CSSE332: Operating Systems

Buffalo, Mark, and Mohammad's Guide to Condition Variables

Last updated: January 24, 2024

1 Introduction

Most of the simple solutions to concurrency problems using condition variables fall into a simple style. This is not necessarily the only way condition variables can be applied correctly but doing it this way makes it easier to see if things are correct. Also limiting your options to one style makes it less overwhelming to start until things start to feel natural and obvious. So I recommend you follow this system.

Note that this guidance is rules of thumb and hints; it's not a substitute for really thinking about your problem. You won't get leverage with the graders by saying *"hey this incorrect thing I did is consistent with the guidance"* or *"the guidance didn't warn me not to incorrect thing I did"*. The goal is to solve the problem without concurrency bugs – any style advice that introduces bugs should be ignored.

Although we can provide hints, you still must figure out your concurrency state, decide when threads need to wait, decide when threads need to signal, etc. That creative problem solving is the essence of writing correct concurrent code, and learning that is the essential skill we hope you will internalize on Exam 2. That kind of problem solving can never be condensed into a simple recipe, that's why some day you'll hopefully get paid the $$.

2 Getting Started

Here are some general guidelines to get you started:

- You'll need one mutex lock, any number of condition variable depending on the number of reasons to wait.

  Note that the mutex is to protect the state of the world (or the concurrency state) and not the critical section. So any time you touch the state of the world, make sure you hold the lock on you!

- You'll also need some number of variables to represent the *concurrency state* (aka, the state of the world). I use concurrency state here to represent part of the (usually global) state that your code accesses to enforce correctness.

  For example, if there are only three threads allowed in the critical section, we need some variable to represent how many threads are currently in that critical section.

- I often have some idea of what the condition variables/concurrency state will be when I start, but end up tweaking them as I design the solution and write the code.

3 In Your Code

Here are some considerations to keep in mind when writing code (or pseudocode):

- If you thread might wait **or** read/update the concurrency state, lock the mutex!
- When holding the lock, you can safely read/write at a consistent time. Usually, you'll want to hold the lock for the entire read/write experience. In other words, there should be only one mutex and that its critical section should be as large as possible; it doesn't matter how fast your multithreaded code is, because it doesn't count if it isn't correct.

  For example, don't lock/unlock to read/write variable `A`, then lock/unlock to read/write variable `B`, as this will tend to introduce bugs where `A` and `B` have an inconsistent state. Similarly, don't read variable `A`, lock, unlock, re-lock, and then write variable `A`.

  Here is a more concrete example:

```
1  void *thread_fn(void *ignore) {
2    pthread_mutex_lock(&lock);
3    while(people_in_room > 5) {
4      pthread_cond_wait(&cv, &lock);
5    }
6    pthread_mutex_unlock(&lock);
7
8    // this is dangerous, this gap in time where you don't hold the lock
9    // might allow another thread to sneak by and change the value of
10   // people_in_room.
11   // Remember that the scheduler hates you! If a context switch is going
12   // to hurt you, then that's when the scheduler will do it!
13
14   pthread_mutex_lock(&lock);
15   people_in_room++;
16   pthread_mutex_unlock(&lock);
17
18   // enter room now!
19   ...
20 }
21
```

- If reading the concurrency state indicates you need to wait, then use wait on an appropriate condition variable **within a while loop** (and this implicitly unlocks).
- After you wake from a wait, remember you are holding the lock again so it is safe to read or write to the concurrency state.
- If it is safe to proceed (as opposed to waiting) unlock the mutex after you know it's safe and after you have updated the state appropriately (see example above).
- Never hold the mutex for a long amount of time (more accurately, I usually say never hold the mutex for an unbounded amount of time). Especially never hold the mutex during one of the sleeps we use to simulate long processing in these problems.

Holding the mutex for a long time makes it unsafe to read/update the concurrency state (because doing so might introduce a long pause or deadlock).

- Never use the mutex to protect something – only use the mutex to safeguard the concurrency state. Use waiting to ensure that all threads do not proceed when it is unsafe to do so.
- **Every wait needs a corresponding signal or signals** (at least one signal or broadcast). Consider carefully whether you wish to use signal or broadcast. Insert the signal/broadcasts where they are needed.

  Try to avoid threads waking up and immediately re-sleeping but sometimes this is unavoidable.
- **Never busy wait**. A process that is unable to proceed because it's waiting for some other process to work should always be waiting for a signal.
- Always grab the lock before signaling. It might seem superfluous but it will help you avoid very weird conditions that are dependent on the implementation of the `pthreads` library.
- Do not make any assumptions about the order in which thing wake up on a `pthread_cond_signal`. From the `POSIX` specification, we know that at least one thread will wake up, but which one is left for the scheduler/implementation to determine.

4 The Pattern

Altogether, the following approach tends to produce patterns that look like this (again, simply guidance here, not substitute for your own thinking/coding):

```
1  /* Ordinary code */
2
3  // Lock mutex
4  pthread_mutex_lock(&lock);
5
6  // read/write concurrency state, maybe signal?
7
8  while(/* undesired condition */) {
9    // wait on some signal with mutex LOCKED
10   pthread_cond_wait(&cv, &lock);
11 }
12 // When at this point, I own the lock!
13
14 // read/write concurrency state, maybe signal?
15
16 // unlock mutex
17 pthread_mutex_unlock(&lock);
18
19 /* Code that has special rules, generally a critical section.
20  * It is not uncommon for us to use sleep here to simulate something taking
21  * place. */
22
23 // lock mutex
24 pthread_mutex_lock(&lock);
25
26 // read/write concurrency state, maybe signal?
27
28 // unlock mutex
```

```
pthread_mutex_unlock(&lock);

/* Ordinary code and other stuff, maybe return? */
```

5 Some Pitfalls

Here is a small list of typical pitfalls that we have seen students attempt to do when they feel stuck on a problem. Generally, doing one of these is not a good idea.

- When reading the man pagees, you might discover new functions that might seem appealing, do not make a point to try out these new functions during an exam. Try those out on your free time.

  For example, some students, when running into a deadlock, discover that `pthread_mutex_trylock` attempts to grab a lock, but does not block if it is not in an unclocked state. This is not the way to solve a deadlock!

- Swapping out `pthread_cond_signal` for `pthread_cond_broadcast` (and vice versa) in the hope that is magically solves a concurrency problem is not a good idea. You should be intentional about your signals/broadcasts.

- Do not try to dance with where your locks/unlocks should go. Changing the locations of your lock/unlock statements is not going to cut it. Go back to your paper and revisit your design.

- Make sure that each `pthread_cond_wait` has at least one corresponding `pthread_cond_signal` (or `pthread_cond_broadcast` if necessary).

- Similarly, make sure that each `pthread_mutex_lock` has exactly one corresponding `pthread_mutex_unlock`.

  Do not forget about the implicit lock/unlock statements in `pthread_cond_wait`.

- Be way of "over-locking". For example, doing something the following is generally a bad idea:

```
void *thread_fn(void *arg) {
  pthread_mutex_lock(&lock);
  while(/* some undesired condition */) {
    pthread_cond_wait(&cv, &lock);
  }

  // Code that has special rules, generally a critical section.

    pthread_cond_signal(&cv);
    pthread_mutex_unlock(&lock);
}

```

  Note that in the code above, the **entire** code block is guarded by a single lock, which renders the condition variables pretty much superfluous. Exacly one thread can be that area at a time, so your code will run completely serially, which defeats the purpose of using condition variables.

- If you need to count the number of waiting threads, be careful not to double count a thread that goes back to sleep. For example,

```
pthread_mutex_lock(&lock);
while(/* some undesired condition */) {
  num_waiting_threads++;
  pthread_cond_wait(&cv, &lock);
}
num_waiting_threads--;
    pthread_mutex_unlock(&lock);

```

This can be dangerous if multiple threads are awakened by a broadcast, but only a subset of them exit the loop. In that case, each thread that waits again will double itself as another waiting thread, and the number of waiting threads will never go back to 0.

You can either do this:

```
pthread_mutex_lock(&lock);
num_waiting_threads++;
while(/* some undesired condition */) {
  pthread_cond_wait(&cv, &lock);
}
num_waiting_threads--;
pthread_mutex_unlock(&lock);

```

or,

```
pthread_mutex_lock(&lock);
while(/* some undesired condition */) {
  num_waiting_threads++;
  pthread_cond_wait(&cv, &lock);
  num_waiting_threads--;
}
pthread_mutex_unlock(&lock);

```