


Condition Variable Guide

Example 1: Why condition variables?

| global beerVolume, beerMugLock | |
|--|---|
| Bartender - wants to fill mug with beer | Patron - wants to drink beer from mug |
| | <pre>Acquire(beerMugLock) // pick up mug If (beerVolume == 0) // no beer! Cannot drink { Sleep?? Wait for beer to appear in mug? } // now there is beer DrinkBeer(); Release(beerMugLock)</pre> |
| <pre>Acquire(beerMugLock) // WAIT!!</pre>  | |

... and so the bartender and the patron both wait forever. The patron is waiting for the **CONDITION** (beer volume to be greater than 0), and the bartender is waiting for the **LOCK** so that they can affect that condition.

CONDITION VARIABLE TO THE RESCUE!



Example 2: Why condition variables?

| global beerVolume, beerMugLock global ConditionVariable mugHasBeer | |
|---|---|
| Bartender - wants to fill mug with beer | Patron - wants to drink beer from mug |
| | Acquire(beerMugLock) // pick up mug If (beerVolume == 0) // no beer! Cannot drink { cv_wait(mugHasBeer) // releases beerMugLock, then block! } // now there is beer DrinkBeer(); Release(beerMugLock) |
| Acquire(beerMugLock) // this is AVAILABLE since cv_wait released it!! beerVolume = 100; // fill mug cv_signal(mugHasBeer) // wake up patron Release(beerMugLock) // hand them beer | |
| | Acquire(beerMugLock) // pick up mug If (beerVolume == 0) // no beer! Cannot drink { cv_wait(mugHasBeer) // wakeup INSIDE cv_wait; // cv_wait calls Acquire(beerMugLock) // before return // PATRON now owns beerMugLock } // now there is beer DrinkBeer(); Release(beerMugLock) |

Bartender and Patron are happy! But this was just two threads ...

Example 3: Producers and Consumers - Why a while loop?

| Producer | Consumer |
|---|---|
| <pre>lock_acquire(mutex) while (count == N) { cv_wait(notfull, mutex) } count ++ cv_signal(notempty, mutex) lock_release(mutex) lock_release(mutex)</pre> | <pre>lock_acquire(mutex) while (count == 0) { cv_wait(notempty, mutex) } count -- cv_signal(notfull, mutex) lock_release(mutex) lock_release(mutex)</pre> |

With two threads, it isn't obvious why this while loop is needed in our producer and consumer code. Let's look at three threads. **Suppose the buffer is FULL (count == N).**

| Producer 1 | Producer 2 | Consumer |
|---|---|---|
| <pre>lock_acquire(mutex) while (count == N) { cv_wait(notfull, mutex)</pre> | | |
| Context Switch [Ready Queue: Consumer, Producer2] | | |
| | | <pre>lock_acquire(mutex) while (count == 0) {} count -- cv_signal(notfull, mutex) // NOTE: Producer1 // goes into the ready // queue. lock_release(mutex) lock_release(mutex)</pre> |
| Context Switch [Ready Queue: Producer1, Consumer] | | |
| | <pre>lock_acquire(mutex) while (count == N) {} count ++ cv_signal(notempty, mutex) lock_release(mutex) lock_release(mutex)</pre> | |

| | | |
|--|---|--|
| | Note that at this point: count == N again! | |
| Context Switch [Ready Queue: Consumer, Producer2] | | |
| <pre>// COUNT == N because // Producer2 added an item!! while (count == N) { cv_wait(notfull, mutex) WAKE UP HERE!! } count ++ cv_signal(notempty, mutex) lock_release(mutex) lock_release(mutex)</pre> | | |

In Producers and Consumers, if we **DID NOT** use the **while-loop** to check the condition it would be possible for Producer1 to add an item to a **FULL BUFFER**. Usually, we must re-check the condition after **cv_wait** returns in case other threads managed to alter the condition before we run!

Why might cv_signal want a pointer to the lock?

Briefly, because you should be the owner of the lock that protects the global condition before you signal that the condition has been met.

1. When you call **cv_wait**, some global condition is not being met. This happens inside of a lock (which ultimately, is used to protect that global condition).
2. Any thread that changes that condition will require that same lock to make that change. So, the thread that makes the condition true, will do so inside of the critical section protected by that one lock.
3. Once a thread makes the condition true, it should immediately signal --- because at that **EXACT** moment in time, the condition is true. This means the signal should happen inside of the critical section protected by the lock. If you were to signal outside of this critical section there is a chance another thread will have changed the condition value prior to your signalling (which is undesirable).

Hence, the signalling thread should be the owner of the lock passed in.

Side Note: various implementations of Mesa-style CVs do NOT do this, but they are not broken. The waiting thread, once returned to the critical section, simply needs to re-check the condition before proceeding --- which you may note, we usually do anyway.