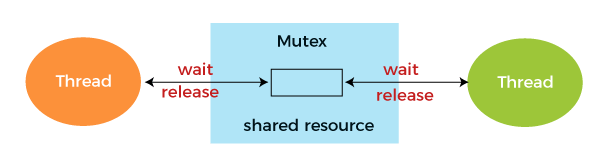
Mutex vs Semaphore

As per operating system terminology, mutex and semaphores are kernel resources that provide synchronization services, also called ***synchronization primitives***. Process synchronization plays an important role in maintaining the consistency of shared data. Both the software and hardware solutions are present for handling critical section problems. But hardware solutions for critical section problems are quite difficult to implement. Mutex and semaphore both provide synchronization services, but they are not the same.

What is Mutex?

Mutex is a mutual exclusion object that synchronizes access to a resource. It is created with a unique name at the start of a program. The mutex locking mechanism ensures only one thread can acquire the mutex and enter the critical section. This thread only releases the mutex when it exits in the critical section.



It is a special type of binary semaphore used for controlling access to the shared resource. It includes a priority inheritance mechanism to avoid extended priority inversion problems. It allows current higher priority tasks to be kept in the blocked state for the shortest time possible. However, priority inheritance does not correct priority inversion but only minimizes its effect.

**Example**

This is shown with the help of the following example,

1. wait (mutex);
2. .....
3. Critical Section
4. .....
5. signal (mutex);

Use of Mutex

A mutex provides mutual exclusion, either producer or consumer who can have the key (mutex) and proceed with their work. As long as the producer fills the buffer, the user needs to wait, and vice versa. In Mutex lock, all the time, only a single thread can work with the entire buffer.

When a program starts, it requests the system to create a mutex object for a given resource. The system creates the mutex object with a unique name or ID. Whenever the program thread wants to use the resource, it occupies lock on mutex object, utilizes the resource and after use, it releases the lock on mutex object. Then the next process is allowed to acquire the lock on the mutex object.

Meanwhile, a process has acquired the lock on the mutex object, and no other thread or process can access that resource. If the mutex object is already locked, the process desiring to acquire the lock on the mutex object has to wait and is queued up by the system till the mutex object is unlocked.

Advantages of Mutex

Here are the following advantages of the mutex, such as:

* Mutex is just simple locks obtained before entering its critical section and then releasing it.
* Since only one thread is in its critical section at any given time, there are no race conditions, and data always remain consistent.

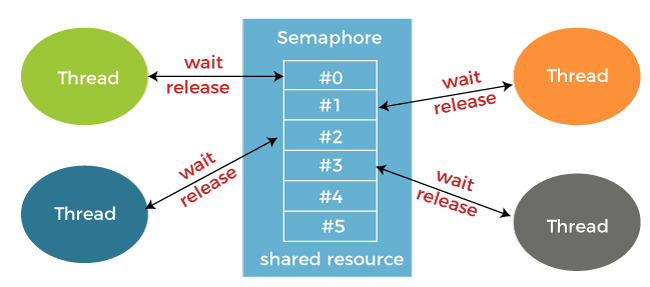
Disadvantages of Mutex

Mutex also has some disadvantages, such as:

* If a thread obtains a lock and goes to sleep or is preempted, then the other thread may not move forward. This may lead to starvation.
* It can't be locked or unlocked from a different context than the one that acquired it.
* Only one thread should be allowed in the critical section at a time.
* The normal implementation may lead to a busy waiting state, which wastes CPU time.

What is Semaphore?

Semaphore is simply a variable that is non-negative and shared between threads. A semaphore is a signaling mechanism, and another thread can signal a thread that is waiting on a semaphore.



A semaphore uses two atomic operations,

**1. Wait:** The wait operation decrements the value of its argument S if it is positive. If S is negative or zero, then no operation is performed.

1. wait(S)
2. {
3. while (S**<**=0);
4. S--;
5. }

**2. Signal for the process synchronization:** The signal operation increments the value of its argument S.

1. signal(S)
2. {
3. S++;
4. }

A semaphore either allows or reject access to the resource, depending on how it is set up.

Use of Semaphore

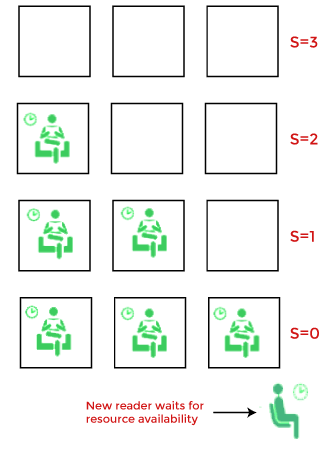
In the case of a single buffer, we can separate the 4 KB buffer into four buffers of 1 KB. Semaphore can be associated with these four buffers, allowing users and producers to work on different buffers simultaneously.

Types of Semaphore

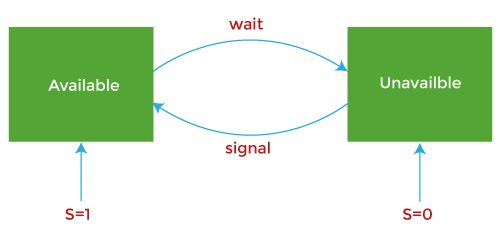
Semaphore is distinguished by the operating system in two categories **Counting semaphore** and **Binary semaphore**.

**1. Counting Semaphore:** The semaphore S value is initialized to the **number of resources** present in the system. Whenever a process wants to access the resource, it performs **the wait()**operation on the semaphore and **decrements** the semaphore value by one. When it releases the resource, it performs **the signal()** operation on the semaphore and **increments** the semaphore value by one.

When the semaphore count goes to 0, it means the processes occupy all resources. A process needs to use a resource when the semaphore count is 0. It executes the **wait()** operation and gets **blocked** until the semaphore value becomes greater than 0.



**2. Binary semaphore:** The value of a semaphore ranges between **0**and **1**. It is similar to mutex lock, but mutex is a locking mechanism, whereas the semaphore is a signaling mechanism. In binary semaphore, if a process wants to access the resource, it performs **the wait()** operation on the semaphore and decrements the value of the semaphore from 1 to 0. When it releases the resource, it performs a **signal()** operation on the semaphore and increments its value to 1. Suppose the value of the semaphore is 0 and a process wants to access the resource. In that case, it performs **wait()** operation and block itself till the current process utilizing the resources releases the resource.



Advantages of Semaphore

Here are the following advantages of semaphore, such as:

* 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 and waiting in semaphore, there is never 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

Semaphores also have some disadvantages, such as:

* 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.
* The Wait and Signal operations require to be executed in the correct order to avoid deadlocks in semaphore.
* Semaphore programming is a complex method, 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
* , and it may cause deadlock or violation of mutual exclusion due to programmer error.

Difference between Mutex and Semaphore

The basic difference between semaphore and mutex is that semaphore is a signalling mechanism, i.e. processes perform wait() and signal() operation to indicate whether they are acquiring or releasing the resource. In contrast, a mutex is a locking mechanism, and the process has to acquire the lock on a mutex object if it wants to acquire the resource. Here are some more differences between semaphore and mutex, such as:

|  |  |  |
| --- | --- | --- |
| **Terms** | **Mutex** | **Semaphore** |
| Definition | The mutex is a locking mechanism, as to acquire a resource, a process needs to lock the mutex object, and while releasing a resource process has to unlock the mutex object. | Semaphore is a signalling mechanism as wait() and signal() operations performed on the semaphore variable indicate whether a process is acquiring or releasing the resource. |
| Existence | A mutex is an object. | Semaphore is an integer variable. |
| Function | Mutex allows multiple program threads to access a single resource but not simultaneously. | Semaphore allows multiple program threads to access a finite instance of resources. |
| Ownership | Mutex object lock is released only by the process that has acquired the lock on the mutex object. | Semaphore value can be changed by any process acquiring or releasing the resource by performing wait() and signal() operation. |
| Categorize | Mutex is not categorized further. | The semaphore can be categorized into counting semaphore and binary semaphore. |
| Operation | The mutex object is locked or unlocked by the process of requesting or releasing the resource. | Semaphore value is modified using wait() and signal() operation apart from initialization. |
| Resources Occupied | If a mutex object is already locked, then the process desiring to acquire resource waits and get queued by the system till the resource is released and the mutex object gets unlocked. | Suppose the process acquires all the resources, and no resource is free. In that case, the process desiring to acquire resource performs wait() operation on semaphore variable and blocks itself till the count of semaphore become greater than 0. |