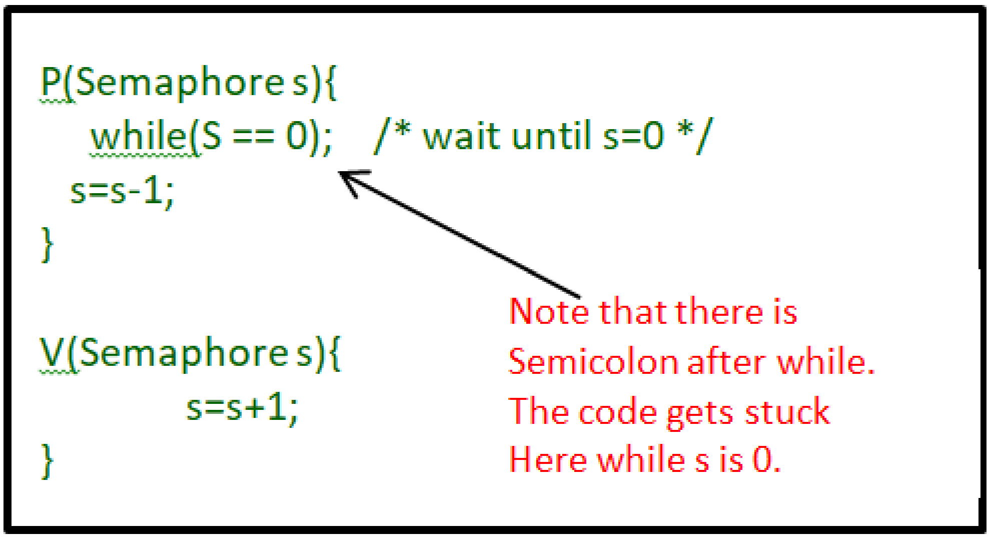
Semaphores in Process Synchronization

Semaphores are just normal variables used to coordinate the activities of multiple processes in a computer system. They are used to enforce mutual exclusion, avoid race conditions, and implement synchronization between processes.
The process of using Semaphores provides two operations: wait (P) and signal (V). The wait operation decrements the value of the semaphore, and the signal operation increments the value of the semaphore. When the value of the semaphore is zero, any process that performs a wait operation will be blocked until another process performs a signal operation.
Semaphores are used to implement critical sections, which are regions of code that must be executed by only one process at a time. By using semaphores, processes can coordinate access to shared resources, such as shared memory or I/O devices.
A semaphore is a special kind of synchronization data that can be used only through specific synchronization primitives. When a process performs a wait operation on a semaphore, the operation checks whether the value of the semaphore is >0. If so, it decrements the value of the semaphore and lets the process continue its execution; otherwise, it blocks the process on the semaphore. A signal operation on a semaphore activates a process blocked on the semaphore if any, or increments the value of the semaphore by 1. Due to these semantics, semaphores are also called counting semaphores. The initial value of a semaphore determines how many processes can get past the wait operation.
Semaphores are of two types:

1. **Binary Semaphore –**
   This is also known as a mutex lock. It can have only two values – 0 and 1. Its value is initialized to 1. It is used to implement the solution of critical section problems with multiple processes.
2. **Counting Semaphore –**
   Its value can range over an unrestricted domain. It is used to control access to a resource that has multiple instances.

Now let us see how it does so.
First, look at two operations that can be used to access and change the value of the semaphore variable.



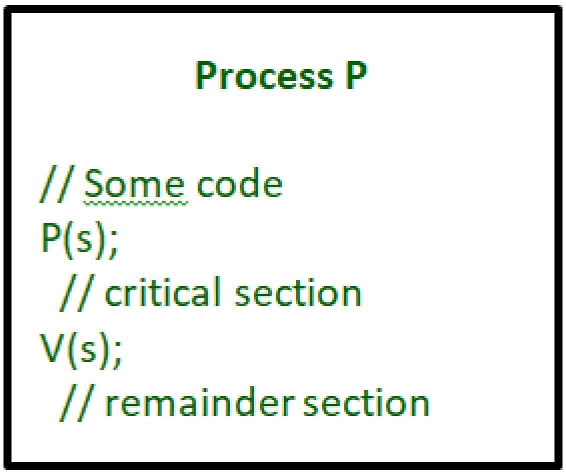
```
P(Semaphore s){
    while(S == 0);    /* wait until s=0 */
   s=s-1;
}

V(Semaphore s){
          s=s+1;
}
```

Note that there is
Semicolon after while.
The code gets stuck
Here while s is 0.

Some points regarding P and V operation:

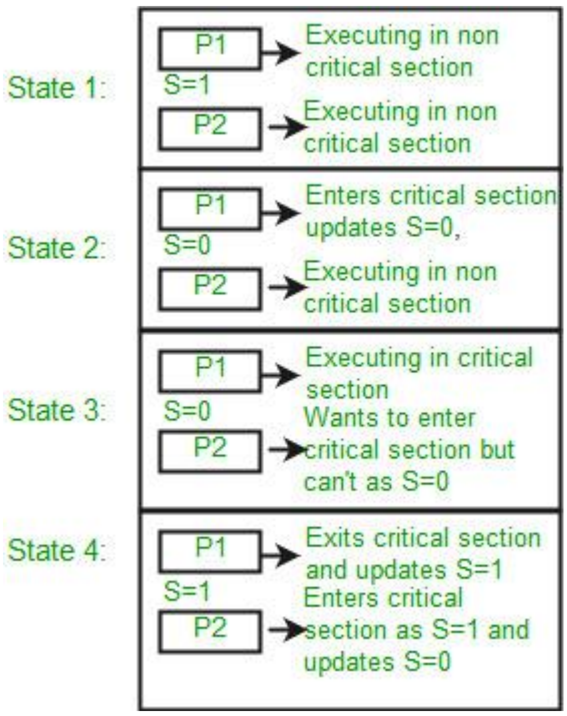
1. P operation is also called wait, sleep, or down operation, and V operation is also called signal, wake-up, or up operation.
2. Both operations are atomic and semaphore(s) is always initialized to one. Here atomic means that variable on which read, modify and update happens at the same time/moment with no pre-emption i.e. in-between read, modify and update no other operation is performed that may change the variable.
3. A critical section is surrounded by both operations to implement process synchronization. See the below image. The critical section of Process P is in between P and V operation.



Process P

```
// Some code
P(s);
  // critical section
V(s);
  // remainder section
```

Now, let us see how it implements mutual exclusion. Let there be two processes P1 and P2 and a semaphore s is initialized as 1. Now if suppose P1 enters in its critical section then the value of semaphore s becomes 0. Now if P2 wants to enter its critical section then it will wait until s > 0, this can only happen when P1 finishes its critical section and calls V operation on semaphore s.
This way mutual exclusion is achieved. Look at the below image for details which is a Binary semaphore.


State 1:
P1
S=1
P2
Executing in non critical section
Executing in non critical section
State 2:
P1
S=0
P2
Enters critical section updates S=0,
Executing in non critical section
State 3:
P1
S=0
P2
Executing in critical section
Wants to enter critical section but can't as S=0
State 4:
P1
S=1
P2
Exits critical section and updates S=1
Enters critical section as S=1 and updates S=0

```
struct semaphore {

    enum value(0, 1);

    // q contains all Process Control Blocks (PCBs)
    // corresponding to processes got blocked
    // while performing down operation.
    Queue<process> q;

};
P(semaphore s)
{
    if (s.value == 1) {
        s.value = 0;
    }
    else {
        // add the process to the waiting queue
        q.push(P) sleep();
    }
}
V(Semaphore s)
{
    if (s.q is empty) {
        s.value = 1;
    }
    else {
        // select a process from waiting queue
        Process p = q.front();
        // remove the process from waiting as it has been
        // sent for CS
        q.pop();
        wakeup(p);
    }
}
```

The description above is for binary semaphore which can take only two values 0 and 1 and ensure mutual exclusion. There is one other type of semaphore called counting semaphore which can take values greater than one.
Now suppose there is a resource whose number of instances is 4. Now we initialize S = 4 and the rest is the same as for binary semaphore. Whenever the process wants that resource it calls P or waits for function and when it is done it calls V or signal function. If the value of S becomes zero then a process has to wait until S becomes positive. For example, Suppose there are 4 processes P1, P2, P3, P4, and they all call wait operation on S(initialized with 4). If another process P5 wants the resource then it should wait until one of the four processes calls the signal function and the value of semaphore becomes positive.
Limitations :

1. One of the biggest limitations of semaphore is priority inversion.
2. Deadlock, suppose a process is trying to wake up another process that is not in a sleep state. Therefore, a deadlock may block indefinitely.
3. The operating system has to keep track of all calls to wait and signal the semaphore.

Problem in this implementation of a semaphore:
The main problem with semaphores is that they require busy waiting, If a process is in the critical section, then other processes trying to enter the critical section will be waiting until the critical section is not occupied by any process. Whenever any process waits then it continuously checks for semaphore value (look at this line while (s==0); in P operation) and waste CPU cycle.
There is also a chance of “spinlock” as the processes keep on spins while waiting for the lock. In order to avoid this another implementation is provided below.
Implementation: Counting semaphore

```
struct Semaphore {

    int value;

    // q contains all Process Control Blocks(PCBs)
    // corresponding to processes got blocked
    // while performing down operation.
    Queue<process> q;

};
P(Semaphore s)
{
    s.value = s.value - 1;
    if (s.value < 0) {

        // add process to queue
        // here p is a process which is currently executing
        q.push(p);
        block();
    }
    else
        return;
}

V(Semaphore s)
{
    s.value = s.value + 1;
    if (s.value <= 0) {

        // remove process p from queue
        Process p = q.pop();
        wakeup(p);
    }
    else
        return;
}
```

In this implementation whenever the process waits it is added to a waiting queue of processes associated with that semaphore. This is done through the system call block() on that process. When a process is completed it calls the signal function and one process in the queue is resumed. It uses the wakeup() system call.

Advantages of Semaphores:

- A simple and effective mechanism for process synchronization
- Supports coordination between multiple processes
- Provides a flexible and robust way to manage shared resources.
- It can be used to implement critical sections in a program.
- It can be used to avoid race conditions.

Disadvantages of Semaphores:

- It Can lead to performance degradation due to overhead associated with wait and signal operations.
- Can result in deadlock if used incorrectly.
- It was proposed by Dijkstra in 1965 which is a very significant technique to manage concurrent processes by using a simple integer

value, which is known as a semaphore. A semaphore is simply an integer variable that is shared between threads. This variable is used to solve the critical section problem and to achieve process synchronization in the multiprocessing environment.

- It can cause performance issues in a program if not used properly.
- It can be difficult to debug and maintain.
- It can be prone to race conditions and other synchronization problems if not used correctly.
- It can be vulnerable to certain types of attacks, such as denial of service attacks.