21. Thread API

How to Create and Control Threads?

Thread Creation

```
#include <pthread.h>
int
pthread_create(pthread_t       *thread,
          const pthread_attr_t *attr,
                void           *(*start_routine)(void*),
                void           *arg);
```

Thread Completion

: if you want to wait for a thread to complete

```
int pthread_join(pthread_t thread, void **value_ptr);
```

second argument : a pointer to the return value you expect to get back

Don't return stack value

```
1  void *mythread(void *arg) {
2      myarg_t *args = (myarg_t *) arg;
3      printf("%d %d\n", args->a, args->b);
4      myret_t oops; // ALLOCATED ON STACK: BAD!
5      oops.x = 1;
6      oops.y = 2;
7      return (void *) &oops;
8  }
```

the variable oops is allocated on the stack of mythread
=> when it returns, the value is automatically deallocated

Not all code that is multi-threaded uses the join routine
: only it needs synchronization

Locks

mutual exclusion to a critical section

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

if another thread holds the lock,
the thread trying to grab the lock will not return from the call until it has acquired the lock
= that the thread holding the lock has released it via the unlock call

```
pthread_mutex_t lock;
pthread_mutex_lock(&lock);
x = x + 1; // or whatever your critical section is
pthread_mutex_unlock(&lock);
```

All locks must be properly initialized to guarantee that they have the correct values to begin with
=> work as desired when lock & unlock are called

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
```

or

```
int rc = pthread_mutex_init(&lock, NULL);
assert(rc == 0); // always check success!
```

+ pthread_mutex_destroy() should also be made (don't need in former)

Check error codes

lock acquisition

```
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_timedlock(pthread_mutex_t *mutex,
                            struct timespec *abs_timeout);
```

- trylock : returns failure if the lock is already held
- timedlock : returns after a timeout or after acquiring the lock, whichever happens first.

Condition Variables

useful when some kind of signaling must take place between threads,
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 condition variable,
we need a **lock** that is associated with this condition.
When calling either of the above routines, this lock should be held

pthread_cond_wait()

puts the calling thread to sleep => waits for some other thread to signal it
usually when something in the program has changed that the now-sleeping thread might 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);
Pthread_mutex_unlock(&lock);
```

doesn't mean doing double-layered lock but just timely release lock
sleeping doens't consume CPU cycles.

What Pthread_cond_wait(%cond, &lock); do exactly?
: The thread always holds the lock during pthread_cond_wait(), except while it's actually waiting for the condition to be signaled.

1. Releases the mutex lock : so other threads can acquire it and modify shared data (like ready).
2. Puts your thread to sleep : waiting for the condition variable cond to be signaled.
3. When the thread wakes up (because another thread called pthread_cond_signal()), it *re-acquires* the mutex lock before pthread_cond_wait() returns.

when wait, it release the lock.
If we DON'T, How the hell the other thread acquire the lock & signal it to wake up? : infinitely stuck in Pthread_mutex_lock(&lock) ;

Why we use while loop?
: simple and safe thing todo.

- it rechecks the condition => perhaps adding a little overhead
- there are some pthread implementations that could spuriously wake up a waiting thread.

=> without rechecking, the waiting thread will continue thinking that the condition has changed even though it has not.

It is safer to view waking up as a hint that something might have changed, rather than an absolute fact.
receiving signal != got the exact condition we actually want

the code to wake a thread, which would run in some other thread

```
Pthread_mutex_lock(&lock);
ready = 1;
Pthread_cond_signal(&cond);
Pthread_mutex_unlock(&lock);
```

What if we just only use a simple flag? : ad hoc synchronization

this in the waiting code:

```
while (ready == 0)
    ; // spin
```

The associated signaling code would look like this:

```
ready = 1;
```

Don't do this.
1. performs poorly in many cases : spinning for a long time just wastes CPU cycles.
2. error prone to synchronize between threads

Use condition variables even when you think you can get away without doing so.

Thread API Guidelines

1. Keep it simple : tricky thread interactions lead to bugs
2. Minimize thread interactions
3. initialize locks and condition variables
4. check your return codes
5. Be careful with how you pass arguments to, and return values from, threads.
6. Each thread has its own stack : shared data must be in the heap or global
7. Always use condition variables to signal between threads
8. Use the manual pages.

Trace threads : Helgrind

valgrind --tool=helgrind ./mainR
Tools like Helgrind are **runtime tools** — they detect problems that **actually happen** during a specific execution.

you need to lock to every critical section access.