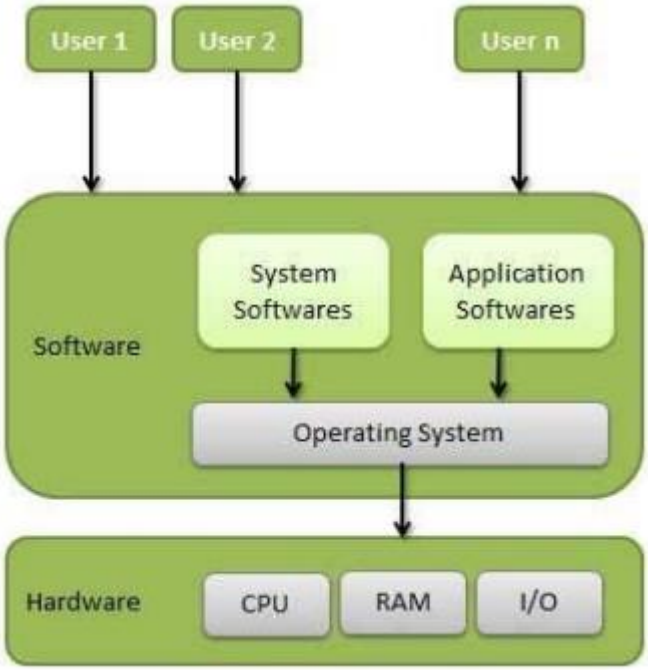
What Is OS ?

An Operating System (OS) is an interface between computer user and computer hardware. An operating system is software which performs all the basic tasks like file management, memory management, process management, handling input and output, and controlling peripheral devices such as disk drives and printers. Some popular Operating Systems include Linux Operating System, Windows Operating System, VMS, OS/400, AIX, z/OS, etc.

Definition:

An operating system is a program that acts as an interface between the user and the computer hardware and controls the execution of all kinds of programs.



Following are some of important functions of an operating System.

- Memory Management• Processor Management• Device Management• File Management• Security• Control over system performance• Job accounting• Error detecting aids• Coordination between other software and users•

An Operating System provides services to both the users and to the programs. It provides programs an environment to execute.

- It provides users the services to execute the programs in a convenient manner.
- Following are a few common services provided by an operating system: Program execution
- I/O operations
- File System manipulation
- Communication
- Error Detection
- Resource Allocation
- Protection

Basic Functions of Operation System:

The various functions of operating system are as follows:

1. **Process Management:**

- A program does nothing unless their instructions are executed by a CPU.A process is a program in execution. A time shared user program such as a complier is a process. A word processing program being run by an individual user on a pc is a process.
- A system task such as sending output to a printer is also a process. A process needs certain resources including CPU time, memory files & I/O devices to accomplish its task.
- These resources are either given to the process when it is created or allocated to it while it is running. The OS is responsible for the following activities of process management.
- Creating & deleting both user & system processes.
- Suspending & resuming processes.
- Providing mechanism for process synchronization.
- Providing mechanism for process communication.
- Providing mechanism for deadlock handling.

2. **Main Memory Management:**

The main memory is central to the operation of a modern computer system. Main memory is a

large array of words or bytes ranging in size from hundreds of thousand to billions. Main memory stores the quickly accessible data shared by the CPU & I/O device. The central processor reads instruction from main memory during instruction fetch cycle & it both reads &writes data from main memory during the data fetch cycle. The main memory is generally the only large storage device that the CPU is able to address & access directly. For example, for the CPU to process data from disk. Those data must first be transferred to main memory by CPU generated E/O calls. Instruction must be in memory for the CPU to execute them. The OS is responsible for the following activities in connection with memory management.

- Keeping track of which parts of memory are currently being used & by whom.
- Deciding which processes are to be loaded into memory when memory space becomes available.
- Allocating &deal locating memory space as needed.

3. **File Management:**

File management is one of the most important components of an OS computer can store information on several different types of physical media magnetic tape, magnetic disk & optical disk are the most common media. Each medium is controlled by a device such as disk drive or tape drive those has unique characteristics. These characteristics include access speed, capacity, data transfer rate & access method (sequential or random).For convenient use of computer system the OS provides a uniform logical view of information storage. The OS abstracts from the physical properties of its storage devices to define a logical storage unit the file. A file is collection of related information defined by its creator. The OS is responsible for the following activities of file management.

- Creating & deleting files.
- Creating & deleting directories.
- Supporting primitives for manipulating files & directories.
- Mapping files into secondary storage.
- Backing up files on non-volatile media.

4. **I/O System Management:**

One of the purposes of an OS is to hide the peculiarities of specific hardware devices from the user. For example, in UNIX the peculiarities of I/O devices are hidden from the bulk of the OS itself by the I/O subsystem. The I/O subsystem consists of:

- A memory management component that includes buffering, catching & spooling.

- A general device- driver interfaces drivers for specific hardware devices. Only the device driver knows the peculiarities of the specific device to which it is assigned.

5. **Secondary Storage Management:**

The main purpose of computer system is to execute programs. These programs with the data they access must be in main memory during execution. As the main memory is too small to accommodate all data & programs & because the data that it holds are lost when power is lost. The computer system must provide secondary storage to back-up main memory. Most modern computer systems are disks as the storage medium to store data & program. The operating system is responsible for the following activities of disk management.

- Free space management.
- Storage allocation.
- Disk scheduling

Because secondary storage is used frequently it must be used efficiently.

Networking:

A distributed system is a collection of processors that don't share memory peripheral devices or a clock. Each processor has its own local memory & clock and the processor communicate with one another through various communication lines such as high speed buses or networks. The processors in the system are connected through communication networks which are configured in a number of different ways. The communication network design must consider message routing & connection strategies are the problems of connection & security.

Protection or security:

If a computer system has multi users & allow the concurrent execution of multiple processes then the various processes must be protected from one another's activities. For that purpose, mechanisms ensure that files, memory segments, CPU & other resources can be operated on by only those processes that have gained proper authorization from the OS.

Command interpretation:

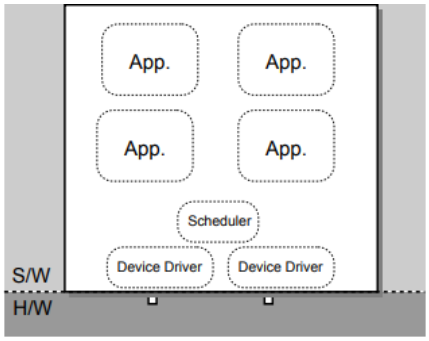
One of the most important functions of the OS is connected interpretation where it acts as the interface between the user & the OS.

Operating System Services

- One set of operating-system services provides functions that are helpful to the user Communications – Processes may exchange information, on the same computer or between computers over a network Communications may be via shared memory or through message passing (packets moved by the OS)
- Error detection – OS needs to be constantly aware of possible errors May occur in the CPU and memory hardware, in I/O devices, in user program For each type of error, OS should take the appropriate action to ensure correct and consistent computing Debugging facilities can greatly enhance the user's and programmer's abilities to efficiently use the system

- Another set of OS functions exists for ensuring the efficient operation of the system itself via resource sharing
- **Resource allocation** - When multiple users or multiple jobs running concurrently, resources must be allocated to each of them
- Many types of resources - Some (such as CPU cycles, main memory, and file storage) may have special allocation code, others (such as I/O devices) may have general request and release code
- **Accounting** - To keep track of which users use how much and what kinds of computer resources
- **Protection and security** - The owners of information stored in a multiuser or networked computer system may want to control use of that information, concurrent processes should not interfere with each other
- **Protection** involves ensuring that all access to system resources is controlled
- **Security** of the system from outsiders requires user authentication, extends to defending external I/O devices from invalid access attempts

Monolithic Operating Systems:



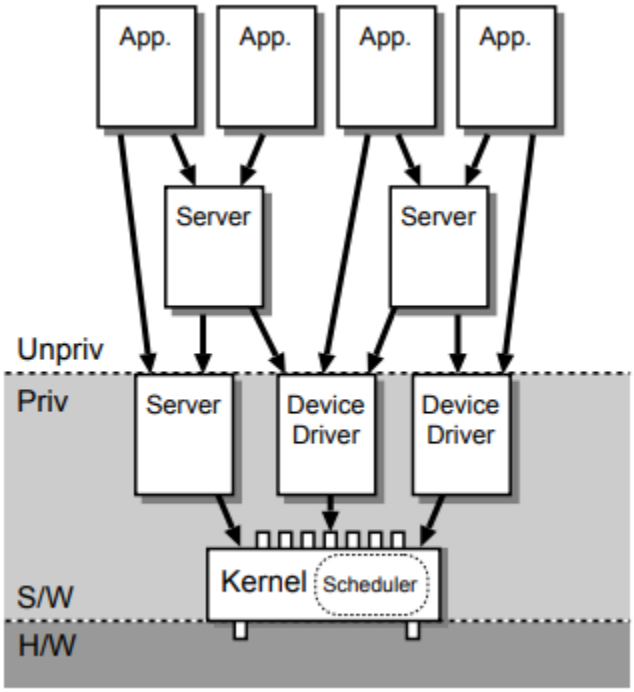
• Oldest kind of OS structure ("modern" examples are DOS, original MacOS)

• Problem: applications can e.g. – trash OS software. – trash another application. – hoard CPU time. – abuse I/O devices. – Etc.

• No good for fault containment (or multi-user).

• Need a better solution.

Microkernel Operating Systems:



- Alternative structure: – push some OS services into servers. – servers may be privileged (i.e. operate in kernel mode).

• Increases both modularity and extensibility.

• Still access kernel via system calls, but need new way to access servers: ⇒ inter-process communication (IPC) schemes

Real time Systems:

Real time system is used when there are rigid time requirements on the operation of a processor or flow of data. Sensors bring data to the computers. The computer analyzes data and adjusts controls to modify the sensors inputs. System that controls scientific experiments, medical imaging systems and some display systems are real time systems. The disadvantages of real time system are: a. A real time system is considered to function correctly only if it returns the correct result within the time constraints. b. Secondary storage is limited or missing instead data is usually stored in short term memory or ROM. c. Advanced OS features are absent. Real time system is of two types such as

• Hard real time systems: It guarantees that the critical task has been completed on time. The sudden task is takes place at a sudden instant of time.

• Soft real time systems: It is a less restrictive type of real time system where a critical task gets priority over other tasks and retains that priority until it computes. These have more limited utility than hard real time systems. Missing an occasional deadline is acceptable e.g. QNX, VX works. Digital audio or multimedia is included in this category. It is a special purpose OS in which there are rigid time requirements on the operation of a processor. A real time OS has well defined fixed time constraints. Processing must be done within the time constraint or the system will fail. A real time system is said to function correctly only if it returns the correct result within the time constraint. These systems are characterized by having time as a key parameter.

Task :

- Task is a piece of code or program that is separate from another task and can be executed independently of the other tasks.
- In embedded systems, the operating system has to deal with a limited number of tasks depending on the functionality to be implemented in the embedded system.

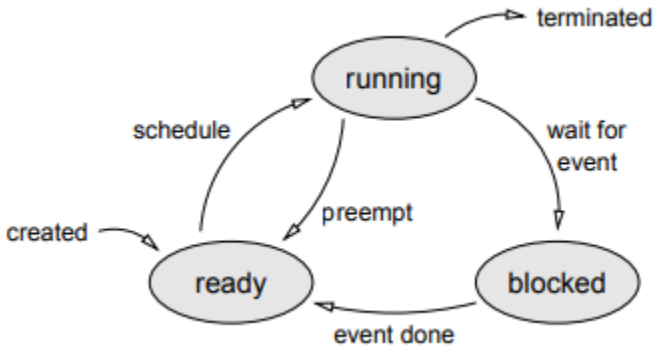
- Multiple tasks are not executed at the same time instead they are executed in pseudo parallel i.e. the tasks execute in turns as the use the processor.
- From a multitasking point of view, executing multiple tasks is like a single book being read by multiple people, at a time only one person can read it and then take turns to read it.
- Different bookmarks may be used to help a reader identify where to resume reading next time.
- An Operating System decides which task to execute in case there are multiple tasks to be executed. The operating system maintains information about every task and information about the state of each task.
- The information about a task is recorded in a data structure called the ***task context.*** When a task is executing, it uses the processor and the registers available for all sorts of processing. When a task leaves the processor for another task to execute before it has finished its own, it should resume at a later time from where it stopped and not from the first instruction. This requires the information about the task with respect to the registers of the processor to be stored somewhere. This information is recorded in the task context.

Task States

In an operation system there are always multiple tasks. At a time only one task can be executed. This means that there are other tasks which are waiting their turn to be executed.

Depending upon execution or not a task may be classified into the following three states:

- **Running state** - Only one task can actually be using the processor at a given time that task is said to be the "running" task and its state is "running state". No other task can be in that same state at the same time
- **Ready state** - Tasks that are not currently using the processor but are ready to run are in the "ready" state. There may be a queue of tasks in the ready state.
- **Waiting state -** Tasks that are neither in running nor ready state but that are waiting for some event external to themselves to occur before the can go for execution on are in the "waiting" state**.**



Process Concept:

Process: A process or task is an instance of a program in execution. The execution of a process must programs in a sequential manner. At any time at most one instruction is executed. The process includes the current activity as represented by the value of the program counter and the content of the processors registers. Also it includes the process stack which contain temporary data (such as method parameters return address and local variables) & a data section which contain global variables.

Difference between process & program:

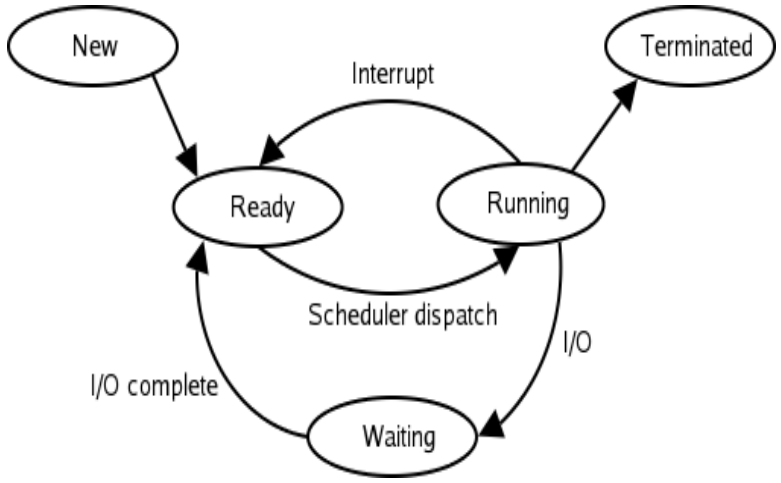
A program by itself is not a process. A program in execution is known as a process. A program is a passive entity, such as the contents of a file stored on disk where as process is an active entity with a program counter specifying the next instruction to execute and a set of associated resources may be shared among several process with some scheduling algorithm being used to determinate when the stop work on one process and service a different one.

Process state: As a process executes, it changes state. The state of a process is defined by the correct activity of that process. Each process may be in one of the following states.

- **New:** The process is being created.
- **Ready:** The process is waiting to be assigned to a processor.
- **Running:** Instructions are being executed.
- **Waiting:** The process is waiting for some event to occur.
- **Terminated:** The process has finished execution.

Many processes may be in ready and waiting state at the same time. But only one process can

be running on any processor at any instant.



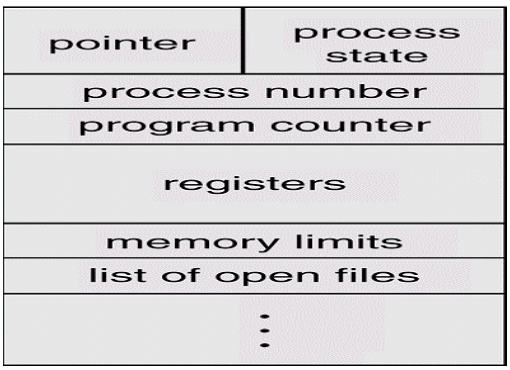
Process scheduling:

Scheduling is a fundamental function of OS. When a computer is multiprogrammed, it has multiple processes completing for the CPU at the same time. If only one CPU is available, then a choice has to be made regarding which process to execute next. This decision making process is known as scheduling and the part of the OS that makes this choice is called a scheduler. The algorithm it uses in making this choice is called scheduling algorithm.

Scheduling queues: As processes enter the system, they are put into a job queue. This queue consists of all process in the system. The process that are residing in main memory and are ready & waiting to execute or kept on a list called ready queue.

Process control block:

Each process is represented in the OS by a process control block. It is also by a process control block. It is also known as task control block.



A process control block contains many pieces of information associated with a specific process. It includes the following informations.

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

Threads :

Applications use concurrent processes to speed up their operation. However, switching between processes within an application incurs high process switching overhead because the size of the process state information is large, so operating system designers developed an alternative model of execution of a program, called a *thread*, that could provide concurrency within an application with less overhead

To understand the notion of threads, let us analyze process switching overhead and see where a saving can be made. Process switching overhead has two components:

- *Execution related overhead:* The CPU state of the running process has to be saved and the CPU state of the new process has to be loaded in the CPU. This overhead is unavoidable.

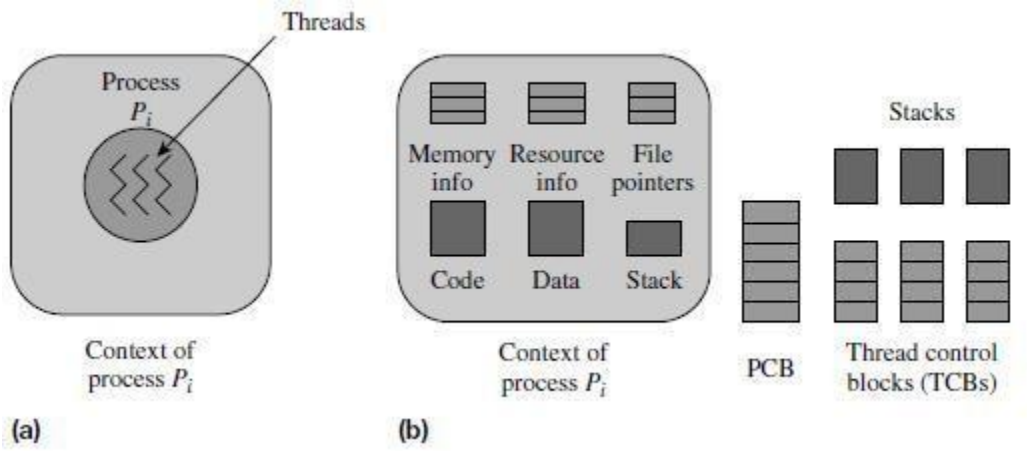
- *Resource-use related overhead:* The process context also has to be switched. It involves switching of the information about resources allocated to the process, such as memory and files, and interaction of the process with other processes. The large size of this information adds to the process switching overhead.

Consider child processes *Pi* and *Pj* of the primary process of an application. These processes inherit the context of their parent process. If none of these processes have allocated any resources of their own, their context is identical; their state information differs only in their CPU states and contents of their stacks. Consequently, while switching between *Pi* and *Pj* ,much of the saving and loading of process state information is redundant. Threads exploit this feature to reduce the switching overhead.

A process creates a thread through a system call. The thread does not have resources of its own, so it does not have a context; it operates by using the context of the process, and accesses the resources of the process through it. We use the phrases ―thread(s) of a process‖ and ―parent process of a thread‖ to describe the relationship between a thread and the process whose context it uses.

Figure illustrates the relationship between threads and processes. In the abstract view of Figure , process*Pi* has three threads,which are represented by wavy lines inside the circle representing process *Pi* . Figure shows an implementation arrangement. Process *Pi* has a context and a PCB. Each thread of *Pi* is an execution of a program, so it has its own stack and a *thread control block* (TCB),which is analogous to the PCB and stores the following information:

1. Thread scheduling information—thread id, priority and state.
2. CPU state, i.e., contents of the PSW and GPRs.
3. Pointer to PCB of parent process.
4. TCB pointer, which is used to make lists of TCBs for scheduling.



POSIX Threads:

POSIX Threads, usually referred to as pthreads, is an execution model that exists independently from a language, as well as a parallel execution model. It allows a program to control multiple different flows of work that overlap in time. Each flow of work is referred to as a *thread*, and creation and control over these flows is achieved by making calls to the POSIX Threads API. POSIX Threads is an API defined by the standard *POSIX.1c, Threads extensions (IEEE Std 1003.1c-1995)*.

Implementations of the API are available on many Unix-like POSIX-conformant operating systems such as FreeBSD, NetBSD, OpenBSD, Linux, Mac OS X, Android[1] and Solaris, typically bundled as a library libpthread. DR-DOS and Microsoft Windows implementations also exist: within the SFU/SUA subsystem which provides a native implementation of a number of POSIX APIs, and also within third-party packages such as *pthreads-w32*,[2] which implements pthreads on top of existing Windows API.

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

void *print_message_function( void *ptr );

main()
{
     pthread_t thread1, thread2;
     char *message1 = "Thread 1";
     char *message2 = "Thread 2";
     int  iret1, iret2;

    /* Create independent threads each of which will execute function */

     iret1 = pthread_create( &thread1, NULL, print_message_function, (void*) message1);
     iret2 = pthread_create( &thread2, NULL, print_message_function, (void*) message2);

     /* Wait till threads are complete before main continues. Unless we  */
     /* wait we run the risk of executing an exit which will terminate   */
     /* the process and all threads before the threads have completed.   */

     pthread_join( thread1, NULL);
     pthread_join( thread2, NULL);

     printf("Thread 1 returns: %d\n",iret1);
     printf("Thread 2 returns: %d\n",iret2);
     exit(0);
}

void *print_message_function( void *ptr )
{
     char *message;
     message = (char *) ptr;
     printf("%s \n", message);
}
```

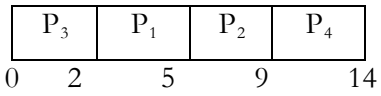
Preemptive Scheduling:

It is the responsibility of CPU scheduler to allot a process to CPU whenever the CPU is in the idle state. The CPU scheduler selects a process from ready queue and allocates the process to CPU. The scheduling which takes place when a process switches from running state to ready state or from waiting state to ready state is called **Preemptive Scheduling**

Shortest Job First Scheduling (SJF) Algorithm: This algorithm associates with each process if the CPU is available. This scheduling is also known as shortest next CPU burst, because the scheduling is done by examining the length of the next CPU burst of the process rather than its total length. Consider the following example:

Process	CPU time
P_1	3
P_2	5
P_3	2
P_4	4

Solution: According to the SJF the Gantt chart will be



The waiting time for process $P_1 = 0$, $P_2 = 2$, $P_3 = 5$, $P_4 = 9$ then the turnaround time for process $P_3 = 0 + 2 = 2$, $P_1 = 2 + 3 = 5$, $P_4 = 5 + 4 = 9$, $P_2 = 9 + 5 = 14$.

Then average waiting time = $(0 + 2 + 5 + 9)/4 = 16/4 = 4$

Average turnaround time = $(2 + 5 + 9 + 14)/4 = 30/4 =$

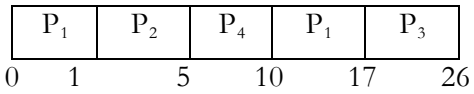
7.5

The SJF algorithm may be either preemptive or non preemptive algorithm. The preemptive SJF is also known as shortest remaining time first.

Consider the following example.

Process	Arrival Time	CPU time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

In this case the Gantt chart will be



The waiting time for

process $P_1 = 10 - 1 = 9$

$P_2 = 1 - 1 = 0$

$P_3 = 17 - 2 = 15$

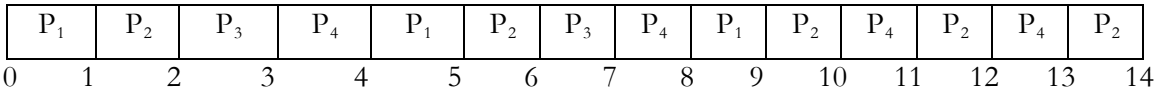
$P_4 = 5 - 3 = 2$

The average waiting time = $(9 + 0 + 15 + 2)/4 = 26/4 = 6.5$

Round Robin Scheduling Algorithm: This type of algorithm is designed only for the time sharing system. It is similar to FCFS scheduling with preemption condition to switch between processes. A small unit of time called quantum time or time slice is used to switch between the processes. The average waiting time under the round robin policy is quiet long. Consider the following example:

Process	**CPU time**
P_1	3
P_2	5
P_3	2
P_4	4

Time Slice = 1 millisecond.



The waiting time for process

$P_1 = 0 + (4 - 1) + (8 - 5) = 0 + 3 + 3 = 6$

$P_2 = 1 + (5 - 2) + (9 - 6) + (11 - 10) + (12 - 11) + (13 - 12) = 1 + 3 + 3 + 1 + 1 + 1 = 10$

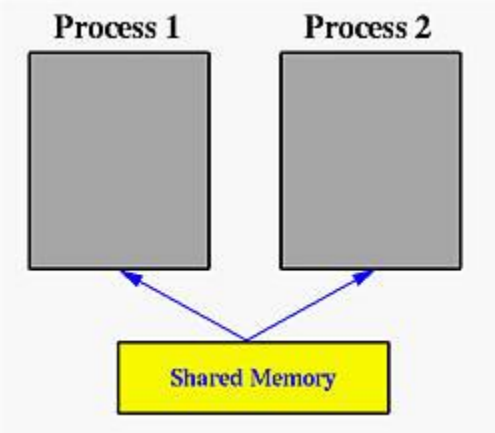
$P_3 = 2 + (6 - 3) = 2 + 3 = 5$

$P_4 = 3 + (7 - 4) + (10 - 8) + (12 - 11) = 3 + 3 + 2 + 1 = 9$

The average waiting time = $(6 + 10 + 5 + 9)/4 = 7.5$

Task Communication :

A ***shared memory*** is an extra piece of memory that is *attached* to some address spaces for their owners to use. As a result, all of these processes share the same memory segment and have access to it. Consequently, race conditions may occur if memory accesses are not handled properly. The following figure shows two processes and their address spaces. The yellow rectangle is a shared memory attached to both address spaces and both process 1 and process 2 can have access to this shared memory as if the shared memory is part of its own address space. In some sense, the original address spaces is "extended" by attaching this shared memory.



Definition - What does *Pipe* mean?

A pipe is a method used to pass information from one program process to another. Unlike other types of inter-process communication, a pipe only offers one-way communication by passing a parameter or output from one process to another. The information that is passed through the pipe is held by the system until it can be read by the receiving process. also known as a **FIFO** for its behavior.

In computing, a named pipe (also known as a **FIFO**) is one of the methods for intern-process communication.

- It is an extension to the traditional pipe concept on Unix. A traditional pipe is "unnamed" and lasts only as long as the process.
- A named pipe, however, can last as long as the system is up, beyond the life of the process. It can be deleted if no longer used.
- Usually a named pipe appears as a file, and generally processes attach to it for inter-process communication. A FIFO file is a special kind of file on the local storage which allows two or more processes to communicate with each other by reading/writing to/from this file.
- A FIFO special file is entered into the filesystem by calling mkfifo() in C. Once we have created a FIFO special file in this way, any process can open it for reading or writing, in the same way as an ordinary file. However, it has to be open at both ends simultaneously before you can proceed to do any input or output operations on it.

Message passing:

Message passing can be **synchronous** or **asynchronous**. Synchronous message passing systems require the sender and receiver to wait for each other while transferring the message. In asynchronous communication the sender and receiver do not wait for each other and can carry on their own computations while transfer of messages is being done.

The advantage to synchronous message passing is that it is conceptually less complex. Synchronous message passing is analogous to a function call in which the message sender is the function caller and the message receiver is the called function. Function calling is easy and familiar. Just as the function caller stops until the called function completes, the sending process stops until the receiving process completes. This alone makes synchronous message unworkable for some applications. For example, if synchronous message passing would be used exclusively, large, distributed systems generally would not perform well enough to be usable. Such large, distributed systems may need to continue to operate while some of their subsystems are down; subsystems may need to go offline for some kind of maintenance, or have times when subsystems are not open to receiving input from other systems.

Message queue:

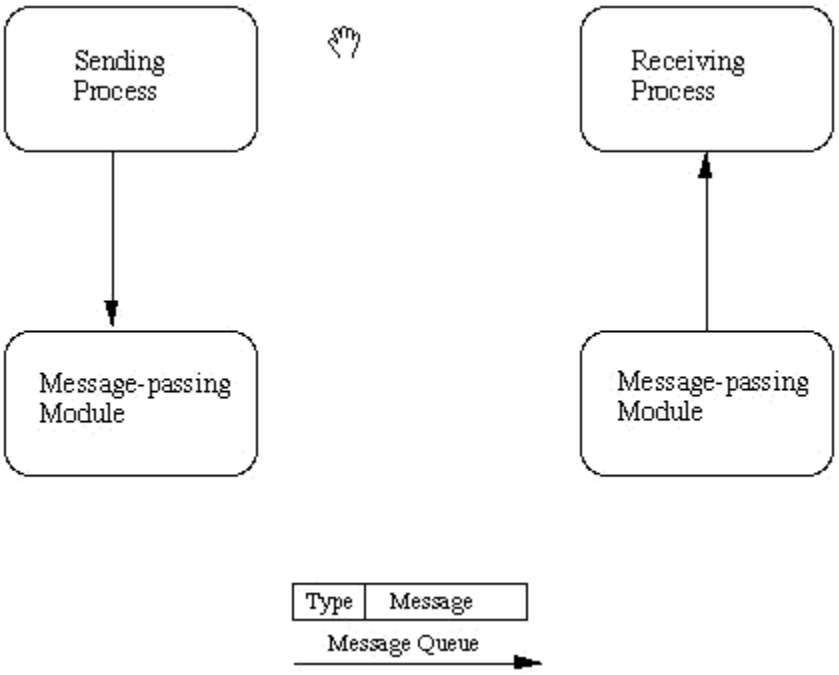
Message queues provide an asynchronous communications protocol, meaning that the sender and receiver of the message do not need to interact with the message queue at the same time. Messages placed onto the queue are stored until the recipient retrieves them. Message queues have implicit or explicit limits on the size of data that may be transmitted in a single message and the number of messages that may remain outstanding on the queue.

Many implementations of message queues function internally: within an operating system or within an application. Such queues exist for the purposes of that system only.[1][2][3]

Other implementations allow the passing of messages between different computer systems, potentially connecting multiple applications and multiple operating systems.[4] These message queueing systems typically provide enhanced resilience functionality to ensure that messages do not get "lost" in the event of a system failure. Examples of commercial implementations of this kind of message queueing software (also known as message-oriented middleware) include IBM WebSphere MQ (formerly MQ Series) and Oracle

Advanced Queuing (AQ). There is a Java standard called Java Message Service, which has several proprietary and free software implementations.

Implementations exist as proprietary software, provided as a service, open source software, or a hardware-based solution.



Mail box:

Mailboxes provide a means of passing messages between tasks for data exchange or task synchronization. For example, assume that a data gathering task that produces data needs to convey the data to a calculation task that consumes the data. This data gathering task can convey the data by placing it in a mailbox and using the SEND command; the calculation task uses RECEIVE to retrieve the data. If the calculation task consumes data faster than the gatherer produces it, the tasks need to be synchronized so that only new data is operated on by the calculation task. Using mailboxes achieves synchronization by forcing the calculation task to wait for new data before it operates. The data producer puts the data in a mailbox and SENDs it. The data consumer task calls RECEIVE to check whether there is new data in the mailbox; if not, RECEIVE calls Pause() to allow other tasks to execute while the consuming task is waiting for the new data.

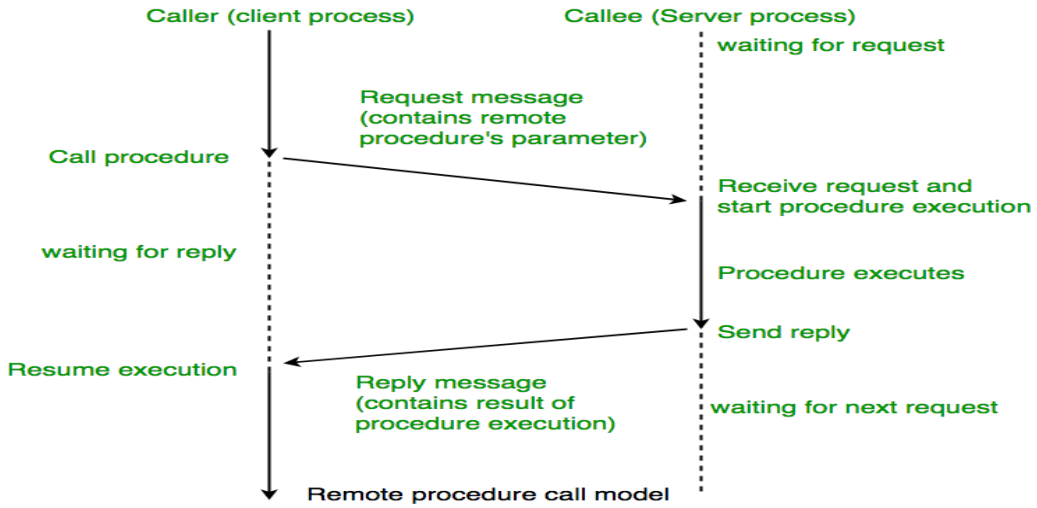
Signaling :

signals are commonly used in POSIX systems. Signals are sent to the current process telling it what it needs to do, such as, shutdown, or that it has committed an exception. A process has several signal-handlers which execute code when a relevant signal is encountered. The ANSI header for these tasks is <signal.h>, which includes routines to allow signals to be raised and read.

Signals are essentially software interrupts. It is possible for a process to ignore most signals, but some cannot be blocked. Some of the common signals are Segmentation Violation (reading or writing memory that does not belong to this process), Illegal Instruction (trying to execute something that is not a proper instruction to the CPU), Halt (stop processing for the moment), Continue (used after a Halt), Terminate (clean up and quit), and Kill (quit now without cleaning up).

RPC:

Remote Procedure Call (RPC) is a powerful technique for constructing **distributed, client-server based applications**. It is based on extending the conventional local procedure calling, so that the **called procedure need not exist in the same address space as the calling procedure**. The two processes may be on the same system, or they may be on different systems with a network connecting them.



The following steps take place during a RPC:

1. A client invokes a **client stub procedure**, passing parameters in the usual way. The client stub resides within the client's own address space.

2. The client stub **marshalls(pack)** the parameters into a message. Marshalling includes converting the representation of the parameters into a standard format, and copying each parameter into the message.

3. The client stub passes the message to the transport layer, which sends it to the remote server machine.

4. On the server, the transport layer passes the message to a server stub, which **demarshalls(unpack)** the parameters and calls the desired server routine using the regular procedure call mechanism.

5. When the server procedure completes, it returns to the server stub **(e.g., via a normal procedure call return)**, which marshalls the return values into a message. The server stub then hands the message to the transport layer.

6. The transport layer sends the result message back to the client transport layer, which hands the message back to the client stub.

7. The client stub demarshalls the return parameters and execution returns to the caller.

Process Synchronization

A co-operation process is one that can affect or be affected by other processes executing in the system. Co-operating process may either directly share a logical address space or be allotted to the shared data only through files. This concurrent access is known as Process synchronization.

Critical Section Problem:

Consider a system consisting of n processes (P_0, P_1, ……P_{n-1}) each process has a segment of code which is known as critical section in which the process may be changing common variable, updating a table, writing a file and so on. The important feature of the system is that when the process is executing in its critical section no other process is to be allowed to execute in its critical section.

The execution of critical sections by the processes is a mutually exclusive. The critical section problem is to design a protocol that the process 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 is followed on exit section. The remaining code is the remainder section.

Example:

```
While (1)
{
Entry Section;
        Critical
Section; Exit
Section;
        Remainder Section;
}
```

A solution to the critical section problem must satisfy the following three conditions.

1. **Mutual Exclusion:** If process P_i is executing in its critical section then no any other process can be executing in their critical section.
2. **Progress:** If no process is executing in its critical section and some process wish to enter their critical sections then only those process that are not executing in their remainder section can enter its critical section next.
3. **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.

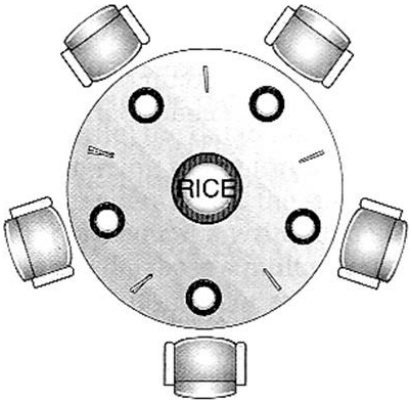
Deadlock:

In a multiprogramming environment several processes may compete for a finite number of resources. A process request resources; if the resource is available at that time a process enters the wait state. Waiting process may never change its state because the resources requested are held by other waiting process. This situation is known as deadlock.

Deadlock Characteristics: In a deadlock process never finish executing and system resources are tied up. A deadlock situation can arise if the following four conditions hold simultaneously in a system.

- **Mutual Exclusion:** At a time only one process can use the resources. If another process requests that resource, requesting process must wait until the resource has been released.
- **Hold and wait:** A process must be holding at least one resource and waiting to additional resource that is currently held by other processes.
- **No Preemption:** Resources allocated to a process can't be forcibly taken out from it unless it releases that resource after completing the task.
- **Circular Wait:** A set $\{P_0, P_1, \ldots\ldots P_n\}$ of waiting state/ process must exists such that P_0 is waiting for a resource that is held by P_1, P_1 is waiting for the resource that is held by P_2 ….. $P_{(n-1)}$ is waiting for the resource that is held by P_n and P_n is waiting for the resources that is held by P_4.

Dining Philosopher Problem: Consider 5 philosophers to spend their lives in thinking & eating. A philosopher shares common circular table surrounded by 5 chairs each occupies by one philosopher. In the center of the table there is a bowl of rice and the table is laid with 6 chopsticks as shown in below figure.



When a philosopher thinks she does not interact with her colleagues. From time to time a philosopher gets hungry and tries to pickup two chopsticks that are closest to her. A philosopher may pickup one chopstick or two chopsticks at a time but she cannot pickup a chopstick that is already in hand of the neighbor. When a hungry philosopher has both her chopsticks at the same time, she eats without releasing her chopsticks. When she finished eating, she puts down both of her chopsticks and starts thinking again. This problem is considered as classic synchronization problem. According to this problem each chopstick is represented by a semaphore. A philosopher grabs the chopsticks by executing the wait operation on that semaphore. She releases the chopsticks by executing the signal operation on the appropriate semaphore

The structure of dining philosopher is as follows:

```
do{
Wait ( chopstick [i]);
Wait (chopstick [(i+1)%5]);
. . . . . .
Eat
. . . . . .
Signal (chopstick [i]);
Signal (chopstick [(i+1)%5]);
. . . . . .
Think
. . . . . .
} While (1);
```

The Integrated Development Environment:

Integrated development environments are designed to maximize programmer productivity by providing tight-knit components with similar user interfaces. IDEs present a single program in which all development is done. This program typically provides many features for authoring, modifying, compiling, deploying and debugging software. This contrasts with software development using unrelated tools, such as vi, GCC or make.

One aim of the IDE is to reduce the configuration necessary to piece together multiple development utilities, instead providing the same set of capabilities as a cohesive unit. Reducing that setup time can increase developer productivity, in cases where learning to use the IDE is faster than manually integrating all of the individual tools. Tighter integration of all development tasks has the potential to improve overall productivity beyond just helping with setup tasks. For example, code can be continuously parsed while it is being edited, providing instant feedback when syntax errors are introduced. That can speed learning a new programming language and its associated libraries.

Some IDEs are dedicated to a specific programming language, allowing a feature set that most closely matches the programming paradigms of the language. However, there are many multiple-language IDEs, such as Eclipse, ActiveState Komodo, IntelliJ IDEA, Oracle JDeveloper, NetBeans, Codenvy and Microsoft Visual Studio. Xcode, Xojo and Delphi are dedicated to a closed language or set of programming languages.

While most modern IDEs are graphical, text-based IDEs such as Turbo Pascal were in popular use before the widespread availability of windowing systems like Microsoft Windows and the X Window System (X11). They commonly use function keys or hotkeys to execute frequently used commands or macros.

A cross compiler is a compiler capable of creating executable code for a platform other than the one on which the compiler is running. For example in order to *compile for* Linux/ARM you first need to obtain its libraries to *compile against*.

A cross compiler is necessary to compile for multiple platforms from one machine. A platform could be infeasible for a compiler to run on, such as for the microcontroller of an embedded system because those systems contain no operating system. In paravirtualization one machine runs many operating systems, and a cross compiler could generate an executable for each of them from one main source.

Cross compilers are not to be confused with a source-to-source compilers. A cross compiler is for cross-platform software development of binary code, while a source-to-source "compiler" just translates from one programming language to another in text code. Both are programming tools.

Uses of cross compilers

The fundamental use of a cross compiler is to separate thebuild environment from target environment. This is useful in a number of situations:

Embedded computers where a device has extremely limited resources. For example, a microwave oven will have an extremely small computer to read its touchpad and door sensor, provide output to a digital display and speaker, and to control the machinery for cooking food. This computer will not be powerful enough to run a compiler, a file system, or a development environment. Since debugging and testing may also require more resources than are available on an embedded system, cross- compilation can be less involved and less prone to errors than native compilation.

Compiling for multiple machines. For example, a company may wish to support several different versions of an operating system or to support several different operating systems. By using a cross compiler, a single build environment can be set up to compile for each of these targets.

Compiling on a server farm. Similar to compiling for multiple machines, a complicated build that involves many compile operations can be executed across any machine that is free, regardless of its underlying hardware or the operating system version that it is running.

Bootstrapping to a new platform. When developing software for a new platform, or the emulator of a future platform, one uses a cross compiler to compile necessary tools such as the operating system and a native compiler.

What is a Disassembler?

In essence, a **disassembler** is the exact opposite of an assembler. Where an assembler converts code written in an assembly language into binary machine code, a disassembler reverses the process and attempts to recreate the assembly code from the binary machine code.

Since most assembly languages have a one-to-one correspondence with underlying machine instructions, the process of disassembly is relatively straight-forward, and a basic disassembler can often be implemented simply by reading in bytes, and performing a table lookup. Of course, disassembly has its own problems and pitfalls, and they are covered later in this chapter.

Many disassemblers have the option to output assembly language instructions in Intel, AT&T, or (occasionally) HLA syntax. Examples in this book will use Intel and AT&T syntax interchangeably. We will typically not use HLA syntax for code examples, but that may change in the future.

Decompilers

Decompilers take the process a step further and actually try to reproduce the code in a high level language. Frequently, this high level language is C, because C is simple and primitive enough to facilitate the decompilation process. Decompilation does have its drawbacks, because lots of data and readability constructs are lost during the original compilation process, and they cannot be reproduced. Since the science of decompilation is still young, and results are "good" but not "great", this page will limit itself to a listing of decompilers, and a general (but brief) discussion of the possibilities of decompilation.

Tools

As with other software, embedded system designers use compilers, assemblers, and debuggers to develop embedded system software. However, they may also use some more specific tools:

For systems using digital signal processing, developers may use a math workbench such as Scilab / Scicos, MATLAB / Simulink, EICASLAB, MathCad, Mathematica,or FlowStone DSP to simulate the mathematics. They might also use libraries for both the host and target which eliminates developing DSP routines as done in DSPnano RTOS.

model based development tool like VisSim lets you create and simulate graphical data flow and UML State chart diagrams of components like digital filters, motor controllers, communication protocol decoding and multi-rate tasks. Interrupt handlers can also be created graphically. After simulation, you can automatically generate C-code to the VisSim RTOS which handles the main control task and preemption of background tasks, as well as automatic setup and programming of on-chip peripherals.

Debugging

Embedded debugging may be performed at different levels, depending on the facilities available. From simplest to most sophisticated they can be roughly grouped into the following areas:

Interactive resident debugging, using the simple shell provided by the embedded operating system (e.g. Forth and Basic)

External debugging using logging or serial port output to trace operation using either a monitor in flash or using a debug server like the Remedy Debugger which even works for heterogeneous multicore systems.

An in-circuit debugger (ICD), a hardware device that connects to the microprocessor via a JTAG or Nexus interface. This allows the operation

of the microprocessor to be controlled externally, but is typically restricted to specific debugging capabilities in the processor.

An in-circuit emulator (ICE) replaces the microprocessor with a simulated equivalent, providing full control over all aspects of the microprocessor.

A complete emulator provides a simulation of all aspects of the hardware, allowing all of it to be controlled and modified, and allowing debugging on a normal PC. The downsides are expense and slow operation, in some cases up to 100X slower than the final system.

For SoC designs, the typical approach is to verify and debug the design on an FPGA prototype board. Tools such as Certus are used to insert probes in the FPGA RTL that make signals available for observation. This is used to debug hardware, firmware and software interactions across multiple FPGA with capabilities similar to a logic analyzer.

Unless restricted to external debugging, the programmer can typically load and run software through the tools, view the code running in the processor, and start or stop its operation. The view of the code may be as HLL source-code, assembly code or mixture of both.

Simulation is the imitation of the operation of a real-world process or system over time.[1] The act of simulating something first requires that a model be developed; this model represents the key characteristics or behaviors/functions of the selected physical or abstract system or process. The model represents the system itself, whereas the simulation represents the operation of the system over time.

Simulation is used in many contexts, such as simulation of technology for performance optimization, safety engineering, testing, training, education, and video games. Often, computer experiments are used to study simulation models.

Key issues in simulation include acquisition of valid source information about the relevant selection of key characteristics and behaviours, the use of simplifying approximations and assumptions within the simulation, and fidelity and validity of the simulation outcomes.

Emulator

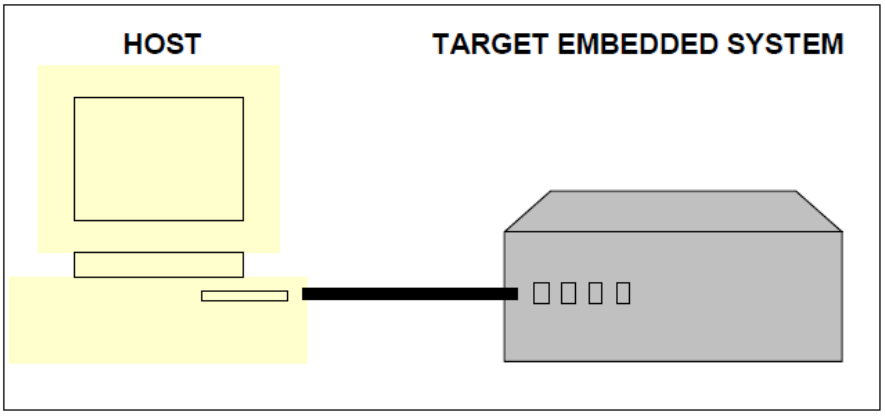
This article is about emulators in computing. For a line of digital musical instruments, see E-mu Emulator. For the Transformers character, see Circuit Breaker (Transformers).#Shattered Glass. For other uses, see Emulation (disambiguation).

DOSBox emulates the command-line interface of DOS.

In computing, an **emulator** is hardware or software or both that duplicates (or *emulates*) the functions of one computer system (the *guest*) in another computer system (the *host*), different from the first one, so that the emulated behavior closely resembles the behavior of the real system (the guest).

The above described focus on exact reproduction of behavior is in contrast to some other forms of computer simulation, in which an abstract model of a system is being simulated. For example, a computer simulation of a hurricane or a chemical reaction is not emulation.

OUT-OF-CIRCUIT : The code to be run on the target embedded system is always developed on the host computer. This code is called the *binary executable image* or simply *hex code*. The process of putting this code in the memory chip of the target embedded system is called Downloading.



There are two ways of downloading the binary image on the embedded system:

1. Using a Device Programmer

A device programmer is a piece of hardware that works in two steps.

Step 1 Once the binary image is ready on the computer, the device programmer is connected to the computer and the binary image is transferred to the device programmer.

Step 2 The microcontroller/microprocessor or memory chip, usually the ROM which is supposed to contain the binary image is placed on the proper socket on the device programmer. The device programmer contains a software interface through which the user selects the target microprocessor for which the binary image has to be downloaded. The Device programmer then transfers the binary image bit by bit to the chip.

2. Using In System Programmer(ISP)

Certain Target embedded platforms contain a piece of hardware called ISP that have a hardware interface to both the computer as well the chip where the code is to be downloaded.

The user through the ISP's software interface sends the binary image to the target board.

This avoids the requirement of frequently removing the microprocessor / microcontroller or ROM for downloading the code if a device programmer had to be used.

DEBUGGING THE EMBEDDED SOFTWARE

- Debugging is the process of eliminating the bugs/errors in software.
- The software written to run on embedded systems may contain errors and hence needs debugging.
- However, the difficulty in case of embedded systems is to find out the bug/ error itself. This is because the binary image you downloaded on the target board was free of syntax errors but

still if the embedded system does not function the way it was supposed to be then it can be either because of a hardware problem or a software problem. Assuming that the hardware is perfect all that remains to check is the software.

- The difficult part here is that once the embedded system starts functioning there is no way for the user or programmer to know the internal state of the components on the target board.

- The most primitive method of debugging is using LEDs. This is similar to using a printf or a cout statement in c/c++ programs to test if the control enters the loop or not. Similarly an LED blind or a pattern of LED blinks can be used to check if the control enters a particular piece of code.

There are other advanced debugging tools like;

a. Remote debugger

b. Emulator

c. Simulator

Remote Debuggers

- Remote Debugger is a tool that can be commonly used for:
- Downloading
- Executing and
- Debugging embedded software
- A Remote Debugger contains a hardware interface between the host computer and the target embedded system.