Chapter 5d: Condition Variables: User-defined Resources

Gene Cooperman

Contents

1 Condition Variables: User-defined Resources **1**
1.1 1. Review: The Acquire/Release Pattern 1
1.2 2. Why Do We Need Condition Variables? 2
1.3 3. The Structure: Acquire, Do Task, Release 2
1.4 4. How `pthread_cond_wait` Works – By Analogy with `sem_wait` 2
1.5 5. Why the `while` Loop? Why Not `if`? 3
1.6 6. Signal vs. Broadcast – and When to Do Neither 3
1.7 7. Putting It All Together: The Thread Lifecycle 3
1.8 8. Summary of Key Ideas 4

THIS IS A FIRST DRAFT OF A WORK IN PROGRESS.

1 Condition Variables: User-defined Resources

This review is still incomplete. To augment this, please refer to:

1. DIAGRAM OF ACQUIRE/RELEASE MODEL FOR CONDITION VARIABLES
2. CHAPTER 30. Condition Variables (in the online textbook at ostep.org)

1.1 1. Review: The Acquire/Release Pattern

All three synchronization primitives – mutexes, semaphores, and condition variables – follow a common pattern: **acquire** a resource, then **release** it.

Mutex (`pthread_mutex_lock` / `pthread_mutex_unlock`): Acquires exclusive access to a critical section. There is exactly one resource – the lock itself.

Semaphore (`sem_wait` / `sem_post`): `sem_wait` acquires one unit of a resource (decrementing an internal count), and `sem_post` releases one unit (incrementing the count). The individual resource units are interchangeable – any one will do. Examples of such resources include: memory addresses in a pool, slots in a bounded buffer, available CPU cores, or sockets to back-end servers that all connect to the same database.

Condition Variable (`pthread_cond_wait` / `pthread_cond_signal` or `pthread_cond_broadcast`): Acquires and releases a *user-defined* resource governed by a *user-defined consistency condition*. This is the key idea – you get to define what "the resource" means.

1.2 2. Why Do We Need Condition Variables?

Mutexes protect a critical section, but they assume a single, simple resource: the lock. What happens when the "resource" is actually defined by **multiple shared variables that must remain consistent with one another**?

Consider a bank transfer: debiting one account and depositing into another must happen atomically. The "resource" here isn't a single counter – it's the *consistent state* across two account balances.

Similarly, consider a readers-writers problem. We might protect three shared variables with a single mutex:

- `num_active_readers`
- `num_active_writers`
- `num_waiting_readers`

No single variable captures whether it's "safe to proceed." The decision depends on a *condition* that is a function of all of them together. For instance, a writer must wait until `num_active_readers == 0 && num_active_writers == 0`. A reader might wait until `num_active_writers == 0`. These are the user-defined consistency conditions that condition variables let us express.

1.3 3. The Structure: Acquire, Do Task, Release

The general pattern for using a condition variable looks like this:

```
pthread_mutex_lock(&mutex);

// ===== ACQUIRE =====
while (!CONDITION) {
    pthread_cond_wait(&condvar, &mutex);
}

// ===== DO TASK =====
// ... perform work while holding the mutex (or release/reacquire around it) ...

// ===== RELEASE =====
// Update shared variables to reflect task completion.
// Then wake waiting threads:
pthread_cond_signal(&condvar);    // wake one thread
// -- or --
pthread_cond_broadcast(&condvar); // wake all threads

pthread_mutex_unlock(&mutex);
```

Here, `CONDITION` is a boolean expression over the shared variables protected by `mutex` (e.g., `num_active_writers == 0`).

1.4 4. How `pthread_cond_wait` Works – By Analogy with `sem_wait`

Recall `sem_wait`: if the semaphore's internal count is greater than zero, it decrements the count and returns immediately. If the count is zero, the calling thread **sleeps** until another thread calls `sem_post`, which increments the count and wakes a sleeper.

`pthread_cond_wait` is analogous, **but with an important difference**: there is no internal count. Instead, the mechanism operates directly on thread states:

1. **The calling thread atomically releases the mutex and enters CPU state SLEEPING.** This is crucial – the release-and-sleep is atomic so that no signal can be lost between unlocking and sleeping.
2. The thread remains in SLEEPING state until another thread calls `pthread_cond_signal` or `pthread_cond_broadcast`.
3. When that signal arrives, the OS interrupts the sleeping thread. Internally, the thread's `sleep()` call frame is interrupted by a signal handler call frame. The OS stops performing the sleep on the thread's behalf and instead dispatches a signal handler. When the signal handler returns to the `sleep()` call frame, `sleep()` returns.
4. **The thread transitions from SLEEPING to RUNNABLE.** It does not yet hold the mutex.
5. The thread then calls `pthread_mutex_lock` internally to **re-acquire the mutex** before `pthread_cond_wait` returns.
6. Once the mutex is acquired and `pthread_cond_wait` returns, execution resumes at the top of the `while` loop, and `CONDITION` is re-evaluated.

The key takeaway: after `pthread_cond_wait` returns, the thread is back at the **end** of the `pthread_cond_wait` call – holding the mutex – and immediately loops back to re-check the condition.

1.5 5. Why the `while` Loop? Why Not `if`?

When a thread wakes up, the condition that caused it to sleep may no longer be the current state of the world. Consider `pthread_cond_broadcast`: **all** sleeping threads wake up, but only one can hold the mutex at a time. By the time the second thread acquires the mutex and checks the condition, the first thread may have already changed the shared variables such that the condition is false again.

Therefore, `CONDITION` must always be rechecked after waking. Using `while` instead of `if` guarantees this. If the condition is still not satisfied, the thread goes right back to sleep.

1.6 6. Signal vs. Broadcast – and When to Do Neither

`pthread_cond_signal(&condvar)` – Wakes **one** sleeping thread (chosen by the OS). This is an optimization: use it when you know that exactly one waiting thread can make progress.

`pthread_cond_broadcast(&condvar)` – Wakes **all** sleeping threads. Each woken thread must re-acquire the mutex (one at a time), re-evaluate `CONDITION`, and either proceed to DO_TASK or go back to sleep.

It is always safe to call `pthread_cond_broadcast`. It may wake threads unnecessarily (they'll just re-check the condition and go back to sleep), but it will never cause incorrect behavior. Use `pthread_cond_signal` as an optimization only when you are sure it is sufficient.

Doing neither is also sometimes correct. During the RELEASE stage, the thread examines the shared variables. If, for example, `num_active_writers > 0` or `num_active_readers > 0`, then perhaps no waiting thread could make progress right now anyway. In that case, the releasing thread may simply exit the critical section without signaling. Other threads currently in DO_TASK will eventually enter RELEASE themselves and signal at that point.

1.7 7. Putting It All Together: The Thread Lifecycle

Consider the full lifecycle of a thread arriving at a condition variable:

1. **Lock the mutex.** Enter the critical section.
2. **Evaluate `CONDITION`.** This condition is a function of *all* the shared variables protected by the mutex.
3. **If `CONDITION` is false:** call `pthread_cond_wait`, which atomically releases the mutex and puts the thread to SLEEPING.
4. **Another thread signals or broadcasts.** The sleeping thread transitions from SLEEPING -> RUNNABLE.
5. **Re-acquire the mutex.** The thread competes with other threads (especially after a broadcast) to lock the mutex.
6. **Re-evaluate `CONDITION`.** If still false, go back to step 3. If true, proceed.
7. **DO TASK.** Perform work, updating shared variables as needed.
8. **RELEASE.** Examine the shared variables. Decide whether to call `signal`, `broadcast`, or neither. Unlock the mutex.

1.8 8. Summary of Key Ideas

- Condition variables let you define your **own resource** and your **own consistency condition** – a boolean function of the shared variables protected by the mutex.
- `pthread_cond_wait` is analogous to `sem_wait`, but there is no internal count. It operates by putting the thread to sleep and later waking it via a signal.
- After waking, a thread must **re-acquire the mutex** and **re-check the condition** (hence the `while` loop).
- `pthread_cond_broadcast` is always correct. `pthread_cond_signal` and skipping the signal entirely are optimizations that require careful reasoning about the shared state.