Threads & Mutex

Computer Systems Programming, Spring 2025

Instructor: Travis McGaha

Teaching Assistants:

Andrew Lukashchuk
Angie Cao
Aniket Ghorpade
Ashwin Alaparthi
Austin Lin
Hassan Rizwan
Lobi Zhao
Pearl Liu
Perrie Quek

pollev.com/tqm

- Any questions on registration I can help with?

Administrivia

- HW08 – Threads
  - Posted☺
  - Due Friday 4/04 at midnight, leaving open till Sunday night tho
  - AG posted soon

Lecture Outline

- Lock Refresh
- Liveness & Deadlocks
- Race Condition vs Data Race
- Spinning & Condition Variable

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();   <- block if locked
// critical section
lock.release();

// non-critical code
```

Poll Everywhere

pollev.com/tqm

- The code below has three functions that could be executed in separate threads. Note that these are not thread entry points, just functions used by threads:
  - Assume that "lock" has been initialized

- Thread-1 executes line 8 while Thread-2 executes line 21.
  Choose one:
  - Could lead to a race condition.
  - There is no possible race condition.
  - The situation cannot occur.

- Thread-1 executes line 15 while Thread-2 executes line 15.
  Choose one:
  - Could lead to a race condition.
  - There is no possible race condition.
  - The situation cannot occur.

```
1   // global variables
2   pthread_mutex_t lock;
3   int g =    0;
4   int k = 0;
5
6   void fun1() {
7     pthread_mutex_lock(&lock);
8     g += 3;
9     pthread_mutex_unlock(&lock);
10    k++;
11  }
12
13  void fun2(int a, int b) {
14    g += a;
15    a += b;
16    k = a;
17  }
18
19  void fun3() {
20    pthread_mutex_lock(&lock);
21    g = k + 2;
22    pthread_mutex_unlock(&lock);
23  }
```

Poll Everywhere

pollev.com/tqm

- The code below has three functions that could be executed in separate threads. Note that these are not thread entry points, just functions used by threads:
  - Assume that "lock" has been initialized

```
1   // global variables
2   pthread_mutex_t lock;
3   int g =    0;
4   int k = 0;
5
6   void fun1() {
7     pthread_mutex_lock(&lock);
8     g += 3;
9     pthread_mutex_unlock(&lock);
10    k++;
11  }
12
13  void fun2(int a, int b) {
14    g += a;
15    a += b;
16    k = a;
17  }
18
19  void fun3() {
20    pthread_mutex_lock(&lock);
21    g = k + 2;
22    pthread_mutex_unlock(&lock);
23  }
```

- Thread-1 executes line 8 while Thread-2 executes line 14
  Choose one:
  - Could lead to a race condition.
  - There is no possible race condition.
  - The situation cannot occur.

- Thread-1 executes line 14 while Thread-2 executes line 16.
  Choose one:
  - Could lead to a race condition.
  - There is no possible race condition.
  - The situation cannot occur.

Lecture Outline

- Lock Refresh
- Liveness & Deadlocks
- Race Condition vs Data Race
- Spinning & Condition Variable

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?

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`

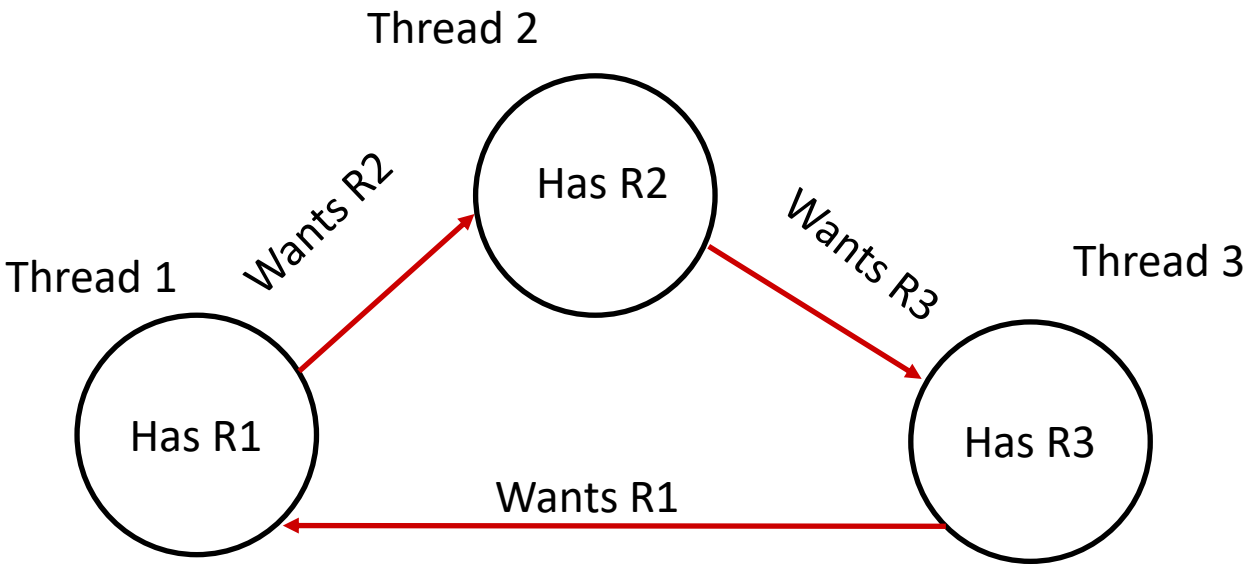
- Note: there are many algorithms for detecting/preventing deadlocks

Deadlock Definition

- A computer has multiple threads, finite resources, and the threads want to acquire those resources
  - Some of these resources require exclusive access
- A threads typically accumulate resources over time
  - If it fails to acquire a resource, it will (by default) wait until it is available before doing anything
- Deadlock: Cyclical dependency on resource acquisition so that none of them can proceed
  - Even if all unblocked threads release, deadlock will continue

Circular Wait Example

- A cycle can exist of more than just two threads:



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.

Liveness: Milk Example – What is the Critical Section?

- What if we use a lock on the refrigerator?
  - Probably overkill – what if roommate wanted to get eggs?
- 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()
```

Poll Everywhere

pollev.com/tqm

- There are at least 4 bad practices/mistakes done with locks in the following code. Find them.
  - Assume `g_lock` and `k_lock` have been initialized and will be cleaned up.
  - Assume that these functions will be called by multi-threaded code.

```
pthread_mutex_t g_lock, k_lock;
int g = 0, k = 0;

void fun1() {
  pthread_mutex_lock(&g_lock);
  g += 3;
  pthread_mutex_unlock(&g_lock);
  k++;
}

void fun2(int a, int b) {
  pthread_mutex_lock(&g_lock);
  g += a;
  pthread_mutex_unlock(&g_lock);
  pthread_mutex_lock(&k_lock);
  a += b;
  pthread_mutex_unlock(&k_lock);
}

void fun3() {
  int c;
  pthread_mutex_lock(&g_lock);
  cin >> c;  // have the user enter an int
  k += c;
  pthread_mutex_unlock(&g_lock);
}
```

Lecture Outline

- Lock Refresh
- Liveness & Deadlocks
- Race Condition vs Data Race
- Spinning & Condition Variable

```
 8 pthread_mutex_t lock;
 9 bool print_ok = false;
10
11 void* producer_thread(void* arg) {
12   pthread_mutex_lock(&lock);
13   print_ok = true;
14   pthread_mutex_unlock(&lock);
15   pthread_exit(nullptr);
16 }
17
18 void* consumer_thread(void* arg) {
19   pthread_mutex_lock(&lock);
20   cout << "print_ok is ";
21   if (print_ok) {
22     cout << "true";
23   } else {
24     cout << "false";
25   }
26   cout << endl;
27   pthread_mutex_unlock(&lock);
28   pthread_exit(nullptr);
29 }
30
31 int main(int argc, char** argv) {
32   pthread_t thd1, thd2;
33   pthread_mutex_init(&lock, nullptr);
34
35   pthread_create(&thd1, nullptr, producer_thread, nullptr);
36   pthread_create(&thd2, nullptr, consumer_thread, nullptr);
37
38   pthread_join(thd1, nullptr);
39   pthread_join(thd2, nullptr);
40
41   pthread_mutex_destroy(&lock);
42   return EXIT_SUCCESS;
43 }
```

pollev.com/tqm

- Does this code have a data race?
  - Can this program enter an “invalid” (unexpected or error) state from having concurrent memory accesses?
- Follow up: Does this code feel good?

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

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

- Need to make sure this communication has no data race or race condition

Poll Everywhere

pollev.com/tqm

- We want to edit the code from the previous example so that consumer thread doesn't exit until print_ok is true.

- How do we do this?

```
 8 pthread_mutex_t lock;
 9 bool print_ok = false;
10
11 void* producer_thread(void* arg) {
12   pthread_mutex_lock(&lock);
13   print_ok = true;
14   pthread_mutex_unlock(&lock);
15   pthread_exit(nullptr);
16 }
17
18 void* consumer_thread(void* arg) {
19   pthread_mutex_lock(&lock);
20   cout << "print_ok is ";
21   if (print_ok) {
22     cout << "true";
23   } else {
24     cout << "false";
25   }
26   cout << endl;
27   pthread_mutex_unlock(&lock);
28   pthread_exit(nullptr);
29 }
```

Lecture Outline

- Lock Refresh
- Liveness & Deadlocks
- Race Condition vs Data Race
- Spinning & Condition Variable

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.
- May be necessary for HW9 so that auto-graders work ☹

Thread Communication: 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`
  - The thread in the loop uses A LOT of cpu just checking until the value is safe
  - Use **`top`** to see CPU util

- 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
  ```

pthread_cond_t Internal Pseudo-Code

- Here is some pseudo code to help understand condition variables

```
int pthread_cond_wait(pthread_cond_t* cond, pthead_mutex_t* mutex) {
  pthread_mutex_unlock(&lock);
  sleep_on_cond(cond);  // sleeps till cond wakes them up
  pthread_mutex_lock(&lock);
  return 0;
}
```

```
int pthread_cond_signal(pthread_cond_t* cond) {
  wakeup_a_thread(cond);  // wake's up a thread sleeping on the cond
  return 0;
}
```

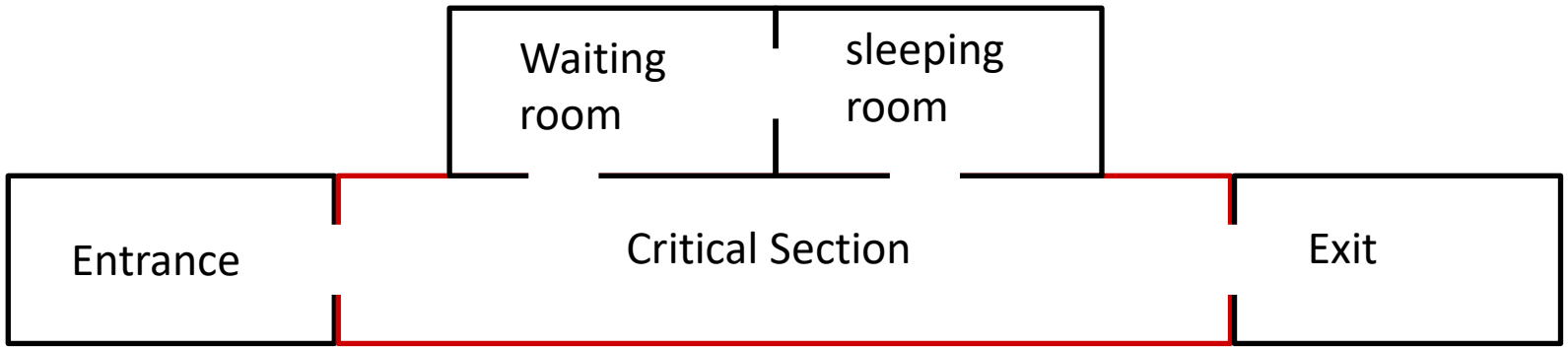
```
int pthread_cond_broadcast(pthread_cond_t* cond) {
  for (thread_sleeping : cond->asleep) { // wake's up all threads
    wakeup(thread_sleeping);
  }
  return 0;
}
```

Demo: `cond.cpp`

- See `cond.cpp`
  - Changes our spinning code to use a condition variable properly
  - No issues with cpu utilization!

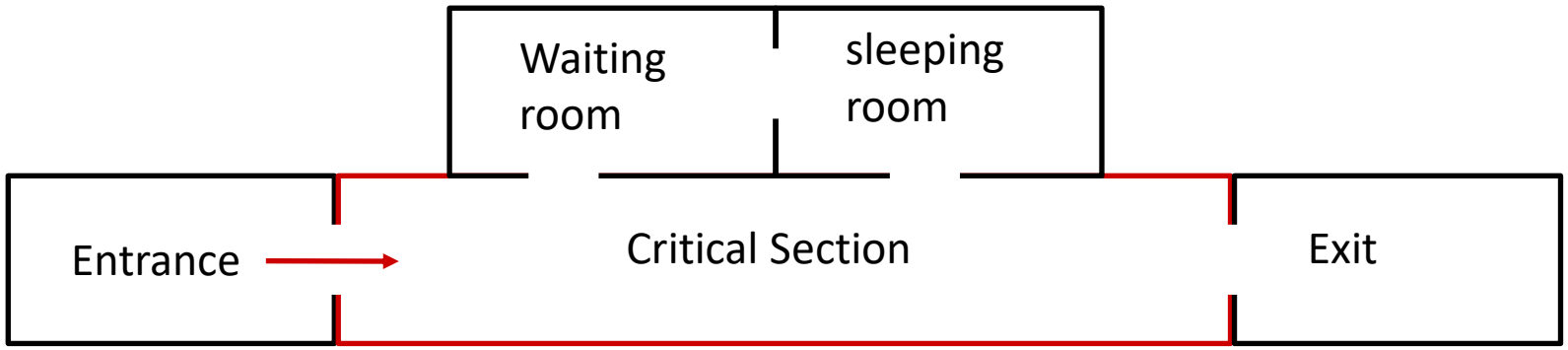
Condition Variable & Mutex Visualization

- 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

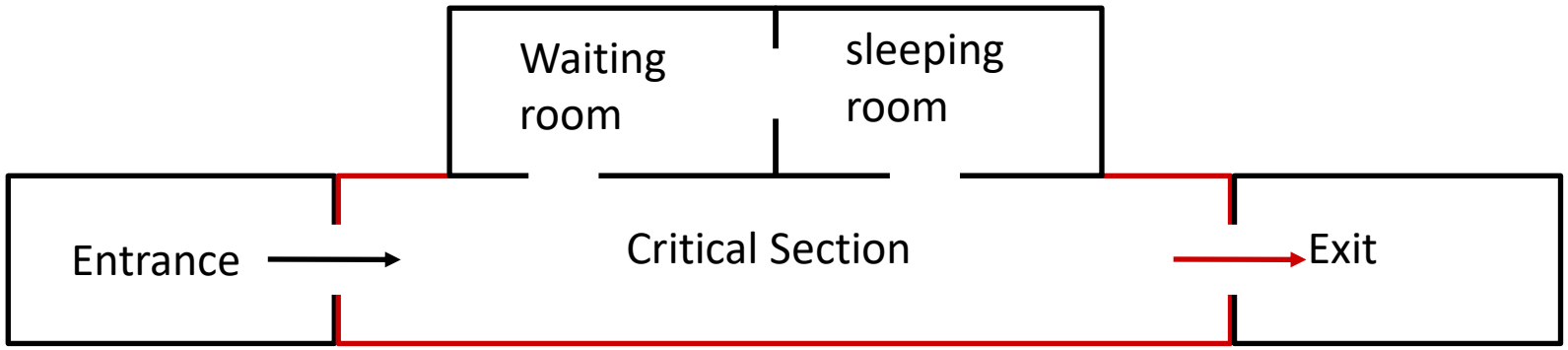


`pthread_mutex_lock`

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



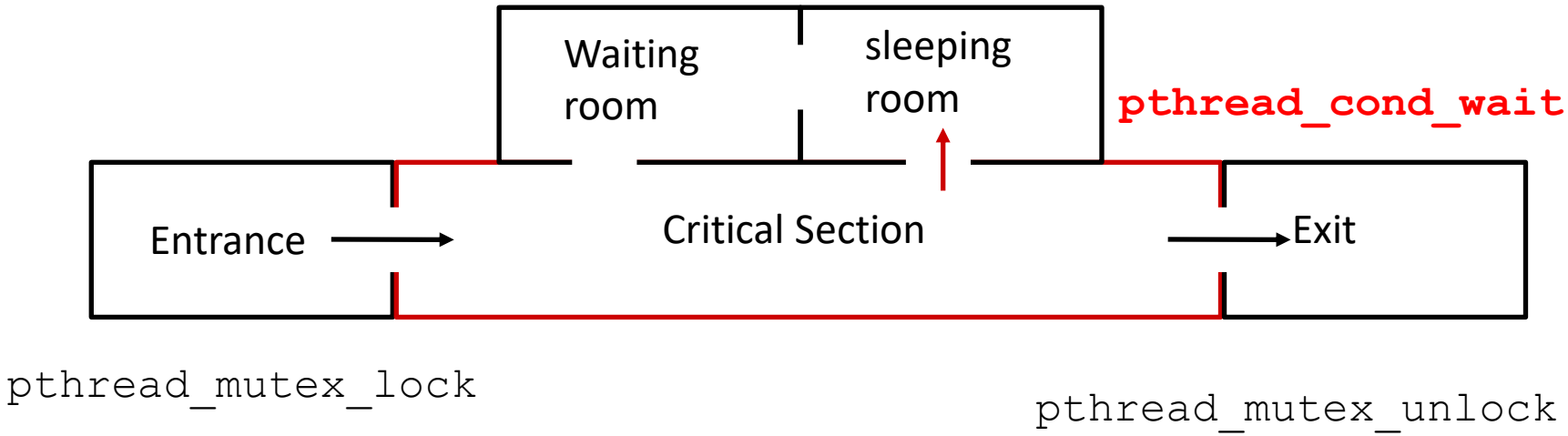
`pthread_mutex_lock`

`pthread_mutex_unlock`

A thread can exit the critical section by acquiring a lock

Condition Variable & Mutex Visualization

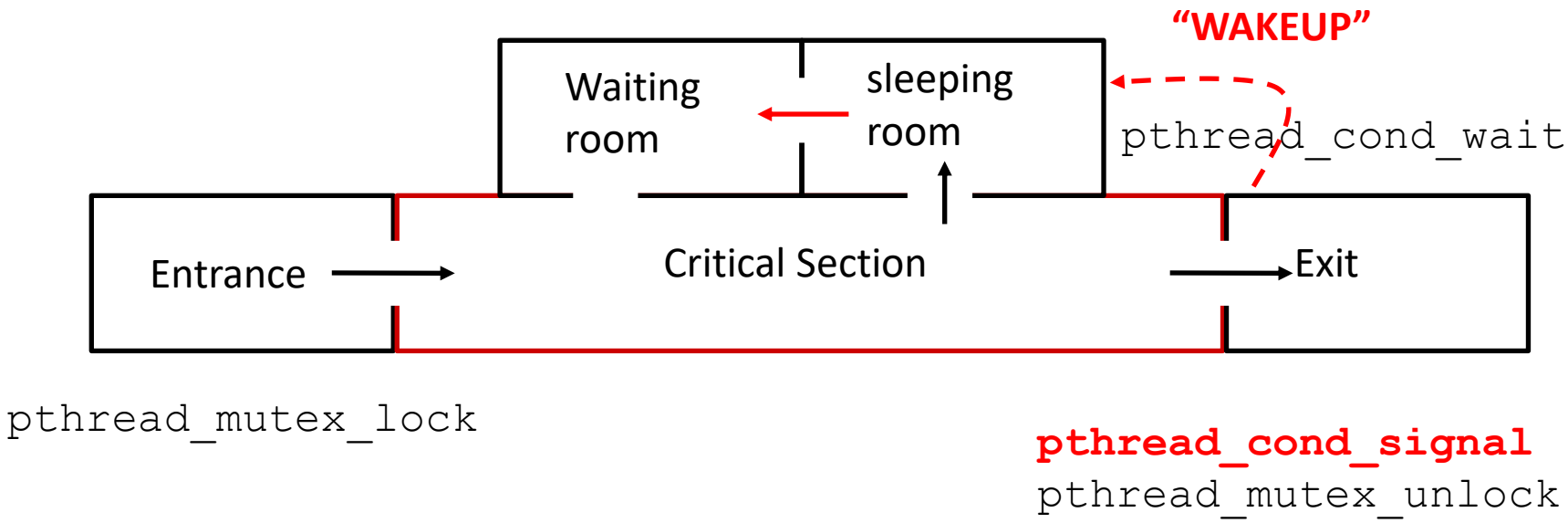
- This is to visualize how we are using condition variables in this example



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

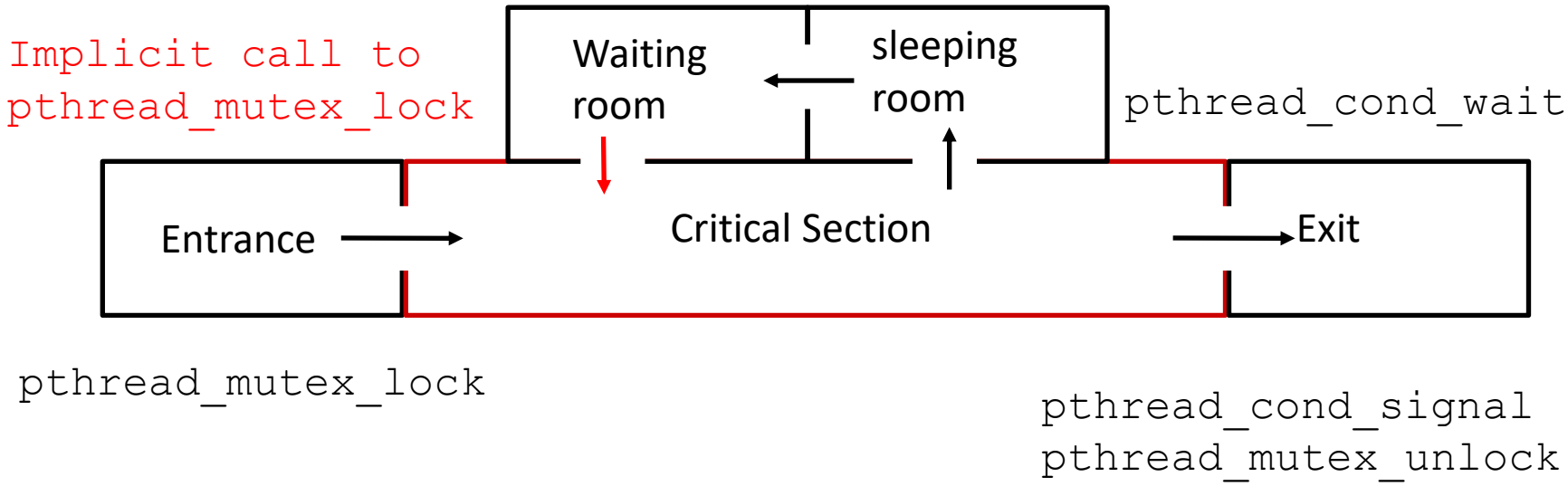
- This is to visualize how we are using condition variables in this example



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

That's all!

- Next time:
  - Condition variables again (if people are confused which they probably will be)!
  - Thread Pools & Efficient Utilization of CPU
  - Basic Parallel Algorithms & Situations