Semaphore

Semaphore

As it is difficult for the application programmer to use these hardware instructions, to overcome this difficulty we use the synchronization tool called Semaphore (that does not require busy waiting).Semaphore is a type of flexible or non-flexible data that is used to control access to common shared resources through multiple processes in the same system as a multitasking operating system.
It is a mechanism that can be used to provide synchronization of tasks. It is a low level synchronization mechanism.

A semaphore is an object that consists of a counter, a waiting list of processes, and two methods: signal and wait.All the modifications to the integer value of the semaphore in the wait() and signal() atomic operations must be executed indivisibly. i.e. when one process changes the semaphore value, no other process will change the same semaphore value simultaneously.

When the count for the semaphore is 0 means that all the resources are being used by some processes. Otherwise resources are available for the processes to allocate .
When a process is currently using a resource means that it blocks the resource until the count becomes > 0.
For example, let us consider two processes P1 and P2 and a semaphore initialized with a value of 1. The value of the semaphore becomes 0 if the process P1 enters the critical section. If the process P2 intends to enter the critical section then the value of the semaphore has to be greater than 0, until that the process has to wait. This is only possible when P1 completes the critical section and calls the Signal operation on the semaphore. Mutual exclusion is obtained this way.

Operation in Semaphore

1. **Wait**
2. **Signal**

Wait
This operation is used to control the entry of a task into the critical section.
The process that wishes to use a resource must perform the wait( ) operation (count is decremented) .

The definition for wait() is as follows:

```
wait (S)
{
      while S <= 0 ; // no-op
      S--;
}
```

Signal
This operation is used to control the exit of a task from a critical section. This increases the value of the argument by 1.
The process that releases a resource must perform the signal() operation (count is incremented) .

The definition for signal() is as follows:

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

Types of semaphore

1. **Counting Semaphore**
2. **Binary Semaphore**

Counting semaphore
The value of the Counting Semaphore can range over an unrestricted domain. Counting Semaphores are used to control the access of given resources each of which consists of some finite no. of instances. This counting semaphore is initialized to the number of resources available.

Binary semaphore
The value of the Binary Semaphore can range between 0 and 1 only.
In some systems the Binary Semaphore is called as Mutex locks, because they are locks to provide mutual exclusion. We can use the Binary Semaphore to deal with critical section problems for multiple processes.

Implementation of semaphore

The main disadvantage of the semaphore definition is, it requires the busy waiting. Because when one process is in the critical section and if another process needs to enter into the critical section must have to loop in the entry code continuously.

Implementation of semaphore with no busy waiting:
*****To overcome the need of the busy waiting we have to modify the definition of wait() and signal() operations. i.e. when a process executes wait() operation and finds that it is not positive then it must wait.
***** Instead of engaging the busy wait, the process blocks itself so that there will be a chance for the CPU to select another process for execution. It is done by block() operation. - Blocked processes are placed in the waiting queue.
*****Later the process that has already been blocked by itself is restarted by using wakeup() operation, so that the process will move from waiting state to ready state. - Blocked processes that are placed in the waiting queue are now placed into the ready queue.
*****To implement the semaphore with no busy waiting we need to define the semaphore of the wait() and signal() operation by using the 'C' Struct.
Which is as follows:

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

- i.e. each semaphore has an integer value stored in the variable "value" and the list of processes list.
- When a process performs the wait() operation on the semaphore then it will add a list of processes to the list .

- When a process performs the signal() operation on the semaphore then it removes the processes from the list.

Semaphore Implementation with no Busy waiting

```
Implementation of wait: (definition of wait with no busy waiting)
wait (S)
{
    Value--;
    if (value < 0)
   {
       add this process to waiting queue
       block();
   }
}
```

```
Implementation of signal: (definition of signal with no busy waiting)
Signal (S)
{
   Value++;
   if (value <= 0)
  {
         remove a process P from the waiting queue
          wakeup(P);
  }
}
```

Advantages

The different advantages of semaphores are given below:

- They do not allow more than one process to enter the critical section. In this way, mutual exclusion is achieved and thus they are extremely efficient than other techniques for synchronization.

- Due to busy waiting in semaphore, there is no wastage of process time and resources. This is because the processes are only allowed to enter the critical section after satisfying a certain condition.
- They are machine-independent as they run in the machine-independent code of the microkernel.
- They allow flexible management of resources.

Disadvantages

The different limitations of semaphores are given below:

- There may be a situation of priority inversion where the processes having low priority get access to the critical section than the processes having higher priority.
- To avoid deadlocks, the wait() and signal() operations have to be executed in the correct order.
- Semaphore programming is complicated and there are chances of not achieving mutual exclusion.

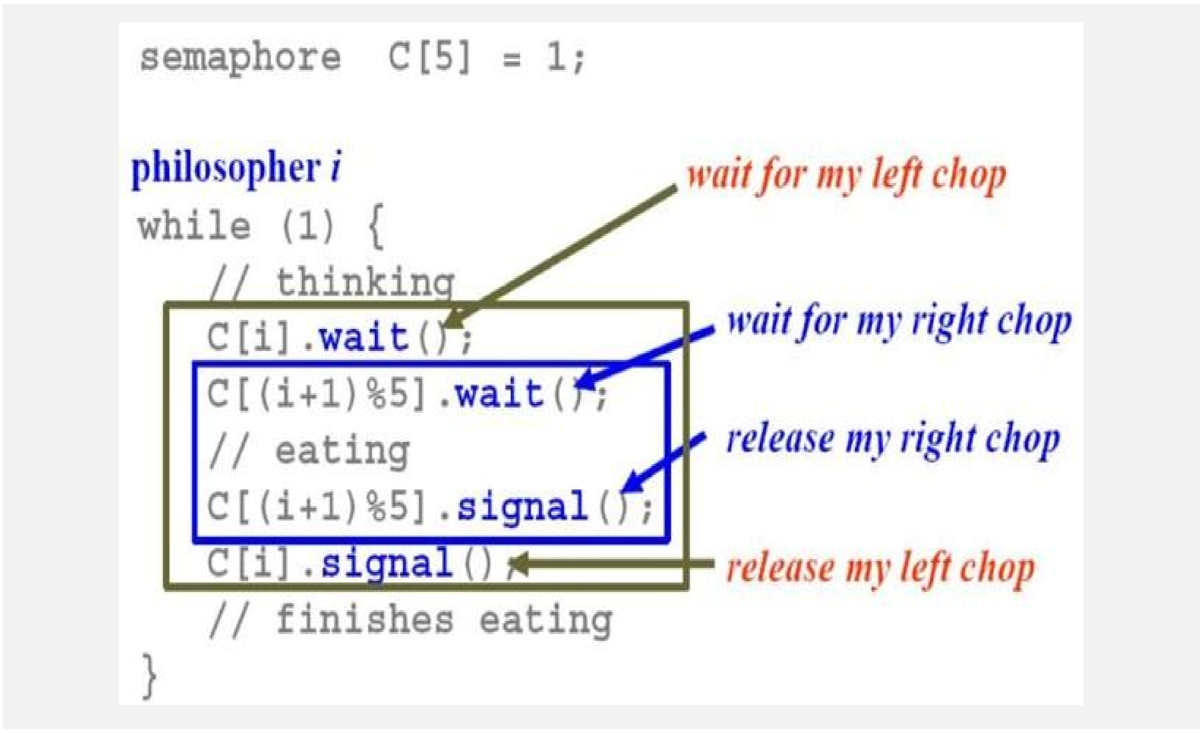
Problem in implementation of semaphore

Dining Philosophers Problem

The problem with dining philosophers is that there are five philosophers who share a round table and eat and think differently. There is a small bowl of rice for each philosopher and 5 chopsticks. The philosopher needs both their right and left chopsticks in order to eat. A hungry philosopher can only eat if there are both sticks available.Otherwise the philosopher puts down his stick and begins to think again.
The dining philosopher is an old synchronization problem as it points to a large class of concurrency control problems.

Solution using semaphore

- Chopsticks are shared items (by two philosophers) and must be protected.
- Each chopstick has a semaphore with initial value 1.
- A philosopher calls wait() before picking up a chopstick and calls signal() to release it.

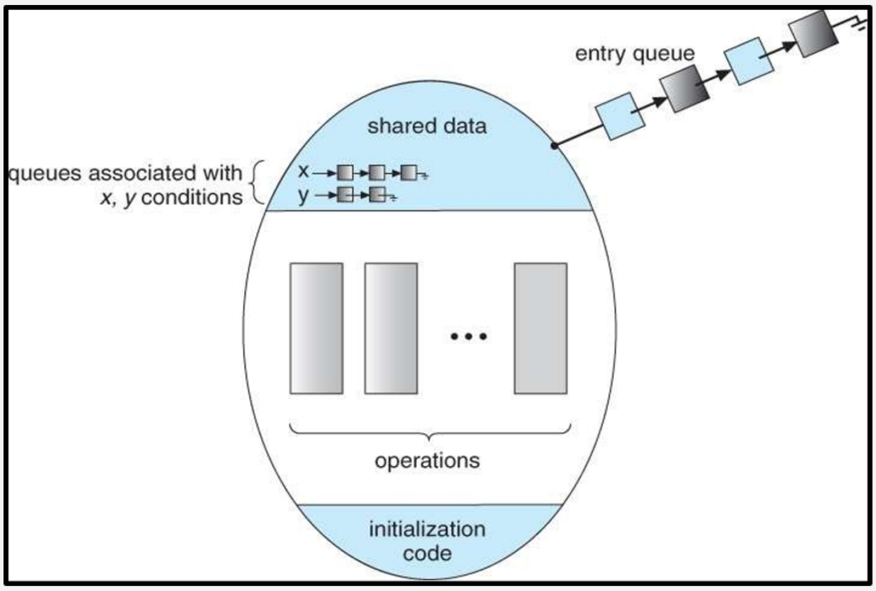


```
semaphore  C[5] = 1;

philosopher i
while (1) {
    // thinking
    C[i].wait();
    C[(i+1)%5].wait();
    // eating
    C[(i+1)%5].signal();
    C[i].signal();
    // finishes eating
}
```

Monitor

In concurrent programming, a monitor is a synchronization construct that allows threads to have both mutual exclusion and the ability to wait (block) for a certain condition to become true. Monitors also have a mechanism for signaling other threads that their condition has been met.



A monitor can have variables of the condition type that can be accessed by wait() and signal() operations only.
The operation x.wait() means that the process making this operation is suspended until another process invokes x.signal().
The operation x.signal() resumes exactly one suspended process and has no effect if there is none.

Syntax:

```
 monitor monitor_name
{
     // shared variable declarations initialization_code (...)
     { ... }
      procedure P1 (...)
      { ... }
      procedure Pn (...)
      { ... }
}
```