IAT-3_OS

PART-B

1. Explain the data structure and detailed design of pass 1 of a two-pass assembler with algorithm.

- **Pass 1 (define symbols)**
  - Assign addresses to all statements in the program
  - Save the addresses assigned to all labels for use in Pass 2
  - Perform assembler directives, including those for address assignment, such as BYTE and RESW

Data Structures

- Operation Code Table (OPTAB)
- Symbol Table (SYMTAB)
- Location Counter(LOCCTR)

OPTABLE

- Mnemonic operation codes ⇔ Machine code
- Contain instruction format and length
  - LOCCTR ← LOCCTR + (instruction length)
- Implementation
  - It is a static table
  - Array or hash table
    - Usually use a hash table (mnemonic opcode as key)

SYMTAB (symbol table)

COPY	1000
FIRST	1000
CLOOP	1003
ENDFIL	1015
EOF	1024
THREE	102D
ZERO	1030
RETADR	1033
LENGTH	1036
BUFFER	1039
RDREC	2039

- Content
  - label name, value, flag, (type, length) etc.
- Characteristic
  - dynamic table (insert, delete, search)
- Implementation
  - hash table, non-random keys, hashing function

LOCCTR

- Initialize to be the beginning address specified in the “START” statement
- LOCCTR ← LOCCTR + (instruction length)
- The current value of LOCCTR gives the address to the label encountered

Pass 1 Algorithm:

```
begin
    read first line
    if OPCODE = 'START' THEN
        begin
            save #[OPERAND] as starting address
            initialize LOCCTR to starting address
            write line to intermediate file
            read next input line
        end {if START}
    else
        initialize LOCCTR to 0
    while OPCODE ≠ 'END' do
        begin
            if this is not a comment line then
                begin
                    if there is a symbol in LABEL field then
```

```
begin
    search SYMTAB for LABEL
    if found then
        set error flag (duplicate symbol)
    else
        insert (LABEL, LOCCTR) into SYMTAB
end {if symbol}
search OPTAB for OPCODE
if found then
    add 3 (instruction length) to LOCCTR
else if OPCODE='WORD' then
    add 3 to LOCCTR
else if OPCODE = 'RESW' then
    add 3* #(OPERAND) to LOCCTR
else if OPCODE = 'RESB' then
    add #(OPERAND) to LOCCTR
else if OPCODE = 'BYTE' then
    begin
        find length of constant in bytes
        add length to LOCCTR
    end {if BYTE}
else
    set error flag (invalid operation code)
end (if not a comment)
write line to intermediate file
read next input line
end {while not END}
write last line to intermediate file.
save (LOCCTR - starting address) as program length
end {pass 1}
```

2. Explain the data structure and detailed design of pass 2 of a two-pass assembler with algorithm.

- **Pass 1 (define symbols)**
  - Assign addresses to all statements in the program
  - Save the addresses assigned to all labels for use in Pass 2
  - Perform assembler directives, including those for address assignment, such as BYTE and RESW

Data Structures

- Operation Code Table (OPTAB)
- Symbol Table (SYMTAB)
- Location Counter(LOCCTR)

OPTABLE

- Mnemonic operation codes ⇔ Machine code
- Contain instruction format and length
  - LOCCTR ← LOCCTR + (instruction length)
- Implementation
  - It is a static table
  - Array or hash table
    - Usually use a hash table (mnemonic opcode as key)

SYMTAB (symbol table)

COPY	1000
FIRST	1000
CLOOP	1003
ENDFIL	1015
EOF	1024
THREE	102D
ZERO	1030
RETADR	1033
LENGTH	1036
BUFFER	1039
RDREC	2039

- Content
  - label name, value, flag, (type, length) etc.
- Characteristic
  - dynamic table (insert, delete, search)
- Implementation
  - hash table, non-random keys, hashing function

LOCCTR

- Initialize to be the beginning address specified in the “START” statement
- LOCCTR ← LOCCTR + (instruction length)
- The current value of LOCCTR gives the address to the label encountered

Write last line to intermediate file
Save (~~LOCCTR~~ - Starting address) as program length
end (Pass 1)

Pass 2 :-

read first input line (from intermediate file)
if OPCODE = 'START' then
begin
write listing line
read next input line
end (if START)
write Header record to object Program
initialise first Text record
While OPCODE ≠ 'END do
begin
if this is not a Comment line then
Search ~~found~~ OPTAB for OPCODE
if found then
begin
if there is a ~~Sy~~ ~~SYMBOL~~ symbol
in OPERAND FIELD then
begin
Search SYMTAB for OPERAND
if found then
Store Symbol value as
operand address

```
else
  begin
    Store 0 as operand address
    set error flag (undefined Symbol)
  end
end (if Symbol)
else
  Store 0 as operand address,
assemble the object Code instruction,
end (if opcode found)
else if OPCODE = 'BYTE' or 'WORD' then
  Convert constant to object Code
if object code will not fit into the current
  text record
   begin
     Write Text record to object Program
     intialize new Text record
   end
add object code to Text record
end (if not Comment)
write listing line
read next input line
end (While not END)
Write last Text record to object ~~Code~~ program
Write End record to object program
Write last Listing line
```

3. Explain the data structure used for macro expansion and write the single pass macro processor algorithm

- * Two new assembler directives are used in macro definition-
  - » MACRO:identify the beginning of a macro definition-MEND:identify the end of a macro definition
  - » * Prototype for the macro
    - – Each parameter begins with '&'

      Name MACRO Parameters

      :

      Body

      :

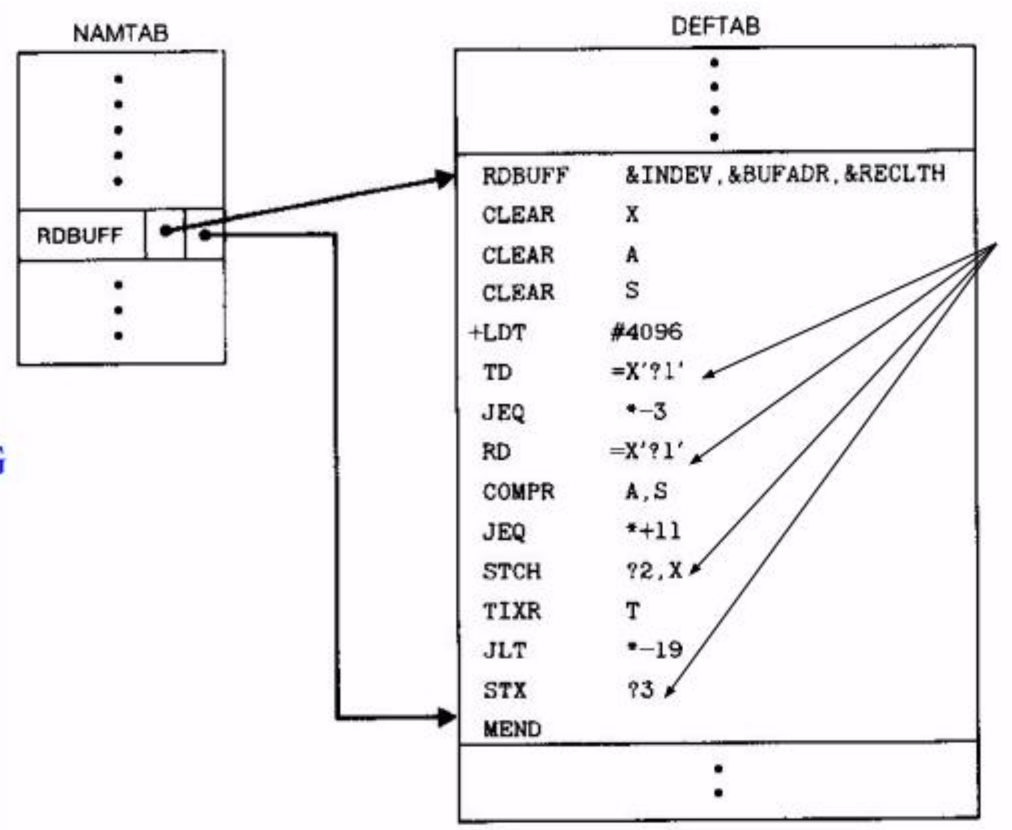
      MEND
    - – Body: the statements that will be generated as the expansion of themacro.

- Each macro invocation statement will be expanded into the statements that form the body of the macro.
- Arguments from the macro invocation are substituted for the parameters in the macro prototype (according to their positions).
  - » In the definition of macro: parameter
  - » In the macro invocation: argument

Comment lines within the macro body will be deleted.

- Macro invocation statement itself has been included as a comment line.
- * The label on the macro invocation statement has been retained as a label on the first statement generated in the macro expansion.
- - We can use a macro instruction in exactly the same way asan assembler language mnemonic.

- DEFTAB
- NAMTAB
- ARGTAB



EXPANDING

Macro Definition

- → Copy Code
- → Parameter Substitution — Positional Substitution / keyword Argument
- → Conditional Macro Expansion

```
begin (Macro Processor)
  EXPANDING != FALSE
  While OPCODE ≠ 'END' do
    begin
      GETLINE
      PROCESSLINE
    end (While)
end (macro processor)

procedure PROCESSLINE
  begin
    Search NAMTAB for OPCODE
    if found then
        EXPAND
    else if OPCODE = 'MACRO' then
        DEFINE
    else write Source line to expand file
  end (PROCEDURE)

procedure DEFINE
  begin
    enter macro name into NAMTAB
    enter macro prototype into DETAB
    LEVEL != 1
    While LEVEL > 0 do
```

```
begin
  GETLINE
  if this is not a Command then
begin
  Subsitute positional notation for parameters
  enter line into DEFTAB
  if OPCODE = 'MACRO' then
     LEVEL := LEVEL + 1
  else if OPCODE = 'MEND' then
     LEVEL := LEVEL - 1
  end (While)
  Store in NAMTAB pointers to beginning &
  end of definition
  end {DEFINE}
procedure EXPAND
  begin
   EXPANDIND := TRUE
  get first line of macro definition {protype}
  from DEFTAB
  Set up arguments from macro invocation in
  AGRGTAB
  Write macro invocation to expanded file as
  a Comment
  While not end of macro definition do
    begin
      GETLINE
      PROCESSLINE
    end {While}
```

```
EXPANDING := FALSE
end {EXPAND}

procedure GETLINE
  begin
    if EXPANDING then
    begin
      get next line of macro definition from DEFTAB
      substitute arguments from ARGTAB for
      positional notation
      end {if}
    else
      read next line from input file
    end {GETLINE}
```

4. Demonstrate the algorithm for simple bootstrap loader.

A Simple Bootstrap Loader

- Bootstrap Loader
  - » **When a computer is first tuned 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**
- Example (SIC bootstrap loader)
  - » **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**

```
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
```

0~9 : 30~39
A~F : 41~46

```
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
```

Break...

5. Explain the different file allocation methods with neat diagram. Mention the advantages and disadvantages.

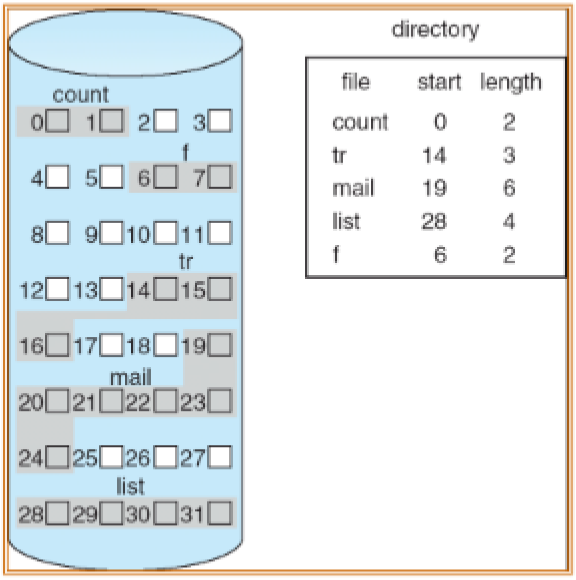
Allocation Methods

- An allocation method refers to how disk blocks are allocated for files:
- **Contiguous allocation**
- **Linked allocation**
- **Indexed allocation**

Contiguous Allocation

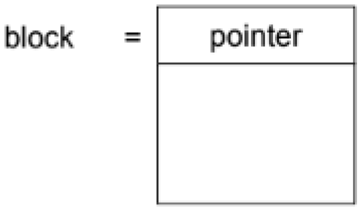
- Each file occupies a set of contiguous blocks on the disk
- Simple – only starting location (block #) and length (number of blocks) are required
- Random access
- Wasteful of space (dynamic storage-allocation problem)
- Files cannot grow

Contiguous Allocation of Disk Space



Linked Allocation

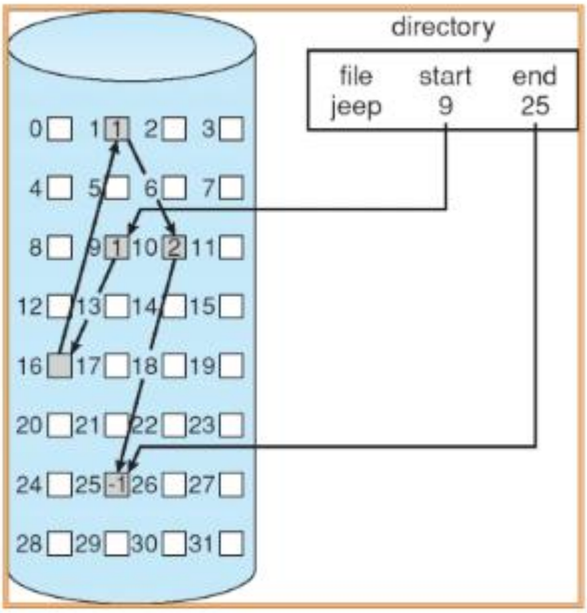
- Each file is a linked list of disk blocks: blocks may be scattered anywhere on the disk.



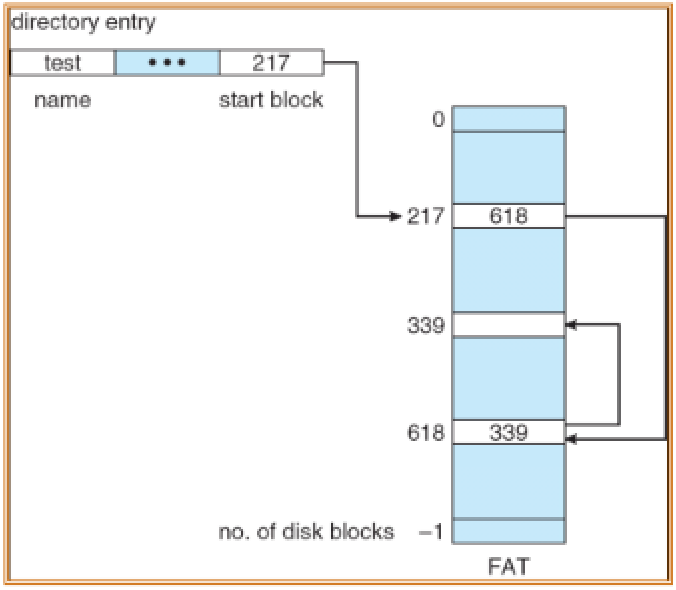
Allocation Methods - Linked

- **Linked allocation** – each file a linked list of blocks
  - File ends at nil pointer
  - No external fragmentation
  - Each block contains pointer to next block
  - No compaction, external fragmentation
  - Free space management system called when new block needed
  - Improve efficiency by clustering blocks into groups but increases internal fragmentation
  - Reliability can be a problem
  - Locating a block can take many I/Os and disk seeks

Linked Allocation



File-Allocation Table



Indexed Allocation

•Linked allocation cannot support efficient direct access, since the pointers to the blocks are scattered with the blocks themselves all over the disk and must be retrieved in order.

•**Indexed allocation** solves this problem by bringing all the pointers together into one location: the **index block**.

•**Each file has its own index block, which is an array of disk-block addresses. The *ith* entry in the index block points to the *ith* block of the file.**

Indexed Allocation

The directory contains the address of the index block

.To find and read the *ith*block, we use the pointer in the *ith* index-block entry.

•**When the file is created, all pointers in the index block are set to null.**

- **When the *ith* block is first written, a block is obtained from the free-space manager, and its address is put in the *i*th index-block entry.**

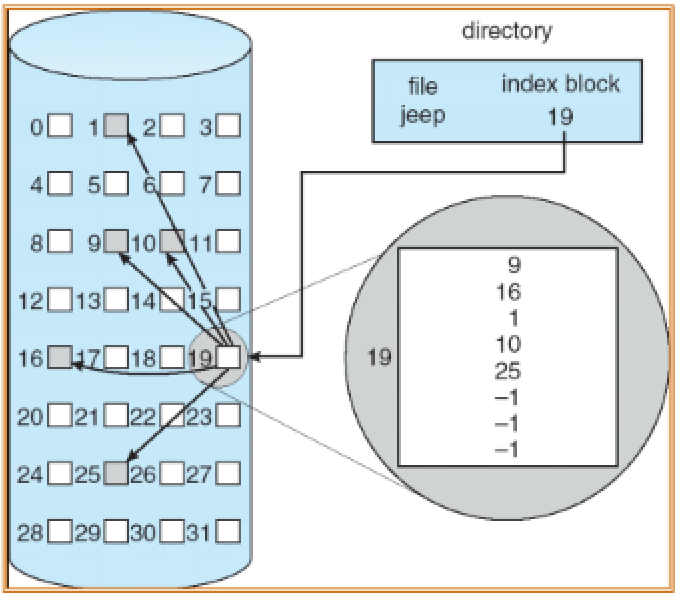
- **If the index block is too small**, however, it will not be able to hold enough pointers for a large file,

- **Mechanisms for this purpose include the following:**

• **Linked scheme** - An index block is normally one disk block. Thus, it can be read and written directly by itself. To allow for large files, we can link together several index blocks.

• **Multilevel index –** A variant of linked representation uses a first-level index block to point to a set of second-level index blocks, which in turn point to the file blocks. To access a block, the operating system uses the first-level index to find a second-level index block and then uses that block to find the desired data block.

Example of Indexed Allocation



6. Explain in detail about free space management with neat sketch

Free-Space Management

- To keep track of free disk space, the system maintains a **free-space list**.

- **The free-space list records all free disk blocks**—those not allocated to some file or directory. To create a file, we search the free-space list for the required amount of space and allocate that space 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.**

Free-Space Management- BIT VECTOR

Each block is represented by 1 bit. If the block is free, the bit is 1; if the block is allocated, the bit is 0.

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 therest of the blocks are allocated. The free-space bit map would be 001111001111110001100000011100000...

- **Advantage**
  - relative simplicity and its efficiency in finding the first free block or *n* consecutive free blocks on the disk

- The first non-0 word is scanned for the first 1 bit, which is the location of the first free block.

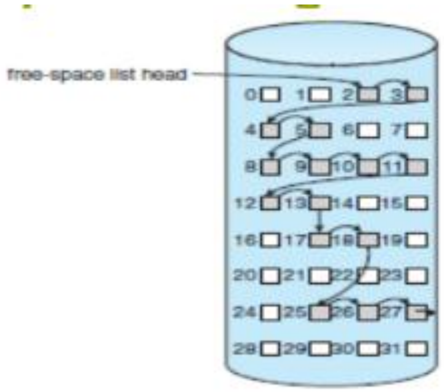
- **The calculation of the block number is (number of bits per word) × (number of 0-value words) + offset of first 1 bit.**

Free-Space Management- LINKED LIST

- **In a Linked List keeping a pointer to the first free block in a special location on the disk and caching it in memory.**

- **This first block contains a pointer to the next free disk block, and so on.**

- 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 17, 18, 25, 26, and 27 were free and the rest of the blocks were allocated.



- In this situation, we would keep a pointer to block 2 as the first free block. Block 2 would contain a pointer to block 3, which would point to block 4, which would point to block 5, which would point to block 8, and so on.

Free-Space Management- GROUPING

- A modification of the free-list approach stores the addresses of *n* free blocks in the first free block.

- **The first *n*−1 of these blocks are actually free.**
- **The last block contains the addresses of another *n* free blocks, and so on.**
- The addresses of a large number of free blocks can now be found quickly, unlike the situation when the standard linked-list approach is used.

Free-Space Management- COUNTING

•Another approach takes advantage of the fact that, generally, several contiguous blocks may be allocated or freed simultaneously, particularly when space is allocated with the contiguous-allocation algorithm or through clustering.

•Thus, rather than keeping a list of *n* free disk addresses, **we can keep the address of the first free block 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.**

7. Outline the life cycle of I/O request in detail with neat sketch

13.5 Transforming I/O Requests to Hardware Operations

- Users request data using file names, which must ultimately be mapped to specific blocks of data from a specific device managed by a specific device driver.
- DOS uses the colon separator to specify a particular device (e.g. C:, LPT:, etc.)
- UNIX uses a ***mount table*** to map filename prefixes (e.g. /usr) to specific mounted devices. Where multiple entries in the mount table match different prefixes of the filename the one that matches the longest prefix is chosen. (e.g. /usr/home instead of /usr where both exist in the mount table and both match the desired file.)
- UNIX uses special ***device files,*** usually located in /dev, to represent and access physical devices directly.
    - Each device file has a major and minor number associated with it, stored and displayed where the file size would normally go.
    - The major number is an index into a table of device drivers, and indicates which device driver handles this device. (E.g. the disk drive handler.)
    - The minor number is a parameter passed to the device driver, and indicates which specific device is to be accessed, out of the many which may be handled by a particular device driver. (e.g. a particular disk drive or partition.)
- A series of lookup tables and mappings makes the access of different devices flexible, and somewhat transparent to users.
- Figure 13.13 illustrates the steps taken to process a (blocking) read request:

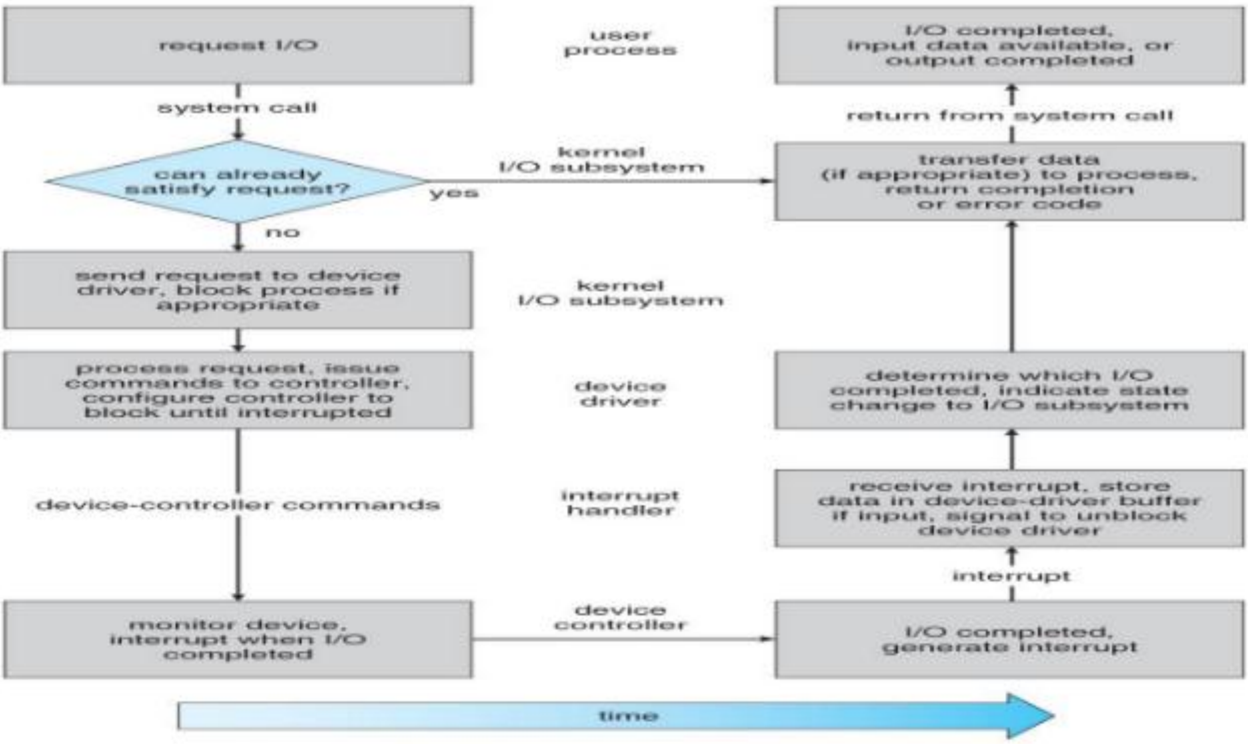


Figure 13.13 - The life cycle of an I/O request.

8. Explain the components of kernel I/O Subsystem in detail.

13.4 Kernel I/O Subsystem

13.4.1 I/O Scheduling

- Scheduling I/O requests can greatly improve overall efficiency. Priorities can also play a part in request scheduling.
- The classic example is the scheduling of disk accesses, as discussed in detail in chapter 12.
- Buffering and caching can also help, and can allow for more flexible scheduling options.
- On systems with many devices, separate request queues are often kept for each device:

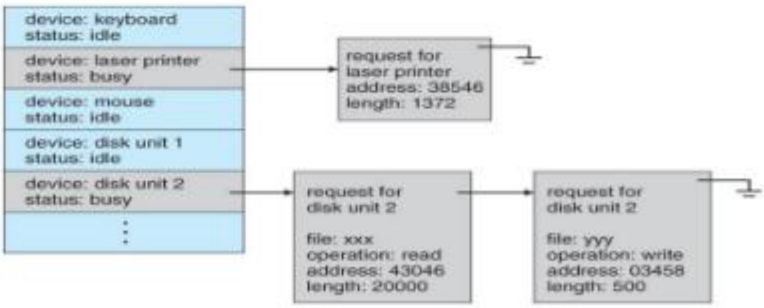


Figure 13.9 - Device-status table.

13.4.2 Buffering

- **Buffering of I/O is performed for (at least) 3 major reasons:**
    1. Speed differences between two devices. (See Figure 13.10 below.) A slow device may write data into a buffer, and when the buffer is full, the entire buffer is sent to the fast device all at once. So that the slow device still has somewhere to write while this is going on, a second buffer is used, and the two buffers alternate as each becomes full. This is known as ***double buffering.*** (Double buffering is often used in (animated) graphics, so that one screen image can be generated in a buffer while the other (completed) buffer is displayed on the screen. This prevents the user from ever seeing any half-finished screen images.)
    2. Data transfer size differences. Buffers are used in particular in networking systems to break messages up into smaller packets for transfer, and then for re-assembly at the receiving side.
    3. To support ***copy semantics.*** For example, when an application makes a request for a disk write, the data is copied from the user's memory area into a kernel buffer. Now the application can change their copy of the data, but the data which eventually gets written out to disk is the version of the data at the time the write request was made.

13.4.3 Caching

- Caching involves keeping a ***copy*** of data in a faster-access location than where the data is normally stored.
- Buffering and caching are very similar, except that a buffer may hold the only copy of a given data item, whereas a cache is just a duplicate copy of some other data stored elsewhere.
- Buffering and caching go hand-in-hand, and often the same storage space may be used for both purposes. For example, after a buffer is written to disk, then the copy in memory can be used as a cached copy, (until that buffer is needed for other purposes.)

13.4.4 Spooling and Device Reservation

- A ***spool (Simultaneous Peripheral Operations On-Line)*** buffers data for (peripheral) devices such as printers that cannot support interleaved data streams.
- If multiple processes want to print at the same time, they each send their print data to files stored in the spool directory. When each file is closed, then the application sees that print job as complete, and the print scheduler sends each file to the appropriate printer one at a time.
- Support is provided for viewing the spool queues, removing jobs from the queues, moving jobs from one queue to another queue, and in some cases changing the priorities of jobs in the queues.
- Spool queues can be general (any laser printer) or specific (printer number 42.)
- OSes can also provide support for processes to request / get exclusive access to a particular device, and/or to wait until a device becomes available.

13.4.5 Error Handling

- I/O requests can fail for many reasons, either transient (buffers overflow) or permanent (disk crash).
- I/O requests usually return an error bit (or more) indicating the problem. UNIX systems also set the global variable ***errno*** to one of a hundred or so well-defined values to indicate the specific error that has occurred. (See errno.h for a complete listing, or man errno.)
- Some devices, such as SCSI devices, are capable of providing much more detailed information about errors, and even keep an on-board error log that can be requested by the host.

13.4.6 I/O Protection

- The I/O system must protect against either accidental or deliberate erroneous I/O.
- User applications are not allowed to perform I/O in user mode - All I/O requests are handled through system calls that must be performed in kernel mode.
- Memory mapped areas and I/O ports must be protected by the memory management system, **but** access to these areas cannot be totally denied to user programs. (Video games and some other applications need to be able to write directly to video memory for optimal performance for example.) Instead the memory protection system restricts access so that only one process at a time can access particular parts of memory, such as the portion of the screen memory corresponding to a particular window.

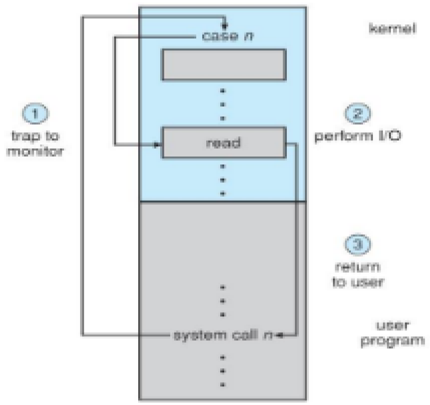


Figure 13.11 - Use of a system call to perform I/O.

13.4.7 Kernel Data Structures

- The kernel maintains a number of important data structures pertaining to the I/O system, such as the open file table.
- These structures are object-oriented, and flexible to allow access to a wide variety of I/O devices through a common interface. (See Figure 13.12 below.)
- Windows NT carries the object-orientation one step further, implementing I/O as a message-passing system from the source through various intermediaries to the device.

PART-C

1.Experiment with the following macro program using conditional statements and expand the macro call with appropriate statements

```
25   RDBUFF   MACRO    &INDEV,&BUFADR,&RECLTH,&EOR,&MAXLTH

26                     IF       (&EOR NE '')
27   &EORCK            SET      1
28                     ENDIF
30                     CLEAR    X
35                     CLEAR    A
38                     IF       (&EORCK EQ 1)
40                     LDCH     =X'&EOR'
42                     RMO      A,S
43                     ENDIF
44                     IF       (&MAXLTH EQ '')
45                    +LDT      #4096
46                     ELSE
47                    +LDT      #&MAXLTH
48                     ENDIF
50   $LOOP             TD       =X'&INDEV'
55                     JEQ      $LOOP
60                     RD       =X'&INDEV'
63                     IF       (&EORCK EQ 1)
65                     COMPR    A,S
70                     JEQ      $EXIT
73                     ENDIF
75                     STCH     &BUFADR,X
80                     TIXR     T
85                     JLT      $LOOP
90   $EXIT             STX      &RECLTH
95                     MEND
```

Macro call

```
RDBUFF   F3,BUF,RECL,04,2048
```

Answer:

```
RDBUFF   F3,BUF,RECL,04,2048


30             CLEAR    X
35             CLEAR    A
40             LDCH     =X'04'
42             RMO      A,S
47            +LDT      #2048
50   $AALOOP   TD       =X'F3'
55             JEQ      $AALOOP
60             RD       =X'F3'
65             COMPR    A,S
70             JEQ      $AAEXIT
75             STCH     BUF,X
80             TIXR     T
85             JLT      $AALOOP
90   $AAEXIT   STX      RECL
```

2. Develop a macro to calculate addition of two numbers and then write a macro to add n numbers.

Addition of two numbers:

```
Sub AddTwoNumbers()
    Dim Num1 As Double
    Dim Num2 As Double
    Dim Result As Double

    Num1 = InputBox("Enter first number:")
    Num2 = InputBox("Enter second number:")

    Result = Num1 + Num2

    MsgBox "The result of " & Num1 & " + " & Num2 & " = " & Result
End Sub
```

Add n numbers:

```
Sub AddNNumbers()
    Dim Count As Integer
    Dim i As Integer
    Dim Num As Double
    Dim Result As Double

    Count = InputBox("How many numbers do you want to add?")

    For i = 1 To Count
        Num = InputBox("Enter number " & i & ":")
        Result = Result + Num
    Next i

    MsgBox "The sum of the " & Count & " numbers entered is " & Result
End Sub
```

3. Experiment with the nested Macro calls and Macro definition with suitable example

Nested Macro calls:

Macro bodies may also contain macro calls, and so may the bodies of those called macros, and so forth. If a macro call is seen throughout the expansion of a macro, the assembler starts immediately with the expansion of the called macro. For this, its its expanded body lines are simply inserted into the expanded macro body of the calling macro, until the called macro is completely expanded. Then the expansion of thecalling macro is continued with the body line following the nested macro call.

Nested macro calls

```
RDBUFF   MACRO    &BUFADR,&RECLTH,&INDEV
.
.        MACRO TO READ RECORD INTO BUFFER
.
         CLEAR    X                CLEAR LOOP COUNTER
         CLEAR    A
         CLEAR    S
        +LDT      #4096            SET MAXIMUM RECORD LENGTH
$LOOP    RDCHAR   &INDEV           READ CHARACTER INTO REG A
         COMPR    A,S              TEST FOR END OF RECORD
         JEQ      $EXIT            EXIT LOOP IF EOR
         STCH     &BUFADR,X        STORE CHARACTER IN BUFFER
         TIXR     T                LOOP UNLESS MAXIMUM LENGTH
         JLT      $LOOP              HAS BEEN REACHED
$EXIT    STX      &RECLTH          SAVE RECORD LENGTH
         MEND
```

(a)

```
RDCHAR   MACRO    &IN
.
.        MACRO TO READ CHARACTER INTO REGISTER A
.
         TD       =X'&IN'          TEST INPUT DEVICE
         JEQ      *-3              LOOP UNTIL READY
         RD       =X'&IN'          READ CHARACTER
         MEND
```

(b)

```
         RDBUFF   BUFFER,LENGTH,F1
```

(c)

Figure 4.11 Example of nested macro invocation.

Nested Macro definition:

Nested macro definition

```
1   MACROS   MACRO    {Defines SIC standard version macros}
2   RDBUFF   MACRO    &INDEV,&BUFADR,&RECLTH
                 .
                 .    {SIC  standard version}
                 .
3            MEND     {End of RDBUFF}
4   WRBUFF   MACRO    &OUTDEV,&BUFADR,&RECLTH
                 .
                 .    {SIC standard version}
                 .
5            MEND     {End of WRBUFF}
                 .
                 .
                 .
6            MEND     {End of MACROS}
```

(a)

```
1   MACROX   MACRO    {Defines  SIC/XE macros}
2   RDBUFF   MACRO    &INDEV,&BUFADR,&RECLTH
                 .
                 .    {SIC/XE version}
                 .
3            MEND     {End of RDBUFF}
4   WRBUFF   MACRO    &OUTDEV,&BUFADR,&RECLTH
                 .
                 .    {SIC/XE version}
                 .
5            MEND     {End of WRBUFF}
                 .
                 .
                 .
6            MEND     {End of MACROX}
```

(b)

Figure 4.3 Example of the definition of macros within a macro body.

4. Construct the Absolute loader with suitable procedure.

Absolute loader:

An absolute loader is the simplest of loaders. Its function is simply to take the output of the assembler and load it into memory. The output of the assembler can be stored on any machine-readable form of storage, but most commonly it is stored on punched cards or magnetic tape, disk, or drum.

Fig. 3.2 Algorithm for an absolute loader

```
Begin
read Header record
verify program name and length
read first Text record
while record type is not '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
```

Memory address	Contents				
0000	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	
0010	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	
⋮	⋮	⋮	⋮	⋮	
0FF0	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	
1000	14103348	20390010	36281030	30101548	
1010	20613C10	0300102A	0C103900	102D0C10	
1020	36482061	0810334C	0000454F	46000003	
1030	000000xx	xxxxxxxx	xxxxxxxx	xxxxxxxx	← COPY
⋮	⋮	⋮	⋮	⋮	
2030	xxxxxxxx	xxxxxxxx	xx041030	001030E0	
2040	205D3020	3FD8205D	28103030	20575490	
2050	392C205E	38203F10	10364C00	00F10010	
2060	00041030	E0207930	20645090	39DC2079	
2070	2C103638	20644C00	0005xxxx	xxxxxxxx	
2080	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	
⋮	⋮	⋮	⋮	⋮	

(b) Program loaded in memory