One last primitive we are going to look at is *condition variables*
One last primitive we are going to look at is *condition variables*

As the name suggests, it is a way a thread can wait until some condition is true
One last primitive we are going to look at is *condition variables*

As the name suggests, it is a way a thread can wait until some condition is true

The idea is that one or more threads can wait on a condition variable until another signals that the required condition is now true
One last primitive we are going to look at is *condition variables*

As the name suggests, it is a way a thread can wait until some condition is true.

The idea is that one or more threads can wait on a condition variable until another signals that the required condition is now true.

The signal can either let just *one* thread continue, or be a *broadcast* that lets all waiting threads continue.
One last primitive we are going to look at is *condition variables*.

As the name suggests, it is a way a thread can wait until some condition is true.

The idea is that one or more threads can wait on a condition variable until another signals that the required condition is now true.

The signal can either let just *one* thread continue, or be a *broadcast* that lets all waiting threads continue.

Condition variables are normally associated with a mutex, and are used *inside* a critical region protected by that mutex.
Concurrent Primitives

Condition Variables

1
get_lock(mx);
condvar_wait(cv, mx);
(wait)
free_lock(mx);

2
get_lock(mx);
condvar_signal(cv);
free_lock(mx);

condvar_wait releases the mutex and waits on the condition variable
Concurrency Primitives
Condition Variables

1
get_lock(mx);
<CR>
condvar_wait(cv, mx);
(wait)
<CR>
free_lock(mx);

2
get_lock(mx);
<CR>
condvar_signal(cv);
free_lock(mx);

condvar_wait releases the mutex and waits on the condition variable

When the other thread signal signals and releases the mutex, the first thread regains the mutex and continues within the critical region
Concurrenty Primitives

Condition Variables

The condition variable allows thread 1 to “step outside” the critical region, letting another thread to enter and do something
Concurrent Primitives

Condition Variables

The condition variable allows thread 1 to “step outside” the critical region, letting another thread to enter and do something.

Conditions variables combine mutual exclusion and synchronisation.
The condition variable allows thread 1 to “step outside” the critical region, letting another thread to enter and do something.

Conditions variables combine mutual exclusion and synchronisation.

Again, not fair on which thread gets to continue if more than one is waiting.
The condition variable allows thread 1 to “step outside” the critical region, letting another thread to enter and do something.

Conditions variables combine mutual exclusion and synchronisation.

Again, not fair on which thread gets to continue if more than one is waiting.

With a broadcast all other threads are marked as ready to run, but only one will regain the lock; the others will blocked on the lock as normal.
Concurrent Primitives

POSIX Condition Variables

```c
#include <pthread.h>
int pthread_cond_init(pthread_cond_t *restrict cond,
                      const pthread_condattr_t *restrict attr);
int pthread_cond_destroy(pthread_cond_t *cond);
int pthread_cond_wait(pthread_cond_t *restrict cond,
                      pthread_mutex_t *restrict mutex);
int pthread_cond_timedwait(pthread_cond_t *restrict cond,
                            pthread_mutex_t *restrict mutex,
                            const struct timespec *restrict abstime);
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
```
As an example of the kind of grungy detail that parallelism has to address: POSIX recognises that there is a nasty implementation detail that would otherwise make implementing condition variables impractical
Concurrency Primitives
POSIX Condition Variables

As an example of the kind of grungy detail that parallelism has to address: POSIX recognises that there is a nasty implementation detail that would otherwise make implementing condition variables impractical.

The specification for `pthread_cond_signal` says

“The `pthread_cond_signal()` function shall unblock at least one of the threads that are blocked on the specified condition variable `cond`”
As an example of the kind of grungy detail that parallelism has to address: POSIX recognises that there is a nasty implementation detail that would otherwise make implementing condition variables impractical.

The specification for `pthread_cond_signal` says

"The `pthread_cond_signal()` function shall unblock at least one of the threads that are blocked on the specified condition variable `cond`"

"at least one": there is a (rare) problem of spurious wakeups that is in general too expensive to avoid.
This just means you have to be a bit formulaic about the use of condition variables and always have a *condition* to test before continuing.

```
1
iteration = 0;
get_lock(mx);
<CR>
it = iteration;
while (it == iteration) condvar_signal(cv, mx);
   condvar_wait(cv, mx);
<CR>
free_lock(mx);
```

```
2
get_lock(mx);
<CR>
iteration++;
condvar_signal(cv, mx);
free_lock(mx);
```

Thread 1 might get awoken spuriously but it doesn’t want to continue until the next iteration.
In general you would test for whatever condition you were waiting for: thread 2 sets the condition, thread 1 should test for it.
In general you would test for whatever condition you were waiting for: thread 2 sets the condition, thread 1 should test for it

Condition variables are very useful, but a bit of a pain to use
We have called these things *primitives*, but we can implement them in terms of each other.
We have called these things *primitives*, but we can implement them in terms of each other

Exercise. Do this
Concurrency Primitives

We have called these things *primitives*, but we can implement them in terms of each other

Exercise. Do this

All eventually go back to the underlying hardware or software support
We have called these things *primitives*, but we can implement them in terms of each other

Exercise. Do this

All eventually go back to the underlying hardware or software support

“Primitive” is actually a good description as they are all very low level
And they do have a cost, thus their use does limit the speedup available
And they do have a cost, thus their use does limit the speedup available

Their overhead can be divided into two parts
Concurrency Primitives

And they do have a cost, thus their use does limit the speedup available.

Their overhead can be divided into two parts:

(a) the time spent blocked as a necessary part of its function, e.g., wait on a lock.
And they do have a cost, thus their use does limit the speedup available.

Their overhead can be divided into two parts:

(a) the time spent blocked as a necessary part of its function, e.g., wait on a lock.

(b) the time spent in executing the code of the primitive.
And they do have a cost, thus their use does limit the speedup available

Their overhead can be divided into two parts

(a) the time spent blocked as a necessary part of its function, e.g., wait on a lock
(b) the time spent in executing the code of the primitive

Note part (a) isn’t really a limitation of the primitive: it’s necessary if it is to work at all. It is (b) that the implementation of a primitive seeks to minimise