# Condition variables

* Condition variables provide yet another way for threads to synchronize.
* While mutexes implement synchronization by controlling thread access to data, *condition variables allow threads to synchronize based upon the actual value of data*.
* A condition variable allows one thread to inform other threads about changes in the state of a shared variable and allows the other threads to wait (block) for such notification.
* Without condition variables, the programmer would need to have threads continually polling (possibly in a critical section), to check if the condition is met. This can be very resource consuming since the thread would be continuously busy in this activity. A condition variable is a way to achieve the same goal without polling.
* A condition variable is always used in conjunction with a mutex lock.
* An example for using condition variables is shown below.

Example

Main Thread

1. Declare and initialize global data/variables which require synchronization (such as nThds)
2. Declare and initialize a condition variable object
3. Declare and initialize an associated mutex
4. Create threads A and B to do work

static int ThreadNum = 0;

static pthread\_mutex\_t mtx = PTHREAD\_MUTEX\_INITIALIZER;

static pthread\_cond\_t cond = PTHREAD\_COND\_INITIALIZER;

Thread A

1. Do work up to the point where a certain condition must occur (such as ThreadNum much reach a value of 32)
2. Lock associated mutex and check value of a global variable
3. Call pthread\_cond\_wait() to perform a blocking wait for signal from Thread B. Note that a call to pthread\_cond\_wait() automatically and atomically unlocks the associated mutex variable so that it can be used by Thread B.
4. When signalled, wake up. Mutex is automatically and atomically locked.
5. Explicitly unlock mutex
6. Continue

s = pthread\_mutex\_lock(&mtx);

if (s != 0) {

printf(“pthread\_mutex\_lock error”);

exit((EXIT\_FAILURE);

}

while (ThreadNum < nThds){

s = pthread\_cond\_wait(&cond, &mtx);

if (s != 0) {

printf(“pthread\_cond\_wait error”);

exit((EXIT\_FAILURE);

}

}

…….

s = pthread\_mutex\_unlock(&mtx);

if (s != 0) {

printf(“pthread\_mutex\_unlock error”);

exit((EXIT\_FAILURE);

}

Thread B

1. Do work
2. Lock associated mutex
3. Change the value of the global variable that Thread A is waiting upon.
4. Check value of the global Thread A wait variable. If it fulfills the desired condition, signal Thread A.
5. Unlock mutex.
6. Continue

s = pthread\_mutex\_lock(&mtx);

if (s != 0) {

printf(“pthread\_mutex\_lock error”);

exit((1);

}

ThreadNum++;

s = pthread\_mutex\_unlock(&mtx);

if (s != 0) {

printf(“pthread\_mutex\_unlock error”);

exit((1);

}

if (ThreadNum == nThds) {

s = pthread\_cond\_signal(&cond);

if (s != 0) {

printf(“pthread\_cond\_signal error”);

exit((1);

}

}

## API

* Condition variables must be declared with type pthread\_cond\_t, and must be initialized before they can be used. There are two ways to initialize a condition variable:

1. Statically, when it is declared. For example:   
   pthread\_cond\_t myconvar = PTHREAD\_COND\_INITIALIZER;
2. Dynamically, with the pthread\_cond\_init() routine. The ID of the created condition variable is returned to the calling thread through the *condition* parameter. This method permits setting condition variable object attributes, *attr*.

Waiting and Signaling on Condition Variables

pthread\_cond\_wait(pthread\_cond\_t \*, pthread\_mutex\_t \*);

pthread\_cond\_signal(pthread\_cond\_t \*);

pthread\_cond\_broadcast(pthread\_cond\_t \*);

All return 0 on success, mpositive error number on error

* pthread\_cond\_wait() blocks the calling thread until the specified *condition* is signalled. This routine should be called while *mutex* is locked, and it will automatically release the mutex while it waits. After signal is received and thread is awakened, *mutex* will be automatically locked for use by the thread. The programmer is then responsible for unlocking *mutex* when the thread is finished with it.

**Recommendation:** Using a WHILE loop instead of an IF statement to check the waited for condition can help deal with several potential problems.

* The pthread\_cond\_signal() routine is used to signal (or wake up) another thread which is waiting on the condition variable. It should be called after *mutex* is locked, and must unlock *mutex* in order for pthread\_cond\_wait() routine to complete.
* The pthread\_cond\_broadcast() routine should be used instead of pthread\_cond\_signal() if more than one thread is in a blocking wait state.

## It is a logical error to call pthread\_cond\_signal() before calling pthread\_cond\_wait().

* Proper locking and unlocking of the associated mutex variable is essential when using these routines. For example:
* Failing to lock the mutex before calling pthread\_cond\_wait() may cause it NOT to block.
* Failing to unlock the mutex after calling pthread\_cond\_signal() may not allow a matching pthread\_cond\_wait() routine to complete (it will remain blocked).

Dynamically Allocated Condition variables

int pthread\_cond\_init (pthread\_cond\_t \* cond,

const pthread\_condattr\_t \* attr);

pthread\_cond\_destroy(pthread\_cond\_t \*);

Returns 0 on success, positive error number on error

* The cond argument identifies the condition variable to be initialized.
* The optional *attr* object is used to set condition variable attributes. There is only one attribute defined for condition variables: process-shared, which allows the condition variable to be seen by threads in other processes. The attribute object, if used, must be of type pthread\_condattr\_t (may be specified as NULL to accept defaults).
* pthread\_cond\_destroy() should be used to free a condition variable that is no longer needed.