Q.2 a. List the major activities of an operating system in regard to proces management and memory management.

Answer:

Major activities of an operating system in respect to Process management are as follows:

- The creation and deletion of both user and system processes.
- The suspension and resumption of processes.
- The provision of mechanisms for process synchronization.
- The provision of mechanisms for process communication.
- The provision of mechanisms for deadlock handling.

Major activities of an operating system in respect to Memory management are as follows:

- Keep track of which parts of memory are currently being used and by whom.
- Decide which processes are to be loaded into memory when memory space becomes available.
- Allocate and deallocate memory space as needed.

b. Distinguish between multiprogramming and multiprocessing systems.

Answer:

A **multiprogramming** operating system is system that allows more than one active user program (or part of user program) to be stored in main memory simultaneously. Multi programmed operating systems are fairly sophisticated. All the jobs that enter the system are kept in the job pool. This pool consists of all processes residing on mass storage awaiting allocation of main memory. If several jobs are ready to be brought into memory, and there is not enough room for all of them, then the system must choose among them. A time-sharing system is a multiprogramming system.

A **multiprocessing** system is a computer hardware configuration that includes more than one independent processing unit. The term multiprocessing is generally used to refer to large computer hardware complexes found in major scientific or commercial applications. The multiprocessor system is characterized by-increased system throughput and application speedup-parallel processing. The main feature of this architecture is to provide high speed at low cost in comparison to uni processor.

c. What is a process? Discuss briefly, the different process states.

Answer:

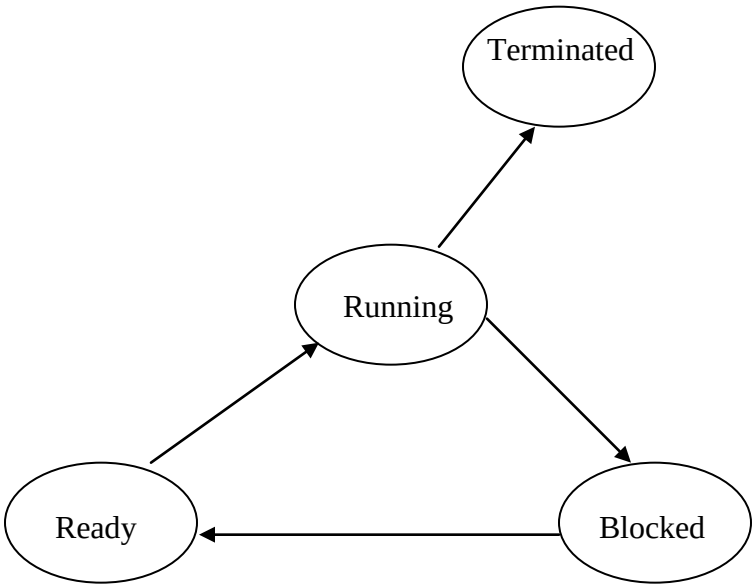
A process or task is a portion of a program in some stage of execution. A program can consist of several processes, each working on their own or as a unit, perhaps communicating with each other. Each process that runs in an operating system is assigned a process control block that holds information about the process, such as a unique process ID (a number used to identify the process), the saved state of the process, the process priority and where it is located in memory.

A process in a computer system may be in one of the possible state as follows:

- Running: A CPU is currently allocated to the process and the process is in execution.

- Blocked: The process is waiting for a request to be satisfied, or an event to occu
  Such a process cannot execute even if a CPU is available.
- Ready: The process is not running, however it can execute if a CPU is allocated to it, means the process is not blocked.
- Terminated: The process has finished its execution.

A state transition is caused by the occurrence of some event in the system. When a process in the running state makes an I/O request, it has to enter blocked state awaiting completion of the I/O. When the I/O completes, the process state changes from blocked to ready. Similar state changes occur when a process makes some request, which cannot be satisfied by OS straightway. The process state changes to blocked until the request is satisfied, when its state changes to ready once again. A ready process becomes running when the CPU allocated to it. The fundamental state transition is shown in figure below.



d. What is cooperating process? Give reasons for providing an environment that allows process cooperation.

Answer:

Processes executing concurrently in the operating system may be either independent process or cooperating processes. A process is independent if it cannot affect or affected by the other processes executing in the system. A process is cooperating if it can affect or affected by the other processes executing in the system. Thus, any process that shares data with other processes is a cooperating process. Cooperating processes require an Interprocess communication mechanism that will allow them to exchange data and information.

Several reasons for providing an environment that allows process cooperation are:

- **Information sharing:** Since several users may be interested in the same piece of information (for instance, a shared file), we must provide an environment to allow concurrent access to such information.
- **Computation speedup:** If we want a particular task to run faster, we must break it into subtasks, each of which will be executing in parallel with the others. Such a speedup can be achieved only if the computer has multiple processing elements (such as CPU's or I/O channels).
- **Modularity:** We may want to construct the system in modular fashion, dividing the system functions into separate processes or threads.
- **Convenience:** Even an individual user may work on many tasks at the same time. For instance, a user may be editing, printing and compiling in parallel.

Q.3 a. State different scheduling criteria that must be kept in mind while choosing different scheduling algorithms.

Answer:

The scheduling criteria that must be kept in mind while choosing different scheduling algorithms include the following:

- **CPU utilization**: We want to keep the CPU as busy as possible. CPU utilization may range from 0 to 100 percent. In a real system, it should range from 40 percent (for a lightly loaded system) to 90 percent (for a heavily used system).
- **Throughput**: If the CPU is busy executing processes, then work is being done. One measure of work is the number of processes completed per time unit, called throughput. For long processes, this rate may be 1 process per hour; for short transactions, throughput might be 10 processes per second.
- **Turnaround time**: From the point of view of a particular process, the important criterion is how long it takes to execute that process. The interval from the time of submission of a process to the time of completion is the turnaround time. Turnaround time is the sum of the periods spent waiting to get into memory, waiting in the ready queue, executing on the CPU, and doing I/O.
- **Waiting time**: The CPU-scheduling algorithm does not affect the amount of time during which a process executes or does I/O; it affects only the amount of time that a process spends waiting in the ready queue. Waiting time is the sum of the periods spent waiting in the ready queue.
- **Response time**: In an interactive system, turnaround time may not be the best criterion. Often, a process can produce some output fairly early, and can continue computing new results while previous results are being output to the user. Thus, another measure is the time from the submission of a request until the first response is produced. This measure, called response time, is the amount of time it takes to start responding, but not the time that it takes to output that response. The turnaround time is generally limited by the speed of the output device.

We want to maximize CPU utilization and throughput, and to minimize turnaround time, waiting time, and response time. In most cases, we optimize the average measure. However, in some circumstances we want to optimize the minimum or maximum values, rather than the average. For example, to guarantee that all users get good service, we may want to minimize the maximum response time.

For interactive systems (such as time-sharing systems), minimizing the variance in the response time is more important than minimizing the average response time. A system with reasonable and predictable response time may be considered more desirable than a system that is faster on the average, but is highly variable.

b. List the different actions taken by time sharing scheduler.

Answer:

Actions of the time sharing scheduler can be summarized as follows:

- The scheduler maintains two separate PCB lists – one for ready processes and other for blocked and swapped-out process.
- The PCB list for ready processes is organized as a queue.
- The PCB of a newly created process can be added to the end of the ready queue.
- The PCB of a terminating process can be simply removed from the system.
- When a running process finishes its time slice, or makes an IO request, its PCB is moved from the ready queue to the blocked / swapped-out list.
- When the IO operation awaited by a process finishes, its PCB is moved from the blocked / swapped-out list to the end of the ready queue.

c. Define Deadlock. Discuss the four necessary conditions for deadlocks to occur.

Answer:

Deadlock is a situation, in which processes never finish executing and system resources are tied up, preventing other jobs form starting. A process requests resources; if the resources are not available at that time, the process enters a wait state. Waiting processes may never again change state, because other waiting processes, thereby causing deadlock, hold the resources they have requested.

The four necessary conditions for deadlocks to occur are mutual exclusion, hold and wait, no preemption and circular wait. If any one of the above four conditions does not hold, then deadlocks will not occur. Thus **prevention of deadlock** is possible by ensuring that at least one of the four conditions cannot hold.

- **Mutual exclusion**: Resources that can be shared are never involved in a deadlock because such resources can always be granted simultaneously access by processes. Hence processes requesting for such a sharable resource will never have to wait. Examples of such resources include read-only files. Mutual exclusion must therefore hold for non-sharable resources. But it is not always possible to prevent deadlocks by denying mutual exclusion condition because some resources are by nature non-sharable, for example printers.
- **Hold and wait**: To avoid hold and wait, the system must ensure that a process that requests for a resource does not hold on to another. There can be two approaches to this scheme:
    - a process requests for and gets allocated all the resources it uses before execution begins.
    - a process can request for a resource only when it does not hold on to any other.

Algorithms based on these approaches have poor resource utilization. This is because resources get locked with processes much earlier than they are actually used and hence not available for others to use as in the first approach. The second approach seems to applicable only when there is assurance about reusability of data and code on the released resources. The algorithms also suffer from starvation since popular resources may never be freely available.

- **No preemption**: This condition states that resources allocated to processes cannot be preempted. To ensure that this condition does not hold, resources could be preempted. When a process requests for a resource, it is allocated the resource if it is available. If it is not, than a check is made to see if the process holding the wanted resource is also waiting for additional resources. If so the wanted resource is preempted from the waiting process and allotted to the requesting process. If both the above is not true that is the resource is neither available nor held by a waiting process, then the requesting process waits. During its waiting period, some of its resources could also be preempted in which case the process will be restarted only when all the new and the preempted resources are allocated to it.
  Another alternative approach could be as follows: If a process requests for a resource which is not available immediately, then all other resources it currently holds are preempted. The process restarts only when the new and the preempted resources are allocated to it as in the previous case.
  Resources can be preempted only if their current status can be saved so that processes could be restarted later by restoring the previous states. Example, CPU memory and main memory. But resources such as printers cannot be preempted, as their states cannot be saved for restoration later.
- **Circular wait**: Resource types need to be ordered and processes requesting for resources will do so in increasing order of enumeration. Each resource type is mapped to a unique integer that allows resources to be compared and to find out the precedence order for the resources. Thus F: R → N is a 1:1 function that maps resources to numbers. For example:
  F (tape drive) = 1, F (disk drive) = 5, F (printer) = 10.
  To ensure that deadlocks do not occur, each process can request for resources only in increasing order of these numbers. A process to start with in the very first instance can request for any resource say R_i. There after it can request for a resource R_j if and only if $F(R_j)$ is greater than $F(R_i)$. Alternately, if $F(R_j)$ is less than $F(R_i)$, then R_j can be allocated to the process if and only if the process releases R_i.
  The mapping function F should be so defined that resources get numbers in the usual order of usage.

Q.4 a. Define critical regions. Give a solution for reader-writers problem using conditional critical regions.

Answer:

A semaphore is a shared integer variable with non negative values which can only be subjected to the following operations:

1. Initialization
2. Indivisible operations P and V

Indivisibility of P and V operations implies that these operations cannot be executed concurrently. This avoids race conditions on the semaphore. Semantics of P and V operations are as follows:

```
P(S)      :    if S > 0
                    then S := S – 1
                    else block the process
                      executing the P operation;

V(S)      :    if there exist process(es) blocked
                    on S
               then wake one blocked process;
               else S := S + !;
```

Readers-writers problem: Let a data object (such as a file or record) is to be shared among several concurrent processes. Readers are the processes that are interested in only reading the content of shared data object. Writers are the processes that may want to update (that is, to read and write) the shared data object. If two readers access the shared data object simultaneously, no adverse effects will result. However if a writer and some other process (either a reader or writer) access the shared object simultaneously, anomaly may arise. To ensure that these difficulties do not arise, writers are required to have exclusive access to the shared object. This synchronization problem is referred to as the readers-writers problem.

Solution for readers-writers problem using conditional critical regions.

Conditional critical region is a high-level synchronization construct. We assume that a process consists of some local data, and a sequential program that can operate on the data. The local data can be accessed by only the sequential program that is encapsulated within same process. One process cannot directly access the local data of another process. Processes can, however, share global data.

Conditional critical region synchronization construct requires that a variable v of type T, which is to be shared among many processes, be declared as

v: shared T;

The variable v can be accessed only inside a region statement of the following form:

region v when B do S;

This construct means that, while statement S is being executed, no other process can access the variable v. When a process tries to enter the critical-section region, the Boolean expression B is evaluated. If the expression is true, statement S is executed. If it is false, the process releases the mutual exclusion and is delayed until B becomes true and no other process is in the region associated with v.

Now, let A is the shared data object. Let readcount is the variable that keeps track of how many processes are currently reading the object A. Let writecount is the variable that keeps track of how many processes are currently writing the object A. Only one writer can update object A, at a given time.

Variables readcount and writecount are initialized to 0. A writer can update the shared object A when no reader is reading the object A.

```
region A when( readcount = = 0 AND writecount = = 0){
                    ……
          writing is performed
              ……    }
```

A reader can read the shared object A unless a writer has obtained permission to update the object A.

```
region A when(readcount >=0 AND writecount = = 0){
              ……
       reading is performed  …… }
```

b. With the help of examples, differentiate between the following:
(i) absolute and relative access paths
(ii) linked and indexed allocation of disk space

Answer: Page Number 566, 570 of Text Book

Q.5 a. Discuss the two approaches used to identify and reuse free memory areas in a heap.

Answer:

The two popular approaches used to identify and reuse free memory areas in a heap are as follows:

- **Use of reference counts:** In the reference count technique, a reference count is associated with each memory area to indicate the number of its active users. The number is incremented when a new user gains access to the memory area and is decremented when a user finishes using it. The memory area is known to be free when its reference count drops to zero. The reference count method can be implemented by providing two library routines allocate and free to implement memory allocation and deallocation, respectively, and a free list to keep track of free areas in memory. The allocate routine performs the following actions: If a request for new memory is received it searches the free list and finds a free memory area of appropriate size to satisfy the request. It then deletes this area from the free list, allocates it to the calling program, sets its reference count to 1 and returns its address to the calling program. If the requested area has already been allocated it simply increments its reference count by 1 and returns its address to the calling program. The free routine decrements the reference counts by 1. If the count becomes 0, it enters the area in the free list. The reference count technique is simple to implement and incurs incremental overheads, i.e. overheads at every allocation and deallocation.

- **Garbage collection:** The garbage collection approach performs reuse of memory differently. It reclaims memory returned by programs only when it runs out of free memory to allocate. The garbage collection algorithm makes two passes over the memory to identify unused areas. In the first pass it traverses all pointers pointing to allocated areas and marks the memory areas, which are in use. In the second pass it finds all unmarked areas and declares them to be free. It can now enter these areas in a free list. In this approach the allocate and free routines do not perform any actions aimed at reuse of memory. Hence the garbage collection overheads are not incremental. Garbage collection overheads are incurred every time the system runs out of free memory to allocate to fresh requests. The overheads become very heavy when most of the available memory is allocated to programs because the garbage collector has to do more work in its mark and free passes and needs to run more often.

b. Describe the First fit, Best fit and Worst fit allocation algorithms. Given memory partitions of 100K, 500K, 200K, 300K, and 600K (in order), how would each of the First-fit, Best-fit, and Worst-fit algorithms place processes of 212K, 417K, 112K and 426K (in order)? Which algorithm makes the most efficient use of memory?

Answer:

The first-fit, best-fit, and worst-fit strategies are the most common ones used to select a free hole from the set of available holes.

First fit: Allocate the first hole that is big enough. Searching can start either at the beginning of the set of holes or where the previous first-fit search ended. We can stop searching as soon as we find a free hole that is large enough.

Best fit: Allocate the smallest hole that is big enough. We must search the entire list, unless the list is kept ordered by size. This strategy produces the smallest leftover hole.

Worst fit: Allocate the largest hole. Again, we must search the entire list, unless it is sorted by size. This strategy produces the largest leftover hole, which may be more useful than the smaller leftover hole from a best-fit approach.

First-fit:

- 212K is put in 500K partition
- 417K is put in 600K partition
- 112K is put in 288K partition (new partition 288K = 500K - 212K)
- 426K must wait

Best-fit:

- 212K is put in 300K partition
- 417K is put in 500K partition
- 112K is put in 200K partition
- 426K is put in 600K partition

Worst-fit:

- 212K is put in 600K partition

- 417K is put in 500K partition
- 112K is put in 388K partition
- 426K must wait

Best-fit algorithm turns out to be the best.

Q.6 a. Explain the different fundamental language processing activities.

Answer:

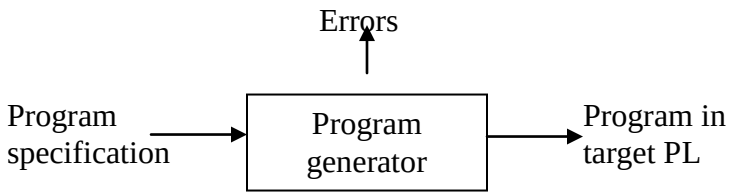
The fundamental languages processing activities can be divided into those that bridge the specification gap and those that bridge the execution gap. These activities are

- Program generation activities
- Program execution activities

Program Generation

A program generation activity aims at automatic generation of a program. The source language is a specification language of an application domain and the target language is a procedure oriented programming language. The following figure depicts the program generation activity. The program generator is a software system which accepts the specification of a program to be generated, and generates a program in a target programming language. Thus, the program generator introduces a new domain between the application and programming language domains known as program generator domain. This specification gap is between the application domain and the program generator domain, which is smaller than the gap between the application domain and the target programming language domain.

This reduction in the specification gap increases the reliability of the generated program. Since the generator domain is close to the application domain, it is easy for the designer or programmer to write the specification of the program to be generated. The harder task of bridging the gap to the programming language domain is performed the generator.



Program Execution

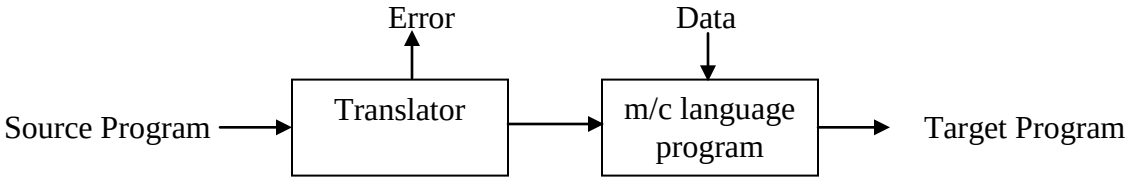
Two popular models for program execution are translation and interpretation.

Program Translation: The program translation model bridges the execution gap by translating a program written in a programming language, called the source program, into an equivalent program in the machine language of the computer system, called the target program. Some characteristics of the program translation model are:

- A program must be translated before it can be executed.

- The translated program may be saved in a file. The saved program may b executed repeatedly.

The program translation model can be as follows:



Program interpretation: The interpreter reads the source program and stores it in its memory. During interpretation it takes a source statement, determines its meaning and performs actions which implement it. This includes computational and inpu-output actions. Some characteristics of the program interpretation model are:

- The source program is retained in the source form itself, i.e. no target program forms exists.
- A statement is analyzed during its interpretation.

b. What properties should a hash function possess to ensure good search performance? Discuss two collision handling techniques.

Answer:

A hash function h should possess the following properties to ensure good search performance:

(i) The hashing function should not be sensitive to the symbols in S_p, that is, it should perform equally well for different source programs. Thus, the value of p_s should only depend on k_p.

(ii) The hashing function h should execute reasonably fast.

Two approaches to collision handling are to accommodate a colliding entry elsewhere in the hash table using a rehashing technique, or to accommodate the colliding entry in a separate table using an overflow technique.

- **Rehashing:** Rehashing technique uses a sequence of hashing functions h1, h2… to resolve collisions. Let a collision occur while probing the table entry whose number is provided by $h_i(s)$. We use $h_{i+1}(s)$ to obtain a new entry number. A popular technique called sequential rehashing uses the recurrence relation

  $$h_{i+1}(s) = h_i(s) \bmod n + 1$$

  to provide a series of hashing functions for rehashing.

  A drawback of rehashing technique is that a colliding entry accommodated elsewhere in the table may be contribute to more collisions. This may lead to clustering of entries in the table.
- **Overflow chaining:** Overflow chaining avoids the problems associated with the clustering effect by accommodating colliding entries in a separate table called the overflow table. Thus, a search which encounters a collision in the primary hash table has to be continued in the overflow table. To facilitate this, a pointer field is

added to each entry in the primary and overflow tables. The entry format is as follows:

Symbol	Other info	Pointer

A single hashing function h is used. All symbols which encounter a collision are accommodated in the overflow table. Symbols hashing into a specific entry of the primary table are chained together using the pointer field. On encountering a collision in the primary table, one chain in the overflow table has to be searched.
The main drawback of the overflow chaining method is the extra memory requirement due to the presence of the overflow table. An organization called scatter table organization is often used to reduce the memory requirements. In this organization, the hash table merely contains pointers, and all symbol entries are stored in the overflow table.

Q.7 a. Compare and contrast non-relocatable program, relocatable program and self-relocatable program.

Answer:

A **non-relocatable** program is a program, which cannot be executed in any memory area other than the area starting on its translated origin. Non relocatability is the result of address sensitive of a program and lack of information concerning the address sensitive instructions in the program. The difference between a relocatable program and a non relocatable program is the availability of information concerning the address sensitive instructions in it.

A **relocatable** program can be processed to relocate it to a desired area of memory.

A **self-relocating** program is a program, which can perform the relocation of its own address sensitive instructions. It contains the following two provisions for this purpose:

1. A table of information concerning the address sensitive instructions exists as a part of the program.
2. Code to perform the relocation of address sensitive instructions also exists as a part of the program. This is called the relocating logic.

The start address of the relocating logic is specified as the execution start address of the program. Thus the relocating logic gains control when the program is loaded in memory for execution. It uses the load address and the information concerning address sensitive instructions to perform its own relocation. Execution control is then transferred to the relocated program.

A **self-relocating** program can execute in area of the memory. This is very important in time sharing operating systems where the load address of a program is likely to be different for different executions.

b. Define top down parsing. Discuss the features that are needed to implement top down parsing. Also, give an algorithm for Operator Precedence Parsing.

Answer:

Top Down Parsing: Top down parsing according to a grammar G attempts to derive string matching a source string through a sequence of derivations starting with the distinguished symbol of G. For a valid source string α, a top down parse thus determines a derivation sequence

$$S \Rightarrow \ldots\ldots \Rightarrow \ldots\ldots \Rightarrow \alpha$$

The following features are needed to implement top down parsing:

- **Source string marker (SSM):** Source string marker points to the first unmatched symbol in the source string.
- **Prediction making mechanism:** This mechanism systematically selects the RHS alternatives of a production during prediction making. It must ensure that any string L_G can be derived from S.
- **Matching and backtracking mechanism:** This mechanism matches every terminal symbol generated during a derivation with the source symbol pointed to by source string marker. Backtracking is performed if the match fails. This involves resetting current sentential form (CSF) and source string marker to earlier values.

Algorithm for Operator Precedence Parsing

Data Structures

Stack: Each stack entry is a record with two field, operator and operand_pointer

Node: A node is a record with three fields, symbol, left_pointer, and right_pointer.

Functions

newnode (operator, l_operand_pointer, r_operand_pointer) creates a node with appropriate pointer fields and returns a pointer to the node.

1. TOS := SB – 1; SSM :=0;
2. Push ‘ ├- ’ on the stack.
3. SSM := SSM + 1;
   If current source symbol is an operator, then goto Step 5.
4. x := newnode(source symbol, null, null);
   TOS.operand_pointer := x;
   Go to step 3;
5. while TOS operator > current operator
   x := newnode(TOS operator, TOS.operand_pointer, TOS.operand_pointer);
   Pop an entry off the stack.
   TOS.operand_pointer := x;
6. if TOS < current operator, then
   Push the current operator on the stack.
   Go to step3;

7. if TOS operator = current operator, then
   if TOS operator = ' ├-', then exit successfully.
   If TOS operator = '(', then
   temp := TOS.operand_pointer;
   Pop an entry off the stack.
   TOS.operand_pointer := temp;
   Go to step 3;
8. if no precedence defined between TOS operator and current operator then report error and exit unsuccessfully.

Q.8 a. Discuss the different data structures used during Pass I of the Assembler.

Answer:

The different data structures used during Pass I of the Assembler are as follows:

- OPTAB A table of mnemonic opcodes and related information
- SYMTAB Symbol table
- LITTAB A table of literals used in the program

OPTAB contains the field's *mnemonic opcode*, *class* and *mnemonic info*. The class field indicates whether the opcode corresponds to an imperative statement (IS), a declaration statement (DL) or an assembler directive (AD). If an imperative statement, the mnemonic info field contains the pair (machine opcode, instruction length), else it contains the id of a routine to handle the declaration or directive statement.

OPTAB

Mnemonic opcode	class	Mnemonic info
MOVER	IS	(04,1)
DS	DL	R#7
START	AD	R#11
	:	

A SYMTAB entry contains the field's address and length.

SYMTAB

symbol	address	length
LOOP	202	1
NEXT	214	1
LAST	216	1
A	217	1
BACK	202	1
B	218	1

A LITTAB entry contains the field's literal and address.

LITTAB

literal	address
= '5'	
= '1'	

Literal no
#1
#3
--

= '1'	

POOLTAB

Processing of an assembly begins with the processing of its label field. If it contains a symbol, the symbol and the value in location counter (LC) is copied into a new entry of SYMTAB. Thereafter, the functioning of Pass I centers around the interpretation of the OPTAB entry for the mnemonic. The class field of the entry is examined to determine whether the mnemonic belongs to the class of imperative, declaration or assembler directives statements. In case of imperative statement, the length of the machine instruction is simply added to the LC. The length is also entered in the SYMTAB entry of the symbol (if any) defined in the statement. This completes the processing of the statement.

For a declaration or assembler directive statement, the routine mentioned in the mnemonic info field is called to perform appropriate processing of the statement. This routine processes the operand field of the statement to determine the amount of memory required by this statement and appropriately updates the LC and the SYMTAB entry of the symbol defined in the statement. Similarly, for an assembler directive the called routine would perform appropriate processing, possibly affecting the value in LC.

The first pass uses LITTAB to collect all literals used in program. Different literal pools are maintained with the help of auxiliary table called POOLTAB. This table contains the literal number of the starting literal of each literal pool. At any stage, the current literal pool is the last pool in LITTAB.

b. Discuss the registers set and control transfer instructions of Intel 8088.

Answer:

Registers set

The Intel 8088 microprocessor supports 8 and 16 bit arithmetic, and also provides special instructions for string manipulation. The CPU of 8088 contains the following registers as shown in figure below.

Each data register is 16 bits in size, split into upper and lower halves. Either half can be used for 8 bit arithmetic, while the two halves together constitute the register for 16 bit arithmetic. The architecture supports stacks for storing subroutine and interrupt return addresses, parameters and other data. The index registers SI and DI are used to index the source and destination addresses in string manipulation instructions. They are provided

with the auto-increment and auto-decrement facility. Two stack pointer registers called SP and BP are provided to address the stack. SP points into the stack implicitly used by the architecture to store subroutine and interrupt return addresses. BP can be used by programmer in any desired manner. Push and Pop instructions are provided for this purpose.

The Intel 8088 provides addressing capability for 1 MB of primary memory. The memory is used to store three components of a program, program code, data and stack. The Code, Stack and Data segment registers are used to contain the start addresses of these three components. The Extra segment register points to another memory area which can be used to store data. To address a memory location, an instruction designates a segment register and provides a 16 bit logical address.

AH	AL	AX
BH	BL	BX
CH	CL	CX
DH	DL	DX

BP
SP

SI
DI

Code
Stack
Data
Extra

Data, Base, Index and segment registers

Control transfer instructions

Two groups of control transfer instructions are supported. These are:

- Calls, jumps and returns
- Iteration control instructions

The calls, jumps and returns can occur within the same segment, or can cross segment boundaries. Intra-segment transfers are preferably assembled using a self-relative displacement. The longer form of intra-segment transfer uses a 16 bit logical address within the segment. Inter-segment transfers indicate a new segment base and an offset. Their execution is expensive since the code segment register has to be modified. Control transfer can be both direct and indirect. The instructions formats are shown as below:

Intra-segment

Opcode	Disp. low	Disp. high

Inter-segment

Opcode	Offset	Offset	Segment	Base

Indirect

Opcode	mod 100 r/m	Disp. low	Disp. high

c. Explain forward and cross references.

Answer:

Information concerning forward references to a symbol is organized in the form of a linked list. Thus, the *forward reference table* (FRT) contains a set of linked lists. The FRT pointer field of a SYMTAB entry field points to the head of the list. Since ordering of FRT entries in not important, for efficiency reasons new entries are added at the beginning of the list. Each FRT entry contains SRTAB# to be used to assemble the forward reference. It also contains the instruction address and a usage code indicating where and how the reference is to be assembled. When the definition of a symbol is encountered, its forward references are processed, and the forward references list is discarded. This minimizes the size of FRT at any time.

A cross references directory is a report produced by the assembler which lists all references to a symbol sorted in the ascending order of the statement numbers. The assembler uses the *cross references table* (CRT) to collect the information concerning references to all symbols in the program. Each SYMTAB entry points to the head and tails of a linked list in the CRT. New entries are added at the end of the list.

Being linked lists, FRT and CRT can be organized in a single memory area. The tables grow from the high end of storage to its low end. The freed entries of FRT are reused by maintaining a free list. The target code generated by the assembler grows from the low end to the storage. Thus, no size restrictions need to be placed on individual tables. The assembler fails to handle a source program only if its target code overlaps with its tables.

Q.9 a. Discuss the following:

(i) Local and Global optimization
(ii) Triples and Quadruples
(iii) Call by value and Call by reference
(iv) Pure and Impure interpreter

Answer:

(i) Local and Global optimization

Local Optimization: The optimizing transformations are applied over small segments of a program consisting of a few segments. Local optimization provides limited benefits at a low cost. The scope of local optimization is a basic block which is an 'essentially sequential' segment in the source program. The cost of local optimization is low because the sequential nature of the basic block simplifies the analysis needed for optimization. The benefits are limited because certain optimizations are beyond the scope of local optimization

Global Optimization: The optimizing transformations are applied over a program unit, i.e. over a function or a procedure. Compared to local optimization, global optimization requires more analysis effort to establish the feasibility of an

optimization. Consider global common subexpression elimination. If some expressio $x * y$ occurs in a set of basic blocks SB of program P, its occurrence in a block $b_j \in$ SB can be eliminated if the following two conditions are satisfied for every execution of P:

- Basic block b_j is executed only after some block $b_k \in$ SB has been executed one or more times.
- No assignments to x or y have been executed after the last evaluation of $x * y$ in block b_k.

(ii) Triples and Quadruples

A **triple** is a representation of an elementary operation in the form of a pseudo machine instruction. Each operand of a triple is either a variable or constant or the result of some evaluation represented by another triple.

Operator	Operand1	Operand2

A **quadruple** represents an elementary evaluation in the following format:

Operator	Operand 1	Operand 2	Result name

Here, result name designates the result of the evaluation it can be used as the operand of another quadruple. This is more convenient than using a number to designate a subexpression.

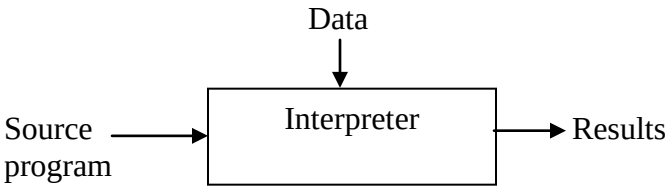
(iii)Call by value and Call by reference

In **Call by value** mechanism, the values of actual parameters are passed to the called function. These values are assigned to the corresponding formal parameters. If a function changes the value of a formal parameter, the change is not reflected on the corresponding actual parameter. This is commonly used for built-in functions of the language. Its main advantage is its simplicity. The compiler can treat formal parameter as a local variable. This simplifies compilation considerably.

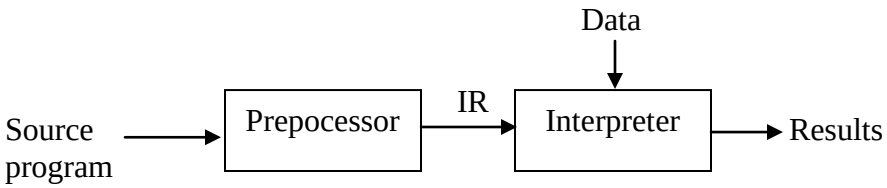
In **Call by reference,** the address of an actual parameter is passed to the called function. If the parameter is an expression, its value is computed and stored in a temporary location and the address of the temporary location is passed to the called function. If the parameter is an array element, its address is similarly computed at the time of call.

(iv)Pure and Impure interpreter

In a **pure interpreter**, the source program is retained in the source form all through its interpretation. This arrangement incurs substantial analysis overheads while interpreting a statement. Schematic of Pure interpreter is shown in figure below:



An **impure interpreter** performs some preliminary processing of the source program to reduce the analysis overheads during interpretation. The preprocessor converts the program to an intermediate representation (IR), which is used during interpretation. This speeds up interpretation as the code component of the IR i.e the IC, can be analyzed more efficiently than the source form of the program.



b. Discuss the issues involved that contribute to the semantics gap between a programming language domain and an execution domain.

Answer:

A compiler bridges the semantic gap between a programming language domain and an execution domain. Following are the issues that contribute to the semantic gap between a programming language domain and an execution domain:

i) Data types
ii) Data structures
iii) Scope rules
iv) Control structures

Data Types

A data type is the specification of

(i) legal values for variables of the type, and
(ii) legal operations on the legal values of the type.

Legal operations of a type typically include operation and a set of data manipulation operations. Semantics of a data type require a compiler to ensure that variables of a type are assigned or manipulated only through legal operations. The following tasks are involved in ensuring this:

1. Checking legality of an operation for the types of its operands. This ensures that a variable is subjected only to the legal operations of its type.
2. Use type conversion operations to convert values of one type into values of another type wherever necessary and permissible according to the rules of a programming language.

3. Use appropriate instruction sequences of the target machine to implement th operations of a type.

```
var
        x, y : real;
        i, j : integer;
begin
        y := 10;
        x := y + 1;
```

While compiling the first assignment statement, the compiler must note that y is a real variable; hence every value stored in y must be a real number. Therefore it must generate code to convert the value 10 to the floating point representation. In the second assignment statement, the addition cannot be performed on the values of y and I straightway as they belong to different types. Hence, the compiler first generates code to convert the value of I to the floating point representation and then generates code to perform the addition as a floating point operation.

Having checked the legality of each operation and determined the need for type conversion operations, the compiler must generate *type specific code* to implement an operation. In a type specific code the value of a variable of type $type_i$ is always manipulated through instructions, which know how values of $type_i$ are represented.

Generation of type specific code achieves two important things. It implements the second half of a type's definition; viz. a value of $type_i$ *is only manipulated through a legal operation of* $type_i$. It also ensures execution efficiency since type related issues do not need explicit handling in the execution domain.

Data Structures

A programming language permits the declaration and use of data structures like arrays, stacks, records, lists etc. To compile a reference to an element of a data structure, the compiler must develop a memory mapping to access the memory word(s) allocated to the element. A record, which is heterogeneous data structure, leads to complex memory mappings. A user defined type requires mappings of a different kind, those that map the values of the type into their representations in a computer, and vice versa.

Scope Rules

Scope rules determine the accessibility of variables declared in different blocks of a program. The scope of a program entity, i.e. a data item, is that part of a program where the entity is accessible. In most languages the scope of a data item is restricted to the program block in which the data item is declared. It extends to an enclosed block unless the enclosed block declares a variable with an identical name.

```
A   x, y : real;
        B   y, z : integer;
                x := y;
```

Variable x of block A is accessible in block A and in the enclosed block B. however, variable y of block A is not accessible in block B since y is redeclared in block B. Thus, the statement x := y uses y of block B.
The compiler performs operations called scope analysis and name resolution to determine the data item designated by the use of a name in the source program. The generated code simply implements the results of the analysis.

Control Structures

The control structure of a language is the collection of the language features for altering the flow of control during program execution. This includes conditional transfer of control, conditional execution, iteration control and procedure calls. The compiler must ensure that a source program does not violate the semantics of control structures.

```
for i := 1 to 100 do
begin
        if i = 10 then
        …..
        …..
end
```

Text Book

Systems Programming and Operating Systems, D. M. Dhamdhere, Tata McGraw-Hill, Second Revised Edition, 2005