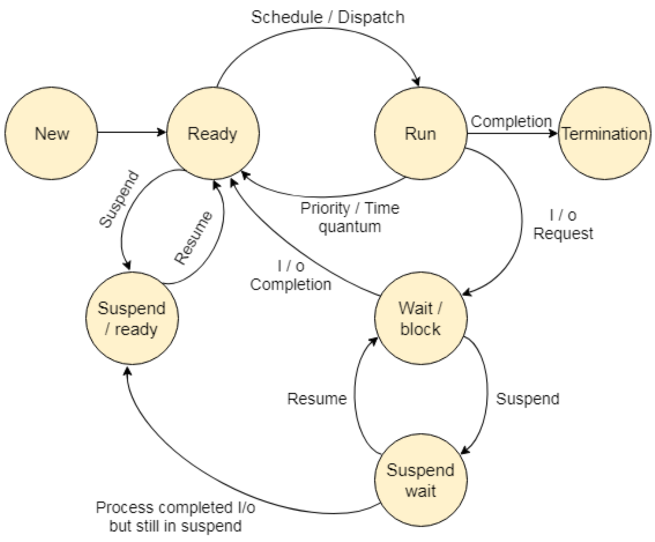
Terminate: The process has finished execution.

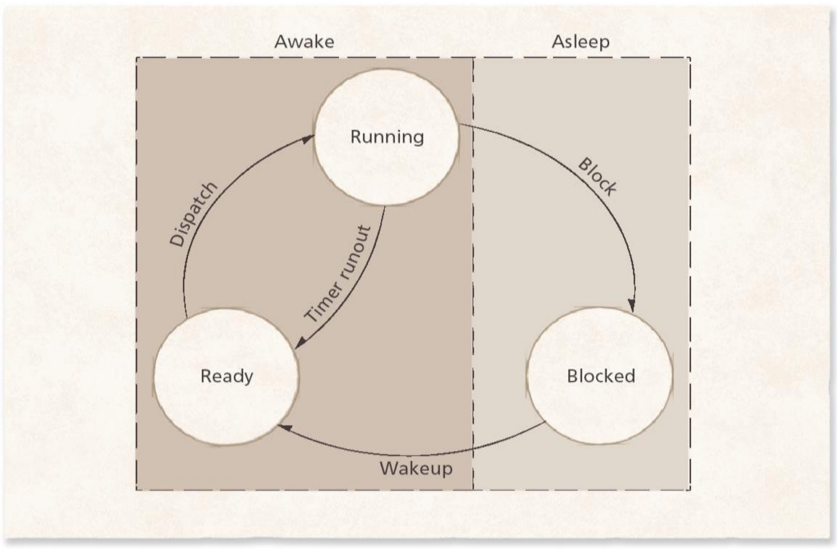
Many processes can be running in any processor at any time. But many processes may be in ready queue waiting for states. Consider the figure below depicts the state diagram of the process states.

In a uniprocessor system only one process may be running at a time. It several may be ready and several blocked. The operating system maintains a ready list of ready processes and a blocked list of blocked processes. The ready list is maintained in priority order. The next process to receive a processor is the first one in the list (i.e., the process with the highest priority). The blocked list is typically unordered - processes do not become unblocked (i.e., ready) in priority order. Unblock in the order in which the events they are waiting for occur.

PROCESS STATE TRANSITIONS

When a user runs a program, processes are created and inserted into the ready list. A process moves toward the head of the list as other processes complete their turns using a processor. When a process reaches the head of the list, a processor becomes available, that process is given a processor.





New to Ready

The operating system creates a process. And prepares the process to be executed by, then the operating system moves the process into the ready queue.

Ready to Running

When it is time to select a process to run, the operating system selects one of the jobs for the ready queue and move the processes from the ready state to the running state. It is said to make a state transition from the ready state to the running state. The act of assigning a processor to the first process on the ready list is called dispatching. It is performed by a system entity called the dispatcher. The state transition is,

dispatch (process name): ready->running

Running to Terminated

When the execution of a process has completed then the operating system terminates that process from running state. Sometimes the operating system terminates the processes for some other reasons also include time limit exceeded, memory unavailable access violation, protection error, I/O failure, data misuse and so on.

Running to Ready

When the time slot for the processor expires or if the processor receives an interrupt signal, then the operating system shifts the running process to the ready state. Processes that are in the ready or running states are said to be awake. To prevent any one process from monopolizing (controlling) the system, either accidentally or maliciously the operating system sets a hardware interrupting clock (also called an interval timer) to allow a process to run for a specific time interval or quantum. The state transition is,

timerrunout (process name): running->ready

For example, process P1 is being executed by the processor, at that time processor, P2 generates an interrupt signal to the processor. Then the processor compares the priorities of process P1 and P2. If P1>P2 then the processor continues executing P1. Otherwise, the processor switches to process P2, and process P1 is moved to the ready state.

Running to Waiting

A process is put into the waiting state if the process needs an event to occur, or an I/O device is to read. The operating system does not provide the I/O or event immediately then the process is moved to the waiting state by the operating system.

Waiting to Ready

A process in the blocked state is moved to the ready state when the event for which it has been waiting occurs.

For example, a process is in running state needs an I/O device, then the process moved to wait or blocked state. When the I/O device is provided by the operating system, the process moved to the ready state from waiting or blocked state.

Running to Block

If a running process initiates an input/output operation before its quantum expires. The running process voluntarily relinquishes the CPU. (i.e the process blocks itself pending the completion of the input/output operation). The state transition is,

block (process name): running->blocked

Block to Ready

The only other allowable state transition in three-state model occurs when an I/O operation (or some other event the process is waiting for) completes. In this case, the operating system transitions the process from the blocked to the ready state. The state transition is,

wakeup (process name): blocked->ready

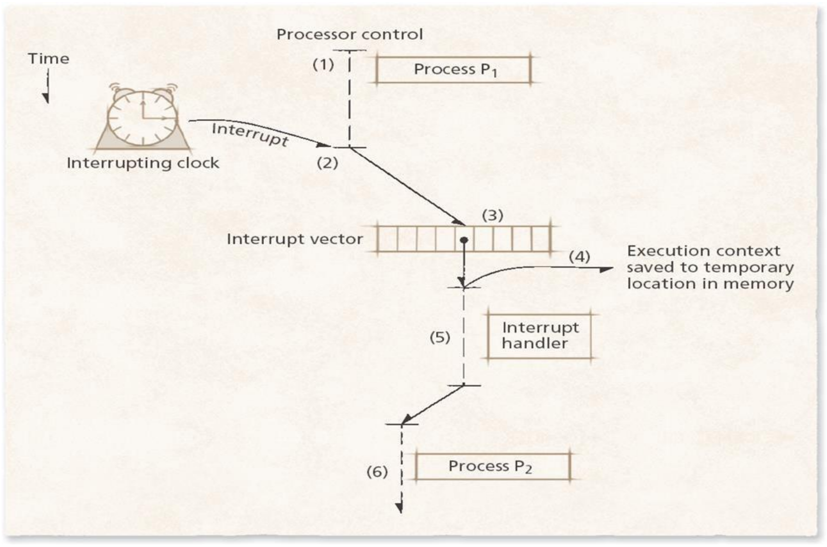
INTERRUPT

Interrupts enable software to respond to signals from hardware. The operating system may specify a set of instructions, called an interrupt handler to be executed in response to each type of interrupt. This allows the operating system to gain control of the processor to manage system resources. Interrupt is called a trap. Synchronous with the operation of the process. For example, dividing by zero or referencing protected memory. Interrupts may also be caused by some event that is unrelated to a process's current instruction. Asynchronous with the operation of the process. For example, the keyboard generates an interrupt when a user presses a key. The mouse generates an interrupt when it moves or when one of its buttons is pressed. Interrupts provide a low-overhead means of gaining the attention of a processor. Polling is an alternative approach for interrupts. Processor repeatedly requests the status of each device. Increases in overhead as the complexity of the system increases.

Difference between polling and interrupts

A simple example microwave oven. A chef may either set a timer to expire after an appropriate number of minutes (the timer sounding after this interval interrupts the chef) The chef may regularly peek through the oven's glass door and watch as the roast cooks (this kind of regular monitoring is an example of polling).

Interrupt processing



Handling Interrupts

1. The interrupt line, an electrical connection between the mainboard and a processor. It becomes active—devices such as timers, peripheral cards and controllers send signals that activate. The interrupt line to inform a processor that an event has occurred (e.g., a period of time has passed or an I/O request has completed). Most processors contain an interrupt controller that orders interrupts according to their priority so that important interrupts are serviced first. Other interrupts are queued until all higher-priority interrupts have been serviced.

2. After the interrupt line becomes active, the processor completes execution of the current instruction, then pauses the execution of the current process. To pause process execution, the processor must save enough information. The process can be resumed at the correct place and with the correct register information.

3. The processor then passes control to the appropriate interrupt handler. Each type of interrupt is assigned a unique value that the processor uses as an index into the interrupt vector, which is an array of pointers to interrupt handlers. The interrupt vector is located in memory that processes cannot access, so that processes cannot modify its contents.

4. The interrupt handler performs appropriate actions based on the type of interrupt.

5. After the interrupt handler completes, the state of the interrupted process is restored.

6. The interrupted process (or some other "next process") executes. It is the responsibility of the operating system to determine whether the interrupted process or some other "next process" executes.

Interrupt classes

There are six interrupt classes. These are,

1. SVC (Supervisor call) interrupts
2. I/O interrupts
3. External interrupts
4. Restart interrupts
5. Program check interrupts
6. Machine check interrupts

SVC (Supervisor call) interrupts

These are initiated by a running process that executes the SVC instruction. It is a user-generated request for a particular system service such as performing input/output. It helps keep the operating system secure from the users. A user may not arbitrarily enter the operating system. The user must request a service through as SVC.

I/O interrupts

These are initiated by the input/output hardware. They signal to the CPU that the status of a device has changed. I/O interrupts are caused when an I/O operation completes, when an I/O error occurs.

External interrupts

These are caused by various events including the expiration of a quantum on an interrupting clock. The pressing of the console's interrupt key by the operator or the receipt of a signal from another processor on a multiprocessor system.

Restart interrupts

These occur when the operator presses the console's restart button. When a restart SIGP (Signal Processor) instruction arrives from another processor on a multiprocessor system.

Program check interrupts

These are caused by a wide range of problems. It may occur as a program's machine language instructions are executed. Example, division by zero, arithmetic overflow or underflow. Data in wrong format attempt to reference a memory location beyond the limits of real storage memory.

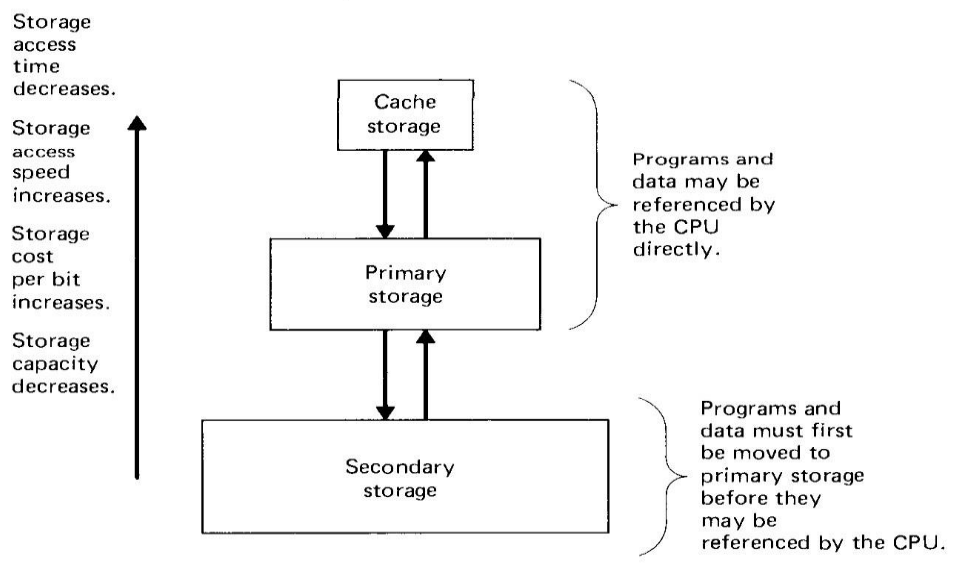
Machine check interrupts

These are caused by malfunctioning (not working) hardware.

STORAGE MANAGEMENT

The term storage management encompasses the technologies and processes organizations use to maximize or improve the performance of their data storage resources. It is a broad category that includes virtualization, replication, mirroring, security, compression, traffic analysis, process automation, storage provisioning and related techniques. The memory management function keeps track of the status of each memory location, either allocated or free. It determines how memory is allocated among competing processes, deciding which gets memory, when they receive it, and how much they are allowed. When memory is allocated, it determines which memory locations will be assigned. It tracks when memory is freed or unallocated and updates the status.

Storage Hierarchy


Storage access time decreases.
Storage access speed increases.
Storage cost per bit increases.
Storage capacity decreases.
Cache storage
Primary storage
Secondary storage
Programs and data may be referenced by the CPU directly.
Programs and data must first be moved to primary storage before they may be referenced by the CPU.

Hierarchical memory organization

Programs and data must be in main memory before the system can execute or reference them. Those that the system does not need immediately may be kept in secondary storage until needed, then brought into main memory for execution or reference.

Secondary storage media, such as tape or disk, are generally far less costly per bit than main memory and have much greater capacity. Main storage may generally be accessed much faster than secondary storage. The memory hierarchy contains levels characterized by the speed and cost of memory in each level. Systems move programs and data back and forth between the various levels. The cache is a high-speed storage that is much faster than main storage.

Cache memory imposes one more level of data transfer on the system. Cache storage is extremely expensive compared with main storage. Programs in main memory are transferred to the cache before being executed—executing programs from cache is much faster than from main memory.

Storage management strategies

Memory management strategies are designed to obtain the best possible use of main memory.

They are divided into:

1. Fetch strategies
2. Placement strategies
3. Replacement strategies

Fetch strategies

It determines when to move the next piece of a program or data to main memory from secondary storage.

It has divided them into two types,

1. Demand fetch strategies
2. Anticipatory fetch strategies

Demand fetch strategy: The system places the next piece of program or data in main memory when a running program references it. Designers believed that because cannot in general predict the paths of execution that programs will take, the overhead involved in making guesses would far exceed expected benefits.

Anticipatory fetch strategies: Today, however, many systems have increased performance by employing anticipatory fetch strategies, which attempt to load a piece of program or data into memory before it is referenced.

Placement strategies

It determines where in main memory the system should place incoming program or data pieces. Consider first-fit, best-fit, and worst-fit memory placement strategies. program and data can be divided into fixed-size pieces called pages. It can be placed in any available page frame.

Replacement strategies

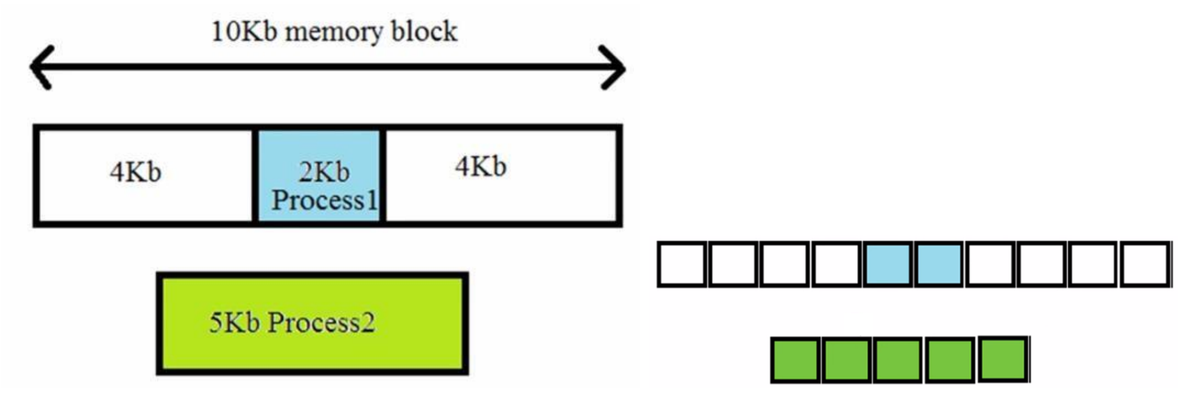
When memory is too full to accommodate a new program, the system must remove some (or all) of a program or data that currently resides in memory. The system's replacement strategy determines which piece to remove.

DEFINITION OF CONTIGUOUS MEMORY ALLOCATION

The operating system and the user's processes both must be accommodated in the main memory. The main memory is divided into two partitions.

1. at one partition the operating system resides
2. at other the user processes reside

In usual conditions, the several user processes must reside in the memory at the same time. It is important to consider the allocation of memory to the processes. The Contiguous memory allocation is one of the methods of memory allocation. In contiguous memory allocation, when a process requests for the memory. A single contiguous section of memory blocks is assigned to the process according to its requirement.



DEFINITION NON-CONTIGUOUS MEMORY ALLOCATION

The Non-contiguous memory allocation allows a process to acquire the several memory blocks at the different location in the memory according to its requirement. The non-contiguous memory allocation also reduces the memory wastage caused due to internal and external fragmentation. As it utilizes the memory holes, created during internal and external fragmentation.

Paging and segmentation are the two ways which allow a process physical address space to be non-contiguous. In non-contiguous memory allocation, the process is divided into blocks (pages or segments) which are placed into the different area of memory space according to the availability of the memory. The non-contiguous memory allocation has an advantage of reducing memory wastage but, it increases the overheads of address translation. The process is placed in a different location in memory, it slows the execution of the memory because time is consumed in address translation.

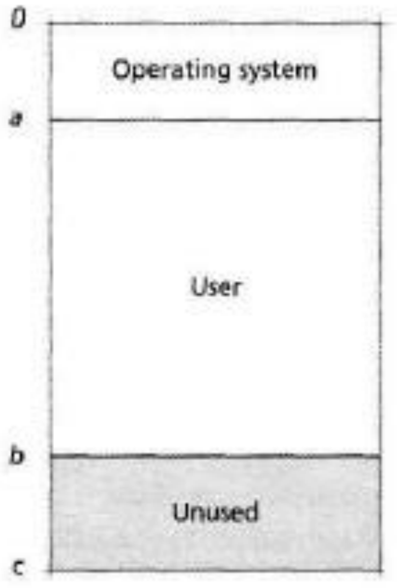
Contiguous versus Non-contiguous Storage allocation

Contiguous Memory Allocation	**Non-Contiguous Memory Allocation**
The contiguous Memory Allocation technique allocates **one single contiguous block of memory** to the process and memory is allocated to the process in a continuous fashion.	The non-Contiguous Memory allocation technique divides the **process into several blocks** and then places them in the **different address space of the memory** that is memory is allocated to the process in a non-contiguous fashion.
In this Allocation scheme, there is **no overhead** in the address translation while the execution of the process.	While in this scheme, there is **overhead** in the address translation while the execution of the process.
In Contiguous Memory Allocation, the process **executes faster** because the whole process is in a sequential block.	In Non-contiguous Memory allocation **execution of the process is slow** as the process is in different locations of the memory.
Contiguous Memory Allocation **is easier for the Operating System** to control.	The non-Contiguous Memory Allocation scheme **is difficult for the Operating System** to control.

Contiguous Memory Allocation	**Non-Contiguous Memory Allocation**
In this, the memory space is divided into **fixed-sized partitions** and each partition is allocated only to a single process.	In this scheme, the process is divided into several blocks and then these blocks are placed in different parts of the memory according to the availability of memory space.
Contiguous memory allocation includes **single partition allocation** and **multi-partition allocation.**	Non-Contiguous memory allocation includes **Paging and Segmentation**.
In this type of **memory allocation**, generally, a table is maintained by the operating system that maintains the list of all **available and occupied partitions** in the memory space.	In this type of memory allocation **generally, a table has to be maintained for each process** that mainly carries the **base addresses of each block** that has been acquired by a process in the memory.
There is wastage of memory in Contiguous Memory allocation.	There is no wastage of memory in Non-Contiguous Memory allocation.
In this type of allocation, swapped-in processes are arranged in the originally allocated space.	In this type of allocation, swapped-in processes can be arranged in any place in the memory.

SINGLE USER CONTIGUOUS STORAGE ALLOCATION

Early computer systems allowed only one person at a time to use a machine. All the machine's resources were dedicated to that user. Billing was straightforward—the user was charged for all the resources whether or not the user's job required them. In fact, the normal billing mechanisms were based on wall clock time. The system operator gave the user machine for some time interval and charged a flat hourly rate. The programmer wrote all the code necessary to implement a particular application, including the highly detailed machine-level input/output instructions.



The memory organization for a typical single-user contiguous memory allocation system

System designers consolidated input/output coding that implemented basic functions into an input/output control system (IOCS). The programmer called IOCS routines (procedures) to do the work instead of having to "reinvent the wheel" for each program. The IOCS greatly simplified and expedited (advanced) the coding process. The implementation of input/output control systems may have been the beginning of today's concept of operating systems.

Advantages and Disadvantages of Single Contiguous Allocation

Advantages

1. Simple Allocation
2. Entire Scheme requires less memory
3. Easy to implement and use

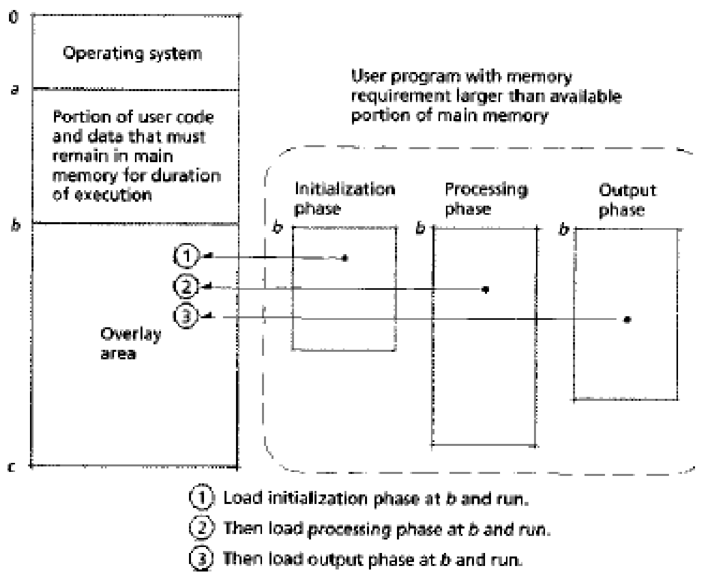
Disadvantages

1. Memory is not fully utilized
2. Processor (CPU) is also not fully utilized
3. User program is being limited to the size available in the main memory

OVERLAYS

How contiguous memory allocation limited the size of programs that could execute on a system? One way in which a software designer could overcome the memory limitation was to create overlays, which allowed the system to execute programs larger than main memory.

Overlay Structure


0
Operating system
a
Portion of user code and data that must remain in main memory for duration of execution
b
Overlay area
c
User program with memory requirement larger than available portion of main memory
Initialization phase
Processing phase
Output phase
1 Load initialization phase at b and run.
2 Then load processing phase at b and run.
3 Then load output phase at b and run.

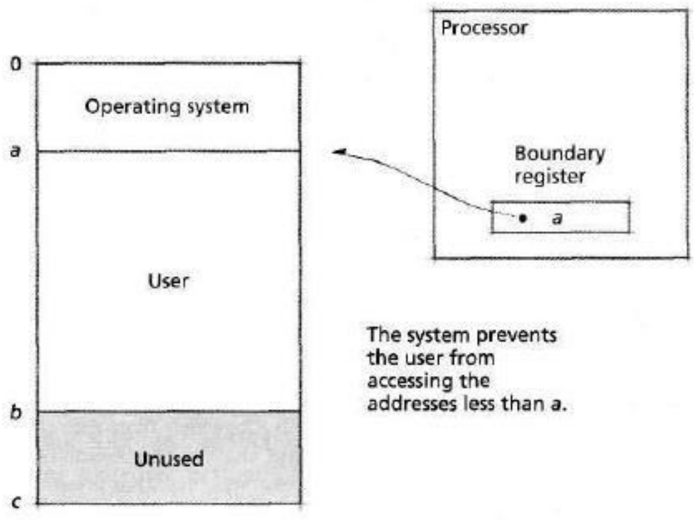
The programmer divides the program into logical sections. When the program does not need the memory for one section. The system can replace some or all of it with the memory for a needed section. Overlays enable programmers to "extend" main memory. However, manual overlay requires careful and time-consuming planning. The programmer often must have detailed knowledge of the system's memory organization. A program with a sophisticated overlay structure can be difficult to modify. Indeed, as programs grew in complexity, by some estimates as much as 40 percent of programming expense were for organizing overlays. It became clear that the operating system needed to insulate the programmer from complex memory management tasks such as overlays.

Protection in a Single-User System

A process can interfere with the operating system's memory - either intentionally or inadvertently (mistake) -by replacing some or all of its memory contents with other data. If it destroys the operating system, then the process cannot proceed. If the process attempts to access memory occupied by the operating system.

Boundary register

The user can detect the problem, terminate execution, possibly fix the problem and re-launch the program. Protection in single-user contiguous memory allocation systems can be implemented with a single boundary register built into the processor.


0
Operating system
a
User
b
Unused
c
Processor
Boundary register
a
The system prevents the user from accessing the addresses less than a.

Memory protection with single-user contiguous memory allocation

The boundary register contains the memory address at which the user's program begins. Each time a process references a memory address, the system determines if the request is for an address greater than or equal to that stored in the boundary register. The hardware that checks boundary addresses operates quickly to avoid slowing instruction execution. The single boundary register represents a simple protection mechanism.

Single-Stream Batch Processing

Early single-user real memory systems were dedicated to one job for more than the job's execution time. Jobs generally required considerable setup time during which the operating system was loaded tapes and disk packs were mounted. When jobs completed, they required considerable teardown time as tapes and disk packs were removed. Designers realized that if they could automate various aspects of job-to-job transition. It could reduce considerably the amount of time wasted between jobs. This led to the development of batch-processing systems.

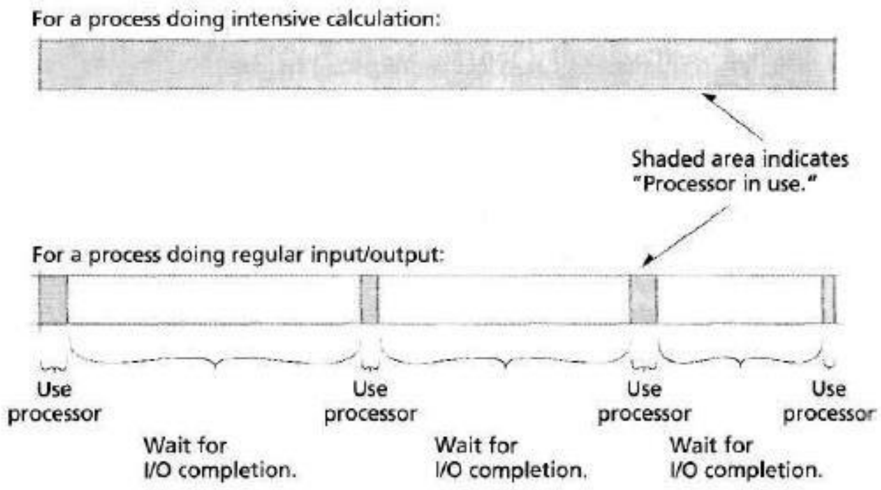
In single stream batch processing, jobs are grouped in batches by loading them consecutively onto tape or disk. A job stream processor reads the job control language statements and facilitates the setup of the next job. Batch-processing systems greatly improved resource utilization and helped demonstrate the real value of operating systems and intensive resource management. Single-stream batch-processing systems were the state of the art in the early 1960s.

REAL MEMORY MANAGEMENT TECHNIQUES

The main memory has to accommodate both the operating system and user space. Now, here the user space has to accommodate various user processes. We also want these several user processes must reside in the main memory at the same time.

- Fixed/Static Partitioning
- Variable/Dynamic Partitioning
- Simple/Basic Paging
- Simple/Basic Segmentation

Fixed partition multiprogramming


For a process doing intensive calculation:
Shaded area indicates "Processor in use."
For a process doing regular input/output:
Use processor
Use processor
Use processor
Use processor
Wait for I/O completion.
Wait for I/O completion.
Wait for I/O completion.

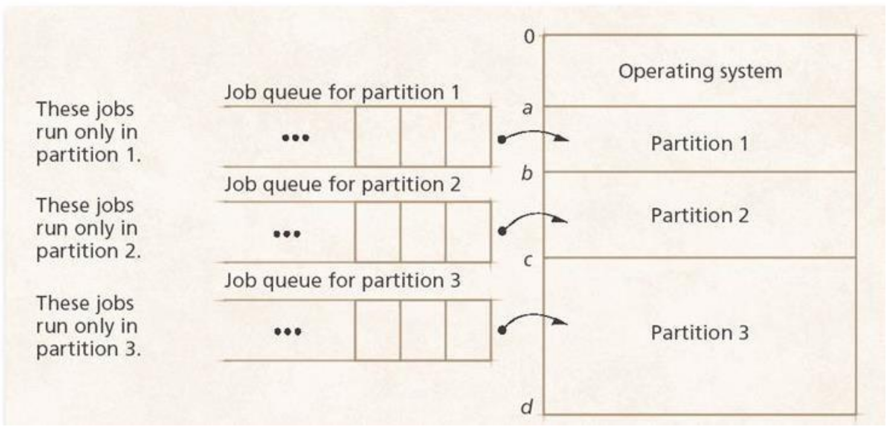
Even with batch-processing operating systems, single-user systems still waste a considerable amount of the computing resource. The program consumes the CPU resource until an input or output is needed. When the I/O request is issued, the job often cannot continue until the requested data is either send or received. Input and output speeds are extremely slow compared with CPU speeds. Increase the utilization of the CPU by intensive management. This time chose to implement multiprogramming systems. Several users simultaneously compete for system resources. The job currently waiting for I/O will produce the CPU to another job ready to calculations if indeed, another job is waiting. Both input/output and CPU calculations can occur simultaneously.

Advantage of Multiprogramming

It is necessary for several jobs to reside in the computer's main storage at once. When one job requests input/output, the CPU may be immediately switched to another and may do calculations without delay. Multiprogramming requires considerably more storage than a single user system. The improved resource use for the CPU. The peripheral devices more than justifies the expense of additional storage.

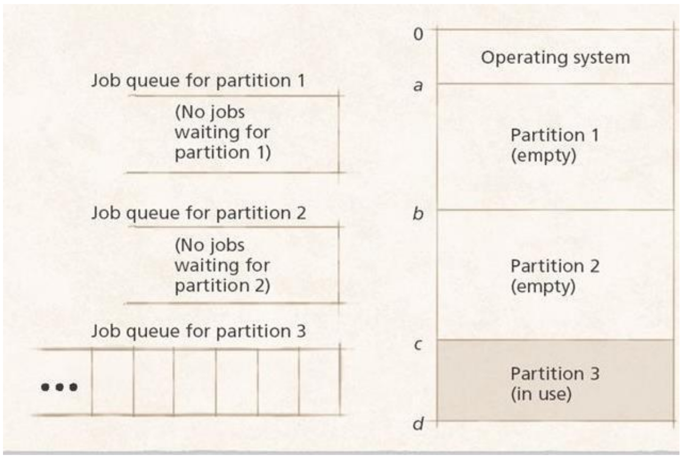
Fixed Partition Multiprogramming: Absolute Translation and Loading

Fixed partition multiprogramming in which main storage was divided into a number of fixed-size partitions. Each partition holds a single job. The CPU was switched rapidly between users to create the illusion of simultaneity.



0
Operating system
a
Job queue for partition 1
These jobs run only in partition 1.
Partition 1
b
Job queue for partition 2
These jobs run only in partition 2.
Partition 2
c
Job queue for partition 3
These jobs run only in partition 3.
Partition 3
d

Jobs were translated with absolute assemblers and compilers to run only in a specific partition. Job was ready to run and its partition was occupied. Then that job had to wait, even if other partitions were available. This resulted in waste of the storage resource. But the OS was relatively straightforward to implement.

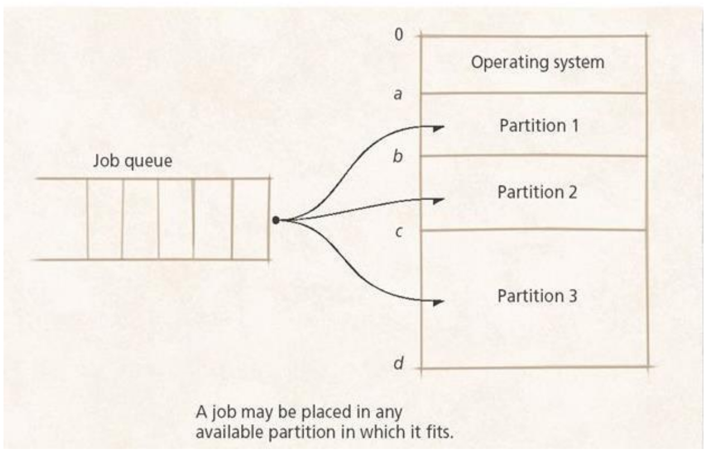


0
Operating system
a
Job queue for partition 1
(No jobs waiting for partition 1)
Partition 1 (empty)
b
Job queue for partition 2
(No jobs waiting for partition 2)
Partition 2 (empty)
c
Job queue for partition 3
Partition 3 (in use)
d

Memory waste under fixed partition multiprogramming with absolute translation and loading

An extreme example of poor storage utilization in fixed partition multiprogramming with absolute translation and loading. Jobs waiting for partition 3 are small and could "fit" in the other partitions. But with absolute translation and loading, these jobs may run only in partition 3. The other two partitions remain empty.

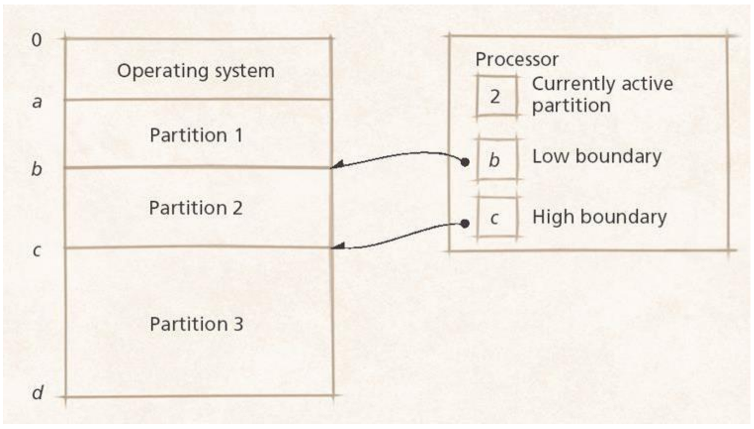
Fixed partition multiprogramming: relocatable translation and loading


0
Operating system
a
Partition 1
b
Job queue
Partition 2
c
Partition 3
d
A job may be placed in any available partition in which it fits.

Relocating compilers, assemblers and loaders are used to produce relocatable programs. It can run in any available partition that is large enough to hold them. This scheme eliminates some of the storage waste characteristic in multiprogramming with absolute translation and loading.

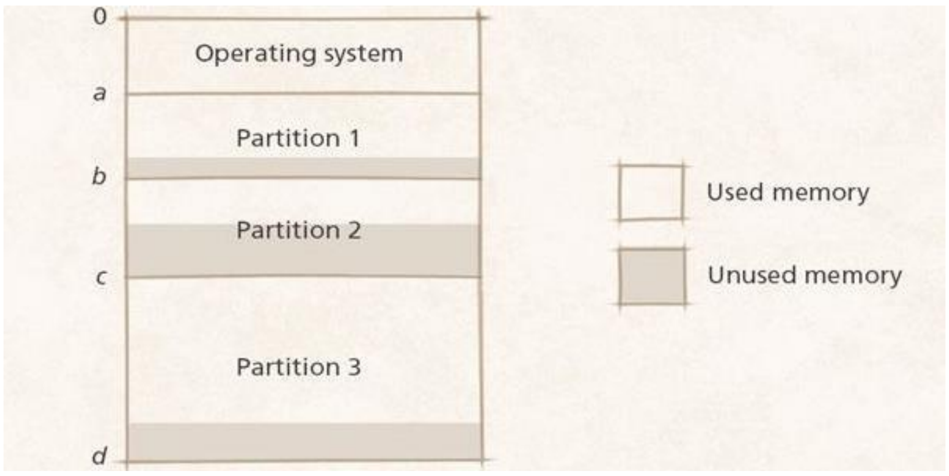
Protection in multiprogramming systems

Allowing Relocation and Transfers between partitions. Protection implemented by the use of several boundary registers: low and high boundary registers, or base register with length. Fragmentation occurs if user programs cannot completely fill a partition - wasteful.



Fragmentation in fixed partition multiprogramming

Storage fragmentation occurs in every computer system. In fixed partition multiprogramming systems, fragmentation occurs. Either user jobs do not completely fill their designed partitions. A partition remains unused if it is too small to hold a waiting job. Consider the warehouse example, multiple jobs of different types (perhaps size) entering storage in different partitions. Several users simultaneously compete for system resources. Switch between I/O jobs and calculation jobs for instance. To take advantage of this sharing of CPU, important for many jobs to be present in main memory.



Internal fragmentation in a fixed partition multiprogramming system

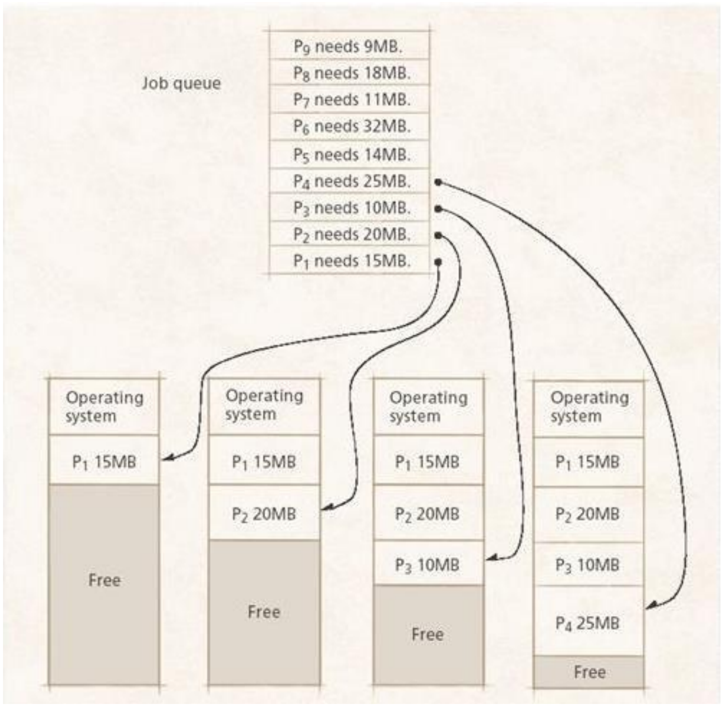
VARIABLE PARTITION MULTIPROGRAMMING

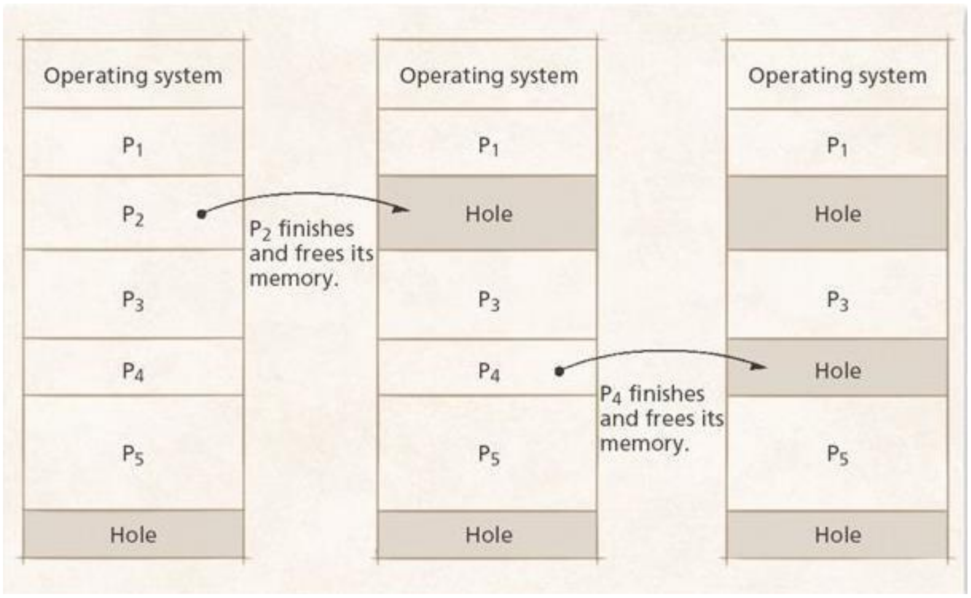
System designers found fixed partitions too respective. It decided that an obvious improvement, to allow jobs to occupy as much storage needed. No fixed boundaries would be observed. Instead, jobs would be given as much storage as they required is called variable partition multiprogramming.

In variable partition multiprogramming the jobs arrive, the scheduling mechanisms decide for proceed. They are given much storage as they need. There is no wastage a job partition is exactly the size of the job. Every storage organization scheme involves some degree of waste.

In variable partition multiprogramming, the waste does not become obvious until jobs start to finish. Leave holes in the main storage. These holes can be used for other jobs. These remaining holes get smaller eventually becoming too small to hold new jobs.

Initial partition assignment in variable partition multiprogramming



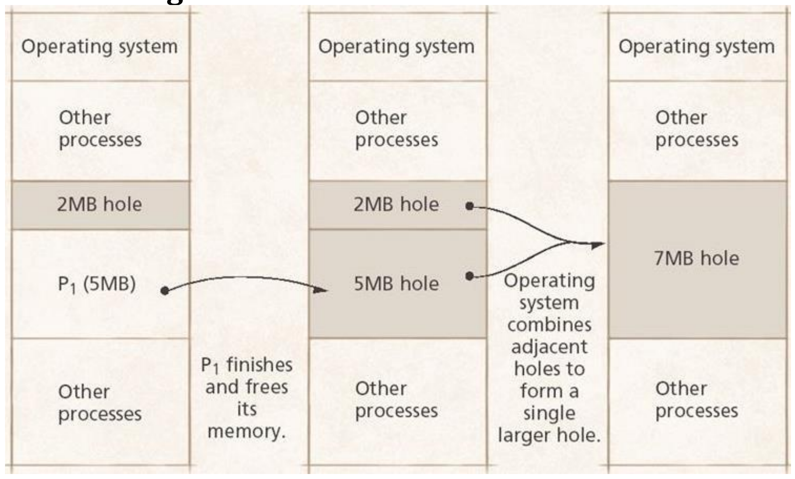


Operating system
P_1
P_2
P_3
P_4
P_5
Hole
P_2 finishes and frees its memory.
P_4 finishes and frees its memory.

Variable partition multiprogramming characteristics

- Coalescing holes
- Storage compaction
- Storage placement strategies

Coalescing holes

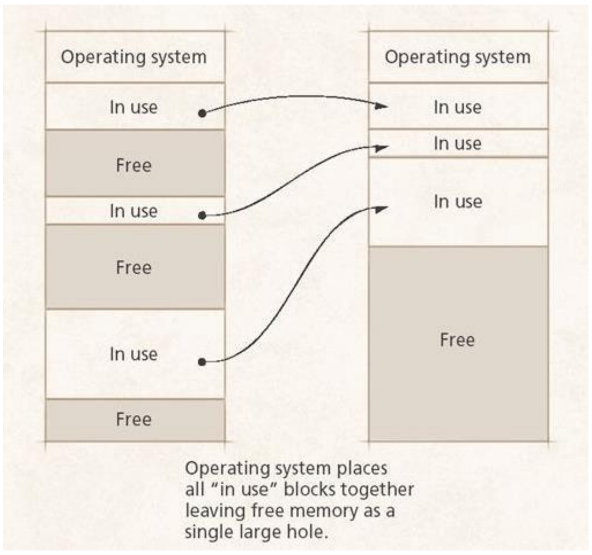


Job finishes in variable partition multiprogramming system, check whether the storage being freed (unrestricted) borders on other free storage areas (holes). The free storage list,

- ❖ An additional hole.
- ❖ A single hole reflecting the merger of the existing hole.
- ❖ New adjacent hole.

The process of merging adjacent holes to form a single larger hole in called coalescing.

Storage compaction


Operating system
In use
Free
In use
Free
In use
Free
Operating system
In use
In use
In use
Free
Operating system places all "in use" blocks together leaving free memory as a single large hole.

Storage compaction in variable partition multiprogramming

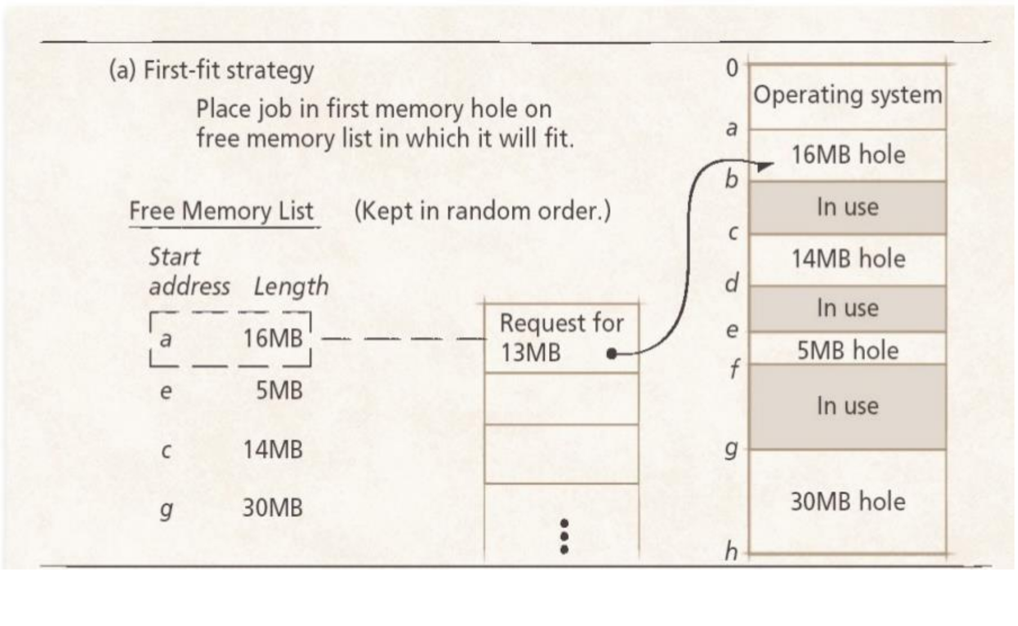
When a job requires a certain amount of main storage no individual hole is large enough to hold the job. Even though the sum of all the holes is larger than the storage needed by the new job. The technique of storage compaction involves moving all occupied areas of storage to one end or the other of main storage. Rearranges memory into a single contiguous block free space. A single contiguous block of occupied space. It is also referred as burping the storage or garbage collection.

Storage placement strategies

1. **Best fit strategy**
   An incoming job is placed in the hole where it best fits (i.e., the amount of free space left is minimal)
2. **First fit strategy**
   Placed in the first available slot large enough to hold the job.
3. **Worst fit strategy**
   Place in storage in the largest slot available. The remaining may still be large enough to hold another job.

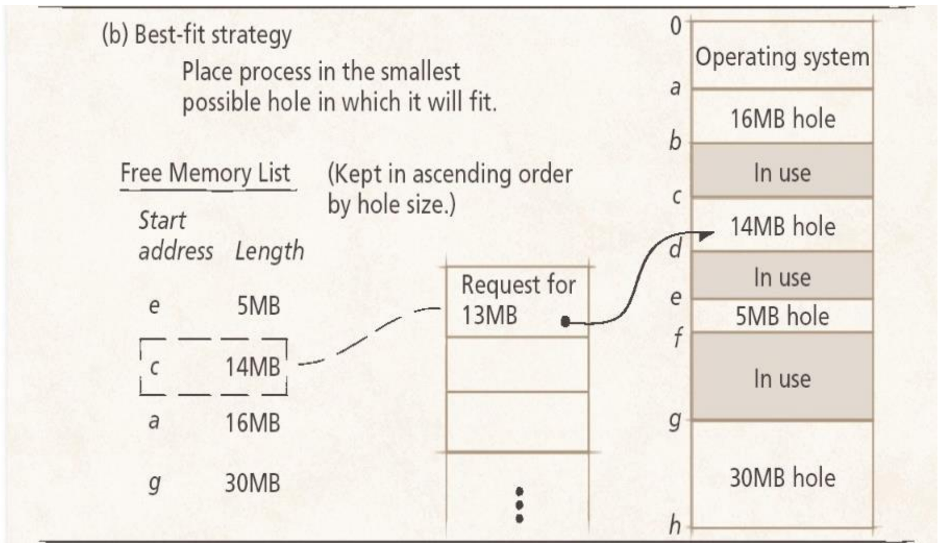
First fit strategy

This method keeps the free/busy list of jobs organized by memory location, low-ordered to high-ordered memory. In this method, first job claims the first available memory with space more than or equal to its size. The operating system doesn't search for appropriate partition but just allocate the job to the nearest memory partition available with sufficient size.



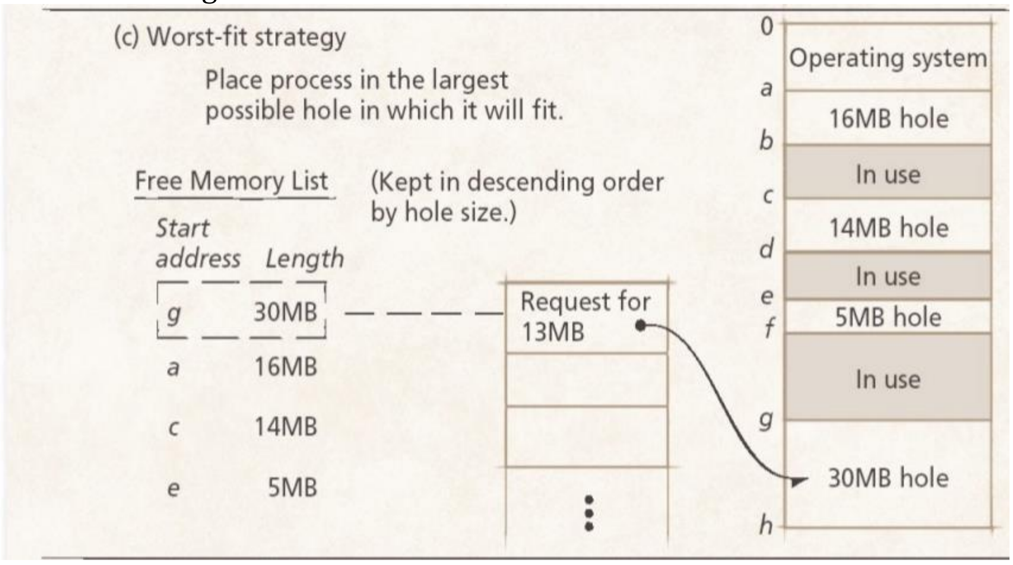
Best fit strategy

This method keeps the free/busy list in order by size – smallest to largest. In this method, the operating system first searches the whole of the memory according to the size of the given job and allocates it to the closest-fitting free partition in the memory, making it able to use memory efficiently. Here the jobs are in the order from smallest job to largest job.



Worst fit strategy

In this allocation technique, the process traverses the whole memory and always search for the largest hole/partition, and then the process is placed in that hole/partition. It is a slow process because it has to traverse the entire memory to search the largest hole.



Unit –IV

Virtual Storage: Virtual Storage Management Strategies – Page Replacement Strategies – Working Sets – Demand Paging – Page Size. Processor Management: Job and Processor Scheduling: Preemptive Vs Non-preemptive scheduling – Priorities – Deadline scheduling.

VIRTUAL STORAGE

Virtual storage management strategies

There are three main strategies namely

Fetch strategies – concerned with when a page or segment should be brought from secondary to primary storage

Placement strategies – concerned with where in primary storage to place an incoming page or segment

Replacement strategies – concerned with deciding which page or segment to displace to make room for an incoming page or segment when primary storage is already fully committed

Page replacement algorithms

There are many page replacement algorithms and the most important three are FIFO, optimal replacement and least recently used. This subsection explains the above three algorithms.

FIFO

The simplest page replacement algorithm is first in first out. In this scheme, when a page must be replaced, the oldest page is chosen. For example consider the page reference string

1, 5, 6, 1, 7, 1, 5, 7, 6, 1, 5, 1, 7

For a three frame case, the FIFO will work as follows. Let all our 3 frames are initially empty.

1 1 1 7 7 7 6 6

5 5 5 1 1 1 7

6 6 6 5 5 5

You can see, FIFO creates eight page faults.

Optimal replacement

In optimal page replacement algorithm, we replace that page which will not be used for the longest period of time. For example for the reference string

1, 5, 6, 1, 7, 1, 5, 7, 6, 1, 5, 1, 7

with 3 frames, the page faults will be as follows

1 1 1 1 1 1

5 5 5 5 5

6 7 6 7

You can see that Optimal replacement, creates six page faults

Least recently used

Most of the case, predicting the future page references is difficult and hence implementing optimal replacement is difficult. Hence there is a need of other scheme which approximates the optimal replacement. Least recently used (LRU) schemes approximate the future uses by the past used pages. In LRU scheme, we replace those pages which have not been used for the longest period of time.

For example for the reference string

1, 5, 6, 1, 7, 1, 5, 7, 6, 1, 5, 1, 7

with 3 frames, the page faults will be as follows

1 1 1 1 1 6 6 6 7

5 5 7 7 7 7 5 5

6 6 5 5 1 1 1

You can see that LRU creates nine page faults

Working sets

If the number of frames allocated to a low-priority process falls below the minimum numberrequired, we must suspend its execution. We should then page out it remaining pages, freeing all of its allocated frames. A process is thrashing if it is spending more time paging than executing.

Thrashing can cause severe performance problems. To prevent thrashing, we must provide a process with as many frames as it needs. There are several techniques available to know how many frame a process needs. Working sets is a strategy which starts by looking at what a program is actually using.

Demand paging

Demand paging is the most common virtual memory system. Demand paging is similar to a paging system with swapping. When we need a program, it is swapped from the backing storage.

There are also lazy swappers, which never swaps a page into memory unless it is needed. The lazy swapper decreases the swap time and the amount of physical memory needed, allowing an increased degree of multiprogramming.

Page size

There is no single best page size. The designers of the Operating system will decide the page
size for an existing machine. Page sizes are usually be in powers of two, ranging from 28 to 212 bytes or words. The size of the pages will affect in the following way.

a) Decreasing the page size increases the number of pages and hence the size of the page table.

b) Memory is utilized better with smaller pages.

c) For reducing the I/O time we need to have smaller page size.

d) To minimize the number of page faults, we need to have a large page size

PROCESSOR MANAGEMENT:

Introduction:

When one or more process is runnable, the operating system must decide which oneto run first. The part of the operating system that makes decision is called the Scheduler; the algorithm it uses is called the Scheduling Algorithm**.**

An operating system has three main CPU schedulers namely the long term scheduler, short term scheduler and medium term schedulers. The long term scheduler determines which jobs are admitted to the system for processing. It selects jobs from the job pool and loads them into memory for execution. The short term scheduler selects from among the jobs in memory which are ready to execute and allocated the cpu to one of them. The medium term scheduler helps to remove processes from main memory and from the active contention for the cpu and thus reduce the degree of multiprogramming.

The cpu scheduler has another component called as dispatcher. It is the module that actually gives control of the cpu to the process selected by the short term scheduler which involves loading of registers of the process, switching to user mode and jumping to the proper location.

Before looking at specific scheduling algorithms, we should think about what the scheduler is trying to achieve. After all the scheduler is concerned with deciding on policy, not providing a mechanism. Various criteria come to mind as to what constitutes a good scheduling algorithm. Some of the possibilities include:

1. Fairness – make sure each process gets its fair share of the CPU.
2. Efficiency (CPU utilization) – keep the CPU busy 100 percent of the time.
3. Response Time [Time from the submission of a request until the first response is produced] – minimize response time for interactive users.
4. Turnaround time [The interval from the time of submission to the time of completion] – minimize the time batch users must wait for output.
5. Throughput [Number of jobs that are completed per unit time] – maximize the number of jobs processed per hour.
6. Waiting time – minimize the waiting time of jobs

Preemptive Vs Non-Preemptive

The Strategy of allowing processes that are logically runnable to be temporarily suspended is called Preemptive Scheduling**.** ie., a scheduling discipline is preemptive if the CPU can be taken away. Preemptive algorithms are driven by the notion of prioritized computation. The process with the highest priority should always be the one currently using the processor. If a process is currently using the processor and a new process with a higher priority enters, the ready list, the process on the processor should be removed and returned the ready list until it is once again the highest-priority process in the system.

Run to completion is also called Nonpreemptive Scheduling. ie., a scheduling discipline is nonpreemptive if, once a process has been given the CPU, the CPU cannot be taken away from that process. In short, Non-preemptive algorithms are designed so that once a process enters the running state(is allowed a process), it is not removed from the processor until it has completed its service time (or it explicitly yields the processor). This leads to race condition and necessitates of semaphores, monitors, messages or some other sophisticated method for preventing them. On the other hand, a policy of letting a process run as long as it is wanted would mean that some process computing π to a billion places could deny service to all other processes indefinitely.

Priorities

A priority is associated with each job, and the cpu is allocated to the job with the highest priority. Priorities are generally some fixed numbers such as 0 to 7 or 0 to 4095. However there is no general agreement on whether 0 is the highest or lowest priority. Priority can be defined either internally or externally. Examples of internal priorities are time limits, memory requirements, number of open files, average I/O burst time, CPU burst time, etc. External priorities are given by the user.

A major problem with priority scheduling algorithms is indefinite blocking or starvation. A solution to this problem is aging. Aging is a technique of gradually increasing the priority of jobs that wait in the system for a long time.

Deadline scheduling

Certain jobs have to be completed in specified time and hence to be scheduled based on deadline. If delivered in time, the jobs will be having high value and otherwise the jobs will be having nil value. The deadline scheduling is complex for the following reasons

a) Giving resource requirements of the job in advance is difficult

b) A deadline job should be run without degrading other deadline jobs

c) In the event of arriving new jobs, it is very difficult to carefully plan resource requirements

d) Resource management for deadline scheduling is really an overhead

UNIT –V

Device and Information Management Disk Performance Optimization: Operation of moving head disk storage – Need for disk scheduling – Seek Optimization – File and Database Systems: File System – Functions – Organization – Allocating and freeing space – File descriptor – Access control matrix.

DEVICE AND DISK MANAGEMENT

Introduction

In multiprogramming systems several different processes may want to use the system's resources simultaneously. For example, processes will contend to access an auxiliary storage device such as a disk. The disk drive needs some mechanism to resolve this contention, sharing the resource between the processes fairly and efficiently.

A magnetic disk consists of a collection of platters which rotate on about a central spindle. These platters are metal disks covered with magnetic recording material on both sides. Each disk surface is divided into concentric circles called *tracks*. Disk divides each track into *sectors*, each typically contains 512 bytes. While reading and writing the *head* moves over the surface of the platters until it finds the track and sector it requires. This is like finding someone's home by first finding the street (track) and then the particular house number (sector). There is one head for each surface on which information is stored each on its own *arm*. In most systems the arms are connected together so that the heads move inunison, so that each head is over the same track on each surface.

The term *cylinder* refers to the collection of all tracks which are under the heads at any time. In order to satisfy an I/O request the disk controller must first move the head to the correct track and sector. Moving the head between cylinders takes a relatively long time so in order to maximize the number of I/O requests which can be satisfied the scheduling policy should try to minimize the movement of the head. On the other hand, minimizing head movement by always satisfying the request of the closest location may mean that some requests have to wait a long time. Thus, there is a trade-off between *throughput* (the average number of requests satisfied in unit time) and *response time* (the average time between a request arriving and it being satisfied).

Need for Disk Scheduling

Access time has two major components namely seek time and rotational latency. *Seektime* is the time for the disk are to move the heads to the cylinder containing the desired sector. *Rotational latency* is the additional time waiting for the disk to rotate the desired sector to the disk head. In order to have fast access time we have to minimize the seek time which is approximately equal to the seek distance.

Disk bandwidth is the total number of bytes transferred, divided by the total time between the first request for service and the completion of the last transfer. The operating system is responsible for using hardware efficiently for the disk drives, to have a fast access time and disk bandwidth. This in turn needs a good disk scheduling.

FILE SYSTEMS AND ORGANIZATION

In computing, a file system (often also written as file system) is a method for storing and organizing computer files and the data they contain to make it easy to find and access them. File systems may use a data storage device such as a hard disk or CD-ROM and involve maintaining the physical location of the files, they might provide access to data on a file server by acting as clients for a network protocol (e.g., NFS, SMB, or 9P clients), or they may be virtual and exist only as an access method for virtual data.

More formally, a file system is a set of abstract data types that are implemented for the storage, hierarchical organization, manipulation, navigation, access, and retrieval of data. File systems share much in common with database technology, but it is debatable whether a file system can be classified as a special-purpose database (DBMS).

Functions of file systems

The most familiar file systems make use of an underlying data storage device that offers access to an array of fixed-size blocks, sometimes called *sectors*, generally 512 bytes each. The file system software is responsible for organizing these sectors into files and directories, and keeping track of which sectors belong to which file and which are not being used.

However, file systems need not make use of a storage device at all. A file system can be used to organize and represent access to any data, whether it be stored or dynamically generated (eg, from a network connection).

Whether the file system has an underlying storage device or not, file systems typically have directories which associate file names with files, usually by connecting the file name to an index into a file allocation table of some sort, such as the FAT in an MS-DOS file system, or an inode in a Unix-like file system. Directory structures may be flat, or allow hierarchies where directories may contain subdirectories. In some file systems, file names are structured, with special syntax for filename extensions and version numbers. In others, file names are simple strings, and per-file metadata is stored elsewhere.

Other bookkeeping information is typically associated with each file within a file system. The length of the data contained in a file may be stored as the number of blocks allocated for the file or as an exact byte count. The time that the file was last modified may be stored as the file's timestamp. Some file systems also store the file creation time, the time it was last accessed, and the time that the file's meta-data was changed. (Note that many early PC operating systems did not keep track of file times.) Other information can include the file's device type (e.g., block, character, socket, subdirectory, etc.), its owner user-ID and group-ID, and its access permission settings (e.g., whether the file is read-only, executable, etc.).

The hierarchical file system was an early research interest of Dennis Ritchie of Unix fame; previous implementations were restricted to only a few levels, notably the IBM implementations, even of their early databases like IMS. After the success of Unix, Ritchie extended the file system concept to every object in his later operating system developments, such as Plan 9 and Inferno.

Traditional file systems offer facilities to create, move and delete both files and directories. They lack facilities to create additional links to a directory (hard links in Unix), rename parent links (".." in Unix-like OS), and create bidirectional links to files.Traditional file systems also offer facilities to truncate, append to, create, move, delete and in-place modify files. They do not offer facilities to prepend to or truncate from the beginning of a file, let alone arbitrary insertion into or deletion from a file. The operations provided are highly asymmetric and lack the generality to be useful in unexpected contexts.

For example, interprocess pipes in Unix have to be implemented outside of the file system because the pipes concept does not offer truncation from the beginning of files.Secure access to basic file system operations can be based on a scheme of access control lists or capabilities. Research has shown access control lists to be difficult to secure properly, which is why research operating systems tend to use capabilities. Commercial file systems still use access control lists.

Arbitrary attributes can be associated on advanced file systems, such as XFS, xt2/ext3, some versions of UFS, and HFS+, using extended file attributes. This feature is implemented in the kernels of Linux, FreeBSD and Mac OS X operating systems, and allows metadata to be associated with the file at the *file system* level. This, for example, could be the author of a document, the character encoding of a plain-text document, or a checksum.

TYPES OF FILE SYSTEMS

File system types can be classified into disk file systems, network file systems and special purpose file systems.

Disk file systems

A *disk file system* is a file system designed for the storage of files on a data storage device, most commonly a disk drive, which might be directly or indirectly connected to the computer. Examples of disk file systems include FAT, FAT32, NTFS, HFS and HFS+, ext2, ext3, ISO 9660, ODS-5, and UDF. Some disk file systems are journaling file systems or versioning file systems.

1.4.2 Flash file systems

A *flash file system* is a file system designed for storing files on flash memory devices. These are becoming more prevalent as the number of mobile devices is increasing, and the capacity of flash memories catches up with hard drives.

While a block device layer can run emulate hard drive behavior and store regular file systems on a flash device, this is suboptimal for several reasons:

Erasing blocks: Flash memory blocks have to be explicitly erased before they can be written to. The time taken to erase blocks can be significant, thus it is beneficial to erase unused blocks while the device is idle.

Random access: Disk file systems are optimized to avoid disk seeks whenever possible, due to the high cost of seeking. Flash memory devices impose no seek latency.

Wear levelling: Flash memory devices tend to "wear out" when a single block is repeatedly overwritten; flash file systems try to spread out writes as evenly as possible.

It turns out that log-structured file systems have all the desirable properties for a flash file system. Such file systems include JFFS2 and YAFFS.

Database file systems

A new concept for file management is the concept of a database-based file system. Instead of, or in addition to, hierarchical structured management, files are identified by their characteristics, like type of file, topic, author, or similar metadata.

Transactional file systems

This is a special kind of file system in that it logs events or transactions to files. Each operation that you do may involve changes to a number of different files and disk structures. In many cases, these changes are related, meaning that it is important that they all be executed at the same time. Take for example a bank sending another bank some money electronically. The bank's computer will "send" the transfer instruction to the other bank and also update its own records to indicate the transfer has occurred. If for some reason the computer crashes before it has had a chance to update its own records, then on reset, there will be no record of the transfer but the bank will be missing some money.

A transactional system can rebuild the actions by resynchronizing the "transactions" on both ends to correct the failure. All transactions can be saved as well, providing a complete record of what was done and where. This type of file system is designed and intended to be fault tolerant, and necessarily incurs a high degree of overhead.

Network file systems

A network file system is a file system that acts as a client for a remote file access protocol, providing access to files on a server. Examples of network file systems include clients for the NFS, SMB protocols, and file-system-like clients for FTP and WebDAV.

1.4.6 Special purpose file systems

A special purpose file system is basically any file system that is not a disk file system or network file system. This includes systems where the files are arranged dynamically by software, intended for such purposes as communication between computer processes or temporary file space.

Special purpose file systems are most commonly used by file-centric operating systems such as Unix. Examples include the procfs (`/proc`) file system used by some Unix variants, which grants access to information about processes and other operating system features. Deep space science exploration craft, like Voyager I & II used digital tape based special file systems. Most modern space exploration craft like Cassini-Huygens used Real-time operating system file systems or RTOS influenced file systems. The Mars Rovers are one such example of an RTOS file system, important in this case because they are implemented in flash memory.

Flat file systems

In a flat file system, there are no subdirectories—everything is stored at the same (root) level on the media, be it a hard disk, floppy disk, etc. While simple, this system rapidly becomes inefficient as the number of files grows, and makes it difficult for users to organise data into related groups.

Like many small systems before it, the original Apple Macintosh featured a flat file system, called Macintosh File System. Its version of Mac OS was unusual in that the file management software (Macintosh Finder) created the illusion of a partially hierarchical filing system on top of MFS. This structure meant that every file on a disk had to have a unique name, even if it appeared to be in a separate folder. MFS was quickly replaced with Hierarchical File System, which supported real directories.

File systems and operating systems

Most operating systems provide a file system, and is an integral part of any modern operating system. Early microcomputer operating systems' only real task was file management — a fact reflected in their names (see DOS). Some early operating systems had a separate component for handling file systems which was called a disk operating system. On some microcomputers, the disk operating system was loaded separately from the rest of the operating system. On early operating systems, there was usually support for only one, native,
unnamed file system; for example, CP/M supports only its own file system, which might be called "CP/M file system" if needed, but which didn't bear any official name at all.

FILE ORGANIZATION

1. A file is organized logically as a sequence of records.

2. Records are mapped onto disk blocks.
3. Files are provided as a basic construct in operating systems, so we assume the existence of an underlying file system.
4. Blocks are of a fixed size determined by the operating system.
5. Record sizes vary.
6. In relational database, tuples of distinct relations may be of different sizes.
7. One approach to mapping database to files is to store records of one length in a given file.
8. An alternative is to structure files to accommodate variable-length records. (Fixedlength is easier to implement.)

ALLOCATING AND FREEZING SPACE

Free Space Management

To keep track of the free space, the file system maintains a free space list which records all disk blocks which are free. We search the free space list to create a file for the required amount of space and allocate it to the new file. This space is then removed from the free space list. When a file is deleted, its disk space is added to the free space list.

Bit-Vector

Frequently, the free-space list is implemented as a bit map or bit vector. Each block is represented by a 1 bit. If the block is free, the bit is 0; if the block is allocated, the bit is 1. For example, consider a disk where blocks 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 17, 18, 25, 26, and 27 are free, and the rest of the blocks are allocated. The free-space bit map would be:
11000011000000111001111110001111…

The main advantage of this approach is that it is relatively simple and efficient to find n consecutive free blocks on the disk. Unfortunately, bit vectors are inefficient unless the entirevector is kept in memory for most accesses. Keeping it main memory is possible for smaller disks such as on microcomputers, but not for larger ones.

Linked List

Grouping

A modification of the free-list approach is to store the addresses of n free blocks in the first free block. The first n-1 of these are actually free. The last one is the disk address of another block containing addresses of another n free blocks. The importance of this implementation is that addresses of a large number of free blocks can be found quickly.

Counting

Another approach is to take advantage of the fact that, generally, several contiguous blocks may be allocated or freed simultaneously, particularly when contiguous allocation is used. Thus, rather than keeping a list of free disk addresses, the address of the first free block is kept and the number n of free contiguous blocks that follow the first block. Each entry in the free-space list then consists of a disk address and a count. Although each entry requires more space than would a simple disk address, the overall list will be shorter, as long as the count is generally greater than 1.

Contiguous allocation

The contiguous allocation method requires each file to occupy a set of contiguous address on the disk. Disk addresses define a linear ordering on the disk. Notice that, with this ordering, accessing block b+1 after block b normally requires no head movement. When head movement is needed (from the last sector of one cylinder to the first sector of the next cylinder), it is only one track. Thus, the number of disk seeks required for accessing contiguous allocated files in minimal, as is seek time when a seek is finally needed.Contiguous allocation of a file is defined by the disk address and the length of the first block.
If the file is n blocks long, and starts at location b, then it occupies blocks b, b+1, b+2, …,
b+n-1. The directory entry for each file indicates the address of the starting block and the
length of the area allocated for this file.

The difficulty with contiguous allocation is finding space for a new file. If the file to be created is n blocks long, then the OS must search for n free contiguous blocks. First-fit, bestfit, and worst-fit strategies (as discussed in Chapter 4 on multiple partition allocation) are the most common strategies used to select a free hole from the set of available holes. Simulations have shown that both first-fit and best-fit are better than worst-fit in terms of both time storage utilization. Neither first-fit nor best-fit is clearly best in terms of storage utilization, but first-fit is generally faster.

These algorithms also suffer from external fragmentation. As files are allocated and deleted, the free disk space is broken into little pieces. External fragmentation exists when enough total disk space exists to satisfy a request, but this space not contiguous; storage is fragmented into a large number of small holes.

Linked allocation

The problems in contiguous allocation can be traced directly to the requirement that the spaces be allocated contiguously and that the files that need these spaces are of different sizes. These requirements can be avoided by using linked allocation.

In linked allocation, each file is a linked list of disk blocks. The directory contains a pointer to the first and (optionally the last) block of the file. For example, a file of 5 block which starts at block 4, might continue at block 7, then block 16, block 10, and finally block 27. Each block contains a pointer to the next block and the last block contains a NIL pointer.
The value -1 may be used for NIL to differentiate it from block 0.

Indexed allocation

The indexed allocation method is the solution to the problem of both contiguous and linked allocation. This is done by bringing all the pointers together into one location called the index block. Of course, the index block will occupy some space and thus could be considered as an overhead of the method. In indexed allocation, each file has its own index block, which is an array of disk sector of addresses. The ith entry in the index block points to the ith sector of the file. The directory contains the address of the index block of a file. To read the ith sector of the file, the pointer in the ith index block entry is read to find the desired sector. Indexed allocation supports direct access, without suffering from external fragmentation. Any free block anywhere on the disk may satisfy a request for more space.

FILE DESCRIPTORS AND ACCESS CONTROL

Aims and Objectives

In this lesson we will learn about the file descriptors and access control.

The objectives of this lesson is to make the candidate aware of the following

a) file descriptors

b) operations on file descriptor

a. creating

b. deriving

c. modifying, etc.

c) access control matrix

Introduction

A file descriptor or file control block is a control block containing information the system needs to manage a file. The file descriptor is controlled by the operating system and is brought to the primary storage when a file is opened. A file descriptor contains information regarding (i) symbolic file name, (ii) location of file, (iii) file organization (sequential, indexed, etc.), (iv) device type, (v) access control data, (vi) type (data file, object program, C source program, etc.), (vii) disposition (temporary or permanent), (viii) date and time of creation, (ix) destroy date, (x) last modified date and time, (xi) access activity counts (number of reads, etc.).

File descriptor in programming

In computer programming, a **file descriptor** is an abstract key for accessing a file. The term is generally used in POSIX operating systems. In Microsoft Windows terminology and in the context of the C standard I/O library, "file handle" is preferred, though the latter case is technically a different object (see below).

In POSIX, a file descriptor is an integer, specifically of the C type int. There are 3 standard POSIX file descriptors which presumably every process (save perhaps a daemon) should expect to have:

Integer value	**Name**
0	Standard Input (stdin)
1	Standard Output (stdout)
2	Standard Error (stderr)

Generally, a file descriptor is an index for an entry in a kernel-resident data structure containing the details of all open files. In POSIX this data structure is called a file descriptor table, and each process has its own file descriptor table. The user application passes the abstract key to the kernel through a system call, and the kernel will access the file on behalf of the application, based on the key. The application itself cannot read or write the file descriptor table directly. In Unix-like systems, file descriptors can refer to files, directories, block or character devices (also called "special files"), sockets, FIFOs (also called named pipes), or unnamed pipes.

The FILE * file handle in the C standard I/O library routines is technically a pointer to a data structure managed by those library routines; one of those structures usually includes an actual low level file descriptor for the object in question on Unix-like systems. Since *file handle* refers to this additional layer, it is not interchangeable with *file descriptor*.

To further complicate terminology, Microsoft Windows also uses the term *file handle* to refer to the more low-level construct, akin to POSIX's file descriptors. Microsoft's C libraries also provide compatibility functions which "wrap" these native handles to support the POSIX-like convention of integer file descriptors as detailed above.

A program is passed a set of ``open file descriptors'', that is, pre-opened files. A setuid/setgid program must deal with the fact that the user gets to select what files are open and to what (within their permission limits). A setuid/setgid program must not assume that opening a new file will always open into a fixed file descriptor id, or that the open will succeed at all. It must also not assume that standard input (stdin), standard output (stdout), and standard error (stderr) refer to a terminal or are even open.

OPERATIONS ON FILE DESCRIPTORS

A modern Unix typically provides the following operations on file descriptors.

Creating file descriptors

open(), open64(), creat(), creat64()

socket()

socketpair()

pipe()

Deriving file descriptors

fileno()

dirfd()

Operations on a single file descriptor

read(), write()

recv(), send()

recvmsg(), sendmsg() (inc. allowing sending FDs)

sendfile()

lseek(), lseek64()

fstat(), fstat64()

fchmod()

fchown()

fdopen()

gzdopen()

ftruncate()

15.4.4 Operations on multiple file descriptors

select(), pselect()

poll(), epoll()

Operations on the file descriptor table

close()

dup()

dup2()

fcntl (F_DUPFD)

fcntl (F_GETFD and F_SETFD)

Operations that modify process state

fchdir(): sets the process's current working directory based on a directory file descriptor

mmap(): maps ranges of a file into the process's address space

File locking

flock()

fcntl (F_GETLK, F_SETLK and F_SETLKW)

lockf()

Sockets

connect()

bind()

listen()

accept(): creates a new file descriptor for an incoming connection

getsockname()

getpeername()

getsockopt(), setsockopt()

shutdown(): shuts down one or both halves of a full duplex connection

Reference :

1. LelandL.Beck,System Software:An Introduction to Systems Programming,Pearson,Third Edition.
2. 2 H.M.Deitel,Operating Systems, 2nd Edition,Perason, 2003.

Prepared by :

M.Balasubramaniyam,
Assistant Professor,
Department of BCA,
Vidyasagar College of arts and science,
Udumalpet.

Reference Website :

www.studoc.com