Q2 (a) What is operating system? Discuss the various features 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 de allocation 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, and 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

© IETE

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.

Q2 (b) Explain the following facilities for implementing interacting processes in programming languages and Operating systems:
(i) Fork-Join primitives
(ii) Unix processes

Answer

Fork-Join primitives

Fork and **Join** are primitives in a higher level programming. The syntax of these primitives is as follows:

```
fork <label>;
join <var>;
```

where <label> is a label associated with some program statement , and <var> is a variable. A statement fork lab1 causes creation of a new process which starts executing at the statement with the label lab1. This process is concurrent with the process which executed the statement fork lab1. A **join** statement synchronizes the birth of a process with termination of one or more processes. Execution of a statement join v; has the following effect:

1. Value of a variable v is decremented by 1.
2. The process executing the join statement terminates.
3. A new process is created if the new value of v is zero. This process begins execution of the statement following the join v statement.

Thus, if v is initialized to some value n, n processes need to execute the statement join v before the new process is created.

The program given below implements the computation result := max(a)/min(a), where a is an array, using the **fork** and **join** primitives. Since m is initialized to 3, join m; will have to be executed by all three processes before the execution of result: = y / x; is initiated.

```
        for i := 1 to 100
                read a[i];
        m := 3;
        fork lab1;
        fork lab2;
        Goto lab3;
lab1 :  x := min(a);
        Goto lab3;
lab2 :  y := max(a);
lab3 :  join m;
        result := y/x;
```

Fork-Join provides a functionally complete facility for control synchronization. Hence it can be used to implement arbitrary synchronizations. However, being unstructured primitives, their use is cumbersome and error-prone.

(iii)Unix processes

Unix permits a user process to create child processes and to synchronize its activities with respect to child processes. The following features are provided for this purpose:

1. **Creation of a child process**

A process creates a child process through the system call fork. fork creates a child process and sets up its execution environment. It then allocates an entry in the process table i.e. PCB; for the newly created process and marks its state as *ready*. fork also returns the id of the child process to its creator; called the parent process. The child process shares the address space and file pointers of the parent process, hence data and files can be directly shared. A child process can in turn create its own child processes, thus leading to the creation of a process tree. The system keeps track of the child-parent relationships throughout the lives of the parent and child processes.

The child process environment (called its context) is a copy of the parent's environment. Hence the child executes the same code as the parent. At creation, the program counter of the child process is set at the instruction at which the fork call returns. The only difference between the parent and the child processes is that in the parent process fork returns with the process id of the child process, while in the child process it returns with a '0'.

2. **Termination of a process**

Any process p_i can terminates itself through the exit system call

exit (*status_call*);

where the value of *staus_code* is saved in the kernel for access by the parent of p_i. If the parent is waiting for the termination of p_i, a signal is sent to it. The child processes of p_i are made the children of a kernel process.

3. **Waiting for termination of a child process.**

A process p_i can wait for the completion of a child process through the system call

wait(*adr*(xyz));

where xyz is available. When a child of p_i terminates (or if one has already terminated) the wait call returns after storing the termination status of the terminated child process into xyz. The wait call returns with a '-1' if p_i has no children.

```
main() {
        int saved_status;
        for ( i=0; i < 3; i++) {
                if (fork() == 0) {
                        /* code for child processes */
                        exit;
                }
        }
        while (wait (&saved_status) != -1);
                        /* all child processes terminated? */
}
```

Q3 (a) Explain Event Control Block (ECB)? With the help of suitable diagram discuss the organization of the different modules of event handler.

Answer
An event control block (ECB) is used facilitate quick identification of a process awaiting an event. It contains all information concerning an event whose occurrence is anticipated by some process in the system. Different fields in the ECB are shown in the figure. The event description field describes the event. Its content can vary depending on the nature of the event. The event description field consists of two parts.

(i) Event class
(ii) Event details

ECB pointer
Event description
Process id of awaiting process

Event Control Block (ECB)

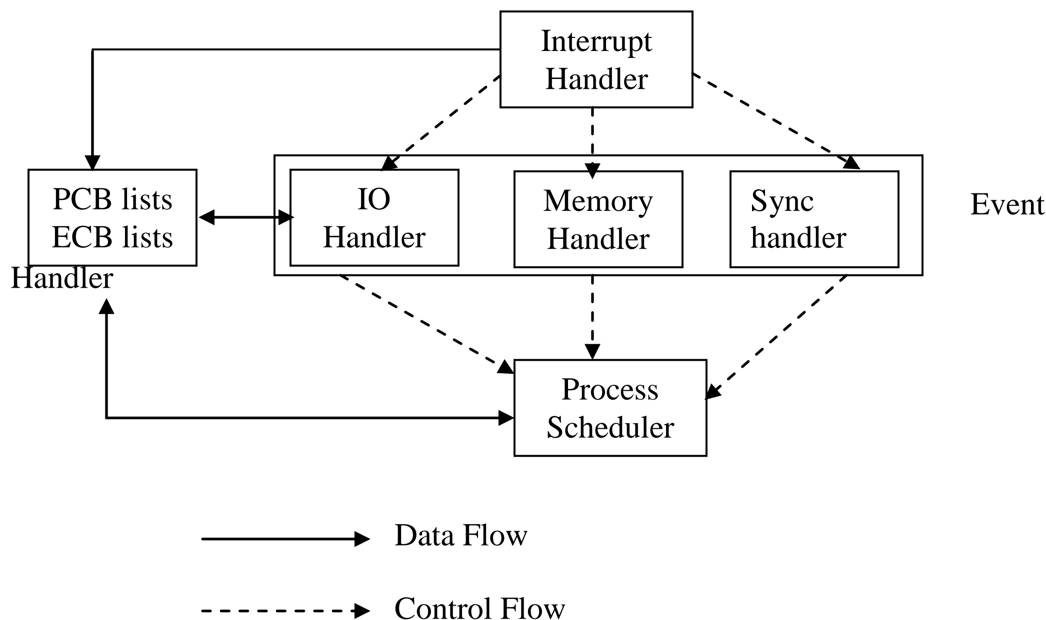
OS kernel uses the ECB pointer field to maintain a list of ECB's. When an event occurs, it scans the list of ECB's to find an ECB with a matching event description. The process id field of the ECB points to the affected process. The state of the process is now changed to reflect the occurrence of the event. To speed this action, scheduler can maintain many ECB lists, e.g. a list of ECB's for awaited inter process messages, a list of ECB's for I/O operations on a specific I/O device, etc. The following example illustrates the use of ECBs to locate the PCB of a process affected by an event.

Kernel actions when process pi requests an I/O operation on some device d, and when the I/O operation completes, can be described as follows:

1) Kernel creates an ECB, and initializes it as follows:
2) The newly created ECB (call as ECB_i) is added to a list of ECB's.
3) The state of p_i is changed to blocked and address of ECB_i is put into the 'Event information' field of p_i's PCB.
4) p_i's PCB may now be shifted to appropriate scheduling list.
5) When the interrupt 'End of I/O on device d' is raised, ECB_i is located by searching for an ECB with a matching event description field.
6) PCB of p_i is located from ECB_i. State of p_i is now changed to ready.

Figure below shows the organization of the modules constituting the event handler component. The interrupt handler gains control when an interrupt signals the occurrence of an event. It saves the PSR and contents of the CPU registers in the PCB of the interrupted process and changes the state of the process from running to ready. It then obtains the event class and event details from the interrupt code field in the saved PSR and invokes the event handler for the appropriate event class. The invoked handler analyses the event and performs appropriate actions. These actions typically result in the creation of a new ECB (when a process makes some request to the OS), or in a change of state for some process. At the end of its processing, the event handler module passes control to the process scheduler, which selects a process and passes its PCB to the CPU dispatcher.

© IETE



Q3 (b) Write down Banker's algorithm for multiple resources? List different Inputs and Data structures used in the algorithm.

Answer

Following is the Banker's algorithm for multiple resources:

Inputs

n	:	Number of processes:
r	:	Number of resource classes;
Blocked	:	**Set of** processes;
Running	:	**Set of** processes;
p_{req}	:	Process making the new resource request;
New_req	:	**array** [1…r] of integer;

Data structures

Max	:	**array** [1..n, 1..r] of integer;
Allocated_resources	:	**array** [1..n, 1..r] of integer;
Requested_resources	:	**array** [1..n, 1..r] of integer;
Total_alloc	:	**array** [1..r] of integer;
Total_exist	:	**array** [1..r] of integer;
Active	:	**Set of** processes;
Simulated_alloc	:	**array** [1..r] of integer;

1. *Active := Running* ∪ *Blocked*
2. **for** k = 1..r
   Requested_resources[*req,k*] := *New_request*[*k*];
3. **for** k = 1..r /* Compute project state */
   Allocated_resources[*req,k*] :=
   Allocated_resources[*req,k*] + *New_request*[*k*];

Total_alloc[*k*] := *Total_alloc*[*k*] + *New_request*[*k*];
4. **for** k = 1..r /* Check if new state is feasible */
 if *Requested_resources*[*req,k*] >
 Total_exist[*k*] − *Total_alloc*[*k*]
 then goto step 6;
5. *Simulated_alloc* := *Total_alloc*;
 while ∃ p_1 ∈ *Active such that* ∀ *k*
 Total_exist[*k*] − *Simulated_alloc*[*k*]
 ≥ *max*[*l,k*] − *Allocated_resources*[*l.k*]
 (a) *Delete* p_1 *from Active;*
 (b) **for** k = 1..r
 Simulated_alloc[*k*] :=
 *Simulated_*alloc[k] − Allocated*_resources*[*l,k*];
6. **if** *Active is empty* **then** /* Projected state is safe */
 for k = 1..r
 Requested_resources[*req,k*] := 0;
 else /* Disallow projected grant and revert to current state */
 for k = 1..r
 Allocated_resources[*req,k*] :=
 Allocated_resources[*req,k*] − *New_request*[*k*];
 Total_alloc[*k*] := *Total_alloc*[*k*] − *New_request*[*k*];

The algorithm keeps a request pending if the projected state is infeasible (Step 4). Else it simulates the grant of the new request (Step 3) and determines its safety (Step5). If the request is safe, its grant is confirmed, else it is nullified (Step 6).

Q4 (a) 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
```

Q4 (b) What is Critical-Section problem? What are the requirements that critical–section problem must satisfy for its solution?

Answer

Consider a system consisting of *n* processes $\{P_0, P_1 \ldots P_{n-1})$. Each process has a segment of code, called a critical section, in which the process may be changing common variables, updating a table, writing a file, and so on. The important feature of the system is that, when one process is executing in its critical section, no other process is to be allowed to execute in its critical section. That is, no two processes are executing in their critical section at the same time. Thus, the execution of critical sections by the processes is *mutually exclusive* in time. The *critical-section problem* is to design a protocol that the processes can use to cooperate. Each process must request permission to enter its critical section. The section of code implementing this request is the **entry section**. The critical section may be followed by an **exit section**. The remaining code is the **remainder section**. The general structure of a typical process P_i is shown below:

```
do {
        | entry section |

                  critical section

        | exit section |

                  remainder section
} while (TRUE);
```

A solution to the critical-section problem must satisfy the following three requirements:

- **Mutual Exclusion:** If process P_i is executing in its critical section, then no other processes can be executing in their critical sections.
- **Progress:** If no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder section can participate in the decision on which will enter its critical section next, and this selection cannot be postponed indefinitely.

Bounded Waiting: There exists a bound on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

Q4 (c) Discuss the different techniques with which a file can be shared among different users?

Answer

Some popular techniques with which a file can be shared among different users are:

1. Sequential sharing: In this sharing technique, a file can be shared by only one program at a time, that is, file accesses by P1 and P2 are spaced out over time. A lock field can be used to implement this. Setting and resetting of the lock at file open and close ensures that only one program can use the file at any time.

2. Concurrent sharing: Here a number of programs may share a file concurrently. When this is the case, it is essential to avoid mutual interference between them. There are three categories of concurrent sharing:

a. **Immutable files**: If a file is shared in immutable mode, none of the sharing programs can modify it. This mode has the advantage that sharing programs are independent of one another.

b. **Single image immutable files**: Here the changes made by one program are immediately visible to other programs. The Unix file system uses this file sharing mode.

c. Multiple image mutable files: Here many programs can concurrently update the shared file. Each updating program creates a new version of the file, which is different from the version created by concurrent programs. This sharing mode can only be used in applications where concurrent updates and the existence of multiple versions are meaningful.

Q5 (a) With the help of example, discuss overlay?

Answer

To enable a process to be larger than the amount of memory allocated to it, we can use overlays. The idea of overlays is to keep in memory only those instructions and data that are needed at any given time. When other instructions are needed, they are loaded into space occupied previously by instructions that are no longer needed.

As an example, consider a two-pass assembler. During pass1, it constructs a symbol table; then, during pass2, it generates machine-language code. We may be able to partition such an assembler into pass1 code, pass2 code, the symbol table, and common support routines used by both pass1 and pass2. Assume that the sizes of these components are as follows:

Pass1	:	70 KB
Pass2	:	80 KB
Symbol table	:	20 KB
Common routines	:	30 KB

To load everything at once, we would require 200 KB of memory. If only 150 KB is available, we cannot run our process. However, notice that pass1 and pass2 do not need to be in memory at the same time. We thus define two overlays: Overlay A is the symbol table, common routines, and pass1, and overlay B is the symbol table, common routines, and pass2.

We add an overlay driver (10 KB) and start with overlay ***A*** in memory. When we finish pass1, we jump to the overlay driver, which reads overlay B into memory, overwriting overlay A, and then transfers control to pass2. Overlay A needs only 120 KB, whereas overlay B needs 130 KB. We can now run our assembler in the 150 KB of memory. It will load somewhat faster because fewer data need to be transferred before execution starts. However, it will run somewhat slower, due to the extra I/O to read the code for overlay B over the code for overlay A.

The code for overlay A and the code for overlay B are kept on disk as absolute memory images, and are read by the overlay driver as needed. Special relocation and linking algorithms are needed to construct the overlays. As in dynamic loading, overlays do not require any special support from the operating system. They can be implemented completely by the user with simple file structures, reading from the files into memory and then jumping to that memory and executing the newly read instructions. The operating system notices only that there is more I/O than usual.

Q5 (b) Consider a paging system with the page table stored in memory

- **(i) If a memory reference takes 200 nanoseconds, how long does a paged reference take?**
- **(iii) If we add associative registers, and 75 percent of all page-table references are found in the associative registers, what is the effective memory reference time? (Assume that finding a page-table entry in the associative registers takes zero time, if the entry is there.)**

Answer

- (i) 400 nanoseconds; 200 nanoseconds to access the page table and 200 nanoseconds to access the word in memory.
- (ii) Effective access time = 0.75 * (200 nanoseconds) + 0.25 * (400 nanoseconds) = 250 nanoseconds.

Q5 (c) What is the cause of thrashing? How does the system detect thrashing?

Answer

Thrashing is caused by under allocation of the minimum number of pages required by a process, forcing it to continuously page fault. The system can detect thrashing by

evaluating the level of CPU utilization as compared to the level of multiprogramming. It can be eliminated by reducing the level of multiprogramming.

Q6 (a) 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.

Q6 (b) Define Grammar of a language. Identify the different classes of grammar. Explain their characteristics and limitations.

Answer

A **formal language grammar** is a set of formation rules that describe which strings formed from the alphabet of a formal language are syntactically valid, within the language. A grammar only addresses the location and manipulation of the strings of the language. It does not describe anything else about a language, such as its semantics.

As proposed by Noam Chomsky, a grammar *G* consists of the following components:

- A finite set *N* of *non terminal symbols*.
- A finite set _ of *terminal symbols* that is disjoint from *N*.
- A finite set *P* of *production rules*, each rule of the form

Where * is the Kleene star operator and denotes set union. That is, each production rule maps from one string of symbols to another, where the first string contains at least one non terminal symbol.

The Chomsky hierarchy consists of the following levels:

• **Type-0 grammars** (unrestricted grammars) include all formal grammars. They generate exactly all languages that can be recognized by a Turing machine. The language that is recognized by a Turing machine is defined as all the strings on which it halts. These languages are also known as the recursively enumerable languages.

• **Type-1 grammars** (context-sensitive grammars) generate the context sensitive languages. These grammars have rules of the form $\alpha A\beta \rightarrow \alpha\gamma\beta$ with *A* a non terminal and α, β and γ are strings of terminals and non terminals. The strings α and β may be empty, but γ must be nonempty. The rule $S \rightarrow \varepsilon$ is allowed if S does not appear on the

right side of any rule. The languages described by these grammars are exactly all languages that can be recognized by a non-deterministic

• **Type-2 grammars** (context-free grammars) generate the context-free languages. These are defined by rules of the form $A \rightarrow \gamma$ with *A* a non terminal and γ a string of terminals and non terminals. These languages are exactly all languages that can be recognized by a non-deterministic pushdown automaton. Context free languages are the theoretical basis for the syntax of most programming languages.

• **Type-3 grammars** (regular grammars) generate the regular languages. Such a grammar restricts its rules to a single non terminal on the left-hand side and a right-hand side consisting of a single terminal, possibly followed by a single non terminal. The rule $S \rightarrow \varepsilon$ is also here allowed if S does not appear on the right side of any rule. These languages are exactly all languages that can be decided by a finite state automaton. Additionally, this family of formal languages can be obtained by regular expressions. Regular languages are commonly used to define search patterns and the lexical structure of programming languages.

Q6 (c) Discuss the different criteria used to classify the data structures used for Language processors?

Answer

The data structures used in language processing can be classified on the basis of the following criteria:

- **Nature of a data structure**: whether a linear or non linear data structure. A *linear data structure* consists of a linear arrangement of elements in the memory. The physical proximity of its elements is used to facilitate efficient search. It requires a contiguous area of memory for its elements. This poses problem in situations where the size of a data structure is difficult to predict and hence the designer is forced to overestimate the memory requirements of a linear data structure to ensure that it does not outgrow the allocated memory. Hence, this leads to wastage of memory. The elements of a *nonlinear data structure* are accessed using pointers. Hence the elements need not occupy contiguous locations of memory, which avoids the memory allocation problem as in linear data structures. However, the nonlinear arrangement of elements leads to lower search efficiency.
- **Purpose of a data structure**: whether a search data structure or an allocation data structure. *Search data structures* are used during language processing to maintain attribute information concerning different entities in the source program. These data structures are characterized by the fact that the entry fro an entity is created only once but may be searched for a large number of times. Search efficiency is therefore very important. *Allocation data structures* are characterized by the fact that the address of the memory area allocated to an entity is known to the user of that entity. Thus no search operations are conducted on them. Speed of allocation or deal location and efficiency of memory utilization are the important criteria for the allocation data structures.
- **Lifetime of a data structure**: whether used during language processing or during target program execution.

Q7 (a) What is parsing? Give difference between top down parsing and bottom up parsing.

Answer
The goal of **parsing** is to determine the syntactic validity of a source string. If the string is valid, a tree is built for use by subsequent phase of compiler.
Top down parsing: Given an input string, top down parsing attempts to derive a string identical to it by successive application of grammar rules to the grammar's distinguished symbol. When such a string is obtained, a tree representing its derivation would be the syntax tree for an input string. Thus if α is input-string, a top down parse determines a derivation sequence.
$s \rightarrow \cdots \rightarrow \cdots \rightarrow \alpha$
Bottom up parsing: A bottom up parse attempts to develop syntax tree for an input string through a sequence of reduction. If the input string can be reduced to the distinguished symbol, the string is valid. If not, error would be detected and indicated during the process of reduction itself.

Q7 (b) What are self-relocating programs? Why self-relocating programs are less efficient then re locatable programs?

Answer Page Number 232 of Text Book

Q7 (c) The translated origin of the assembly program P is 500. If the program is loaded for execution in the memory area starting with the address 900, calculate the relocation factor of P.

Answer
The relocation factor of P is defined as
$relocation_{factor_p} = l_{origin_p} - t_{origin_p}$
$= 900 - 500$
$= 400$

Q8 (a) What is assembly language? What are the basic features provided by assembly language that simplifies programming as compared to machine language?

Answer
An **assembly language** is a machine dependent, low level programming language which is specific to a certain computer system (or a family of computer systems)
Compared to the machine language of a computer system, it provides three basic features which simplify programming:

- ***Mnemonic operation codes:*** Use of mnemonic operation codes (also called mnemonic opcodes) for machine instructions eliminates the need to memorize numeric operation codes. It also enables the assembler to provide helpful diagnostics, for example indication of misspelt operation codes.

- ***Symbolic operands:*** Symbolic names can be associated with data or instructions. These symbolic names can be used as operands in assembly statements. The assembler performs memory bindings to these names; the programmer need not know any details of the memory bindings performed by the assembler. This leads to a very important advantage during program modification.
- ***Data declarations:*** Data can be declared in a variety of notations, including the decimal notation. This avoids manual conversion of constants into their internal machine representation, for example, conversion of -5 into $(11111010)_2$ or 10.5 into $(41A80000)_{16}$.

Q8 (b) 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

© IETE

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*>]

Q9 (a) Consider the following program segment:

```
main()
{
int i, j;
float x, y;
    y = 10; ........................(A)
    i = 5;
    x = y + i; .....................(B)
}
```

Explain what action the compiler must take during the compilation of assignment statements marked as (A) and (B)?

Answer

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' straightaway as they belong to different types. Hence compiler must first generate code to convert the value of 'i' to the floating point representation and then generate code to perform the addition as a floating point operation.

Q9 (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).

Q9 (c) Give an account of the issue pertaining to compilation of "if" statement in C language?

Answer

Control structures like if cause significant semantic gap between the PL domain and the execution domain because the control transfers are implicit rather than explicit. This semantic gap is bridged in two steps as follows:

Step 1: Control structure is mapped into an equivalent program containing explicit **goto's**. Since the destination of a **goto** may not have a label in the source program, the compiler generates its own labels and put them against the appropriate statements. For example, the equivalent of (a) given below is (b) where int_1, int_2 are labels generated by compiler for its own purposes.

```
if (e1) then
        S1;
else
        S2;
S3;
```

(a)

```
     if (e1) then goto int1;
     S2;
     goto int2;
int1 : S1;
int2 : S3;
```

(b)

Step 2: These programs are translated into assembly programs.

The first step need not be carried out explicitly. It can be implied in the compilation action.

Text Book

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