29

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 29.1.

```
1  void *
2  child(void *arg) {
3      printf("child\n");
4      // XXX how to indicate we are done?
5      return NULL;
6  }
7
8  int
9  main(int argc, char *argv[]) {
10     printf("parent: begin\n");
11     pthread_t c;
12     Pthread_create(&c, NULL, child, NULL); // create child
13     // XXX how to wait for child?
14     printf("parent: end\n");
15     return 0;
16 }
```

Figure 29.1: 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 29.2. This solution will generally work, 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.

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?

```
1  volatile int done = 0;
2
3  void *
4  child(void *arg) {
5      printf("child\n");
6      done = 1;
7      return NULL;
8  }
9
10 int
11 main(int argc, char *argv[]) {
12     printf("parent: begin\n");
13     pthread_t c;
14     Pthread_create(&c, NULL, child, NULL); // create child
15     while (done == 0)
16         ; // spin
17     printf("parent: end\n");
18     return 0;
19 }
```

Figure 29.2: Parent Waiting For Child: Spin-based Approach

29.1 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 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);
```

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 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 29.3) 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

```
1  int done  = 0;
2  pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
3  pthread_cond_t c  = PTHREAD_COND_INITIALIZER;
4
5  void thr_exit() {
6      Pthread_mutex_lock(&m);
7      done = 1;
8      Pthread_cond_signal(&c);
9      mutex_unlock(&m);
10 }
11
12 void *child(void *arg) {
13     printf("child\n");
14     thr_exit();
15     return NULL;
16 }
17
18 void thr_join() {
19     Pthread_mutex_lock(&m);
20     while (done == 0)
21         Pthread_cond_wait(&c, &m);
22     Pthread_mutex_unlock(&m);
23 }
24
25 int main(int argc, char *argv[]) {
26     printf("parent: begin\n");
27     pthread_t p;
28     Pthread_create(&p, NULL, child, NULL);
29     thr_join();
30     printf("parent: end\n");
31     return 0;
32 }
```

Figure 29.3: Parent Waiting For Child: Use A Condition Variable

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, 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 `thr_join()`, which checks `done` and sees that it 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 we 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`. For example, what if the code looked like the example below. Would this work? (think about it!)

```
void thr_exit() {
    mutex_lock(&m);
    Pthread_cond_signal(&c);
    mutex_unlock(&m);
}

void thr_join() {
    mutex_lock(&m);
    Pthread_cond_wait(&c, &m);
    mutex_unlock(&m);
}
```

Unfortunately this approach does not work. Imagine the case where the child runs immediately and calls `thr_exit()` 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, waking, and locking 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 thr_exit() {
    done = 1;
    Pthread_cond_signal(&c);
}

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

The issue here is an even trickier 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.

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.

CODING 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.

29.2 The Producer/Consumer (Bound Buffer) Problem

The next synchronization 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 this very producer/consumer problem that led Dijkstra and his co-workers to invent the generalized semaphore (which can be used as either a lock or a condition variable) [D01]; we will learn more about semaphores in a future chapter.

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.

```
1  int buffer;
2  int count = 0; // initially, empty
3
4  void put(int value) {
5      assert(count == 0);
6      count = 1;
7      buffer = value;
8  }
9
10 int get() {
11     assert(count == 1);
12     count = 0;
13     return buffer;
14 }
```

Figure 29.4: The Put and Get Routines (Version 1)

This arrangement occurs in many real systems. For example, in a multi-threaded web server, a producer puts HTTP requests into a work queue (i.e., the bounded buffer); consumer threads take requests out of this queue and process them.

A bounded buffer is also used when you pipe the output of one program into another (e.g., `grep foo file.txt | wc -l`). This example runs two processes concurrently; `grep` writes lines from `file.txt` with the string `foo` in them 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 `grep` process is the producer; the `wc` process is the consumer; between them is an in-kernel 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.

The first thing we need is a shared buffer, into which a producer puts data, and out of which a consumer takes data. Let's just use a single integer for simplicity (you can certainly imagine placing a pointer to a data structure into this slot instead), and the two inner routines to put a value into the shared buffer, and to get a value out of the buffer. See Figure 29.4 for details.

Pretty simple, no? The `put()` routine assumes the buffer is empty

```
1  void *producer(void *arg) {
2      int i;
3      int loops = (int) arg;
4      for (i = 0; i < loops; i++) {
5          put(i);
6      }
7  }
8
9  void *consumer(void *arg) {
10     int i;
11     while (1) {
12         int tmp = get();
13         printf("%d\n", tmp);
14     }
15 }
```

Figure 29.5: Producer/Consumer Threads (Version 1)

(and checks this with an assertion), and then simply puts a value into the shared buffer and marks it full by setting `count` to 1. The `get()` routine does the opposite, setting the buffer to empty (i.e., setting `count` to 0) and returning the value.

Now we need to write some routines that know when it is OK to access the buffer to either put data into it or get data out of it. The conditions for this should be obvious: only put data into the buffer when `count` is zero (i.e., when the buffer is empty), and only get data from the buffer when `count` is one (i.e., when the buffer is full). If we write the synchronization code such that a producer puts data into a full buffer, or a consumer gets data from an empty one, we have done something wrong (and in this code, an assertion will fire).

This work is going to be done by two types of threads, one set of which we'll call the **producer** threads, and the other set which we'll call **consumer** threads. Figure 29.5 shows the code for a producer that puts an integer into the shared buffer `loops` number of times, and a consumer that gets the data out of that shared buffer (forever), each time printing it out.

A Broken Solution

Now imagine that we have just a single producer and a single consumer. Obviously the `put()` and `get()` routines have critical sections within them, as `put()` updates the buffer, and `get()` reads from it. However, putting a lock around the code doesn't work; we need something more. Not surprisingly, that something more is some condition variables. Let's try to throw some in there and

```
1  cond_t  cond;
2  mutex_t mutex;
3
4  void *producer(void *arg) {
5      int i;
6      for (i = 0; i < loops; i++) {
7          mutex_lock(&mutex);
8          if (count == 1)
9              cond_wait(&cond, &mutex);
10         put(i);
11         cond_signal(&cond);
12         mutex_unlock(&mutex);
13     }
14 }
15
16 void *consumer(void *arg) {
17     int i;
18     for (i = 0; i < loops; i++) {
19         mutex_lock(&mutex);
20         if (count == 0)
21             cond_wait(&cond, &mutex);
22         int tmp = get();
23         cond_signal(&cond);
24         mutex_unlock(&mutex);
25         printf("%d\n", tmp);
26     }
27 }
```

Figure 29.6: Producer/Consumer: Single CV and If Statement

see what happens. In this (broken) first try (Figure 29.6), we have a single condition variable `cond` and an associated lock `mutex`.

Let's understand the signaling between producers and consumers. When a producer wants to fill the buffer, it first waits for the buffer to be empty (lines 7–9). The consumer has the same logic, but waits for the buffer to become full (19–21).

With just a single producer and a single consumer, the code above works. However, if we have more than one of these threads, the solution has some problems. Can you 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, where we assume there are two consumers (T_{c1} and T_{c2} and one producer, T_p. First, a consumer (T_{c1}) runs; it acquires the lock (line 19), checks if any buffers are ready for consumption (line 20), and finding that none are, waits (line 21) (which thus releases the lock). Then a producer (T_p) runs. It acquires the lock

(line 7), checks if all buffers are full (line 8), and finding that not to be the case, goes ahead and fills a buffer (line 10). Then, the producer signals that a buffer has been filled. Critically, this moves the first consumer (T_{c1}) from sleeping on a condition variable to the ready queue; T_{c1} is now able to run (but not yet running). The producer then finishes, unlocking the mutex (line 12) and continuing to loop.

Here is where the problem occurs: another consumer (T_{c2}) comes along and consumes the one existing value in the buffer (it runs from line 19 through line 25, skipping the wait at 21 because the buffer was full). Now T_{c1} runs; just before returning from the wait it re-acquires the lock and then returns. It then calls `get()` (line 22), but there are no buffers to consume! An assertion triggers, and the code has not worked as desired. Clearly, we should have somehow prevented T_{c1} from trying to consume because T_{c2} had snuck in and consumed the one value in the buffer that had been produced.

The problem arises for a simple reason: after the producer woke T_{c1}, but *before* T_{c1} ever ran, the state of the bounded buffer changed (thanks to T_{c2}). 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 value has been placed in the buffer), 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.

Better, But Still Broken: While, Not If

Fortunately, this fix is easy (Figure 29.7): change the `if` to a `while`. Think about why this works; now consumer T_{c1} wakes up and (with the lock held) immediately re-checks the state of the shared variable (line 20). If the buffer is empty at that point, the consumer simply goes back to sleep (line 21). The corollary `if` is also changed to a `while` in the producer (line 8).

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 always safe to do so.

```
1  cond_t  cond;
2  mutex_t mutex;
3
4  void *producer(void *arg) {
5      int i;
6      for (i = 0; i < loops; i++) {
7          mutex_lock(&mutex);
8          while (count == 1)
9              cond_wait(&cond, &mutex);
10         put(i);
11         cond_signal(&cond);
12         mutex_unlock(&mutex);
13     }
14 }
15
16 void *consumer(void *arg) {
17     int i;
18     for (i = 0; i < loops; i++) {
19         mutex_lock(&mutex);
20         while (count == 0)
21             cond_wait(&cond, &mutex);
22         int tmp = get();
23         cond_signal(&cond);
24         mutex_unlock(&mutex);
25         printf("%d\n", tmp);
26     }
27 }
```

Figure 29.7: Producer/Consumer: Single CV and While

However, this code still has a bug, the second of two problems mentioned above. Can you see it? It has something to do with the fact that there is only one condition variable. Try to figure out what the problem is, before reading ahead. DO IT!

Let's confirm you figured it out correctly. The problem occurs when two consumers run first (T_{c1} and T_{c2}), and both go to sleep (line 21). Then, a producer runs, put a value in the buffer, wakes one of the consumers (say T_{c1}), and then goes back to sleep. Now we have one consumer ready to run (T_{c1}), and two threads sleeping on a condition (T_{c2} and T_p).

The consumer T_{c1} then wakes (returning from `wait()` at line 21), re-checks the condition (line 20), and finding the buffer full, consumes the value (line 22). This consumer then, critically, signals on the condition, waking one thread that is sleeping. However, which thread should be woken?

Because the consumer has emptied the buffer, it clearly should wake the producer. However, if it wakes the consumer T_{c2} (definitely possible depending on how the wait queue is managed), we have a problem. Specifically, the consumer T_{c2} will wake up and find the buffer empty (line 20), and go back to sleep (line 21). The producer T_p, which has a value to put into the buffer, is left sleeping. The other consumer thread, T_{c1}, also goes back to sleep. All three threads are left sleeping, a clear correctness bug.

The Single Buffer Producer/Consumer Solution

The solution here is once again a small one: use *two* condition variables, instead of one, in order to properly signal which type of thread should wake up when the state of the system changes. Figure 29.8 shows the resulting code.

```
1  cond_t  empty, fill;
2  mutex_t mutex;
3
4  void *producer(void *arg) {
5      int i;
6      for (i = 0; i < loops; i++) {
7          mutex_lock(&mutex);
8          while (count == 1)
9              cond_wait(&empty, &mutex);
10         put(i);
11         cond_signal(&fill);
12         mutex_unlock(&mutex);
13     }
14 }
15
16 void *consumer(void *arg) {
17     int i;
18     for (i = 0; i < loops; i++) {
19         mutex_lock(&mutex);
20         while (count == 0)
21             cond_wait(&fill, &mutex);
22         int tmp = get();
23         cond_signal(&empty);
24         mutex_unlock(&mutex);
25         printf("%d\n", tmp);
26     }
27 }
```

Figure 29.8: Producer/Consumer: Two CVs and While

```
1  int buffer[MAX];
2  int fill  = 0;
3  int use   = 0;
4  int count = 0;
5
6  void put(int value) {
7      buffer[fill] = value;
8      fill = (fill + 1) % MAX;
9      count++;
10 }
11
12 int get() {
13     int tmp = buffer[use];
14     use = (use + 1) % MAX;
15     count--;
16     return tmp;
17 }
```

Figure 29.9: The Final Put and Get Routines

In the code above, producer threads wait on the condition **empty**, and signals **fill**. Conversely, consumer threads wait on **fill** and signal **empty**. By doing so, the second problem above is avoided by design: a consumer can never accidentally wake a consumer, and a producer can never accidentally wake a producer.

The Final Producer/Consumer Solution

We now have a working producer/consumer solution, albeit not a fully general one. The last change we make is to enable more concurrency and efficiency; specifically, we add more buffer slots, so that multiple values can be produced before sleeping, and similarly multiple values can be consumed before sleeping. With just a single producer and consumer, this approach is more efficient as it reduces context switches; with multiple producers or consumers (or both), it even allows concurrent producing or consuming to take place, thus increasing parallelism.

The first change for this final solution is within the buffer structure itself and the corresponding `put()` and `get()` (Figure 29.9). We also slightly change the conditions that producers and consumers check in order to determine whether to sleep or not. Figure 29.10 shows the final waiting and signaling logic. Basically, a producer only sleeps if all the buffers are currently filled (line 8); similarly, a consumer only sleeps if all the buffers are currently empty (line 20).

```
1  cond_t empty, fill;
2  mutex_t mutex;
3
4  void *producer(void *arg) {
5      int i;
6      for (i = 0; i < loops; i++) {
7          mutex_lock(&mutex);
8          while (count == MAX)
9              cond_wait(&empty, &mutex);
10         put(i);
11         cond_signal(&fill);
12         mutex_unlock(&mutex);
13     }
14 }
15
16 void *consumer(void *arg) {
17     int i;
18     for (i = 0; i < loops; i++) {
19         mutex_lock(&mutex);
20         while (count == 0)
21             cond_wait(&fill, &mutex);
22         int tmp = get();
23         cond_signal(&empty);
24         mutex_unlock(&mutex);
25         printf("%d\n", tmp);
26     }
27 }
```

Figure 29.10: The Final Working Solution

CODING TIP: WHILE (NOT IF) FOR CONDITIONS

When checking for a condition in a multi-threaded program, using a `while` loop is always correct; using an `if` statement only might be, depending on the semantics of signaling. Thus, always use `while` and your code will behave as expected.

Using while loops around conditional checks also handles the case where **spurious wakeups** occur. In some thread packages, due to details of the implementation, it is possible that two threads get woken up though just a single signal has taken place [L11]. Spurious wakeups are further reason to re-check the condition a thread is waiting on.

```
1  // how many bytes of the heap are free?
2  int bytesLeft = MAX_HEAP_SIZE;
3
4  // need lock and condition too
5  cond_t  c;
6  mutex_t m;
7
8  void *
9  allocate(int size) {
10     lock(&m);
11     while (bytesLeft < size)
12         cond_wait(&c, &m);
13     void *ptr = ...; // get mem from heap
14     bytesLeft -= size;
15     unlock(&m);
16     return ptr;
17 }
18
19 void free(void *ptr, int size) {
20     lock(&m);
21     bytesLeft += size;
22     cond_signal(&c); // whom to signal??
23     unlock(&m);
24 }
```

Figure 29.11: Covering Conditions: An Example

29.3 Covering Conditions

Before closing, we'll look at one more example of how condition variables can be used. This code study is drawn from Lampson and Redell's paper on Pilot [LR80], the same group who first implemented the "Mesa" semantics described above (the language they used was called Mesa, and hence the name).

The problem they ran into is best shown via simple example, in this case in a simple multi-threaded memory allocation library. Below is a code snippet which demonstrates the issue.

As you might see in the code snippet, when a thread calls into the memory allocation code, it might have to wait in order for more memory to become free. Conversely, when a thread frees memory, it signals that more memory is free. However, our code above has a problem: which waiting thread (there can be more than one) should be woken up?

Consider the following scenario. Assume there are zero bytes free; thread T_a calls `allocate(100)`, followed by thread T_b which calls

`allocate(10)`. Both T_a and T_b thus wait on the condition and go to sleep; there aren't enough free bytes to satisfy either request.

At that point, assume a third thread, T_c, comes along and calls `free(50)`. Unfortunately, when it calls signal to wake a waiting thread, it might not wake the correct waiting thread, T_b, which is waiting for only 10 bytes to be freed (T_a should still wait, as not enough memory is yet free). Thus, the code above does not work, as the thread waking other threads does not know which thread (or threads) to wake up.

The solution suggested by Lampson and Redell is straightforward: replace the `cond_signal()` call in the code above with a call to `cond_broadcast()`, which wakes up *all* waiting threads. By doing so, we guarantee that any threads that should be woken are. The downside, of course, can be a negative performance impact, as we might needlessly wake up many other waiting threads that shouldn't (yet) be awake. Those threads will simply wake up, re-check the condition, and then go immediately back to sleep.

Lampson and Redell call such a condition a **covering condition**, as it covers all the cases where a thread needs to wake up (conservatively); the cost, as we've discussed, is that too many threads might be woken. The astute reader might also have noticed we could have used this approach earlier (see the producer/consumer problem with only a single condition variable). However, in that case, a better solution was available to us, and thus we used it. In general, if you find that your program only works when you change your signals to broadcasts (but you don't think it should need to), you probably have a bug; fix it! But in cases like the memory allocator above, broadcast may be the most straightforward solution available.

29.4 Summary

We have seen the introduction of another important synchronization primitive beyond locks: condition variables. By allowing threads to sleep when some program state is not as desired, CVs enable us to neatly solve a number of important synchronization problems, including the famous (and still important) producer/consumer problem, as well as covering conditions. A more dramatic concluding sentence would go here, such as "He loved Big Brother" [O49].

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
The famous paper that introduced the producer/consumer problem.

[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 even "a stack"!

[H74] "Monitors: An Operating System Structuring Concept"
C.A.R. Hoare
Communications of the ACM, 17:10, pages 549–557, October 1974
Hoare did a fair amount of theoretical work in concurrency. However, he is still probably most known for his work on Quicksort, the coolest sorting algorithm in the world, at least according to these authors.

[L11] "Pthread_cond_signal Man Page"
Available: http://linux.die.net/man/3/pthread_cond_signal
March, 2011
The Linux man page shows a nice simple example of why a thread might get a spurious wakeup, due to race conditions within the signal/wakeup code.

[LR80] "Experience with Processes and Monitors in Mesa"
B.W. Lampson, D.R. Redell
Communications of the ACM. 23:2, pages 105-117, February 1980
A terrific paper about how to actually implement signaling and condition variables in a real system, leading to the term "Mesa" semantics for what it means to be woken up; the older semantics, developed by Tony Hoare [H74], then became known as "Hoare" semantics, which is hard to say out loud in class with a straight face.

[O49] "1984"
George Orwell, 1949, Secker and Warburg
A little heavy-handed, but of course a must read. That said, we kind of gave away the ending by quoting the last sentence. Sorry! And if the government is reading this, let us just say that we think that the government is "double plus good".