Semaphores Solutions in Operating System

A **Semaphore** can be described as an object that consists of a counter, a waiting list of processes, Signal and Wait functions. The most basic use of semaphore is to initialize it to 1. When a thread wants to enter a critical section, it calls down and enter the section. When another thread tries to do the same thing, the operation system will put it to the sleep because the value of semaphore is already zero due to previous call to down. When first thread is finished with the critical section, it calls up, which wakes up the other thread that's waiting to enter.
Logically semaphore S is an integer variable that, apart from initialization can only be accessed through two atomic operations:

- **Wait(S) or P :** If the semaphore value is greater than 0, decrement the value. Otherwise, wait until the value is greater than 0 and then decrement it.
- **Signal(S) or V :** Increment the value of Semaphore

Semaphore Solution with Busy Waiting :
If a process is in critical section, the other process that tries to enter its critical section must loop continuously in the entry code. The classic definitions for wait and signal are –

```
wait (S) {
while (S<=0);
S--;
}
signal (S) {
S++;
}
```

Implementing Semaphores : Critical Section of n process problems

```
Shared Data : semaphore mutex ;   // initially mutex=1
Process P :
do {
wait (mutex) ;
<critical section>
signal (mutex) ;
<remainder section>
} while (1)
```

Semaphore Solution with Block and Wake up :

In Busy-wait problem processes waste CPU cycles while waiting to enter their critical sections. Modify wait operation into the block operation. The process can block itself rather than bus-waiting. Place the process into wait queue associated with the critical section. Modify signal operation into the wakeup operation. Change the state of process from wait to ready.
When process executes wait operation and finds that semaphore value is not positive, the process can block itself. Block operation places the process in waiting queue associated with semaphore. A process that is blocked waiting on a semaphore should be restarted when the other process executes signal operation. The blocked process should be restarted by a wakeup operation which put that process in ready queue.

To implement semaphore we define the Semaphore as record :

```
typedef struct {
int value ;
struct process *L ;
} semaphore ;
```

Assume two operations :

- **block :** suspends the process that invokes it.
- **wakeup (P) :** resumes the execution of the blocked process (P)

Semaphore operations defined as :

```
wait (S) {
S.value -- ;
if ( S.value < 0 ) {
add this process to S.L ;
block ;
}  }
signal (S) {
S.value ++ ;
if ( S.value <= 0 ) {
removes a process P from S.L ;
wakeup (P) ;
} }
```

Advantages :

- Semaphores are simple to implement and machine independent
- Corrections is easy to determine
- Semaphores acquire many resources simultaneously
- Can have many different critical sections with different semaphores
- No waste of resources due to busy waiting

Drawbacks :

- Access to semaphores can come from anywhere in a program
- There is no linguistic connection between the semaphore and data to which the semaphore control access
- improper use of the semaphores will lead to deadlock or starvation
- There is no control or guarantee of proper usage

Busy Waiting:

- Busy waiting is a problem in multiprogramming systems where processes constantly check to see if a semaphore is zero. This continual looping wastes CPU cycles that other processes could use productively.
- Busy-wait semaphores are appropriate in multiprocessor systems where the waiting process has its own processor and little contention.
- In a busy-waiting process, a process is blocked and placed on a waiting queue where it does not consume resources. Once the conditions are satisfied, the process is restarted and placed on a ready queue.
- Busy-waiting is also called a spinlock because the process spins while waiting for the lock. A spinlock keeps checking the lock, while a mutex puts threads waiting for the lock into sleep (blocked). A busy-waiting thread wastes CPU cycles, while a blocked thread does not.

  https://www.geeksforgeeks.org/busy-waiting-in-os/