**Set 2 Question 1:** Please explain, in details, how semaphores work and what is the primary purpose of semaphores? Enumerate and differentiate with other mechanisms that provide similar functionalities

**Answer 1:** Semaphores are used for process synchronization to control number of processes simultaneously in the critical section. Semaphores are proposed by Dijkstra for managing concurrent processes by using the value of a simple integer variable to synchronize the progress of interacting processes.

**Working of Semaphores –** Semaphores are integer variables accessed only through two low standard atomic operations, **wait** and **signal** designated by P(S)  and V(S)  respectively. These 2 operations must be atomic so no preemption can happen. The semaphore is a variable that can hold only a non-negative Integer value, shared between all the threads, with operations **wait** and **signal**, which work as follow:

P(S): if S >= 1 then S = S - 1;

else <block and enqueue the process> ;

**Wait**: This operation decrements the value of its argument S, if it is greater than or equal to zero, and allows the process to enter into critical section. In the case of the negative or zero value, no operation is executed and the process is blocked. Hence, the process is not allowed to enter into the critical section controlling the number of processes in the critical section. As this operation can block the process to enter into critical section so it is called Wait operation.

V(S): if <some process is blocked on the queue> then <unblock a process>

else S = S + 1;

**Signal:** This operation increments the value of its argument S, as there is no more process blocked on the queue. If there are processes still in the blocked queue then it allows one of the blocked processes to enter into the critical section. These processes entered into blocked queue while they are trying to enter in the critical section but blocked by wait signal. As this operation allows the process into critical section so it is called Signal operation.

Semaphores are of 2 types –

1. **Binary Semaphores -** It is the type of semaphore used for implementing mutual exclusion, hence it is often called a **mutex**. A binary semaphore is initialized to 1 and only takes the values 0 and 1 during the execution of a program. In Binary Semaphore, the wait operation works only if the value of semaphore = 1, and the signal operation succeeds when the semaphore= 0. Binary Semaphores are easier to implement than counting semaphores.
2. **Counting Semaphores-** These are used to implement **bounded concurrency**. The Counting semaphores can range over an **unrestricted domain**. These can be used to control access to a given resource that consists of a finite number of Instances. Here the semaphore count is used to indicate the number of available resources. If the resources are added then the semaphore count automatically gets incremented and if the resources are removed, the count is decremented. Counting Semaphore has no mutual exclusion.

Both types of semaphores work in a similar manner, before entering into critical section of code wait operation is performed and after leaving critical section signal operation is performed, as shown below :

**while(true){**

**Wait(S)**

**Critical Section of Code**

**Signal(S)**

**Program Code**

**}**

**Other mechanisms that provide process synchronization – Busy waiting synchronization mechanisms**

**Busy waiting,** also known as spinning, or busy looping is a process synchronization technique**in which a process/task waits and constantly checks for a condition to be satisfied before proceeding with its execution**. In busy waiting, a process executes instructions that test for the entry condition to be true, such as the availability of a lock or resource in the computer system.

**Busy waiting can be inefficient because the looping procedure is a waste of computer resources**. In addition, the system is left idle while waiting. This is particularly wasteful if the task/process at hand is of low priority. In that case, resources that can be diverted to complete high-priority tasks are hogged by a low-priority task in busy waiting. Examples of busy waiting synchronization mechanisms are TSL (Test Set Lock) and Peterson’s Solution.

**Differences between Semaphores and busy waiting synchronization mechanisms –**

* Semaphores use sleep function call but busy waiting solutions do not use it.
* Semaphores are efficient but busy waiting solutions are inefficient. 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. But in busy waiting solutions same if condition is evaluated repeatedly which is wastage of computer resources.
* Semaphores are complicated so the wait and signal operations must be implemented in the correct order to prevent deadlocks. Peterson’s solution, which is a busy waiting mechanism, is deadlock free.
* Busy-waiting is preferable when scheduling overhead is larger than expected wait time. If the scheduling overhead of processor is low then semaphores are preferred.
* Semaphores may lead to a priority inversion where low priority processes may access the critical section first and high priority processes later.[[1]](#endnote-2)

1. https://www.tutorialspoint.com/semaphores-in-operating-system [↑](#endnote-ref-2)