# Conditional Variable vs Semaphore



When should one use a semaphore and when should one use a conditional variable (CondVar) ?

multithreading operating-system mutual-exclusion

edited Jan 12 at 15:03
UmNyobe
10.2k 3 19 49

asked Aug 18 '10 at 14:19
doron
10.7k 2 21 50

## 3 Answers

Locks are used for mutual exclusion. When you want to ensure that a piece of code is atomic, put a lock around it. You could theoretically use a binary semaphore to do this, but that's a special case.

Semaphores and condition variables build on top of the mutual exclusion provide by locks and are used for providing synchronized access to shared resources. They can be used for similar purposes.

A condition variable is generally used to avoid busy waiting (looping repeatedly while checking a condition) while waiting for a resource to become available. For instance, if you have a thread (or multiple threads) that can't continue onward until a queue is empty, the busy waiting approach would be to just doing something like:

```
//pseudocode
while(!queue.empty())
{
   sleep(1);
}
```

The problem with this is that you're wasting processor time by having this thread repeatedly check the condition. Why not instead have a synchronization variable that can be signaled to tell the thread that the resource is available?

```
//pseudocode
syncVar.lock.acquire();

while(!queue.empty())
{
   syncVar.wait();
}

//do stuff with queue

syncVar.lock.release();
```

Presumably, you'll have a thread somewhere else that is pulling things out of the queue. When the queue is empty, it can call `syncVar.signal()` to wake up a random thread that is sitting asleep on `syncVar.wait()` (or there's usually also a `signalAll()` or `broadcast()` method to wake up all the threads that are waiting).

I generally use synchronization variables like this when I have one or more threads waiting on a single particular condition (e.g. for the queue to be empty).

Semaphores can be used similarly, but I think they're better used when you have a shared resource that can be available and unavailable based on some integer number of available things. Semaphores are good for producer/consumer situations where producers are allocating resources and consumers are consuming them.

Think about if you had a soda vending machine. There's only one soda machine and it's a shared resource. You have one thread that's a vendor (producer) who is responsible for keeping the machine stocked and N threads that are buyers (consumers) who want to get sodas out of the machine. The number of sodas in the machine is the integer value that will drive our semaphore.

Every buyer (consumer) thread that comes to the soda machine calls the semaphore `down()` method to take a soda. This will grab a soda from the machine and decrement the count of available sodas by 1. If there are sodas available, the code will just keep running past the `down()` statement without a problem. If no sodas are available, the thread will sleep here waiting to be notified of when soda is made available again (when there are more sodas in the machine).

The vendor (producer) thread would essentially be waiting for the soda machine to be empty. The vendor gets notified when the last soda is taken from the machine (and one or more consumers are potentially waiting to get sodas out). The vendor would restock the soda machine with the semaphore `up()` method, the available number of sodas would be incremented each time and thereby the waiting consumer threads would get notified that more soda is available.

The `wait()` and `signal()` methods of a synchronization variable tend to be hidden within the `down()` and `up()` operations of the semaphore.

Certainly there's overlap between the two choices. There are many scenarios where a semaphore or a condition variable (or set of condition variables) could both serve your purposes. Both semaphores and condition variables are associated with a lock object that they use to maintain mutual exclusion, but then they provide extra functionality on top of the lock for synchronizing thread execution. It's mostly up to you to figure out which one makes the most sense for your situation.

That's not necessarily the most technical description, but that's how it makes sense in my head.

edited Aug 18 '10 at 18:59

answered Aug 18 '10 at 16:35

Brent Nash
**7,045** 23 40

4 Wow!! Excellent answer. – user373215 Aug 18 '10 at 18:45

That was really helpful. Thanks Brent. – ashu Mar 4 '14 at 4:19

3 Great answer, I would like to add from other so answers:Semaphore is used to control the number of threads executing. There will be fixed set of resources. The resource count will gets decremented every time when a thread owns the same. When the semaphore count reaches 0 then no other threads are allowed to acquire the resource. The threads get blocked till other threads owning resource releases. In short, the main difference is how many threads are allowed to acquire the resource at once ? Mutex --its ONE. Semaphore -- its DEFINED_COUNT, ( as many as semaphore count) – berkay Aug 17 '14 at 6:47

4 Just to elaborate on why there's this while loop instead of a simple if: something called **spurios wakeup**. Quoting this wikipedia article: "*One of the reasons for this is a spurious wakeup; that is, a thread might be awoken from its waiting state even though no thread signaled the condition variable*" – Vladislavs Burakovs Aug 28 '14 at 19:25

@VladislavsBurakovs Good point! I think it's also helpful for the case where a broadcast wakes up more threads than there are resources available (e.g. broadcast wakes up 3 threads, but there are only 2 items in the queue). – Brent Nash Aug 28 '14 at 20:11



Semaphores can be used to implement exclusive access to variables, however they are meant to be used for synchronization. Mutexes, on the other hand, have a semantics which is strictly related to mutual exclusion: only the process which locked the resource is allowed to unlock it.

Unfortunately you cannot implement synchronization with mutexes, that's why we have condition variables. Also notice that with condition variables you can unlock all the waiting threads in the same instant by using the broadcast unlocking. This cannot be done with semaphores.

answered Aug 18 '10 at 16:44


Dacav
**2,865** 19 42

I file condition variables under monitor synchronization. I've generally seen semaphores and monitors as two different synchronization styles. There are differences between the two in terms of how much state data is inherently kept and how you want to model code - but there really isn't any problem that can be solved by one but not the other.

I tend to code towards monitor form; in most languages I work in that comes down to mutexes, condition variables, and some backing state variables. But semaphores would do the job too.

edited Aug 18 '10 at 16:49

answered Aug 18 '10 at 16:44


Justin R
**191** 5

1 This would be a better answer if you explained what "monitor form" is. – Steven Lu May 5 '14 at 16:43