# Chapter 27 Thread API

Single thread

- passed a few arguments via myarg_t struct
- To return values, myret_t struct
- Main thread is waiting due to the pthread_join()
  - pthread_join() is needed to capture the output from the mythread() function.
- Once thread finished running
  - Main thread resumes because pthread_join returns
  - Main thread can now access the returned values in myret_t struct

To return values from the mythread() function, we had to allocate myret_t in the heap

If we allocated myret_t on the stack, we would pop the mythread() functions stack frame from the stack, and access the myret_t struct would no longer be valid accesses. As indicated by the compiler warning.

Not much point in doing what we just did…

Much easier to do with simple procedure call!

But it is common that one or more threads wait for all to complete!

Not all multi-threaded programs use join.

MT web server:

- create many worker thread
- Use main thread to accept network connections
- Pass them on to worker threads
- Do this indefinitely

Where to use Join:

- Parallel programs: create threads to execute tasks in parallel, and main thread use join to wait for all to complete before moving on to the next stage in the computation.

**27.3 Locks**

Functions for providing mutual exclusion to critical sections via locks.

```
int pthread_mutux_lock(pthread_mutex_t *mutex);
int pthread_mutux_unlock(pthread_mutex_t *mutex);
```

Ex.

```
int rc = pthread_mutex_init(&lock, NULL);
assert(rc == 0);

pthread_mutex_lock(&lock);
x = x + 1;  // here goes your critical section
pthread_mutex_unlock(&lock);
```

Intent of this code:
- if no other thread holds the lock (when pthread_mutex_lock()) is called
    - the thread will acquire the lock and enter the critical section
- otherwise, another thread holds the lock
    - the thread blocks until it has acquired the lock
    - (this means: doesn't return from pthread_mutex_lock())
    - Implies that the thread holding the lock eventually called unlock()

Many threads may be stuck waiting inside the lock acquisition function
Only the thread that acquired the lock should call unlock.

PS: When done with lock; should call pthread_mutex_destroy().

### 27.4 Condition Variables

**Condition variables (CV)** are useful when threads need to signal certain events to other threads.
- .e.g if one thread is waiting for another to do something before it can continue

```
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
int pthread_cond_signal(pthread_cond_t *cond);
```

To use a CV function above, must hold a lock associated with this condition.

cond_wait(): puts calling thread to sleep; waiting for another thread to signal it.
(when some condition has changed that it may care about)

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

```
pthread_mutex_lock(&lock);
while (ready == 0) {
        pthread_cond_wait(&cond, &lock);
}
// do what ready means for us to do… (Critical section)
pthread_mutex_unlock(&lock);
```

Check if ready != 0 before we can do something…
If ready is still 0, wait (goes to sleep) for other thread to signal (wake it up).

Other thread:

```
pthread_mutex_lock(&lock);
// doing stuff
ready = 1;
pthread_cond_signal(&cond);
pthread_mutex_unlock(&lock);
```

Important: both threads must hold the lock when signaling or waiting.

Note: the cond_wait() takes both cond and lock as argument.

This is because wait() calls unlock() to release the lock when going to sleep.

Otherwise: how could the other thread acquire the lock and signal it to wake up?

However, before returning after being woken, cond_wait() re-acquires the lock,
- this ensures that when it (the thread that waited) is running it has the lock.

**Thread API Guidelines**

- Keep it simple
  - Any code to lock or signal between threads should be as simple as possible
  - Tricky thread interactions lead to subtle bugs (deadlocks)
  - (This applies to channel-based interactions as well — Go)
- Minimize thread interactions
  - Keep the number of ways in which threads interact to a minimum
  - Each interaction should be carefully though out and constructed with well-known patterns
- Initialize locks and condition variables (in C)
  - Failure to initialize: sometimes works fine and sometimes fails in strange ways
- Check return codes:
  - Return codes in C and Unix contain info: should be checked
- Be careful about how you pass arguments to and return values from threads
  - If pass by reference to a variable allocated on the stack: you are doing it wrong!
- Each thread has its own stack
  - A thread's locally allocated variables should be considered private
  - To share data between threads
    - Allocate space on the heap

    - Or use a global variable
- Always use condition variables to signal between threads
    - Don’t use a simple flag!
    - Good alternative to CVs: CSP and channels in Go.