**Please explain, in detail, how semaphores work and what is the primary purpose of semaphores? Enumerate and differentiate with other mechanisms that supply similar functionalities.**

Semaphore was devised by Dijkstra in 1962-63. Semaphore is a synchronizing tool, which allows application programs to deal with critical section problems and synchronizing. Semaphore is of two types, binary and counting semaphore. Binary semaphore is a special type of counting semaphore with value restricted to 0 and 1. Whereas counting semaphore can have unrestricted values. Semaphore was also implemented in two forms one is busy waiting and the other in without busy waiting. We will be discussing the semaphore without busy waiting since it will be more efficient and used. In semaphore three operations are allowed initialization, signal and wait[[1]](#endnote-2).

Semaphore is mainly used for two reasons:

1. Can be used to control a given number of resource instances among multiple processes. The semaphore can be initialized with number of resource instances available. Then every process can perform a wait operation on the semaphore to get resources and signal to release resources. If all the resource instances are acquired, then the value of the semaphore will be zero and the next process must wait for any of the processes which have resources to release it.
2. Semaphore started as a general synchronizing tool. Rather than a mutex kind of usage. For example, if we want to execute B in Pj only after A in Pi. We can do so by initializing semaphore to 0 and by following the code:

Process I Process j

A wait(flag)

signal(flag) B

**Working of Semaphore**

To implement a semaphore, we define a structure as:

Struct semaphore {int value; struct process \*list;}

Each semaphore has a value and list of waiting processes pid. It compromises of two operations wait(S) and signal(S). Let’s look at them now.

wait (semaphore \*S) {

S->value--. f(S->value < 0) {add this process to S->list and block ()}

}

signal (semaphore \*S) { S→value++; if(S->value <= 0) {remove the process P from S->list and wakeup(P)} }

The list can be any ordered Data structure but to ensure bounded waiting we use FIFO (first in first out) queue. But semaphore doesn’t define what data structure we need to use.

Now let’s focus on working. If the value of the semaphore is negative it implies number of processes waiting in the queue for resources. And if the value of the semaphore is positive it implies number of available resources. If the value of the semaphore is zero, then there are no available resources left and no one is in the queue as well. For keeping the queue, we can use Process control block of the processes. One of the important concerns about semaphore is that they need to be executed atomically. To do so in a single core system we can simply disable the interrupt. We can’t roll back in operating systems like we can do in Databases. Either we must execute completely or not execute it at all there is no system of roll back in Operating systems. For multi core architecture if we disable interrupt in all cores, it will be a significant loss to the throughput. In Symmetric Multiprocessing system we need to supply alternate locking techniques like spin locks to make sure operations in semaphore run atomically. By this we moved busy waiting from entry section to critical section. So, it is important to accept that busy waiting is not all together removed but performance will be improved compared to the semaphore with busy waiting solution. One important thing to note is deadlock may be possible if semaphore is not used properly.

**Let’s explore other mechanisms that supply similar functionalities.**

1. **The Big Lock-** It will lock the kernel so only one thread will run at a time. It is not suitable for the desktop and higher power CPU, but this is suitable for IoT (internet of things) devices, where very few times process requires kernel, and this makes sense[[2]](#endnote-3). It can’t provide synchronization like semaphore.
2. **Hardware Solutions-** Test\_and\_set lock and swap lock are provided by hardware for the locking mechanism. Some hardware also supply compare\_and\_exchange lock. Test\_and\_set and swap lock are a bounded waiting solution whereas semaphores are not busy waiting solutions. Because once the lock is freed anyone can take it. We can change this scenario by keeping a queue for the process and then bounded waiting will be ensured to some extent. Also, there may be a chance of deadlock. This can provide synchronization to one instance of resource not multiple instance of resources. Multiple locks are needed to ensure synchronization for multiple process and multiple instance of resources. Whereas semaphore need one.
3. **Spin Lock-** It is a busy waiting solution where the lock variable can take only Boolean values. It is basically a mutex with busy waiting. It is used for a short duration and has exceptionally low overhead. It can be implemented using test\_and\_set lock as follows:

void spin\_lock (spinlock\_t \*lock) { while(test\_and\_set(&(lock->value), LOCKED) ==LOCKED); }

void spin\_unlock(spinlock\_t \*lock) { assert(test\_and\_set(&(lock->value), UNLOCKED)==LOCKED); }

1. **Mutex-** It is a binary semaphore. It is used when the duration of locking period is high, and process can sleep instead of busy waiting. The difference between semaphore and mutex is that one mutex can’t provide synchronization to many resources with multiprocessor environment whereas semaphore can do that.
2. **Read Copy Update (RCU) [[3]](#endnote-4)-** It is an alternative of the lock and used for synchronization. Here whenever a process tries to write/update all other processes must see that that and update the changes in their own data structure to avoid inconsistencies. This is an example of space time tradeoff. We are using more space and reducing time. This will supply a situation where there is no need for synchronization. It is fast but maintaining updates is difficult. Semaphore don’t need a separate copy of data structure whereas read copy update need. After an update in RCU update need to be propagated whereas in semaphore there is no such case.
3. **Software Solutions-**  There are algorithms like Peterson algorithms and Bakery algorithms which allow synchronization and don't depend on the hardware and can be extended to multiple processes as well. Semaphore internally require hardware support in form of spin lock whereas software solutions didn’t. Software solutions are completely architectural neutral whereas semaphore are not in some sense.
4. **Monitor-** It is a module that has shared variables, condition and procedure. Condition works on wait and signal. At one time only one process is allowed inside the module. To change the shared memory processor, you need to do so by using procedures. After the work is done the entered process will do a signal and next process in the waiting queue will get a chance to enter in the module and make changes. All the changes are to be made via procedure. Semaphore does not have procedure whereas monitor has and shared variable can be modified using that only.

1. OPERATING SYSTEM PRINCIPLES, 7TH ED, Abraham Silberschatz, Peter Baer Galvin, Greg Gagne [↑](#endnote-ref-2)
2. Class Lectures by Dr. Manish Shrivastava [↑](#endnote-ref-3)
3. https://en.wikipedia.org/wiki/Read-copy-update [↑](#endnote-ref-4)