30

Condition Variables

Thus far we have developed the notion of a lock and seen how one can be properly built with the right combination of hardware and OS support. Unfortunately, locks are not the only primitives that are needed to build concurrent programs.

In particular, there are many cases where a thread wishes to check whether a **condition** is true before continuing its execution. For example, a parent thread might wish to check whether a child thread has completed before continuing (this is often called a `join()`); how should such a wait be implemented? Let's look at Figure 30.1.

```
void *child(void *arg) {
    printf("child\n");
    // XXX how to indicate we are done?
    return NULL;
}

int main(int argc, char *argv[]) {
    printf("parent: begin\n");
    pthread_t c;
    Pthread_create(&c, NULL, child, NULL); // child
    // XXX how to wait for child?
    printf("parent: end\n");
    return 0;
}
```

Figure 30.1: **A Parent Waiting For Its Child**

What we would like to see here is the following output:

```
parent: begin
child
parent: end
```

We could try using a shared variable, as you see in Figure 30.2. This solution will generally work, but it is hugely inefficient as the parent spins

```
volatile int done = 0;

void *child(void *arg) {
    printf("child\n");
    done = 1;
    return NULL;
}

int main(int argc, char *argv[]) {
    printf("parent: begin\n");
    pthread_t c;
    Pthread_create(&c, NULL, child, NULL); // child
    while (done == 0)
        ; // spin
    printf("parent: end\n");
    return 0;
}
```

Figure 30.2: **Parent Waiting For Child: Spin-based Approach**

and wastes CPU time. What we would like here instead is some way to put the parent to sleep until the condition we are waiting for (e.g., the child is done executing) comes true.

> THE CRUX: HOW TO WAIT FOR A CONDITION
>
> In multi-threaded programs, it is often useful for a thread to wait for some condition to become true before proceeding. The simple approach, of just spinning until the condition becomes true, is grossly inefficient and wastes CPU cycles, and in some cases, can be incorrect. Thus, how should a thread wait for a condition?

30.1 Definition and Routines

To wait for a condition to become true, a thread can make use of what is known as a **condition variable**. A **condition variable** is an explicit queue that threads can put themselves on when some state of execution (i.e., some **condition**) is not as desired (by **waiting** on the condition); some other thread, when it changes said state, can then wake one (or more) of those waiting threads and thus allow them to continue (by **signaling** on the condition). The idea goes back to Dijkstra's use of "private semaphores" [D68]; a similar idea was later named a "condition variable" by Hoare in his work on monitors [H74].

To declare such a condition variable, one simply writes something like this: `pthread_cond_t c;`, which declares `c` as a condition variable (note: proper initialization is also required). A condition variable has two operations associated with it: `wait()` and `signal()`. The `wait()` call is executed when a thread wishes to put itself to sleep; the `signal()` call

```c
int done  = 0;
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t c  = PTHREAD_COND_INITIALIZER;

void thr_exit() {
    Pthread_mutex_lock(&m);
    done = 1;
    Pthread_cond_signal(&c);
    Pthread_mutex_unlock(&m);
}

void *child(void *arg) {
    printf("child\n");
    thr_exit();
    return NULL;
}

void thr_join() {
    Pthread_mutex_lock(&m);
    while (done == 0)
        Pthread_cond_wait(&c, &m);
    Pthread_mutex_unlock(&m);
}

int main(int argc, char *argv[]) {
    printf("parent: begin\n");
    pthread_t p;
    Pthread_create(&p, NULL, child, NULL);
    thr_join();
    printf("parent: end\n");
    return 0;
}
```

Figure 30.3: **Parent Waiting For Child: Use A Condition Variable**

is executed when a thread has changed something in the program and thus wants to wake a sleeping thread waiting on this condition. Specifically, the POSIX calls look like this:

```c
pthread_cond_wait(pthread_cond_t *c, pthread_mutex_t *m);
pthread_cond_signal(pthread_cond_t *c);
```

We will often refer to these as `wait()` and `signal()` for simplicity. One thing you might notice about the `wait()` call is that it also takes a mutex as a parameter; it assumes that this mutex is locked when `wait()` is called. The responsibility of `wait()` is to release the lock and put the calling thread to sleep (atomically); when the thread wakes up (after some other thread has signaled it), it must re-acquire the lock before returning to the caller. This complexity stems from the desire to prevent certain race conditions from occurring when a thread is trying to put itself to sleep. Let's take a look at the solution to the join problem (Figure 30.3) to understand this better.

There are two cases to consider. In the first, the parent creates the child thread but continues running itself (assume we have only a single processor) and thus immediately calls into `thr_join()` to wait for the child thread to complete. In this case, it will acquire the lock, check if the child is done (it is not), and put itself to sleep by calling `wait()` (hence releasing the lock). The child will eventually run, print the message "child", and call `thr_exit()` to wake the parent thread; this code just grabs the lock, sets the state variable `done`, and signals the parent thus waking it. Finally, the parent will run (returning from `wait()` with the lock held), unlock the lock, and print the final message "parent: end".

In the second case, the child runs immediately upon creation, sets `done` to 1, calls signal to wake a sleeping thread (but there is none, so it just returns), and is done. The parent then runs, calls `thr_join()`, sees that `done` is 1, and thus does not wait and returns.

One last note: you might observe the parent uses a `while` loop instead of just an `if` statement when deciding whether to wait on the condition. While this does not seem strictly necessary per the logic of the program, it is always a good idea, as we will see below.

To make sure you understand the importance of each piece of the `thr_exit()` and `thr_join()` code, let's try a few alternate implementations. First, you might be wondering if we need the state variable `done`. What if the code looked like the example below? (Figure 30.4)

Unfortunately this approach is broken. Imagine the case where the child runs immediately and calls `thr_exit()` immediately; in this case, the child will signal, but there is no thread asleep on the condition. When the parent runs, it will simply call wait and be stuck; no thread will ever wake it. From this example, you should appreciate the importance of the state variable `done`; it records the value the threads are interested in knowing. The sleeping, waking, and locking all are built around it.

```
void thr_exit() {
    Pthread_mutex_lock(&m);
    Pthread_cond_signal(&c);
    Pthread_mutex_unlock(&m);
}

void thr_join() {
    Pthread_mutex_lock(&m);
    Pthread_cond_wait(&c, &m);
    Pthread_mutex_unlock(&m);
}
```

Figure 30.4: **Parent Waiting: No State Variable**

```
void thr_exit() {
    done = 1;
    Pthread_cond_signal(&c);
}

void thr_join() {
    if (done == 0)
        Pthread_cond_wait(&c);
}
```

Figure 30.5: **Parent Waiting: No Lock**

Here (Figure 30.5) is another poor implementation. In this example, we imagine that one does not need to hold a lock in order to signal and wait. What problem could occur here? Think about it[1]!

The issue here is a subtle race condition. Specifically, if the parent calls `thr_join()` and then checks the value of `done`, it will see that it is 0 and thus try to go to sleep. But just before it calls wait to go to sleep, the parent is interrupted, and the child runs. The child changes the state variable `done` to 1 and signals, but no thread is waiting and thus no thread is woken. When the parent runs again, it sleeps forever, which is sad.

Hopefully, from this simple join example, you can see some of the basic requirements of using condition variables properly. To make sure you understand, we now go through a more complicated example: the **producer/consumer** or **bounded-buffer** problem.

TIP: ALWAYS HOLD THE LOCK WHILE SIGNALING

Although it is strictly not necessary in all cases, it is likely simplest and best to hold the lock while signaling when using condition variables. The example above shows a case where you *must* hold the lock for correctness; however, there are some other cases where it is likely OK not to, but probably is something you should avoid. Thus, for simplicity, **hold the lock when calling signal**.

The converse of this tip, i.e., hold the lock when calling wait, is not just a tip, but rather mandated by the semantics of wait, because wait always (a) assumes the lock is held when you call it, (b) releases said lock when putting the caller to sleep, and (c) re-acquires the lock just before returning. Thus, the generalization of this tip is correct: **hold the lock when calling signal or wait**, and you will always be in good shape.

[1] Note that this example is not "real" code, because the call to `pthread_cond_wait()` always requires a mutex as well as a condition variable; here, we just pretend that the interface does not do so for the sake of the negative example.