OS (Process Synchronization)

★★Process Synchronization

Process Synchronization was introduced to handle problems that arose while multiple process executions.

Process is categorized into two types on the basis of synchronization and these are given below:

- Independent Process
- Cooperative Process

Independent Processes: Two processes are said to be independent if the execution of one process does not affect the execution of another process.

Cooperative Processes: Two processes are said to be cooperative if the execution of one process affects the execution of another process. These processes need to be synchronized so that the order of execution can be guaranteed.

☆☆Process Synchronization

It is the task phenomenon of coordinating the execution of processes in such a way that no two processes can have access to the same shared data and resources.

- It is a procedure that is involved in order to preserve the appropriate order of execution of cooperative processes.
- In order to synchronize the processes, there are various synchronization mechanisms.
- Process Synchronization is mainly needed in a multi-process system when multiple processes are running together and more than one process try to gain access to the same shared resource or any data at the same time.

☆☆Race Condition

At the time when more than one process is either executing the same code or accessing the same memory or any shared variable; In that condition, there is a possibility that the output or the value of the shared variable is wrong so for that purpose all the processes are **doing the race** to say that my output is correct. This condition is commonly known as a **race condition**. As several processes access and process the manipulations on the same data in a concurrent manner and due to which the outcome depends on the particular order in which the access of data takes place.

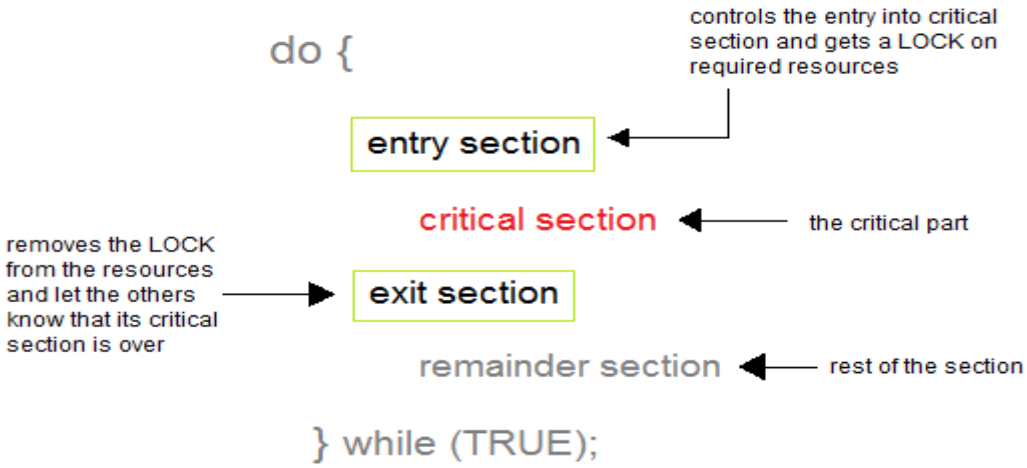
Mainly this condition is a situation that may occur inside the critical section. Race condition in the critical section happens when the result of multiple thread execution differs according to the order in which the threads execute. But this condition is critical sections can be avoided if the critical

section is treated as an atomic instruction. Proper thread synchronization using locks or atomic variables can also prevent race conditions.

★★Critical Section Problem

A **Critical Section** is a part of a Program (or code segment) which tries to access shared resources and has to be executed as an atomic action. It means that in a group of cooperating processes, at a given point of time, only one process must be executing its critical section. If any other process also wants to execute its critical section, it must wait until the first one finishes.

The **entry** to the critical section is mainly handled by wait() function while the **exit** from the critical section is controlled by the signal() function.



Entry Section: In this section mainly the process requests for its entry in the critical section.

Exit Section: This section is followed by the critical section.

☆☆The solution to the Critical Section Problem

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

1. Mutual Exclusion: Out of a group of cooperating processes, only one process can be in its critical section at a given point of time.

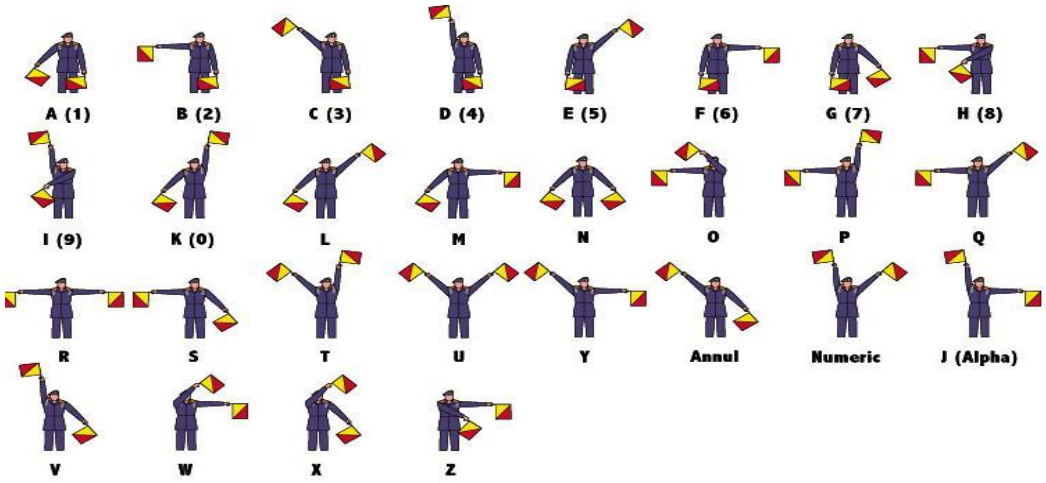
2. Progress: If no process is in its critical section, and if one or more threads want to execute their critical section then any one of these threads must be allowed to get into its critical section.

3. Bounded Waiting: After a process makes a request for getting into its critical section, there is a limit for how many other processes can get into their critical section, before this process's request is granted. So after the limit is reached, the system must grant the process permission to get into its critical section.

★★**Semaphore (**General Sense**)**

In General Sense: A system of sending message by holding the arms/flags in certain positions.

Example: Earlier used in guiding train. In modern age, it is like "traffic-lights".



★★**Semaphore (**In OS**)**

- **Semaphore (S)** is a variable used to control access to a common resource by multiple processes in multitasking operating system.

- It is used to solve the critical section problem by using two atomic operations, wait and signal that are used for process synchronization.

The definitions of wait and signal are as follows −

- **Wait**: The wait operation decrements the value of its argument **S**, if it is positive. If **S** is negative or zero, then no operation is performed. This Operation mainly helps to control the entry of a task into the critical section. In the case of the negative or zero value, no operation is executed.

```
wait(S)
{
  while (S>=0);
  S--;
}
```

- **Signal**: Increments the value of its argument S, as there is no more process blocked on the queue. This Operation is mainly used to control the exit of a task from the critical section.

```
signal(S)
{
S++;
}
```

☆☆Types of Semaphores

There are two main types of semaphores i.e. counting semaphores and binary semaphores. Details about these are given as follows –

- **Counting Semaphores**: These are **integer value** semaphores and have an **unrestricted value domain**. These semaphores are used to coordinate the resource access, where the semaphore count is the number of available resources. If the resources are added, semaphore count automatically incremented and if the resources are removed, the count is decremented.
- **Binary Semaphores**

The binary semaphores are like counting semaphores but their value is restricted to **0 and 1**. The wait operation only works when the semaphore is 1 and the signal operation succeeds when semaphore is 0. It is sometimes easier to implement binary semaphores than counting semaphores.

☆☆Advantages of Semaphores

Some of the advantages of semaphores are as follows –

- Semaphores allow only one process into the critical section. They follow the mutual exclusion principle strictly and are much more efficient than some other methods of synchronization.
- There is no resource wastage because of busy waiting in semaphores as processor time is not wasted unnecessarily to check if a condition is fulfilled to allow a process to access the critical section.
- Semaphores are implemented in the machine independent code of the microkernel. So they are machine independent.

☆☆Disadvantages of Semaphores

Some of the disadvantages of semaphores are as follows –

- Semaphores are **complicated** so the wait and signal operations must be implemented in the correct order to prevent deadlocks.
- Semaphores are impractical for last scale use as their use leads to loss of modularity. This happens because the wait and signal operations prevent the creation of a structured layout for the system.
- Semaphores may lead to a priority inversion where low priority processes may access the critical section first and high priority processes later.

★★Classical Problems of Synchronization

In this section, we present some of the **classical synchronization problems.** These problems are used **for testing** nearly every **newly proposed synchronization scheme**.

We will discuss the following **three problems**:

1. Bounded Buffer (Producer-Consumer) Problem
2. The Readers Writers Problem
3. Dining Philosophers Problem

☆☆Bounded Buffer Problem

Because the buffer pool has a maximum size, this problem is often called the **Bounded buffer problem**.

- This problem is generalized in terms of the **Producer Consumer problem**, where a **finite** buffer pool is used to exchange messages between producer and consumer processes.
- **Solution** to this problem is, creating two counting semaphores "full" and "empty" to keep track of the current number of full and empty buffers respectively.
- In this method, Producers mainly produces **a product** and consumers consume the **product**, but both can use of one of the containers each time.
- The main complexity of this problem is that we must have to maintain the count for both empty and full containers that are available.

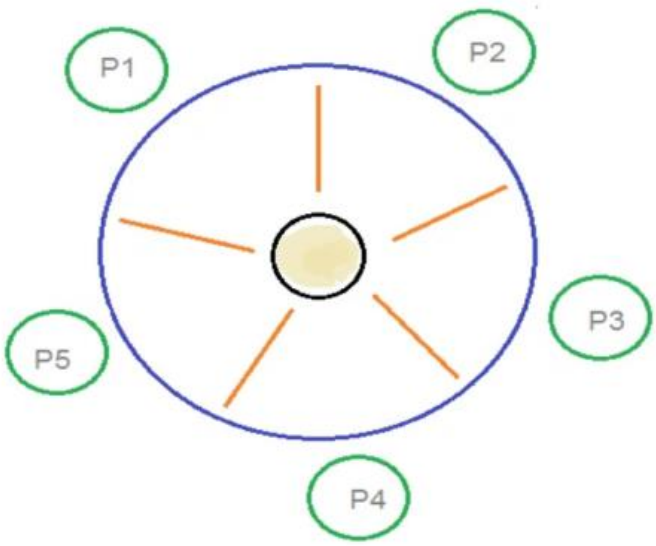
☆☆The Readers Writers Problem

- In this problem there are some processes(called **readers**) that only read the shared data, and never change it, and there are other processes(called **writers**) who may change the data in addition to reading, or instead of reading it.
- There are various type of readers-writers problem, most centered on relative priorities of readers and writers.
- The main complexity with this problem occurs from allowing more than one reader to access the data at the same time.

☆☆**Dining Philosophers Problem**

- The dining philosopher's problem involves the allocation of limited resources to a group of processes in a deadlock-free and starvation-free manner.

- There are five philosophers sitting around a table, in which there are five chopsticks/forks kept beside them and a bowl of rice in the center, when a philosopher wants to eat, he uses two chopsticks - one from their left and one from their right. When a philosopher wants to think, he keeps down both chopsticks at their original place.



The possible solutions for this are:

- A philosopher must be allowed to pick up the chopsticks only if both the left and right chopsticks are available.

- Allow only four philosophers to sit at the table. That way, if all the four philosophers pick up four chopsticks, there will be one chopstick left on the table. So, one philosopher can start eating and eventually, two chopsticks will be available. In this way, deadlocks can be avoided.

★★Monitor in OS

Monitor in OS (operating system) is **a synchronization construct** that enables multiple processes or threads to coordinate actions and ensures that they are not interfering with each other or producing unexpected results. Also, it ensures that only one thread is executed at a critical code section.

☆☆Why are Monitors Used?

Monitors in operating systems are used to manage access to shared resources, like files or data, among multiple processes. They ensure that only one process can use a resource simultaneously, preventing conflicts and data corruption. Monitors simplify synchronization and protect data integrity, making it easier for programmers to create reliable software.

They serve as "guards" for critical code sections, ensuring that no two processes can enter them simultaneously. Monitors are like traffic lights that control access to resources, preventing crashes and ensuring a smooth flow of data and tasks in an operating system.

☆☆Syntax and Pseudo-code of Monitor in OS

Below is the pseudo-code of the monitor in OS:

```
monitor monitor_name{

    //declaring shared variables

  variables_declaration;
  condition_variables;

  procedure p1(...) {
   ...
  };

  procedure p2(...){
    ...
  };
  ...
  procedure pn(...){
    ..
  };

  {
    Initailisation Code();
  }

}
```

In the above pseudo code, **monitor_name** is the name of the monitor, and **p1, p2, pn** are the procedures that provide access to shared data and variables enclosed in monitors. They help ensure that only one thread is accessed at a time for using the shared resource. This help prevents race condition and synchronization problems.

☆☆Advantages of Monitors in OS

There are various advantages of a monitor in OS, some of which are mentioned below:

- **Mutual Exclusion:** It is automatic in monitors. It ensures that at a time, only one thread can access the shared resource.
- **Modularity**: Monitors are helpful for encapsulating individual modules of an extensive system, which helps in maintenance.
- **Parallel programming:** Monitors help in making parallel programming easy.
- **Less error-prone:** Monitors are less prone to errors as compared to semaphores. A semaphore can be defined as an integer variable. It is shared among various processes.

☆☆Disadvantages of Monitors in OS

While monitors can be helpful in some situations, it also has some disadvantages. Some are listed below:

- **Difficult debugging:** Debugging programs that use monitors can be tedious.
- **Single process limitation**: Monitors are implemented within a thread or single process. This also means that synchronization access to shared resources is not possible across multiple threads.