

## 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 an example (Figure 22.1).

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

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

We could try using a shared variable, as follows you see in Figure 22.2. This solution will work (note that you don't even have to lock the variable) but it is hugely inefficient, as the parent spins 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. And thus a new primitive: the **condition variable**.

```

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); // create child
    // XXX how to wait for child?
    printf("parent: end\n");
    return 0;
}

```

Figure 22.1: A Parent waiting for its Child

## 22.1 Definition and Routines

A **condition variable** is an explicit queue that threads can put themselves on when some state of execution is not as desired; some other thread, when it changes said state, can then wake one (or more) of those waiting threads and thus allow them to continue.

To declare such a condition variable, one simply writes something like this:

```
pthread_cond_t c;
```

which declares `c` as a condition variable.

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 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:

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

```
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); // create child
    while (done == 0)
        ; // spin
    printf("parent: end\n");
    return 0;
}
```

Figure 22.2: A Parent waiting for its Child

We will just 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 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. The reason for this complexity is 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 join example now solved with a condition variable in Figure 22.3 to make some sense of this (the code omits the prefix `pthread_` from declarations for brevity).

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 `myjoin()` 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 `cond_wait()` (hence releasing the

```
int done = 0;
mutex_t m;
cond_t c;

// called by child
void myexit() {
    mutex_lock(&m);
    done = 1;
    signal(&c);
    mutex_unlock(&m);
}

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

// called by parent
void myjoin() {
    mutex_lock(&m);
    if (done == 0)
        cond_wait(&c, &m);
    mutex_unlock(&m);
}

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

Figure 22.3: A Parent waiting for its Child

lock). The child will eventually run, print the message “child”, and call `myexit()` 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 `cond_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, and thus sets `done` to 1, calls `signal` to wake a sleeping thread (but there is none, so this just returns), and is done. The parent then runs, calls `myjoin()`, which checks `done` and sees that it is 1 and thus does not wait and returns.

To make sure we understand the importance of each piece of the `myexit()` and `myjoin()` code, let’s try a few alternate implementations. First, you might be wondering if we need the state variable `done`. For example, what if the code looked like the example below. Would this work? (think about it!)

```
void myexit() {
    mutex_lock(&m);
    signal(&c);
    mutex_unlock(&m);
}

void myjoin() {
    mutex_lock(&m);
    cond_wait(&c, &m);
    mutex_unlock(&m);
}
```

Unfortunately this does not work. Imagine the case where the child runs immediately and calls `myexit()` right away; in this case, the child will signal but there is no thread asleep on the condition. Thus, when the parent runs, it will simply call `wait` and be stuck. No thread will ever wake it. From this example, you should be able to understand the importance of the state variable `done`; it records the value the threads are interested in knowing. The sleeping and locking really all are built around it.

Here 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!)

```
void myexit() {
    done = 1;
    signal(&c);
}
```

```
void myjoin() {
    if (done == 0)
        cond_wait(&c);
}
```

The issue here is an even trickier race condition. Specifically, if the parent calls `myjoin()` 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 simply goes to sleep, forever.

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

## 22.2 Producer/Consumer

The final problem we will confront in this note is known as the **producer/consumer** problem, or sometimes as the **bounded buffer** problem, which was also first posed by Dijkstra [D72]. Indeed, it was the producer/consumer problem that led to the invention of the basic primitives of synchronization [D01].

Imagine one or more producer threads and one or more consumer threads. Producers produce data items and wish to place them in a buffer; consumers grab data items out of the buffer consume the data in some way.

This arrangement occurs in many places within real systems. For example, in a multithreaded web server, a producer puts HTTP requests into a work queue (i.e., the bounded buffer); a thread pool of consumers each take a request out of the work queue and process the request. Similarly, when you use a piped command in a UNIX shell, as follows:

```
prompt> cat notes.txt | wc -l
```

This example runs two processes concurrently; `cat` writes the body of the file `notes.txt` to what it thinks is standard output; instead, however, the UNIX shell has redirected the output to what is called a UNIX pipe (created by the `pipe()` system call). The other end of this pipe is connected to the standard input of the process `wc`, which simply counts the number of lines in the input stream and prints out the result. Thus, the `cat` process is the producer, and the `wc` process is the consumer. Between them is a bounded buffer.

Because the bounded buffer is a shared resource, we must of course require synchronized access to it, lest a race condition arise. To begin to understand this problem better, let us examine some actual code:

```
int buffer[MAX];
int fill      = 0;
int use      = 0;
int numfilled = 0;

void put(int value) {
    buffer[fill] = value; // line F1
    fill = (fill + 1) % MAX; // line F2
    numfilled++;
}

int get() {
    int tmp = buffer[use]; // line G1
    use = (use + 1) % MAX; // line G2
    numfilled--;
    return tmp;
}
```

Figure 22.4: The Put and Get Routines

In this example, we assume that the shared buffer `buffer` is just an array of integers (this could easily be generalized to arbitrary objects, of course), and that the `fill` and `use` integers are used as indices into the array, and are used to track where to both put data (`fill`) and get data (`use`).

Let us assume in this simple example that we have just two threads, a producer and a consumer, and that the producer just

writes some number of integers into the buffer which the consumer removes from the buffer and prints:

```
void *producer(void *arg) {
    int i;
    int loops = (int) arg;
    for (i = 0; i < loops; i++) {
        put(i);
    }
}

void *consumer(void *arg) {
    int i;
    while (1) {
        int tmp = get();
        printf("%d\n", tmp);
    }
}
```

Figure 22.5: The Producer and Consumer Threads

Finally, Figure 22.6 presents the main body of the program, which simply creates the one producer and a variable number of consumer threads and waits for them to finish. The producer is puts 1e6 (1 million) integers in the bounded buffer; each consumer takes one of these out at a time and then prints it. The slight ugliness of the code is that it does not (as shown) terminate, for the sake of simplicity (see if you can change the code to make consumers exit after all the data has been produced).

If the program is run with `loops = 5`, and just one producer and one consumer, what we'd like to get is the producer "producing" 0, 1, 2, 3, and 4, and the consumer getting them and subsequently printing them in that order. However, without synchronization, we may not get that. For example, imagine if the consumer thread runs first; it will call `get()` to get data that hasn't even been produced yet, and thus not work as desired. Things get worse when you add multiple producers or consumers, as there could be race conditions in the update of the use or fill indices. Clearly, something is missing.

```
int loops = 0;

int main(int argc, char *argv[] {
    assert(argc == 3);
    loops = atoi(argv[1]);
    numconsumers = atoi(argv[2]);

    pthread_t pid[MAXTHREADS], cid[MAXTHREADS];
    // create threads (producer and consumers)
    Pthread_create(&pid, NULL, producer, NULL);
    for (int i = 0; i < numconsumers; i++)
        Pthread_create(&cid[i], NULL, consumer, NULL);

    // wait for producer and consumers to finish
    Pthread_join(pid, NULL);
    for (int i = 0; i < numconsumers; i++) {
        Pthread_join(cid[i], NULL);
    }
    return 0;
}
```

Figure 22.6: The Main Body of Code

### First Try: Single Condition + If Statement

Let's now try to use a condition variable to solve this problem. To do that, we'll put some calls into the code either wait for the buffer to be empty (or full) and then signals when a buffer is full (or empty). Of course, the proper solution depends on how we go about such code.

Let's take a first try (Figure 22.7). Try to read and understand this code. As it turns out, there are two problems with this solution. See if you can figure them out!

OK, you gave up. Let's understand the first problem. It has to do with the `if` statement before the `wait`. Imagine the following interleaving of threads (assuming 2 consumers and 1 producer). First, a consumer (Ca) runs; it acquires the lock (c1), checks if any buffers are ready for consumption (c2), and finding that none are, waits (c3) (which thus releases the lock). Then a producer (P) runs. It acquires the lock (p1), checks if all buffers

```

cond_t cond;
mutex_t mutex;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        mutex_lock(&mutex);           // p1
        if (numfilled == MAX)         // p2
            cond_wait(&cond, &mutex); // p3
        put(i);                       // p4
        cond_signal(&cond);           // p5
        mutex_unlock(&mutex);         // p6
    }
}

void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        mutex_lock(&mutex);           // line c1
        if (numfilled == 0)           // c2
            cond_wait(&cond, &mutex); // c3
        int tmp = get();              // c4
        cond_signal(&cond);           // c5
        mutex_unlock(&mutex);         // c6
        printf("%d\n", tmp);
    }
}

```

Figure 22.7: Producer and Consumer: Single CV + If Statement

are full (p2), and finding that not to be the case, goes ahead and fills a buffer (p4). Then, the producer signals that a buffer has been filled. Critically, this moves the first consumer (Ca) from sleeping to the ready queue; Ca is now able to run (but not yet running). The producer P then finishes up, unlocking the mutex (p6) and returning back to the top of the loop.

Here is where the problem occurs: another consumer (Cb) comes along and consumes the one existing buffer (it runs through c1, c2, c4, c5, and c6). Now Ca tries to run; just before returning from the wait it re-acquires the lock and then returns. It then calls `get()` (c4), but there are no buffers to consume! Clearly,

we should have somehow prevented Ca from trying to consume because Cb had snuck in and consumed the one buffer that had been produced.

The problem arises for a simple reason: after the producer woke Ca, but *before* Ca ever ran, the state of the bounded buffer changed (thanks to Cb). Signaling a thread only wakes them up; it is thus a *hint* that the state of the world has changed (in this case, that a buffer has been produced), but there is no guarantee that when the woken thread runs, the state will still be as desired. This interpretation of what a signal means is often referred to as **Mesa semantics**, after the first research that built a condition variable in such a manner [LR80]; the contrast, referred to as **Hoare semantics**, is harder to build but provides a stronger guarantee that the woken thread will run immediately upon being woken [H74]. Virtually every system ever built employs Mesa semantics.

Fortunately, the fix is easy: change the `if` to a `while`. Think about why this works; now consumer Ca wakes up and (with the lock held) immediately rechecks the state of the shared variable (c2). If the buffer is empty, the consumer simply goes back to sleep (c3).

Thus, thanks to Mesa semantics, a simple rule to remember with condition variables is to **always use while loops**. Sometimes you don't have to, but it is almost always safe to do so.

#### CODING TIP: WHILE (NOT IF) FOR CONDITIONS

Using a `while` loop is always correct; using an `if` statement only might be, depending on the semantics of signaling (Mesa vs. Hoare). Thus, always use `while` and your code will behave as expected.

## Second Try: Two Conditions

Unfortunately, the code has another problem. It has to do with the fact that there is only a single condition variable. See if you

can figure out a case where the presence of a single condition could cause problems. Do it!

```

cond_t empty, fill;
mutex_t mutex;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        mutex_lock(&mutex);           // p1
        while (numfilled == MAX)     // p2
            cond_wait(&empty, &mutex); // p3
        put(i);                       // p4
        cond_signal(&fill);           // p5
        mutex_unlock(&mutex);         // p6
    }
}

void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        mutex_lock(&mutex);           // c1
        while (numfilled == 0)        // c2
            cond_wait(&fill, &mutex); // c3
        int tmp = get();              // c4
        cond_signal(&empty);          // c5
        mutex_unlock(&mutex);         // c6
        printf("%d\n", tmp);
    }
}

```

Figure 22.8: Producer and Consumer: Working Solution

Now here is an example of why it doesn't work. Imagine two consumers (Ca and Cb) run first, and both get stuck waiting. Then, a producer runs, fills a single buffer, wakes a single consumer (Ca), and then tries to fill again but finds the buffer full (assume here that `MAX=1`). Thus, we have a producer waiting for an empty buffer, a consumer waiting for a full buffer (Cb), and a consumer Ca (who had been waiting) about to run because it has been woken.

The consumer (Ca) then runs and consumes the buffer. When it signals, though, it wakes a single thread that is waiting on the

condition. Because there is only a single condition variable, the consumer might wake the waiting *consumer* (Cb), instead of the waiting producer. We are out of luck.

How do we fix this? It's also not too hard. We need to be more careful exactly whom we wake when the state of the system changes. Specifically, we need different condition variables for producers and consumers to sleep on.

### The Working Solution

Thus, we arrive at the working solution for the producer/consumer problem (Figure 22.8). The code isn't much different than what we began with, but the subtle changes are the difference between a working and buggy solution.

## 22.3 Summary

We have seen the introduction of another important synchronization primitive beyond locks: condition variables. By allowing threads to sleep when the state of the system is not as they desire, conditions enable us to neatly solve a number of important synchronization problems. A more dramatic concluding sentence would go here.

## References

- [D72] "Information Streams Sharing a Finite Buffer"  
E.W. Dijkstra  
Information Processing Letters 1: 179180, 1972  
Available: <http://www.cs.utexas.edu/users/EWD/ewd03xx/EWD329.PDF>
- [D01] "My recollections of operating system design"  
E.W. Dijkstra  
April, 2001  
Available: <http://www.cs.utexas.edu/users/EWD/ewd13xx/EWD1303.PDF>  
*A fascinating read for those of you interested in how the pioneers of our field came up with some very basic and fundamental concepts, including ideas like "interrupts" and "a stack"*
- [H74] "Monitors: An Operating System Structuring Concept"  
C.A.R. Hoare  
Communications of the ACM. Volume 17, Number 10. pages 549-557. October 1974.
- [LR80] "Experience with Processes and Monitors in Mesa"  
B.W. Lampson, D.R. Redell  
Communications of the ACM. Volume 23, Number 2. pages 105-117. February 1980.