3. PROCESS CO-ORDINATION

Process Control Block (PCB)

When a process is being executed, the operating system must store information about a process in a PCB. When scheduler switches CPU from executing one process to another, context switcher saves the contents of all processor registers in process descriptor. That is, the context of a process is represented in a PCB of a process. It contains information about a process such as;

- **Pointer**: Pointer points to another process control block. Pointer is used for maintaining the scheduling list.
- **Process State**: Process state may be new, ready, running, waiting and so on.
- **Program Counter**: It indicates the address of the next instruction to be executed for this process.
- **Event information**: For a process in the blocked state this field contains information concerning the event for which the process is waiting.
- **CPU register**: It indicates general purpose registers, stack pointers and index registers. Number of register and type of register totally depends upon the computer architecture.
- **Memory Management Information**: This information may include the value of base and limit register. This information is useful for de-allocating the memory when the process terminates.
- **Accounting Information**: This information includes the amount of CPU and real time used, time limits, job or process numbers etc.

Concurrent Processes

In a single-processor multiprogramming system, processes are interleaved in time to yield the appearance of simultaneous execution. Even though there is a certain amount of overhead involved in switching back and forth between processes, interleaved execution provides benefits in processing efficiency and in program structuring. Concurrent processes (processes executing concurrently) in operating system may either be independent or cooperating processes. A process is **independent** if it cannot affect or be affected by other processes executing in the system. Any process that that does not share data with any other process is independent. A process is **cooperating** if it can affect or be affected by other processes executing in the system. Any

process that shares data with other processes is a cooperating process. The OS supports concurrency for the following reasons;

- **Information sharing**: Several processes may be interested in the same piece of information (e.g. a shared file) and thus concurrent access to such files must be allowed.
- **Computation speedup**: for a particular process to run faster, it is broken into sub-processes (threads) which shall be executed in parallel with each other.
- **Modularity**: System functions are divided into separate functions or threads.
- **Convenience**: Allowing the CPU to work on several tasks at the same time.

Cooperating process usually, however, face two major problems. These problems include;

- **Starvation**: A process is made to wait for a resource indefinitely.
- **Deadlock**: This occurs when two or more processes block waiting an event to occur which is under the control of the other blocking process.

To overcome these two problems then process synchronization must be ensured.

Process Synchronization

When two processes are writing or reading some shared data and the final result depends on the order of execution of instructions, i.e. results depends on who runs precisely when, leads to occurrence of a race condition. Suppose two processes P1 and P2 share a global variable B. At some point in execution, P1 updates B to value 1 and at some point in execution P2 updates B to value 2. The two tasks are in a race to write variable B. In this example, the "loser" of the race (process that updates last) determines the final value of B.

The scheduling algorithm must determine the relative timing of cooperating processes. Code executed by a process can be grouped into sections some of which require access to shared resources and others that do not. Each process has a segment of code, called a Critical Section. The section of the code that requires access to shared resources is called a **Critical section.** It is the section in which a 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 sections at the same time. Each process must request permission to enter its critical section. The section of code implementing this request is the **Entry section.** This is followed by an exit section. The remaining code is the **remainder section.** This can be illustrated as shown below

```
do {
    entry section
        critical section
    exit section
        remainder section
} while (TRUE);
```

To avoid race condition occurring, a mechanism is needed to synchronize execution within critical sections. Mutual exclusion then needs to be satisfied, i.e. when a process is in a critical section that accesses a set of shared resources no other process should be in a critical section at the same time. A process waiting to enter a critical section need to wait a finite time and a process that terminates outside its critical section should not prevent other processes to enter into critical section.

Requirements for mutual exclusion

To provide mutual exclusion, the following requirements must be met;

- Only one process at a time is allowed into its critical section among all processes that have critical sections.
- A process that stops in its non-critical section must do so without interfering with other processes.
- A process requiring access to a critical section should not be delayed indefinitely to starvation or deadlock.
- When no process is in its critical section, any process that request entry must be permitted to enter its critical section without delay.
- A process must remain inside its critical section for a finite time only.

Methods of handling mutual exclusion

- **Use of hardware solution by interrupt disabling**: Each process that is entering in its critical section disables all interrupts before entering in its critical section. With interrupt disabled the CPU cannot be switched to other processes.

- **Use of software solution by using tests and set instructions**: In this method, mutual exclusion is achieved when a process that wants to enter in its critical section first tests a lock. If a lock is 0 it means that no process is in critical process. Then the process sets the lock to 1 and now enters in critical section. If the lock is already 1, the process just wait until the lock variable becomes 0.
- **Use of semaphore**: This was proposed by Dijkstra in 1965. A semaphore is a simple integer variable which can take non-negative values and upon which two operations called wait and signal are defined. Entry into critical section is controlled by wait operation and exit from critical section by signal operation. If a semaphore has a non-zero value this indicates the availability of resource whereas if else is zero, the resource is not available. Assuming S is the variable

  Wait (S)
  If S > 0 // resource is available
  Set S to S – 1
  Else if S = 0 // resource is not available
  //block the calling process (waits)
  Else
  Set S to S + 1 // release the resource
- **Use of monitors**: Semaphore provide general purpose solution for controlling access to critical section but this does not guarantee mutual exclusion on deadlock. A monitor is a programming language construct that guarantees appropriate access to critical sections. This code is placed before and after critical sections to control access to critical section and is generated by the compiler. Inside the monitor is a collection of uninitialized code, shared data objects and functions. Processes that may wish to access shared data object must do so through the execution of monitor functions. A process enters into the monitor by invoking one of its procedures. If a process is executing in the monitor then no other process is allowed to invoke the monitor but it is suspended until the monitor becomes available. **Wait** operation suspends process invokement and the **Signal** operation allows a waiting process to re-enter the monitor at the point its execution was suspended.

Inter-Process Communication

Processes need to communicate with each other for the purposes of information sharing and files are the mechanism for information sharing. Information written to a file by one process can be read by another. This information that can be shared is limited by file capacity of the file system. Many operating system support an explicit system for sending messages between processes.

A **Message** is a collection of data objects consisting of a fixed size header and a variable or constant length body which can be managed by a process and delivered to its destination. There are two fundamental models of inter-process communication, namely; Shared memory and Message passing.

- **Shared memory:** In this model, a region of memory that is shared by cooperating processes is established. Typically, a shared-memory region resides in the address space of the process creating the shared-memory segment. Other processes that wish to communicate using this shared-memory segment must attach it to their address space. Recall that, normally, the operating system tries to prevent one process from accessing another process's memory. Shared memory requires that two or more processes agree to remove this restriction. They can then exchange information by reading and writing data in the shared areas. The form of the data and the location are determined by these processes and are not under the operating system's control. The processes are also responsible for ensuring that they are not writing to the same location simultaneously. Shared memory allows maximum speed and convenience of communication, as it can be done at memory speeds when within a computer
- **Message Passing:** communication takes place by means of messages exchanged between the cooperating processes. Message passing provides a mechanism to allow processes to communicate and to synchronize their actions without sharing the same address space and is particularly useful in a distributed environment, where the communicating processes may reside on different computers connected by a network. A message-passing facility provides at least two operations: send (message) and receive (message). Message passing is useful for exchanging smaller amounts of data, because no conflicts need be avoided. It is also easier to implement than is shared memory for inter-computer communication.