Threads

Computer Systems Programming, Spring 2024

Instructor: Travis McGaha

TAs:

CV Kunjeti	Lang Qin
Felix Sun	Sean Chuang
Heyi Liu	Serena Chen
Kevin Bernat	Yuna Shao

Administrivia

- HW1 is due this Friday
  - Already out
  - Everything you need has been covered
  - Auto-grader should be out sometime today
- HW2 to be released over the weekend
- Check-in was due before lecture today

Poll Everywhere

pollev.com/tqm

- Any questions? How is your Valentine’s day?

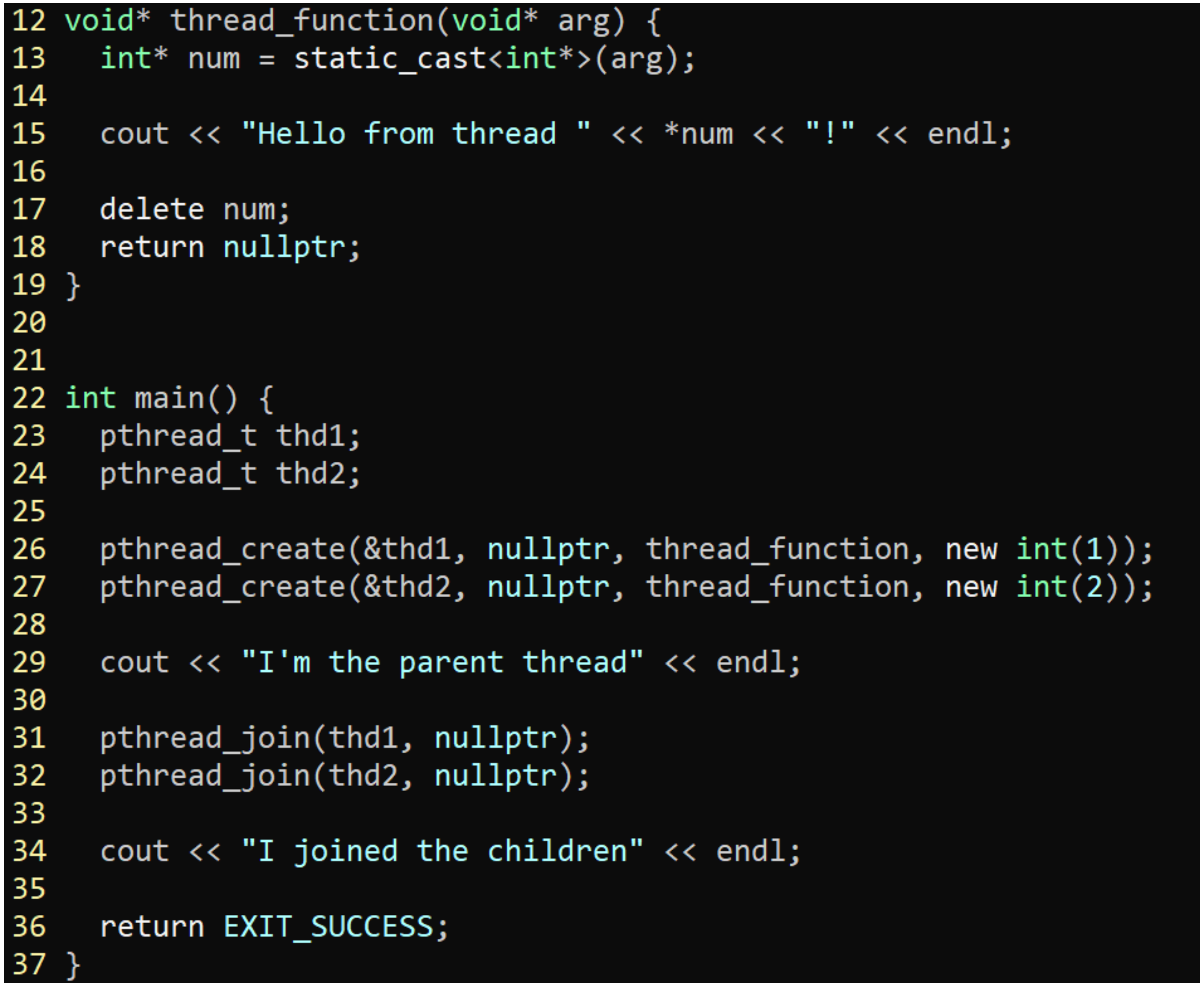
Lecture Outline

- **Data Races Continued**
- Locks & mutexes
- Liveness & deadlocks
- Condition Variables

Poll Everywhere

pollev.com/tqm

- How many possible outputs does this code have?

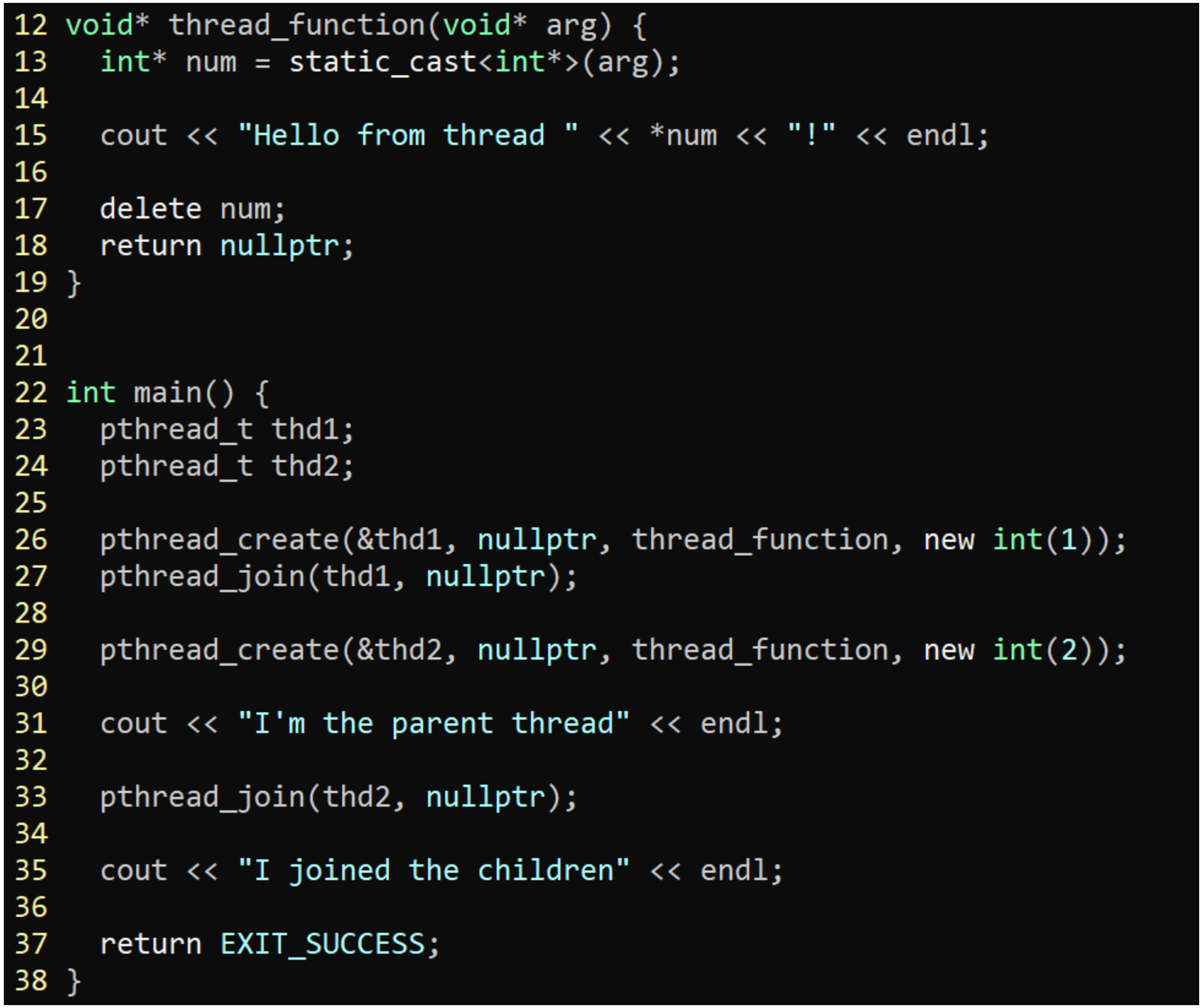


```
void* thread_function(void* arg) {
  int* num = static_cast<int*>(arg);

  cout << "Hello from thread " << *num << "!" << endl;

  delete num;
  return nullptr;
}


int main() {
  pthread_t thd1;
  pthread_t thd2;

  pthread_create(&thd1, nullptr, thread_function, new int(1));
  pthread_create(&thd2, nullptr, thread_function, new int(2));

  cout << "I'm the parent thread" << endl;

  pthread_join(thd1, nullptr);
  pthread_join(thd2, nullptr);

  cout << "I joined the children" << endl;

  return EXIT_SUCCESS;
}
```

Poll Everywhere

pollev.com/tqm

- How many possible outputs does this code have?



```
void* thread_function(void* arg) {
  int* num = static_cast<int*>(arg);

  cout << "Hello from thread " << *num << "!" << endl;

  delete num;
  return nullptr;
}


int main() {
  pthread_t thd1;
  pthread_t thd2;

  pthread_create(&thd1, nullptr, thread_function, new int(1));
  pthread_join(thd1, nullptr);

  pthread_create(&thd2, nullptr, thread_function, new int(2));

  cout << "I'm the parent thread" << endl;

  pthread_join(thd2, nullptr);

  cout << "I joined the children" << endl;

  return EXIT_SUCCESS;
}
```

Data Races

- Two memory accesses form a data race if different threads access the same location, and at least one is a write, and they occur one after another
  - Means that the result of a program can vary depending on chance (which thread ran first?)

Data Race Example

- If your fridge has no milk, then go out and buy some more
  - What could go wrong?

```
if (!milk) {
   buy milk
}
```

- If you live alone:

- If you live with a roommate:

Poll Everywhere

pollev.com/tqm

- Idea: leave a note!
  - Does this fix the problem?

```
if (!note) {
  if (!milk) {
    leave note
    buy milk
    remove note
  }
}
```

A. Yes, problem fixed

B. No, could end up with no milk

C. No, could still buy multiple milk

D. We’re lost…

Poll Everywhere

pollev.com/tqm

- Idea: leave a note!
  - Does this fix the problem?

We can be interrupted between checking note and leaving note ☹

```
if (!note) {
  if (!milk) {
    leave note
    buy milk
    remove note
  }
}
```

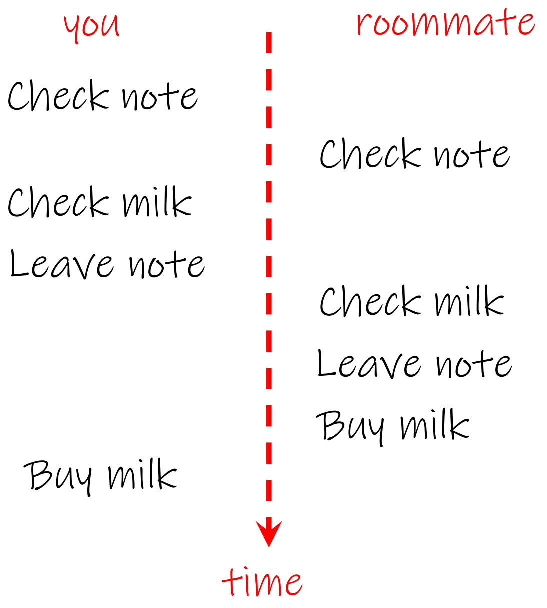
A. Yes, problem fixed

B. No, could end up with no milk

[✓ C.] No, could still buy multiple milk

D. We're lost…

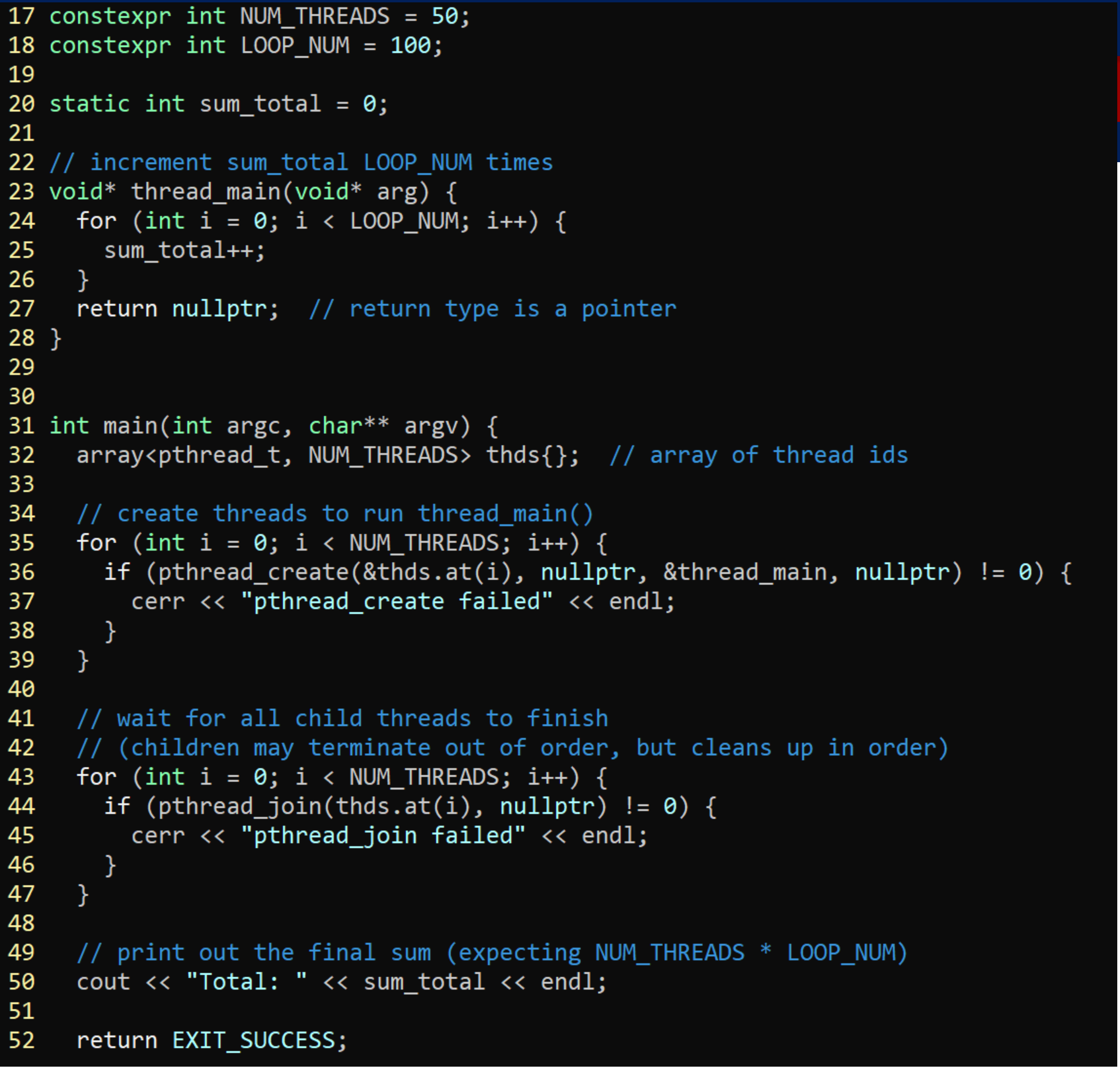
*There are other possible scenarios that result in multiple milks



you	roommate
Check note	
	Check note
Check milk	
Leave note	
	Check milk
	Leave note
	Buy milk
Buy milk	

Threads and Data Races

- Data races might interfere in painful, non-obvious ways, depending on the specifics of the data structure
- Example: two threads try to read from and write to the same shared memory location
  - Could get "correct" answer
  - Could accidentally read old value
  - One thread's work could get "lost"
- Example: two threads try to push an item onto the head of the linked list at the same time
  - Could get "correct" answer
  - Could get different ordering of items
  - Could break the data structure! ☠



```cpp
constexpr int NUM_THREADS = 50;
constexpr int LOOP_NUM = 100;

static int sum_total = 0;

// increment sum_total LOOP_NUM times
void* thread_main(void* arg) {
  for (int i = 0; i < LOOP_NUM; i++) {
    sum_total++;
  }
  return nullptr;  // return type is a pointer
}


int main(int argc, char** argv) {
  array<pthread_t, NUM_THREADS> thds{};  // array of thread ids

  // create threads to run thread_main()
  for (int i = 0; i < NUM_THREADS; i++) {
    if (pthread_create(&thds.at(i), nullptr, &thread_main, nullptr) != 0) {
      cerr << "pthread_create failed" << endl;
    }
  }

  // wait for all child threads to finish
  // (children may terminate out of order, but cleans up in order)
  for (int i = 0; i < NUM_THREADS; i++) {
    if (pthread_join(thds.at(i), nullptr) != 0) {
      cerr << "pthread_join failed" << endl;
    }
  }

  // print out the final sum (expecting NUM_THREADS * LOOP_NUM)
  cout << "Total: " << sum_total << endl;

  return EXIT_SUCCESS;
```

pollev.com/tqm

- What is the expected output of this code? Is there a data-race?

Increment Data Race

- What seems like a single operation `++sum total` is actually multiple operations in one. The increment looks something like this in assembly:

```
LOAD  sum_total into R0
ADD   R0 R0 #1
STORE R0 into sum_total
```

- What happens if we context switch to a different thread while executing these three instructions?
- See `total.cpp`
  - Data race between threads
- **Reminder: Each thread has its own registers to work with. Each thread would have its own R0**

Increment Data Race

- Consider that sum_total starts at 0 and two threads try to execute `++sum total`

`sum_total = 0`

Thread 0 `R0 = 0`

```
LOAD  sum_total into R0
```

Thread 1

Increment Data Race

- Consider that sum_total starts at 0 and two threads try to execute `++sum_total`

sum_total = 0

Thread 0 **R0 = 0**

```
LOAD  sum_total into R0
```

Thread 1 **R0 = 0**

```
LOAD  sum_total into R0
```

Increment Data Race

- Consider that sum_total starts at 0 and two threads try to execute `++sum_total`

sum_total = 0

Thread 0 **R0 = 0**

```
LOAD   sum_total into R0
```

Thread 1 **R0 = 1**

```
LOAD   sum_total into R0
ADD    R0 R0 #1
```

Increment Data Race

- Consider that sum_total starts at 0 and two threads try to execute `++sum_total`

sum_total = 1

Thread 0 **R0 = 0**

```
LOAD  sum_total into R0
```

Thread 1 **R0 = 1**

```
LOAD  sum_total into R0
ADD   R0 R0 #1
STORE R0 into sum_total
```

Increment Data Race

- Consider that sum_total starts at 0 and two threads try to execute `++sum total`

sum_total = 1

Thread 0 **R0 = 1**

```
LOAD   sum_total into R0



ADD    R0 R0 #1

```

Thread 1 **R0 = 1**

```
LOAD   sum_total into R0
ADD    R0 R0 #1
STORE  R0 into sum_total
```

Increment Data Race

- Consider that sum_total starts at 0 and two threads try to execute `++sum_total`

sum_total = 1

Thread 0 **R0 = 1**

```
LOAD  sum_total into R0




ADD   R0 R0 #1
STORE R0 into sum_total
```

Thread 1 **R0 = 1**

```
LOAD  sum_total into R0
ADD   R0 R0 #1
STORE R0 into sum_total
```

- With this example, we could get 1 as an output instead of 2, even though we executed ++sum_total twice

Lecture Outline

- Data Races Continued
- **Locks & mutexes**
- Liveness & deadlocks
- Condition Variables

Synchronization

- Synchronization is the act of preventing two (or more) concurrently running threads from interfering with each other when operating on shared data
  - Need some mechanism to coordinate the threads
    - "Let me go first, then you can go"
  - Many different coordination mechanisms have been invented

- Goals of synchronization:
  - Liveness – ability to execute in a timely manner (informally, "something good eventually happens")
  - Safety – avoid unintended interactions with shared data structures (informally, "nothing bad happens")

These are VERY related

First concern we will be looking at with locks

Atomicity

- Atomicity: An operation or set of operations on some data are *atomic* if the operation(s) are indivisible, that no other operation(s) on that same data can interrupt/interfere.

- Aside on terminology:
  - Often interchangeable with the term "Linearizability"
  - Atomic has a different (but similar-ish) meaning in the context of data bases and ACID.

Lock Synchronization

- Use a “Lock” to grant access to a *critical section* so that only one thread can operate there at a time
  - Executed in an uninterruptible (*i.e.* atomic) manner

- Lock Acquire
  - Wait until the lock is free, then take it

- Lock Release
  - Release the lock
  - If other threads are waiting, wake exactly one up to pass lock to

- Pseudocode:

```
// non-critical code

lock.acquire();
// critical section
lock.release();

// non-critical code
```

loop/idle if locked

pthreads and Locks

- Another term for a lock is a mutex ("mutual exclusion")
  - `pthread.h` defines datatype `pthread_mutex_t`

```
int pthread_mutex_init(pthread_mutex_t* mutex,
                       const pthread_mutexattr_t* attr);
```

  - Initializes a mutex with specified attributes

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

  - Acquire the lock – blocks if already locked Un-blocks when lock is acquired

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

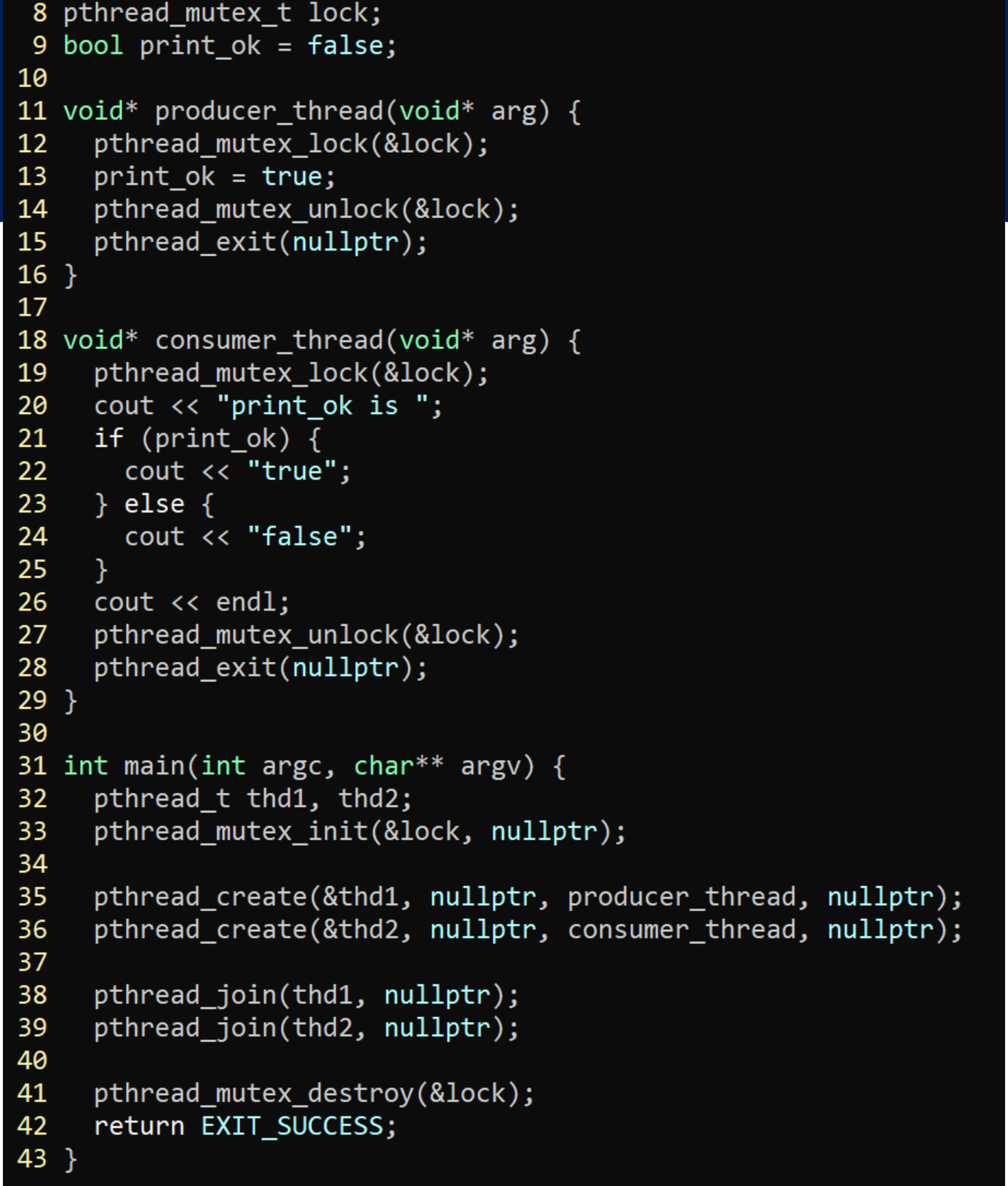
  - Releases the lock

```
int pthread_mutex_destroy(pthread_mutex_t* mutex);
```

  - "Uninitializes" a mutex – clean up when done

pthread Mutex Examples

- See `total.cpp`
  - Data race between threads
- See `total_locking.cpp`
  - Adding a mutex fixes our data race
- How does `total_locking` compare to sequential code and to `total`?
  - Likely *slower* than both– only 1 thread can increment at a time, and must deal with checking the lock and switching between threads
  - One possible fix: each thread increments a local variable and then adds its value (once!) to the shared variable at the end
    - See `total_locking_better.cpp`



```
pthread_mutex_t lock;
bool print_ok = false;

void* producer_thread(void* arg) {
  pthread_mutex_lock(&lock);
  print_ok = true;
  pthread_mutex_unlock(&lock);
  pthread_exit(nullptr);
}

void* consumer_thread(void* arg) {
  pthread_mutex_lock(&lock);
  cout << "print_ok is ";
  if (print_ok) {
    cout << "true";
  } else {
    cout << "false";
  }
  cout << endl;
  pthread_mutex_unlock(&lock);
  pthread_exit(nullptr);
}

int main(int argc, char** argv) {
  pthread_t thd1, thd2;
  pthread_mutex_init(&lock, nullptr);

  pthread_create(&thd1, nullptr, producer_thread, nullptr);
  pthread_create(&thd2, nullptr, consumer_thread, nullptr);

  pthread_join(thd1, nullptr);
  pthread_join(thd2, nullptr);

  pthread_mutex_destroy(&lock);
  return EXIT_SUCCESS;
}
```

pollev.com/tqm

- Does this code have a data race?
  - Can this program enter an “invalid” (unexpected or error) state?

Race Condition vs Data Race

- Data-Race: when there are concurrent accesses to a shared resource, with at least one write, that can cause the shared resource to enter an invalid or “unexpected” state.

- Race-Condition: Where the program has different behaviour depending on the ordering of concurrent threads. This can happen even if all accesses to shared resources are “atomic” or “locked”

- The previous example has no data-race, but it does have a race condition

Lecture Outline

- Data Races Continued
- Locks & mutexes
- **Liveness & deadlocks**
- Condition Variables

Liveness

- Liveness: A set of properties that ensure that threads execute in a timely manner, despite any contention on shared resources.

- When `pthread_mutex_lock();` is called, the calling thread blocks (stops executing) until it can acquire the lock.
  - What happens if the thread can never acquire the lock?

Milk Example – Granularity & Liveness

- What if we use a lock on the refrigerator?
  - Probably overkill – what if roommate wanted to get eggs?
  - Code would still be live, but slower

- For performance reasons, only put what is necessary in the critical section
  - Only lock the milk
  - But lock *all* steps that must run uninterrupted (*i.e.* must run as an atomic unit)

```
fridge.lock()
if (!milk) {
  buy milk
}
fridge.unlock()
```

↓

```
milk_lock.lock()
if (!milk) {
  buy milk
}
milk_lock.unlock()
```

Liveness Failure: Releasing locks

- If locks are not released by a thread, then other threads cannot acquire that lock

- See `release_locks.cpp`
  - Example where locks are not released once critical section is completed.

Liveness Failure: Deadlocks

- Consider the case where there are two threads and two locks
  - Thread 1 acquires lock1
  - Thread 2 acquires lock2
  - Thread 1 attempts to acquire lock2 and blocks
  - Thread 2 attempts to acquire lock1 and blocks

Neither thread can make progress ☹

- See `milk_deadlock.cpp`

- Note: there are many algorithms for detecting/preventing deadlocks

Liveness Failure: Mutex Recursion

- What happens if a thread tries to re-acquire a lock that it has already acquired?

- See `recursive_deadlock.cpp`

- By default, a mutex is not re-entrant.
  - The thread won't recognize it already has the lock, and block until the lock is released

Aside: Recursive Locks

- Mutex's can be configured so that you it can be re-locked if the thread already has locked it. These locks are called *recursive locks* (sometimes called *re-entrant locks*).

- Acquiring a lock that is already held will succeed
- To release a lock, it must be released the same number of times it was acquired

- Has its uses, but generally discouraged.

Lecture Outline

- Data Races Continued
- Locks & mutexes
- Liveness & deadlocks
- **Condition Variables**

Aside: sleep()

- `unistd.h` defines the function:

```
unsigned int sleep(unsigned int seconds);
```

  - Makes the calling thread sleep for the specified number of seconds, resuming execution afterwards

- Useful for manipulating scheduling for testing and demonstration purposes
  - Also for asynchronous/non-blocking I/O, but not covered in this course.

- Necessary for HW2 so that auto-graders work ☹

Thread Communication

- Sometimes threads may need to communicate with each other to know when they can perform operations

- Example: Producer and consumer threads
  - One thread creates tasks/data
  - One thread consumes the produced tasks/data to perform some operation
  - The consumer thread can only produce things once the producer has produced them

Naïve Solution

- Consider the example where a thread must wait to be notified before it can print something out and terminate

- Possible solution: “Spinning”
  - Infinitely loop until the producer thread notifies that the consumer thread can print

- See `spinning.cpp`

- Alternative: Condition variables

Condition Variables

- Variables that allow for a thread to wait until they are notified to resume

- Avoids waiting clock cycles “spinning”

- Done in the context of mutual exclusion
  - a thread must already have a lock, which it will temporarily release while waiting
  - Once notified, the thread will re-acquire a lock and resume execution

pthreads and condition variables

- `pthread.h` defines datatype `pthread_cond_t`

-
```
int pthread_cond_init(pthread_cond_t* cond,
                      const pthread_condattr_t* attr);
```
  - Initializes a condition variable with specified attributes

-
```
int pthread_cond_destroy(pthread_cond_t* cond);
```
  - "Uninitializes" a condition variable – clean up when done

pthreads and condition variables

- `pthread.h` defines datatype `pthread_cond_t`

-
  ```
  int pthread_cond_wait(pthread_cond_t* cond,
                        pthread_mutex_t* mutex);
  ```
  - Atomically releases the mutex and blocks on the condition variable. Once unblocked (by one of the functions below), function will return and calling thread will have the mutex locked

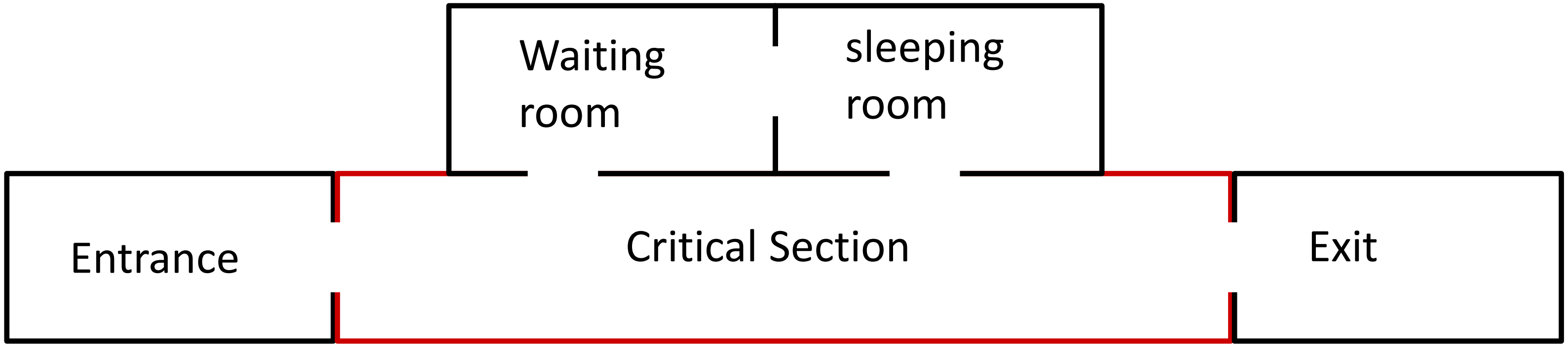
-
  ```
  int pthread_cond_signal(pthread_cond_t* cond);
  ```
  - Unblock at least one of the threads on the specified condition

-
  ```
  int pthread_cond_broadcast(pthread_cond_t* cond);
  ```
  - Unblock all threads blocked on the specified condition

- See `cond.cpp`

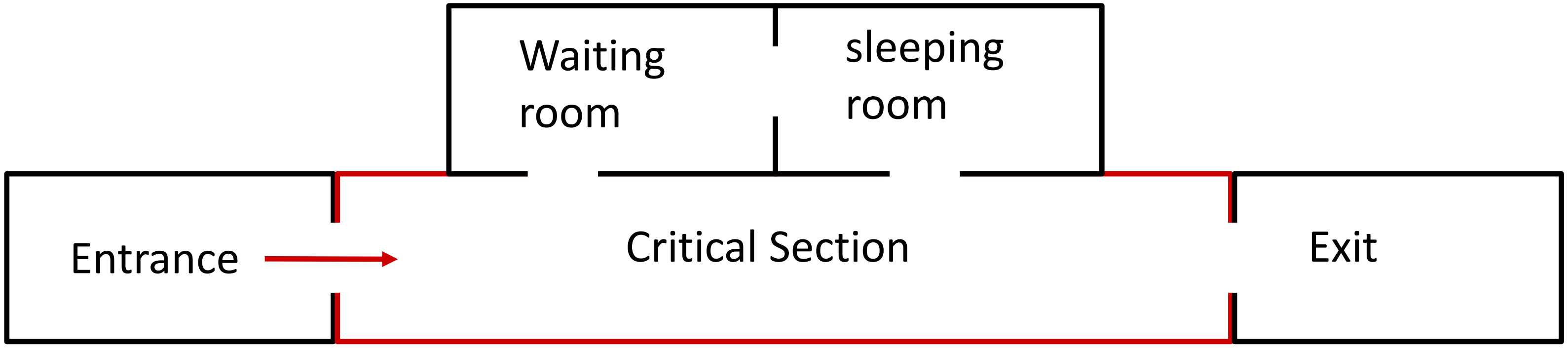
Condition Variable & Mutex Visualization

- This is to visualize how we are using condition variables in this example



Condition Variable & Mutex Visualization

- This is to visualize how we are using condition variables in this example

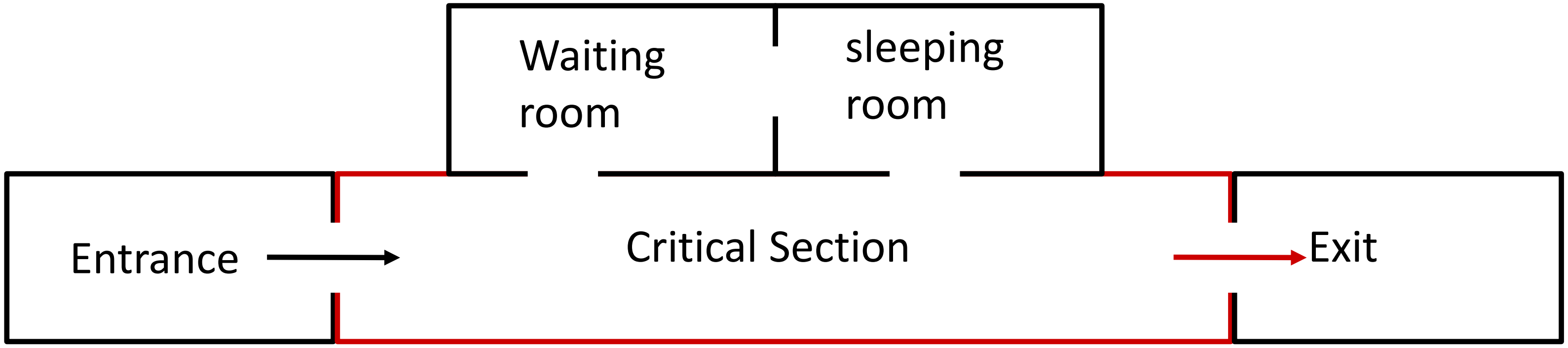


`pthread_mutex_lock`

A thread enters the critical section by acquiring a lock

Condition Variable & Mutex Visualization

- This is to visualize how we are using condition variables in this example



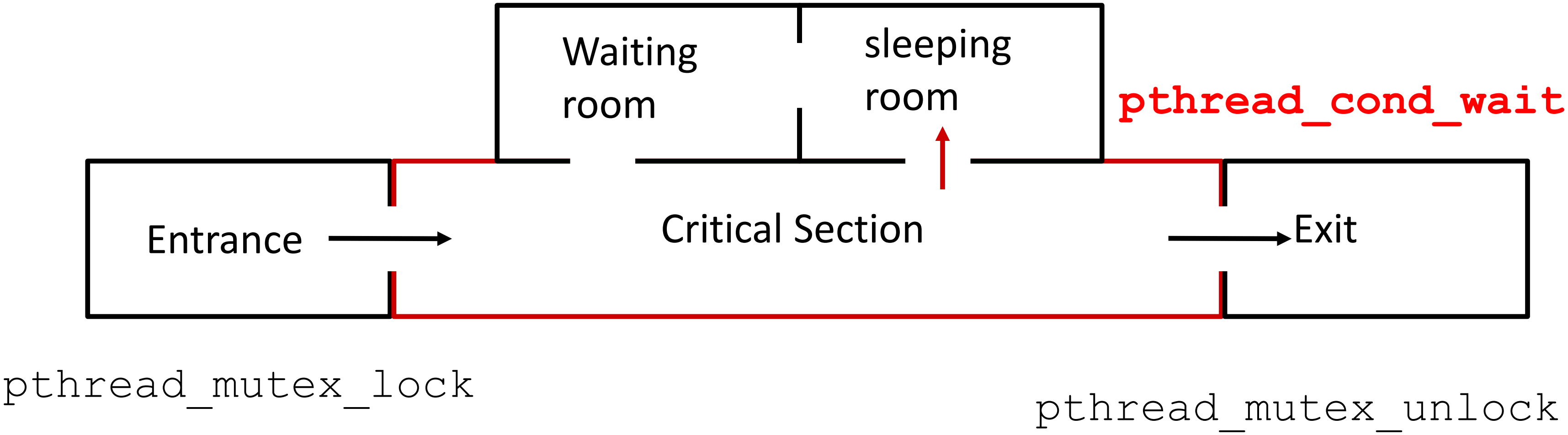
`pthread_mutex_lock`

`pthread_mutex_unlock`

A thread can exit the critical section by acquiring a lock

Condition Variable & Mutex Visualization

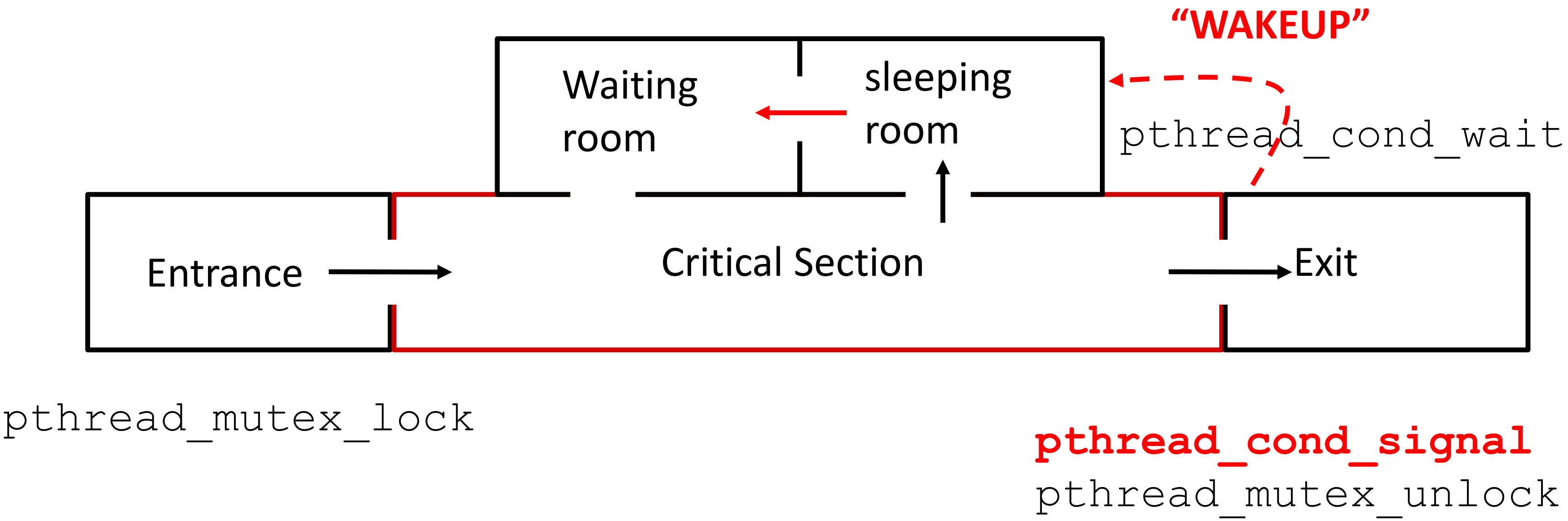
- This is to visualize how we are using condition variables in this example



If a thread can't complete its action, or must wait for some change in state, it can "go to sleep" until someone wakes it up later.
It will release the lock implicitly when it goes to sleep

Condition Variable & Mutex Visualization

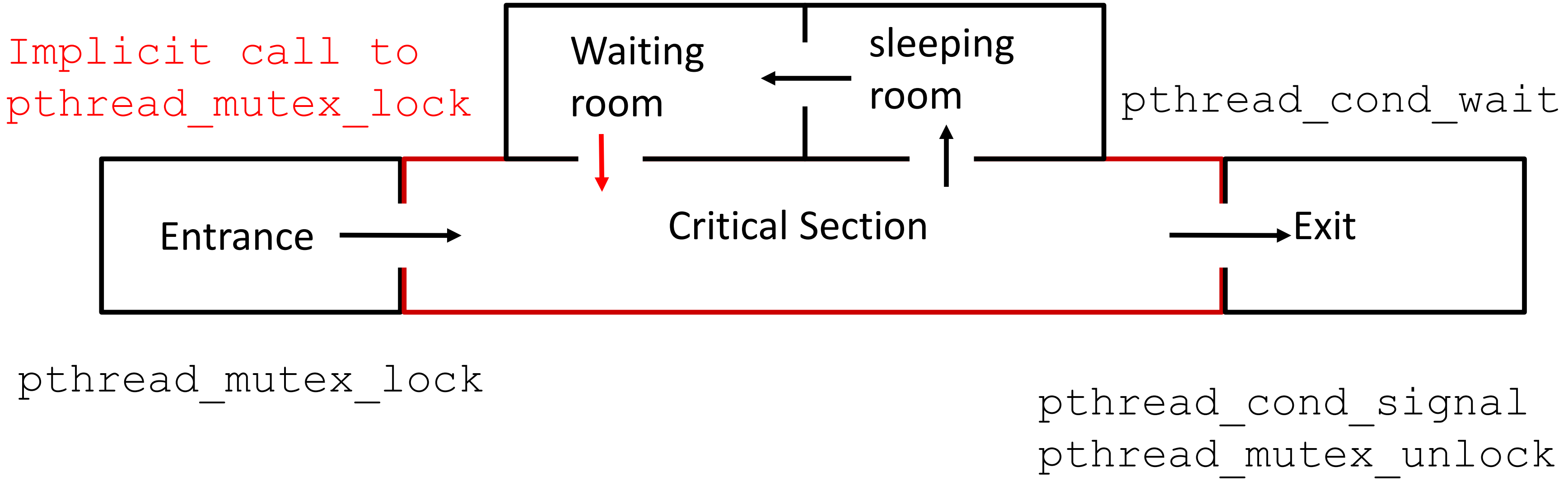
- This is to visualize how we are using condition variables in this example



When a thread modifies state and then leaves the critical section, it can also call pthread_cond_signal to wake up threads sleeping on that condition variable

Condition Variable & Mutex Visualization

- This is to visualize how we are using condition variables in this example



One or more sleeping threads wake up and attempt to acquire the lock.
Like a normal call to pthread_mutex_lock the thread will block until it can acquire the lock

Aside: Things left out

- MANY things left out of this lecture

- Synchronization methods:
  - Semaphores
  - Monitors

- Concurrency properties
  - ACID (databases)
  - CAP theorem

- A lot more concurrency stuff covered in CIS 5050 ☺