Q.2 a. What is an operating system? Discuss the various functions of operating system.

Answer:

An **operating system** is system software that provides interface between user and hardware. The operating system provides the means for the proper use of resources (CPU, memory, I/O devices, data and so on) in the operation of the computer system. An operating system provides an environment within which other programs can do useful work.

Various functions of operating system are as follows:

(1) Process management: A process is a program in execution. It is the job, which is currently being executed by the processor. During its execution a process would require certain system resources such as processor, time, main memory, files etc. OS supports multiple processes simultaneously. The process management module of the OS takes care of the creation and termination of the processes, assigning resources to the processes, scheduling processor time to different processes and communication among processes.

(2) Memory management module: It takes care of the allocation and deallocation of the main memory to the various processes. It allocates main and secondary memory to the system/user program and data. To execute a program, its binary image must be loaded into the main memory. Operating System decides.

(a) Which part of memory are being currently used and by whom.
(b) Which process to be allocated memory.
(c) Allocation and de allocation of memory space.

(3) I/O management: This module of the OS co-ordinates and assigns different I/O devices namely terminals, printers, disk drives, tape drives etc. It controls all I/O devices, keeps track of I/O request, issues command to these devices. I/O subsystem consists of

(i) Memory management component that includes buffering, caching and spooling.
(ii) Device driver interface
(iii) Device drivers specific to hardware devices.

(4) File management: Data is stored in a computer system as files. The file management module of the OS would manage files held on various storage devices and transfer of files from one device to another. This module takes care of creation, organization, storage, naming, sharing, backup and protection of different files.

(5) Scheduling: The OS also establishes and enforces process priority. That is, it determines and maintains the order in which the jobs are to be executed by the computer system. This is so because the most important job must be executed first followed by less important jobs.

(6) Security management: This module of the OS ensures data security and integrity. That is, it protects data and program from destruction and unauthorized access. It keeps different programs and data which are executing concurrently in the memory in such a manner that they do not interfere with each other.

(7) Processor management: OS assigns processor to the different task that must be performed by the computer system. If the computer has more than one processor idle, one of the processes waiting to be executed is assigned to the idle processor. OS maintains internal time clock and log of system usage for all the users. It also creates error message and their debugging and error detecting codes for correcting programs.

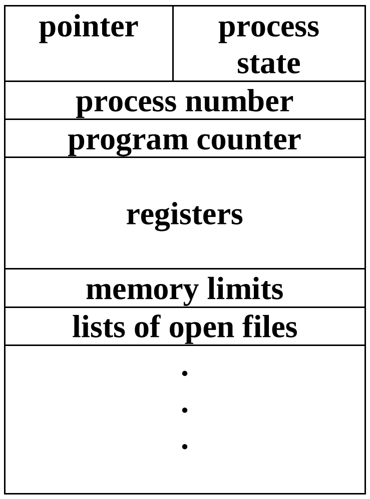
(b) Write the different ways in which the pthread terminates?
Answer
A pthread terminates in one of the following ways:
(i) When it calls the routine `pthread-exit(status-code).`
(ii) When it returns from its `start_routine` (`pthread_exit` is implicitly called).
(iii) When it is cancelled by another pthread by calling the `pthread_cancel` routine.
(iv) When the process creating the pthread terminates.

(c) Explain the various component of Process Control Block?
Answer
Each process is represented in the operating system by a **process control block** (PCB), also called a task control block. A process control block is shown in the figure below. It contains many pieces of information associated with a specific process, as follows:

- **Process state:** The state may be new, ready, running, waiting, halted, and so on.
- **Program counter:** The counter indicates the address of the next instruction to be executed for this process.
- **CPU registers:** The registers vary in number and type, depending on the computer architecture. They include accumulators, index registers, stack pointers, and general-purpose registers, plus any condition-code information. Along with the program counter, this state information must be saved when an interrupt occurs, to allow the process to be continued correctly afterward.
- **CPU scheduling information:** This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters.
- **Memory-management information:** This information may include such information as the value of the base and limit registers, the page tables, or the segment tables, depending on the memory system used by the operating system.
- **Accounting information:** This information includes the amount of CPU and real time used, time limits, account numbers, job or process numbers, and so on.

- **IO Status information:** The information includes the list of I/O devices allocated to this process, a list of open files, and so on.

The PCB simply serves as the repository for any information that may vary from process to process.



Process Control Block

Q3 (a) List the various tasks involving Process scheduling?
Answer
Process scheduling involves following tasks:

(i) Creating new processes
(ii) Monitoring processes
(iii) Selecting a process for execution (process scheduling)
(iv) Allocating the CPU to the selected process (process dispatching)
(v) Deallocating the CPU from a process and determining the new state of process (process pre-emption)
(vi) Termination of processes
(vii) Supporting communication and synchronization requirements of processes
(viii) Interfacing with other OS modules which affect process states

(b) Write the actions taken by the time sharing scheduler?
Answer
Actions of the time sharing scheduler can be summarized as follows:

(i) The scheduler maintains two separate PCB lists – one for ready processes and another for blocked and swapped-out processes.
(ii) The PCB list for ready processes is organized as a queue.
(iii) The PCB of a newly created process can be added to the end of the ready queue.
(iv) The PCB of a terminating process can be simply removed from the system.
(v) The scheduler always selects the PCB at the head of the ready queue.
(vi) When a running process finishes its time slice, or makes an IO request, its PCB is moved from ready queue to the blocked / swapped-out list.

(vii) 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) Give the algorithm for deadlock detection. Also mention the inputs and data structures used in algorithm.
Answer Page Number 384-385 of Text Book

Q4 (a) Give a solution for reader-writers problem using conditional critical regions.
Answer: Page Number 409 of Text Book

(b) What is a semaphore? Explain binary semaphore with the help of an example?
Answer
A **semaphore** is a synchronization tool that provides a general-purpose solution to controlling access to critical sections. A semaphore is an abstract data type (ADT) that defines a nonnegative integer variable which, apart from initialization, is accessed only through two standard operations: wait and signal. The classical definition of wait in pseudo code is

```
wait(S){
        while(S<=0)
        ; // do nothing
        S--;
}
The classical definitions of signal in pseudocode is
        signal(S){
        S++;
}
```

A binary semaphore is one that only takes the values 0 and 1. These semaphores are used to implement mutual exclusion. The following program code illustrates a Critical-section implementation using a binary semaphore. The main program declares a semaphore named *mutex* (and initializes it to 1) and initiates two concurrent processes. Each process performs a P(*mutex*) to gain entry to the CS, and a V(*mutex*) while exiting from the CS. Since *mutex* is initialized to 1, only one process can be in the CS at any time.

```
var mutex : semaphore := 1;

   Parbegin
      Repeat                              Repeat
          P(mutex);                           P(mutex);
          { Critical Section}                 { Critical Section}
          V(mutex);                           V(mutex);
          {Remainder of the cycle}            {Remainder of the cycle}
       forever;                           forever;

   Parend
```

end

Process p_i *Process p_j*

(c) Discuss the following:

(i) FS Actions at Open

When a user program U executes the call

`open (<filename> …);`

where <*filename*> is an access path for a file, FS determines two items of information:

1. Pointer to FCB of the directory containing the file (*directory FCB pointer*)
2. *Internal id* of the file.

The internal id is passed back to the application program for use during file processing. The directory FCB pointer is used to update the directory while closing a newly created file. FS uses the following procedure to determine these items of information:

1. If the access path is absolute, locate the FCB of the FS root directory. Else, locate the FCB of the current directory. Set a pointer called *directory FCB pointer* to point at this FCB.
2. (a) Search for the next component of the access path in the directory represented by *directory FCB pointer*. Give an error if the access path is not valid,
   (b) Create an FCB in a new entry of the AFT for the file described by the access path component.
   (c) Set a pointer called *file FCB pointer* to point at this FCB,
   (d) If more components exist in the access path, set *directory FCB pointer = file FCB pointer* and repeat Step 2.
   (e) Set *internal id* of the file to the offset of *file FCB pointer* in AFT.
3. (a) For existing file, initialize its FCB using the information in the directory entry of the file. This includes copying the pointer to the FMT of the file.
   (b) For a new file, allocate a disk block to contain its FMT. Set the address of this disk block in the FCB.
4. Return the *internal id* of the file <*filename*> to the application program.

Step 2 is called *access path resolution*. In the interests of access efficiency, FS may copy some part of the FMT into memory while opening the file (i.e. in Step 3).

(ii) FS Actions at Close

When the application program executes the statement

`close` (*internal_id, …*);

FS performs the following actions:

1. If the file has been newly created or updated,
   (a) If a newly created file, use *directory FCB pointer* to locate the FCB of the directory in which the file is to exist. Create an entry for the file in this directory. Copy the FMT of the new file into this entry. If the directory entry contains a pointer to FMT rather than FMT itself, the

FMT is first written on the disk and its disk address is entered in the directory entry.

(b) If an updated file, update the directory entry of the file (in the directory pointed to by *directory FCB pointer*) to reflect the change in size etc.

(c) *file FCB pointer* := *directory FCB pointer;*
directory FCB pointer :=
Address (FCB of the parent of the Directory);
If necessary, repeat Step 1(b).

2. The FCB of the file and the FCB's of the directories containing it are erased from the AFT. Internal id's assigned to them can now be reused for other files.

Q5 (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.

Q6 (a) With the help of suitable example, explain forward reference?

Answer

A forward reference of a program entity is a reference to the entity which precedes its definition in the program.

While processing a statement containing a forward statement, a language processor does not possess all relevant information concerning the referenced entity. This creates difficulties in synthesizing the equivalent target statements. This problem can be solved by postponing the generation of target code until more information concerning the entity becomes available. Postponing the generation of target code may also reduce memory requirements of the language processor and simplify its organization.

Consider the following statements

```
percent_profit := (profit * 100) / cost_price;

....................
....................
long profit;
```

to be a part of some program in some programming language. The statement `long profit;` declares `profit` to have a double precision value. The reference to `profit` in the assignment statement constitutes a forward reference because the declaration of `profit` occurs later in the program. Since the type of `profit` is not known while processing the assignment, correct code cannot be generated for it in a statement-by-statement manner.

(b) Define Intermediate Representation? What are the desirable properties of Intermediate Representation?
Answer
"An intermediate representation (IR) is a representation of a source program which reflects the effect of some, but not all, analysis and synthesis tasks performed during language processing".

Desirable properties of an IR are:

- Ease of use: IR should be easy to construct and analyze.
- Processing efficiency: Efficient algorithms must exist for constructing and analyzing the IR.
- Memory efficiency: IR must be compact.

Like the pass structure of language processors, the nature of intermediate representation is influenced by many design and implementation considerations.

(c) Explain the two approaches of collision handling methods?
Answer
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.

Q7 (a) Define the following:
(i) Finite state automaton (FSA)
(ii) Deterministic finite state automaton (DFA)

Answer

(i) Finite state automaton (FSA)

A finite state automaton is a triple (S, ∑, T) where

S is a finite set of states, one of which is the initial state s_{init}, and or more of which are the final states.
∑ is the alphabet of source symbols.
T is a finite set of state transitions defining transitions out of each $s_i \in S$ on encountering the symbols of ∑.

(ii) Deterministic finite state automaton (DFA)

A deterministic finite state automaton (DFA) is an FSA such that $t_1 \in T$, $t_1 \equiv (s_i, symb, s_j)$ implies $\nexists\, t_2 \in T$, $t_2 \equiv (s_i, symb, s_k)$.

(b) Discuss LL(1) parser? Show the parser table for an LL(1) parser for Grammar given below.

E ::= T E′
E′ ::= +T E′ | ε
T ::= V T′
T′ ::= * V T′ | ε
V ::= < id >

Answer

An LL(1) parser is a table driven parser for left-to-left parsing. The ‘1’ in LL(1) indicates that the grammar uses a look-ahead of one source symbol i.e., the prediction to be made is determined by the next source symbol. A major advantage of LL(1) parsing is its amenability to automatic construction by a parser generator.

The following table shows the parser table or an LL(1) parser for the grammar given:

Non terminal	***Source symbol***			
	<id>	+	*	-|
E	$E \Rightarrow TE'$			
E′		$E' \Rightarrow +TE'$		$E' \Rightarrow \varepsilon$
T	$T \Rightarrow VT'$			
T′		$T' \Rightarrow \varepsilon$	$T' \Rightarrow *VT'$	$T' \Rightarrow \varepsilon$
V	$V \Rightarrow$ <id>			

The parsing table (PT) has a row for each NT $\in$ SNT and a column for each T $\in \sum$. A parsing table entry PT (nt_i, t_j) indicates what prediction should be made if nt_i is the leftmost NT in a sentential form and t_j is the next source symbol. A blank entry in PT indicates an error situation. A source string is assumed to be enclosed between the symbol ‘|-’ and ‘-|’. Hence the parser starts with the sentential form |- E -|.

(c) What are the different information`s contained by the object module of a program to relocate and link the program with other programs?

Answer

The object module of a program contains all necessary information to relocate and link the program with other programs. The object module of a program consists of four components:

1. *Header:* The header contains translated origin, size and execution start address of P.
2. *Program:* This component contains the machine language program corresponding to P.
3. *Relocation table* (RELOCTAB)*:* This table describes IRR_P. Each RELOCTAB entry contains a single field:
   Translated address: Translated address of an address sensitive instruction.
4. *Linking table* (LINKTAB)*:* This table contains information concerning the public definitions and external references in P. each LINKTAB entry contains three fields:

Symbol	:	Symbolic name
Type	:	PD/EXT indicating whether public definition or external reference
Translated address	:	For a public definition, this is the address of first memory word allocated to the symbol. For an external reference, it is the address of the memory word which is required to contain the address of the symbol.

Q8 (a) Explain the following Assembler Directives:-

(i) ORIGIN
(ii) EQU
(iii) LTORG
(iv) START & END

Answer

(i) ORIGIN

The syntax of this directive is

ORIGIN *<address spec>*

where <*address spec*> is an <*operand spec*> *or* <constant>. This directive indicates that *location counter* (LC) should be set to the address given by <address spec>. The ORIGIN statement is useful when the target program does not consist of consecutive memory words. The ability to use an <*operand* spec> in the ORIGIN statement provides the ability to perform LC processing in a *relative* rather than *absolute* manner.

(ii) EQU

The EQU statement has the syntax

<symbol> EQU <address spec>

where <address spec> is an <operand spec> or <constant>.

The EQU statement defines the symbol to represent <address spec>. This differs from the DC/DS statement as no LC processing is implied. Thus EQU simply associates the name <symbol> with <address spec>.

(iii) LTORG

The LTORG statement permits a programmer to specify where literals should be placed. By, default assembler places the literals after the END statement. At every LTORG statement, as also at the END statement, the assembler allocates memory to the literals of a literal pool. The pool contains all literals used in the program since the start of the program or since the last LTORG statement.

The LTORG directive has very little relevance for the simple assembly language. The need to allocate literals at intermediate points in the program rather than at the end is critically felt in a computer using a base displacement mode of addressing.

(iv) START & END

The START directive indicates that the first word of the target program generated by the assembler should be placed in the memory word with address <*constant*>.

START <*constant*>

The END directive indicates the end of the source program. The optional <*operand spec*> indicates the address on the instruction where the execution of the program should begin.

END [<*operand spec*>]

(b) 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	
	:	

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'	
= '1'	

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.

POOLTAB

Literal no
#1
#3
--

Q9 (a) Discuss the issues involved that contributes 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 the 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.

```
     ┌ x, y : real;
     │         ┌ y, z : integer;
     │         │
     │         │
  A  │     B   │      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
```

(b) What are the features used by compiler during implementing function calls?

Answer
The compiler uses a set of features to implement function calls. These are described below:

- *Parameter list:* The parameter list contains a descriptor for each actual parameter of the function call. The notation D_p is used to represent the descriptor corresponding to the formal parameter p.
- *Save area:* The called function saves the contents of CPU registers in this area before beginning its execution. The register contents are restored from this area before returning from the function.
- *Calling conventions:* These are execution time assumptions shared by the called function and its caller(s). The conventions include the following:
    a) How the parameter list is accessed.
    b) How the save area is accessed.
    c) How the transfers of control at call and return are implemented.
    d) How the function value is returned to the calling program.

Most machine architectures provide special instructions to implement items c) and d).

(c) Compare and contrast the following:
(i) Static and Dynamic memory allocation

In **static memory allocation**, memory is allocated to a variable before the execution of a program begins. Static memory allocation is typically performed during compilation. No memory allocation or deallocation actions are performed during the execution of a program. Thus, variables remain permanently allocated; allocation to a variable exists even if the program unit in which it is defined is not active.

In **dynamic memory allocation**, memory bindings are established and destroyed during the execution of a program. Dynamic memory allocation is of two types; automatic allocation and program controlled allocation.

- In automatic dynamic allocation, memory is allocated to the variables declared in a program unit when the program unit is entered during execution and is deallocated when the program unit is exited. Thus the same memory area may be used for the variables of different program units. It is also possible that different memory areas may be allocated to same variable in different activations of a program unit.
- In program controlled dynamic allocation, a program can allocate or deallocate memory at arbitrary points during its execution.

(ii) **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.

Text Book

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