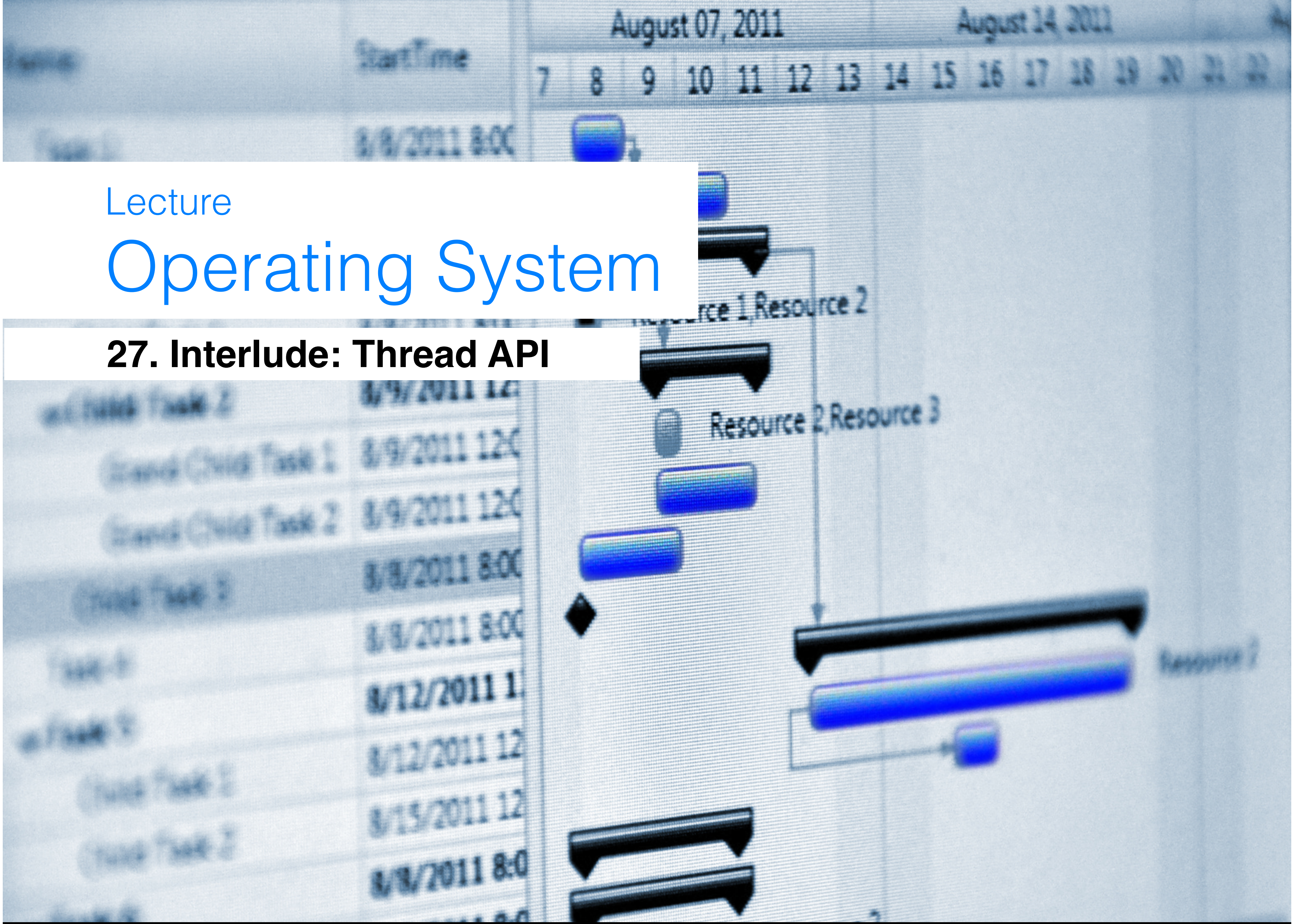


Lecture

Operating System

27. Interlude: Thread API

Professor Dr. Michael Mächtel

27. Interlude: Thread API

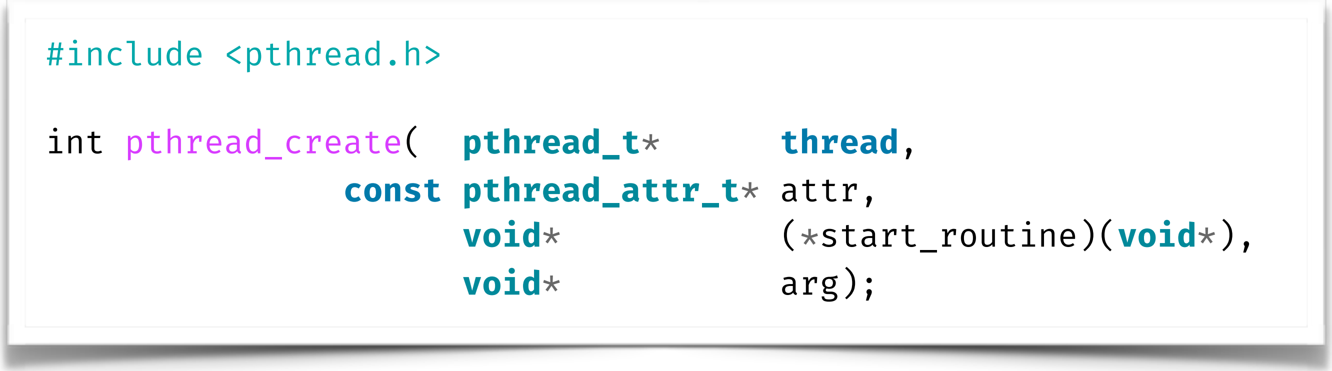
1. **Create**
2. **Wait**
3. **Lock**
4. **CV**

27. Interlude: Thread API

1. **Create**
2. Wait
3. Lock
4. CV

Thread Creation

- How to create and control threads?
- `pthread_create(thread, attr, start_routine, arg)`
    - `thread`: Used to interact with this thread.
    - `attr`: Used to specify any attributes this thread might have.
        - Stack size, Scheduling priority, …
    - `start_routine`: the function this thread start running in.
    - `arg`: the argument to be passed to the function (start routine)
        - a void pointer allows us to pass in any type of argument.



```
#include <pthread.h>

int pthread_create(  pthread_t*       thread,
               const pthread_attr_t* attr,
                     void*            (*start_routine)(void*),
                     void*            arg);
```

Thread Creation (Cont.)

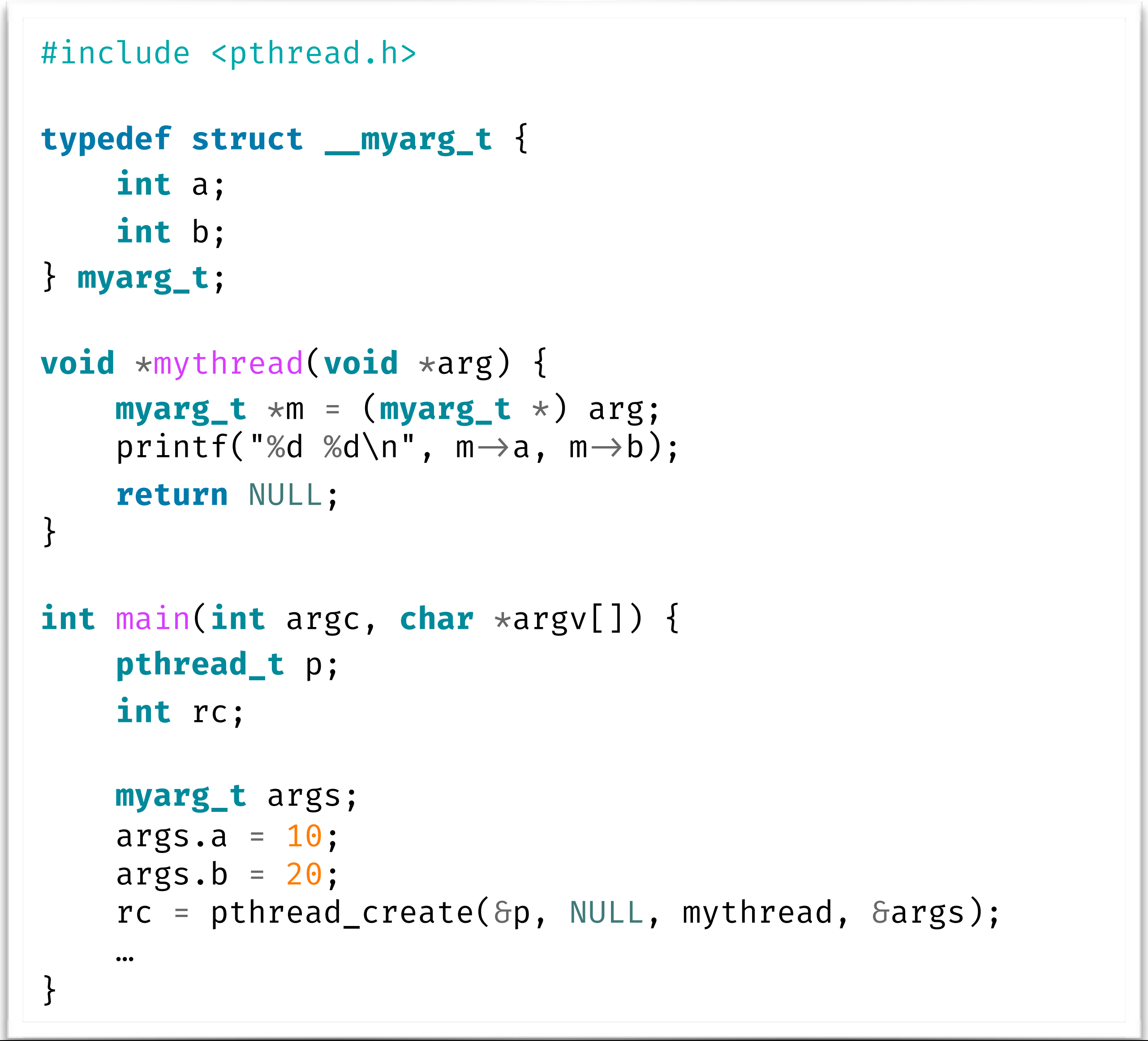
- If start_routine instead required another type argument, the declaration would look like this:
  - An integer argument:

```
int pthread_create(…, // first two args are the same
                     void*  (*start_routine)(int),
                     int    arg);
```

  - Return an integer:

```
int pthread_create(…, // first two args are the same
                     int  (*start_routine)(void*),
                     void*    arg);
```

Example: Creating a Thread



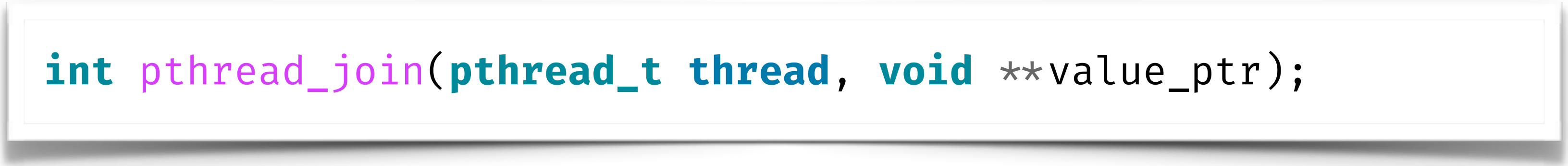
```c
#include <pthread.h>

typedef struct __myarg_t {
    int a;
    int b;
} myarg_t;

void *mythread(void *arg) {
    myarg_t *m = (myarg_t *) arg;
    printf("%d %d\n", m->a, m->b);
    return NULL;
}

int main(int argc, char *argv[]) {
    pthread_t p;
    int rc;

    myarg_t args;
    args.a = 10;
    args.b = 20;
    rc = pthread_create(&p, NULL, mythread, &args);
    …
}
```

27. Interlude: Thread API

1. Create
2. **Wait**
3. Lock
4. CV

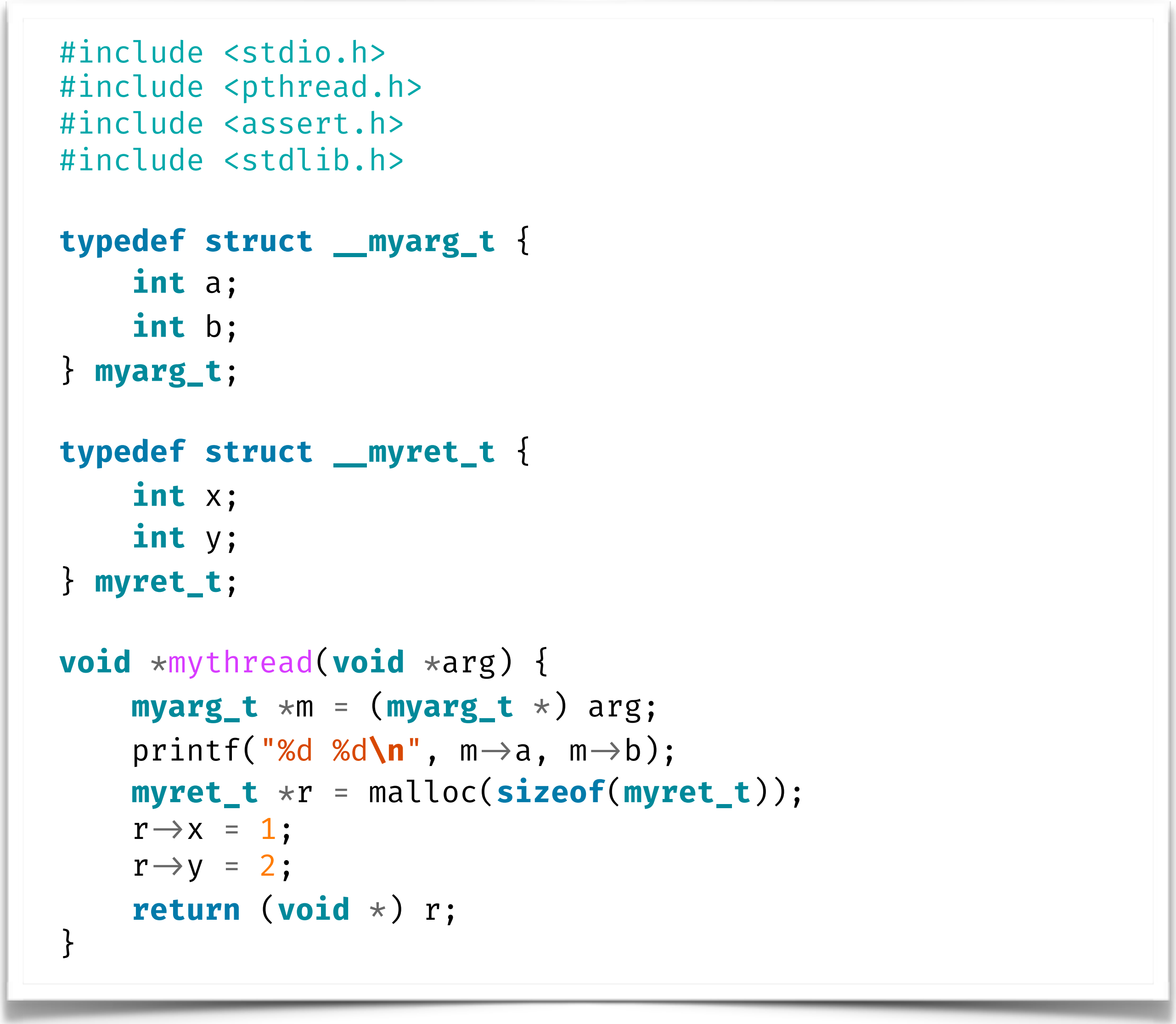
Wait for a thread to complete

- `pthread_join(thread, value_ptr)`
  - `thread`: Specify which thread to wait for
  - `value_ptr`: A pointer to the return value
    - Because pthread_join() routine changes the value, you need to pass in a pointer to that value.



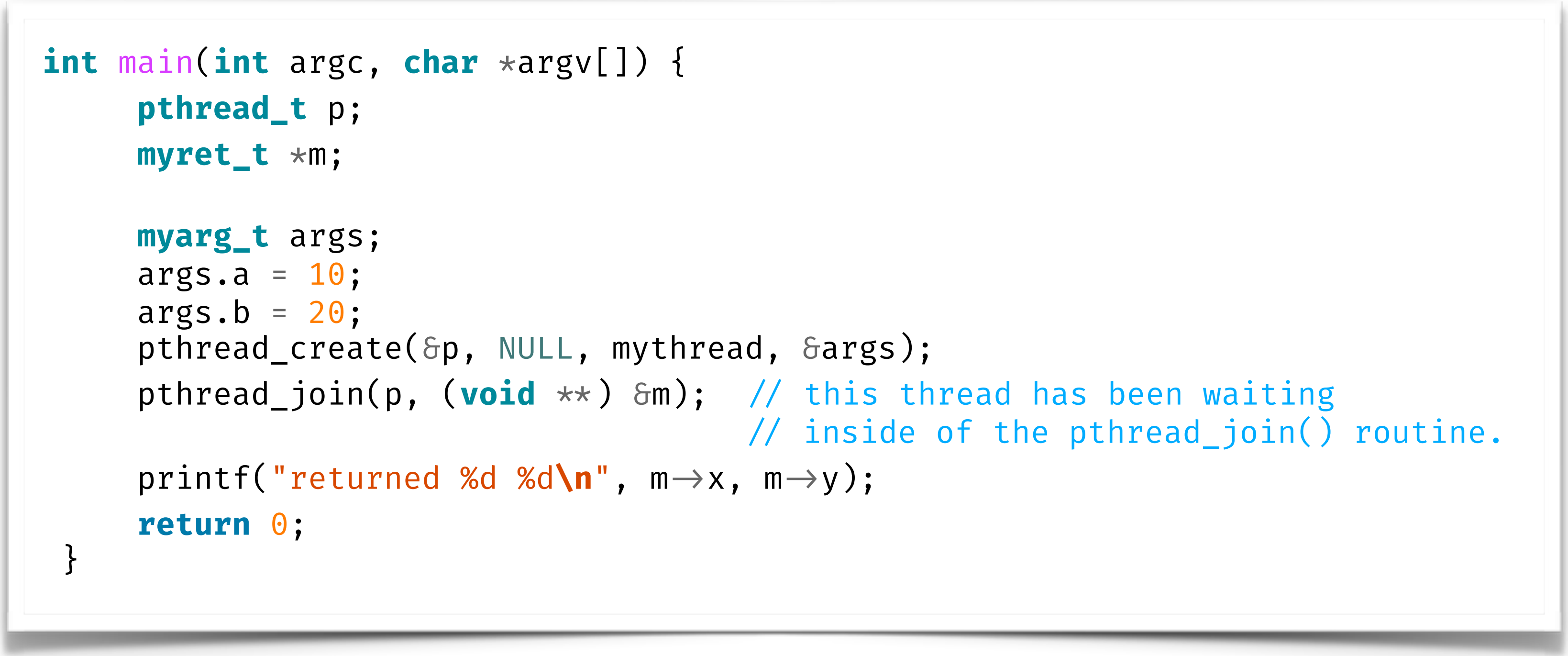
```
int pthread_join(pthread_t thread, void **value_ptr);
```

Example: Waiting for Thread Completion



```
#include <stdio.h>
#include <pthread.h>
#include <assert.h>
#include <stdlib.h>

typedef struct __myarg_t {
    int a;
    int b;
} myarg_t;

typedef struct __myret_t {
    int x;
    int y;
} myret_t;

void *mythread(void *arg) {
    myarg_t *m = (myarg_t *) arg;
    printf("%d %d\n", m->a, m->b);
    myret_t *r = malloc(sizeof(myret_t));
    r->x = 1;
    r->y = 2;
    return (void *) r;
}
```

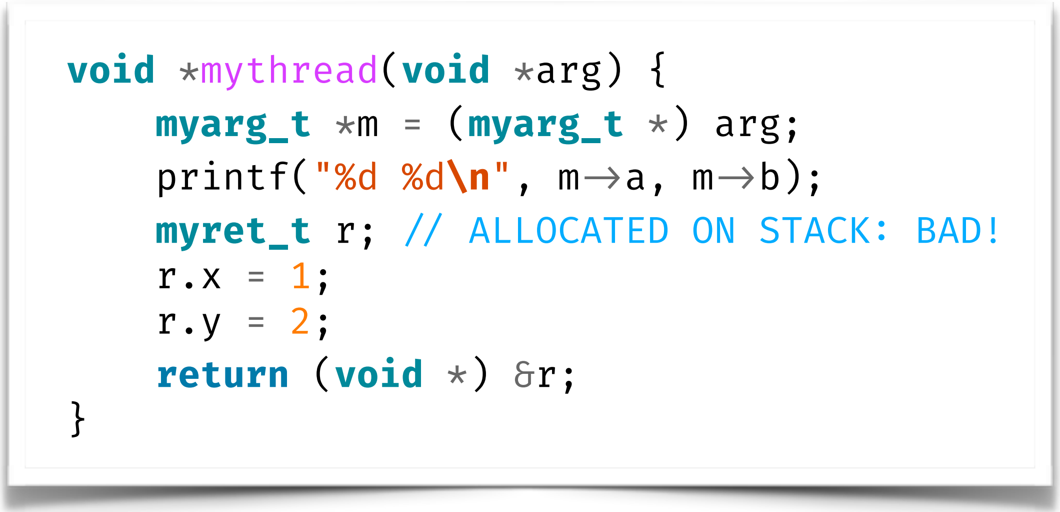
Example: Waiting for Thread Completion



```
int main(int argc, char *argv[]) {
     pthread_t p;
     myret_t *m;

     myarg_t args;
     args.a = 10;
     args.b = 20;
     pthread_create(&p, NULL, mythread, &args);
     pthread_join(p, (void **) &m);  // this thread has been waiting
                                     // inside of the pthread_join() routine.
     printf("returned %d %d\n", m->x, m->y);
     return 0;
 }
```

Example: Dangerous code

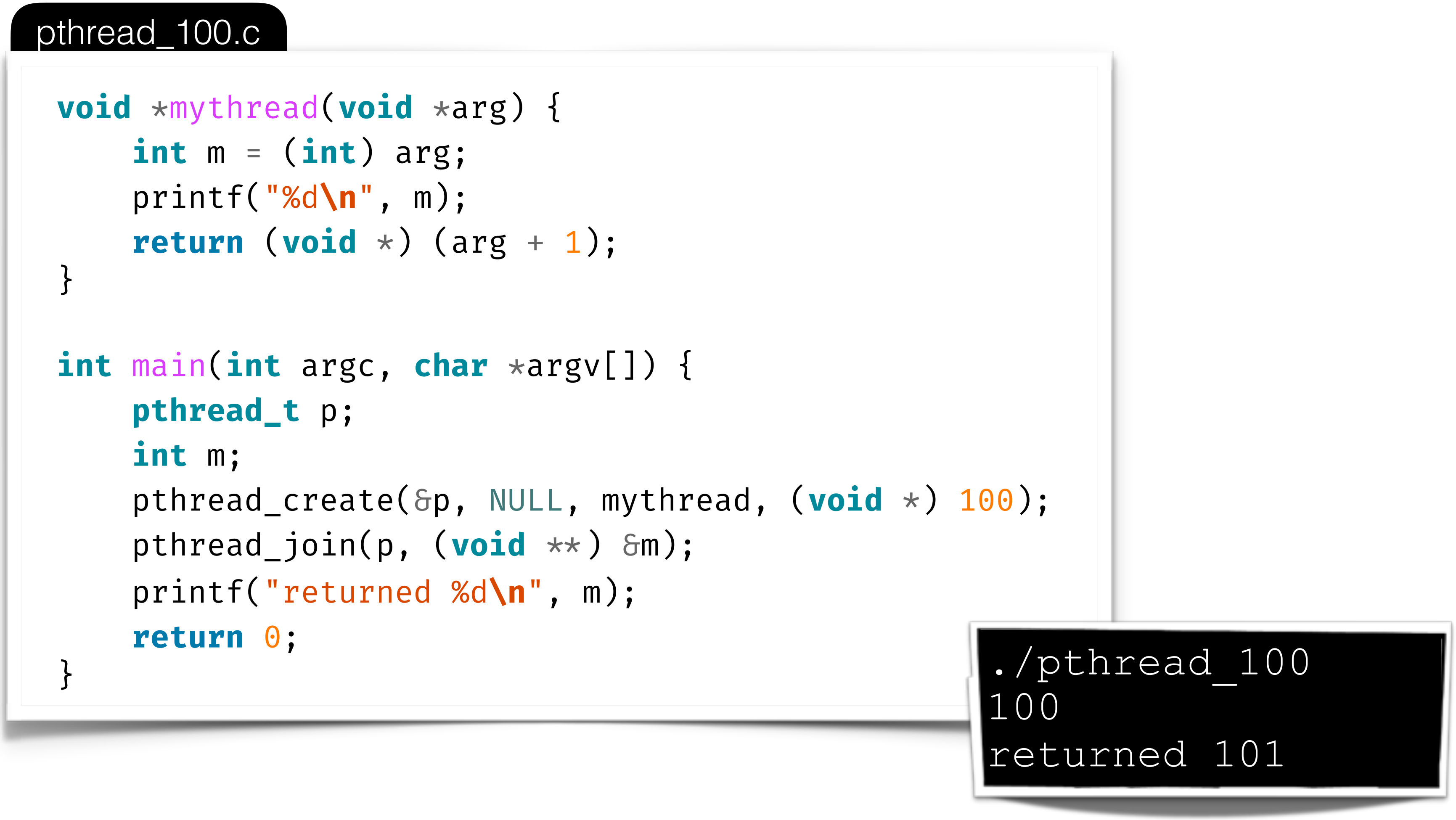
- Be careful with how values are returned from a thread.
  - When the variable **r** returns, it is automatically de-allocated.



```
void *mythread(void *arg) {
    myarg_t *m = (myarg_t *) arg;
    printf("%d %d\n", m->a, m->b);
    myret_t r; // ALLOCATED ON STACK: BAD!
    r.x = 1;
    r.y = 2;
    return (void *) &r;
}
```

Simpler Argument Passing to a Thread

- Just passing in a single value



pthread_100.c

```
void *mythread(void *arg) {
    int m = (int) arg;
    printf("%d\n", m);
    return (void *) (arg + 1);
}

int main(int argc, char *argv[]) {
    pthread_t p;
    int m;
    pthread_create(&p, NULL, mythread, (void *) 100);
    pthread_join(p, (void **) &m);
    printf("returned %d\n", m);
    return 0;
}
```

```
./pthread_100
100
returned 101
```

27. Interlude: Thread API

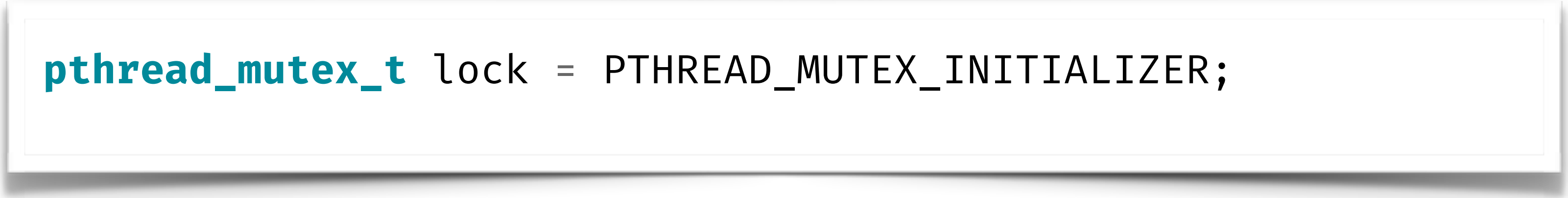
1. Create
2. Wait
3. **Lock**
4. CV

Locks

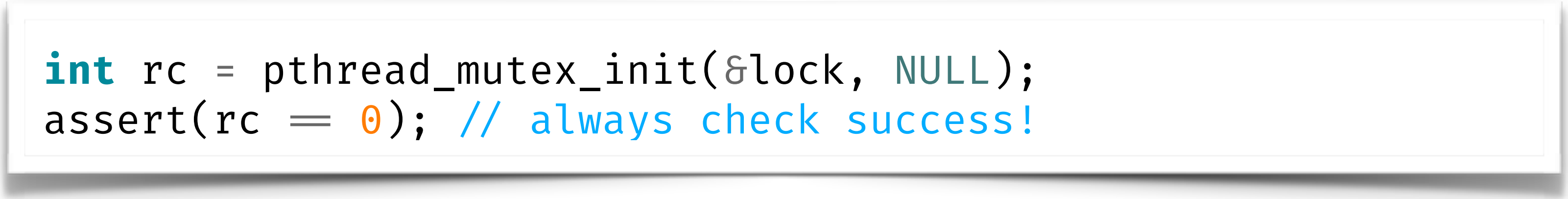
- Provide **mutual exclusion** to a critical section
- Interface:
  - `pthread_mutex_lock(mutex)`
  - `pthread_mutex_unlock(mutex)`
- No other thread holds the lock
  - the thread will acquire the lock and **enter the critical section**.
- If another thread hold the lock
  - the thread will **not return from the call** until it has acquired the lock.

Initialize Locks

- All locks must be properly initialized.
    - One way: using `PTHREAD_MUTEX_INITIALIZER`
    - The dynamic way: using `pthread_mutex_init()`

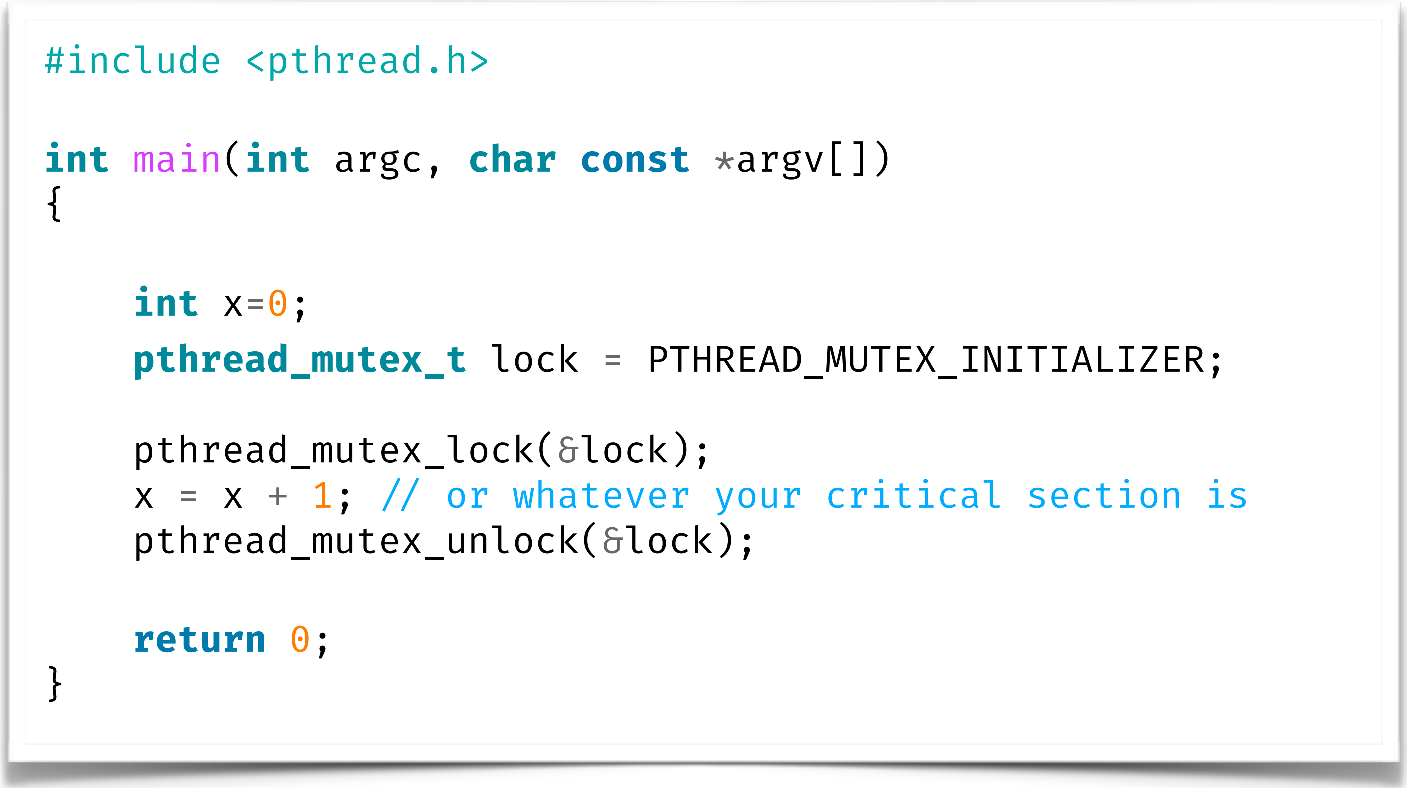


```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
```



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

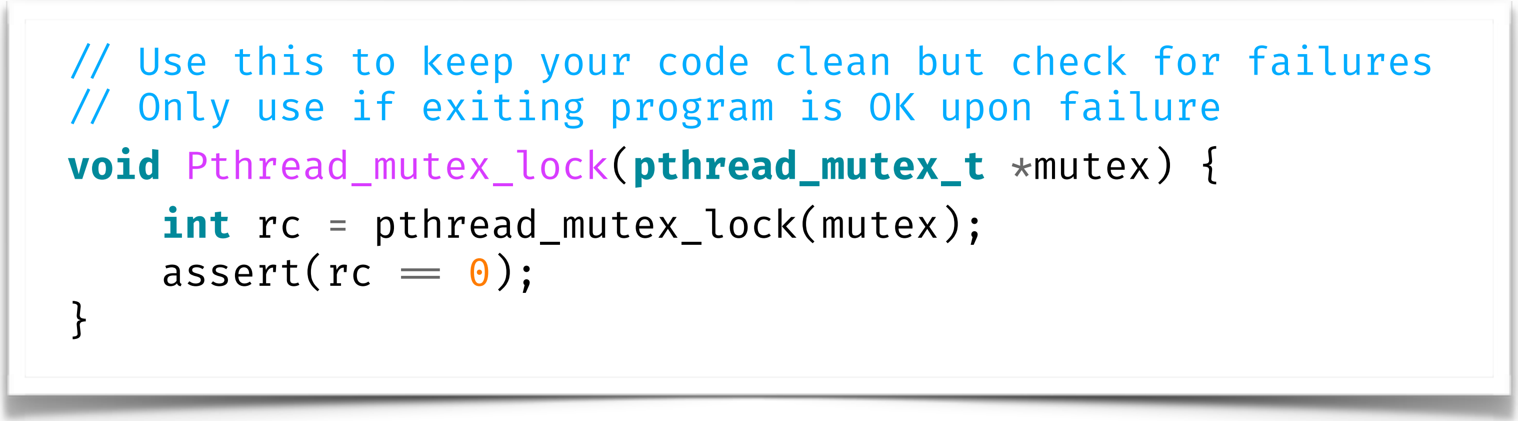
Example: Locks



```
#include <pthread.h>

int main(int argc, char const *argv[])
{

    int x=0;
    pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

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

    return 0;
}
```

Check errors code when using locks

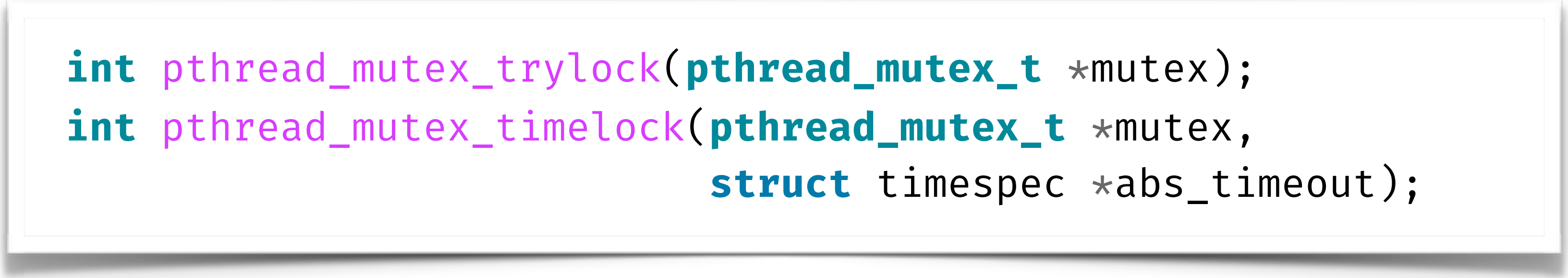
- An example wrapper



```
// Use this to keep your code clean but check for failures
// Only use if exiting program is OK upon failure
void Pthread_mutex_lock(pthread_mutex_t *mutex) {
    int rc = pthread_mutex_lock(mutex);
    assert(rc == 0);
}
```

More calls to lock

- These two calls are also used in **lock acquisition**
  - `trylock()`: return failure if the lock is already held
  - `timelock()`: return after a timeout or after acquiring the lock



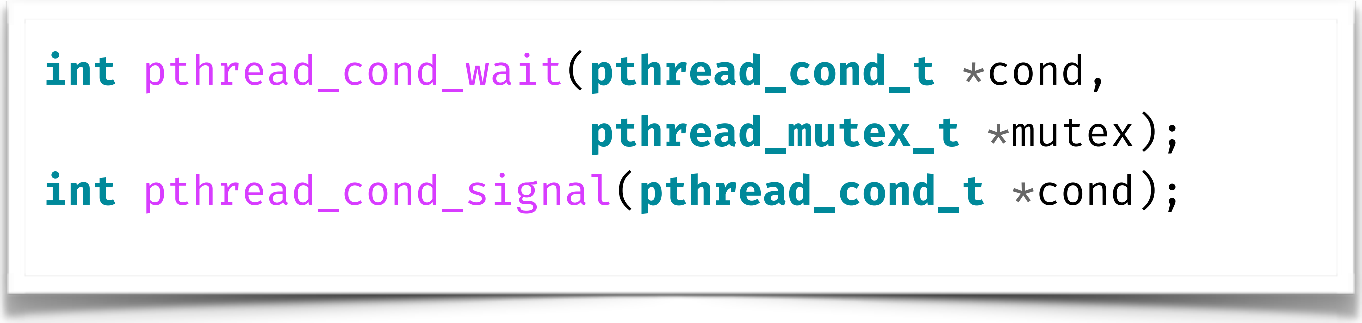
```
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_timelock(pthread_mutex_t *mutex,
                           struct timespec *abs_timeout);
```

27. Interlude: Thread API

1. Create
2. Wait
3. Lock
4. **CV**

Condition variables

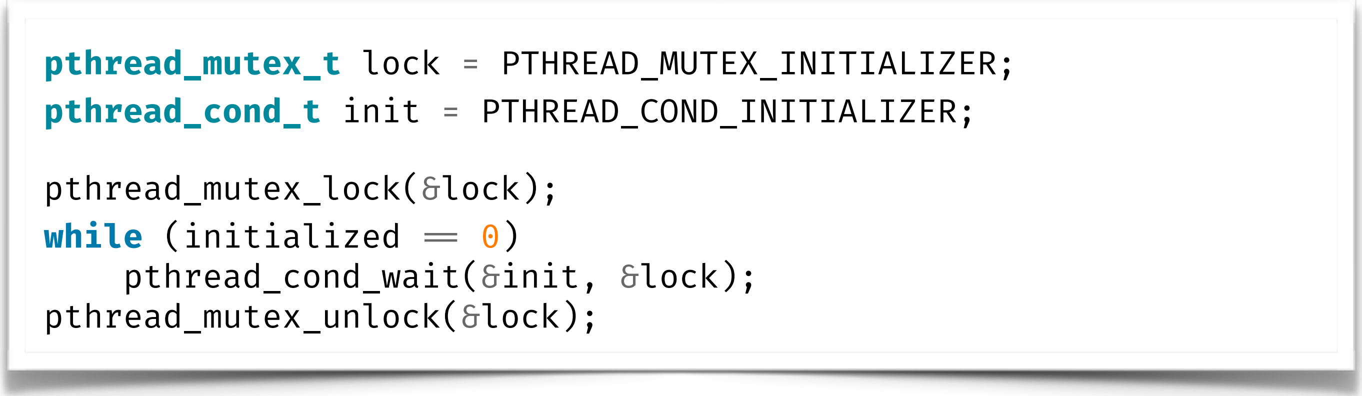
- **Condition variables** are useful when some kind of **signaling** must take place between threads.
- *pthread_cond_wait()*:
  - Put the calling thread to sleep.
  - Wait for some other thread to signal it.
- *pthread_cond_signal()*:
  - Unblock at least one of the threads that are blocked on the condition variable



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

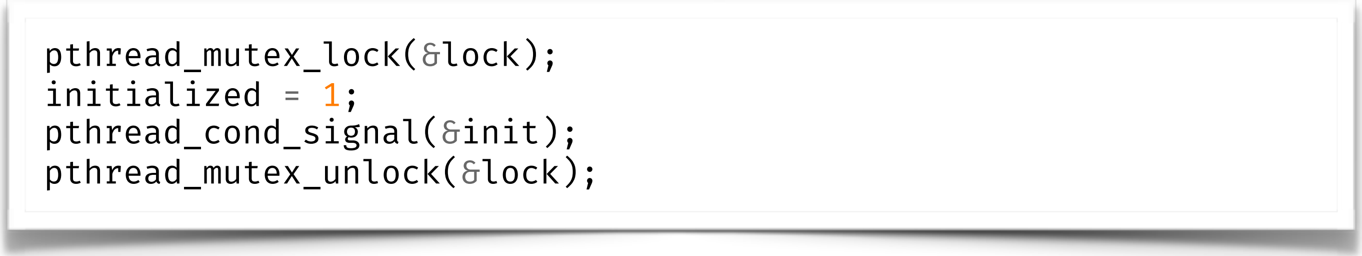
Using wait and signal

- A thread calling wait routine:
  - The wait call **releases the lock** when putting said caller to sleep.
  - Before returning after being woken, the wait call **re-acquire the lock.**



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

pthread_mutex_lock(&lock);
while (initialized == 0)
    pthread_cond_wait(&init, &lock);
pthread_mutex_unlock(&lock);
```

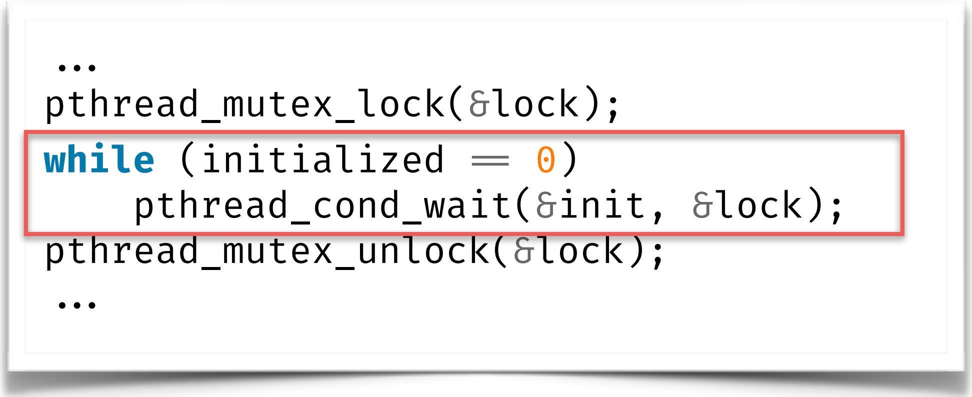
- A thread calling signal routine:



```
pthread_mutex_lock(&lock);
initialized = 1;
pthread_cond_signal(&init);
pthread_mutex_unlock(&lock);
```

Coming back from wait ....

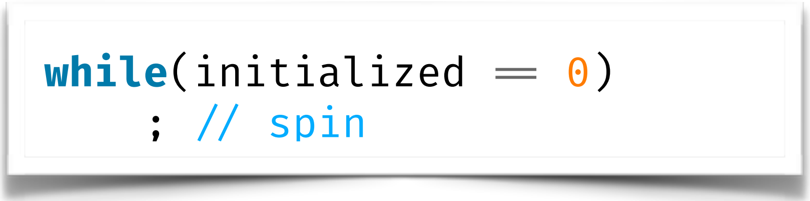
- The waiting thread **re-checks** the condition **in a while loop**, instead of a simple if statement.
    - Without rechecking, the waiting thread will continue thinking that the condition has changed **even though it has not**.



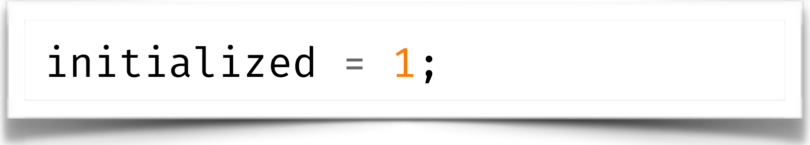
```
...
pthread_mutex_lock(&lock);
while (initialized == 0)
    pthread_cond_wait(&init, &lock);
pthread_mutex_unlock(&lock);
...
```

Don't ever do this

- A thread calling wait routine:
- A thread calling signal routine:
- It performs poorly in many cases.
  - just wastes CPU cycles.
- It is error prone.



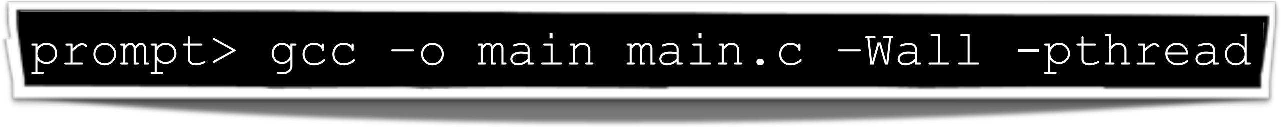
```
while(initialized == 0)
    ; // spin
```



```
initialized = 1;
```

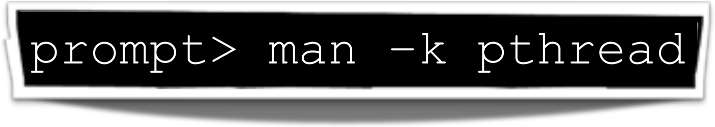
Compiling and Running

- To compile them, you must include the header `pthread.h`
  - Explicitly link with the **pthreads library**, by adding the **–pthread** flag.



```
prompt> gcc -o main main.c -Wall -pthread
```

- For more information,



```
prompt> man -k pthread
```

Thanks

Questions