**What is Semaphore?**

**Semaphore** is simply a variable that is non-negative and shared between threads. A semaphore is a signaling mechanism, and a thread that is waiting on a semaphore can be signaled by another thread. It uses two atomic operations, 1) Wait, and 2) Signal for the process synchronization.

A semaphore either allows or disallows access to the resource, which depends on how it is set up.

In this Operating System (OS) tutorial, you will learn:

**Characteristic of Semaphore**

Here, are characteristic of a semaphore:

* It is a mechanism that can be used to provide synchronization of tasks.
* It is a low-level synchronization mechanism.
* Semaphore will always hold a non-negative integer value.
* Semaphore can be implemented using test operations and interrupts, which should be executed using file descriptors.

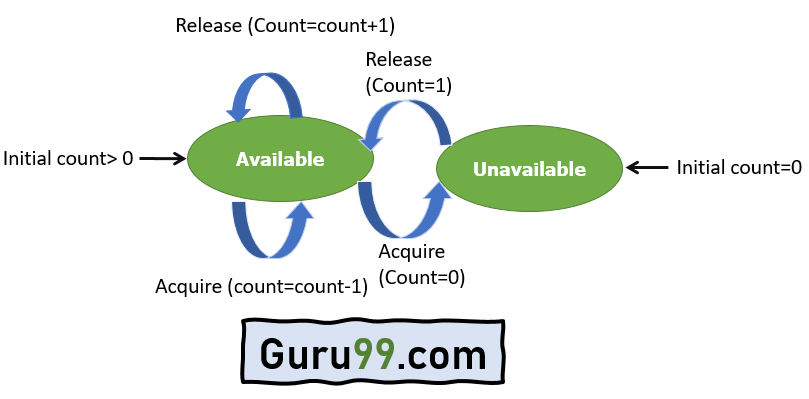
**Types of Semaphores**

The two common kinds of semaphores are

* Counting semaphores
* Binary semaphores.

**Counting Semaphores**

This type of Semaphore uses a count that helps task to be acquired or released numerous times. If the initial count = 0, the counting semaphore should be created in the unavailable state.

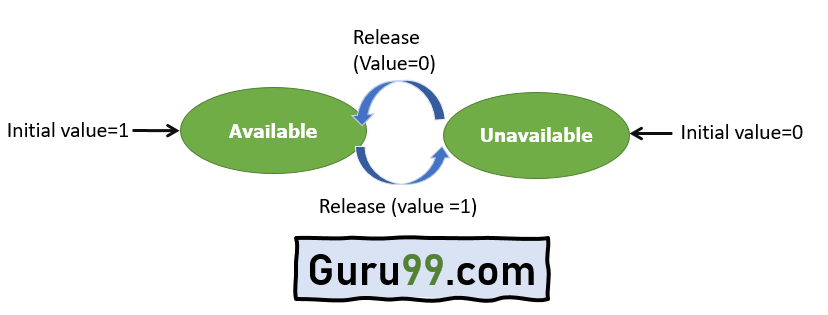


Counting Semaphores

However, If the count is > 0, the semaphore is created in the available state, and the number of tokens it has equals to its count.

**Binary Semaphores**

The binary semaphores are quite similar to counting semaphores, but their value is restricted to 0 and 1. In this type of semaphore, the wait operation works only if semaphore = 1, and the signal operation succeeds when semaphore= 0. It is easy to implement than counting semaphores.



Binary Semaphores

**Example of Semaphore**

The below-given program is a step by step implementation, which involves usage and declaration of semaphore.

Shared var mutex: semaphore = 1;

Process i

begin

.

.

P(mutex);

execute CS;

V(mutex);

.

.

End;

**Wait and Signal Operations in Semaphores**

Both of these operations are used to implement process synchronization. The goal of this semaphore operation is to get mutual exclusion.

**Wait for Operation**

This type of semaphore operation helps you to control the entry of a task into the critical section. However, If the value of wait is positive, then the value of the wait argument X is decremented. In the case of negative or zero value, no operation is executed. It is also called P(S) operation.

After the semaphore value is decreased, which becomes negative, the command is held up until the required conditions are satisfied.

Copy CodeP(S)

{

while (S<=0);

S--;

}

**Signal operation**

This type of Semaphore operation is used to control the exit of a task from a critical section. It helps to increase the value of the argument by 1, which is denoted as V(S).

Copy CodeP(S)

{

while (S>=0);

S++;

}

**Counting Semaphore vs. Binary Semaphore**

Here, are some major differences between counting and binary semaphore:

|  |  |
| --- | --- |
| **Counting Semaphore** | **Binary Semaphore** |
| No mutual exclusion | Mutual exclusion |
| Any integer value | Value only 0 and 1 |
| More than one slot | Only one slot |
| Provide a set of Processes | It has a mutual exclusion mechanism. |

**Difference between Semaphore vs. Mutex**

|  |  |  |
| --- | --- | --- |
| **Parameters** | **Semaphore** | **Mutex** |
| Mechanism | It is a type of signaling mechanism. | It is a locking mechanism. |
| Data Type | Semaphore is an integer variable. | Mutex is just an object. |
| Modification | The wait and signal operations can modify a semaphore. | It is modified only by the process that may request or release a resource. |
| Resource management | If no resource is free, then the process requires a resource that should execute wait operation. It should wait until the count of the semaphore is greater than 0. | If it is locked, the process has to wait. The process should be kept in a queue. This needs to be accessed only when the mutex is unlocked. |
| Thread | You can have multiple program threads. | You can have multiple program threads in mutex but not simultaneously. |
| Ownership | Value can be changed by any process releasing or obtaining the resource. | Object lock is released only by the process, which has obtained the lock on it. |
| Types | Types of Semaphore are counting semaphore and binary semaphore and | Mutex has no subtypes. |
| Operation | Semaphore value is modified using wait () and signal () operation. | Mutex object is locked or unlocked. |
| Resources Occupancy | It is occupied if all resources are being used and the process requesting for resource performs wait () operation and blocks itself until semaphore count becomes >1. | In case if the object is already locked, the process requesting resources waits and is queued by the system before lock is released. |

**Advantages of Semaphores**

Here, are pros/benefits of using Semaphore:

* It allows more than one thread to access the critical section
* Semaphores are machine-independent.
* Semaphores are implemented in the machine-independent code of the microkernel.
* They do not allow multiple processes to enter the critical section.
* As there is busy waiting in semaphore, there is never a wastage of process time and resources.
* They are machine-independent, which should be run in the machine-independent code of the microkernel.
* They allow flexible management of resources.

**Disadvantage of semaphores**

Here, are cons/drawback of semaphore

* One of the biggest limitations of a semaphore is priority inversion.
* The operating system has to keep track of all calls to wait and signal semaphore.
* Their use is never enforced, but it is by convention only.
* In order to avoid deadlocks in semaphore, the Wait and Signal operations require to be executed in the correct order.
* Semaphore programming is a complicated, so there are chances of not achieving mutual exclusion.
* It is also not a practical method for large scale use as their use leads to loss of modularity.
* Semaphore is more prone to programmer error.
* It may cause deadlock or violation of mutual exclusion due to programmer error.

**Summary:**

* Semaphore is defined as a variable that is non-negative and shared between threads.
* It is a mechanism that can be used to provide synchronization of tasks.
* Counting semaphore uses a count that helps task to be acquired or released numerous times.
* The binary semaphores are quite similar to counting semaphores, but their value is restricted to 0 and 1.
* Wait operation helps you to control the entry of a task into the critical section
* Signal semaphore operation is used to control the exit of a task from a critical section
* Counting Semaphore has no mutual exclusion whereas Binary Semaphore has Mutual exclusion
* Semaphore means a signaling mechanism whereas Mutex is a locking mechanism
* Semaphore allows more than one thread to access the critical section
* One of the biggest limitations of a semaphore is priority inversion.