Condition Variables & Concurrency

Computer Operating Systems, Spring 2024

Instructor: Travis McGaha

Head TAs: Nate Hoaglund & Seungmin Han

TAs:

Adam Gorka	Haoyun Qin	Kyrie Dowling	Ryoma Harris
Andy Jiang	Jeff Yang	Oliver Hendrych	Shyam Mehta
Charis Gao	Jerry Wang	Maxi Liu	Tom Holland
Daniel Da	Jinghao Zhang	Rohan Verma	Tina Kokoshvili
Emily Shen	Julius Snipes	Ryan Boyle	Zhiyan Lu

Administrivia

- Lecture on Thursday will be on Zoom (more threads stuff)
- Back to in-person next week!!!!!

- PennOS
  - You have the first milestone, which needs to be done sometime this week
  - Your group (or at least most of your group) needs to meet with your assigned TA and display the expectations laid out in the PennOS Specification
  - Everyone should have already contacted their group, I sent emails to every group that had to be filled by course staff (let us know if you haven't gotten this)

- Checkin to be released, due before lecture next week

Poll Everywhere

pollev.com/tqm

- Any questions, comments or concerns from last lecture?

Lecture Outline

- **Data Race & Mutex Practice**
- Intro to Deadlocks
- Producer & Consumer Problem
- Condition Variables
- Monitors
- Reader/Writer Problem

Poll Everywhere

pollev.com/tqm

What is the range of values that g can have at the end of the program?

```
int g = 0;
void *worker(void *ignore) {
  for (int k = 1; k <= 3; k++) {
    g = g + k;
  }
  printf("g = %d\n", g);
  return NULL;
}

int main() {
  pthread_t t1, t2;
  int ignore;
  ignore = pthread_create(&t1, NULL, &worker, NULL);
  ignore = pthread_create(&t2, NULL, &worker, NULL);
  pthread_join(t1, NULL);
  pthread_join(t2, NULL);
  return EXIT_SUCCESS;
}
```

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

```c
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 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.

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 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.

```
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 }
```

Lecture Outline

- Data Race & Mutex Practice
- **Intro to Deadlocks**
- Producer & Consumer Problem
- Condition Variables
- Monitors
- Reader/Writer Problem

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.c`
  - 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.c`

- 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.c`
- 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 Race & Mutex Practice
- Intro to Deadlocks
- **Producer & Consumer Problem**
- Condition Variables
- Monitors
- Reader/Writer Problem

Producer & Consumer Problem

- Common design pattern in concurrent programming.
  - There are at least two threads, at least one producer and at least one consumer.
  - The producer threads create some data that is then added to a shared data structure
  - Consumers will process and remove data from the shared data structure

- We need to make sure that the threads play nice

Aside: C++ deque

- I am using a c++ **`deque`** for this example so that we don't have to write our own data structure. This is not legal C

- Deque is a double ended queue, you can push to the front or back and pop from the front or back

```
// global deque of integers
// will be initialized to be empty
deque<int> dq {};

int main() {
  dq.push_back(3);        // adds 3
  int val = dq.at(0);     // access index 0
  dq.pop_front()          // delete first element
  printf("%d\n", val);    // should print 3
}
```

Producer Consumer Example

- Does this work?
- Assume that two threads are created, one assigned to each function

```
deque<int> dq {};

void* producer_thread(void* arg) {
  while (true) {
    dq.push_back(long_computation());
  }
}

void* consumer_thread(void* arg) {
  while (true) {
    while (dq.size() == 0) {
      // do nothing
    }
    int val = dq.at(0);
    dq.pop_front();
    do_something(val);
  }
}
```

Poll Everywhere

pollev.com/tqm

- How do we use mutex to fix this? To make sure that the threads access dq safely.
  - You are only allowed to add calls to pthread_mutex_lock and pthread_mutex_unlock
  - Can add other mutexes if needed

```
deque<int> dq {};
pthread_mutex_t dq_lock;

void* producer_thread(void* arg) {
  while (true) {
    dq.push_back(long_computation());
  }
}

void* consumer_thread(void* arg) {
  while (true) {
    while (dq.size() == 0) {
      // do nothing
    }
    int val = dq.at(0);
    dq.pop_front();
    do_something(val);
  }
}
```

Any issue?

- The code is correct, but do we notice anything wrong with this code?

- Maybe a common inefficiency that I have told you about several times before (just in other contexts?)

- Then consumer code "busy waits" when there is nothing for it to consume.
  - It is particularly bad if we have multiple consumers, the locks make the busy waiting of the consumers sequential and use more CPU resources.

Lecture Outline

- Data Race & Mutex Practice
- Intro to Deadlocks
- Producer & Consumer Problem
- **Condition Variables**
- Monitors
- Reader/Writer Problem

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

Condition Variables

- Condition Variables exist so that:
  - Threads can wait for a shared variable to change
  - Threads can notify waiting threads that a change has been made to the shared variable, and that they can stop waiting

- Since condition variables are used to manage access of a shared variable, it is utilized with a mutex (lock).
  - For a thread to wait, it must first have the associated lock. While the thread waits, it gives up the lock
  - For a thread to signal threads sleeping on a condition variable, it must also have the associated lock.
  - When a thread is notified, it will resume executing once it can re-acquire the lock.

pthreads and condition variables

- `pthread.h` defines datatype `pthread_cond_t`

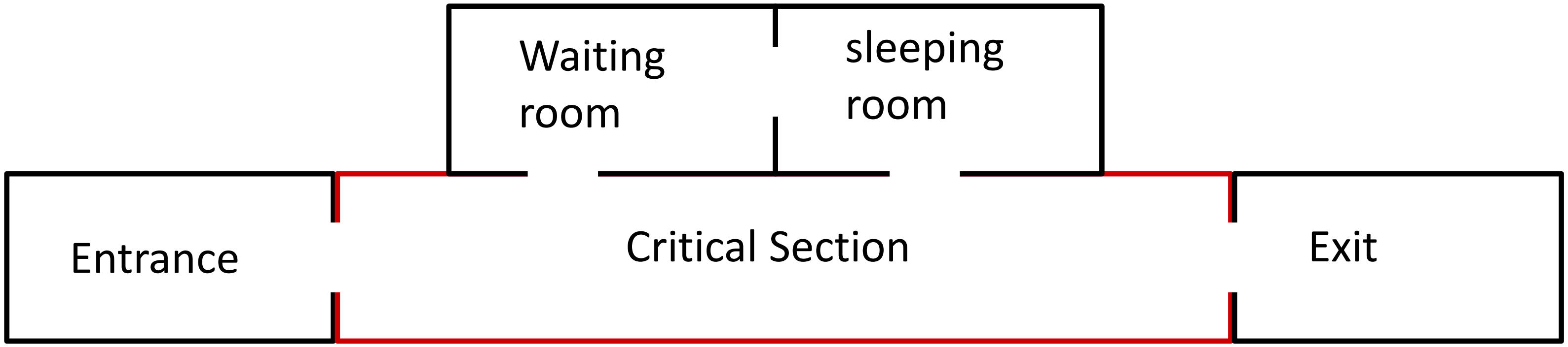
- ```
  int pthread_cond_init(pthread_cond_t* cond,
                        const pthread_condattr_t* attr);
  ```
  - Initializes a condition variable with specified attributes
- ```
  int pthread_cond_destroy(pthread_cond_t* cond);
  ```
  - "Uninitializes" a condition variable – clean up when done

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
  ```

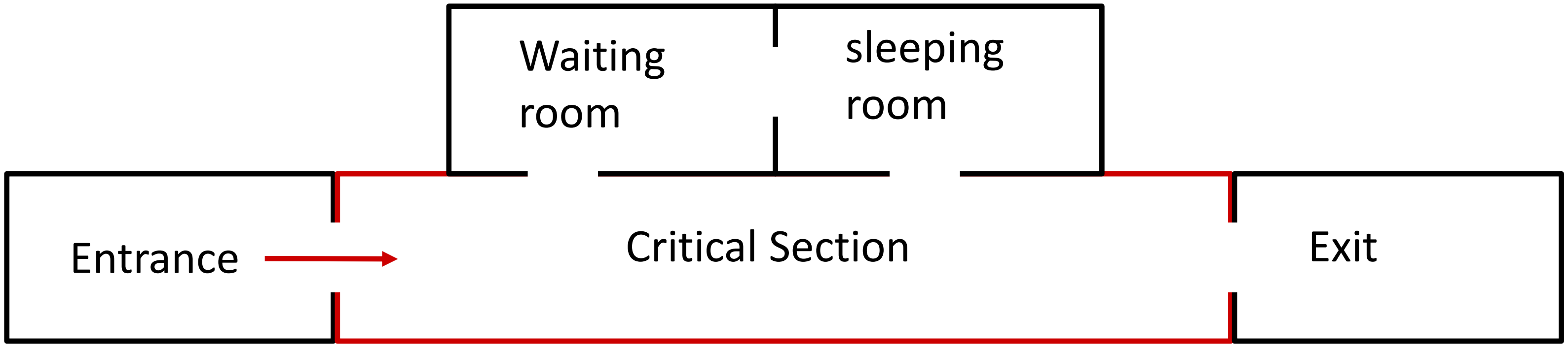
Condition Variable & Mutex Visualization

- 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

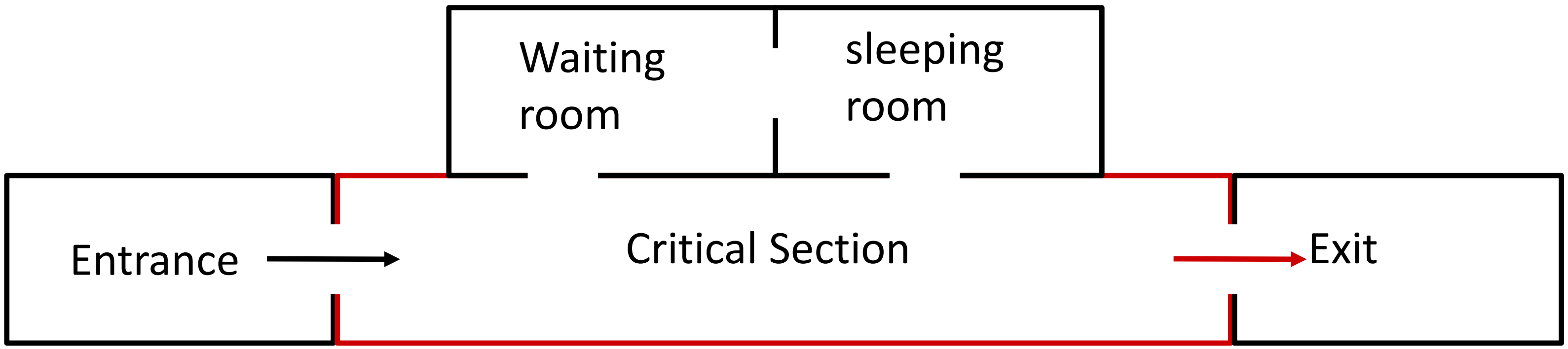


`pthread_mutex_lock`

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



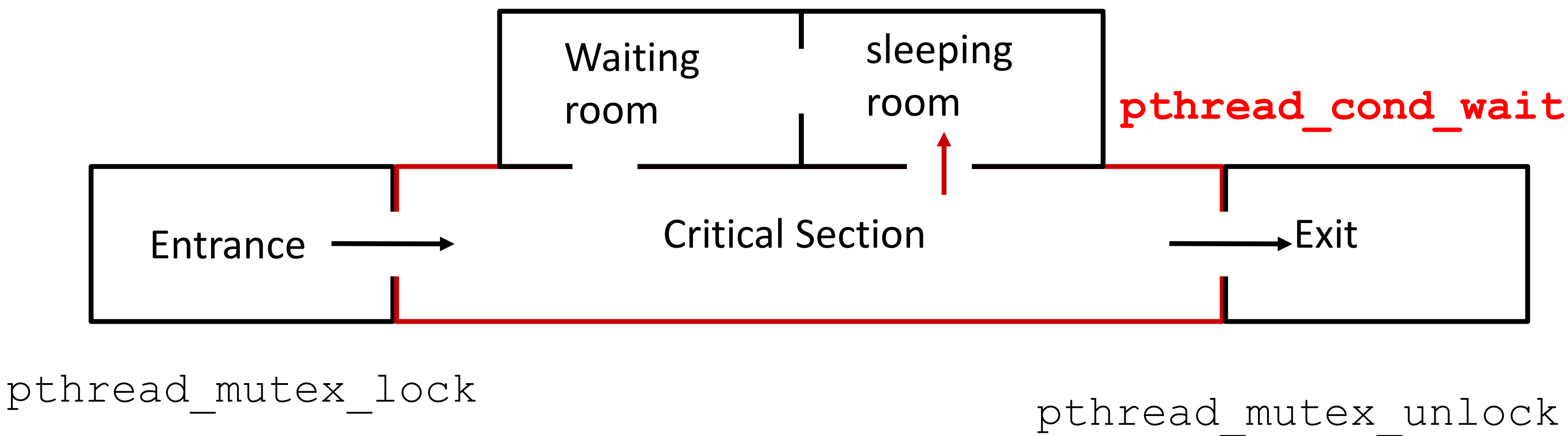
`pthread_mutex_lock`

`pthread_mutex_unlock`

A thread can exit the critical section by acquiring a lock

Condition Variable & Mutex Visualization

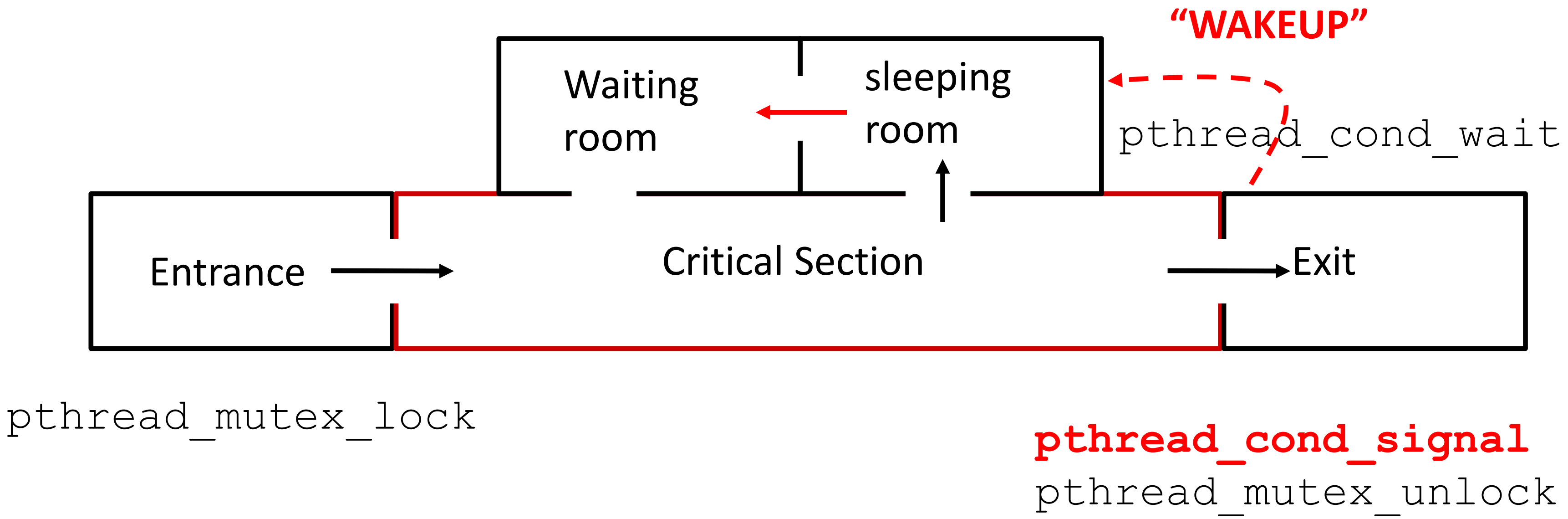
- This is to visualize how we are using condition variables in this example



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.

Condition Variable & Mutex Visualization

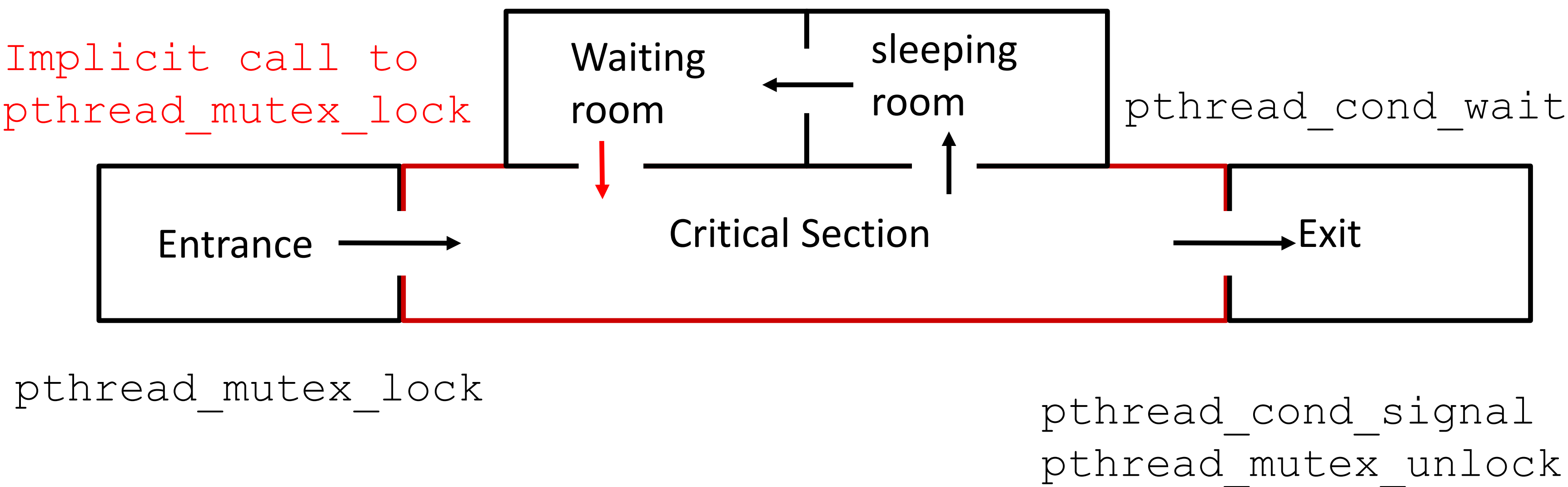
- This is to visualize how we are using condition variables in this example



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

Revisiting Producer Consumer

- Demo: producer_and_consumer.cpp
  - Original producer and consumer code
  - One thread reads a line from stdin and puts it in the deque
  - The other thread gets that string and prints it
  - The consumer thread spins while doing this

- Demo: cond.cpp
  - Consumer and producer uses condition variable
  - Consumer waits if there is no value to process
  - Producer notifies any sleeping threads
  - No more spinning ☺

Poll Everywhere

pollev.com/tqm

- We still need a while loop in the consumer, even with condition variables.
- Why is this needed? Why may our code be incorrect if we don't have one?

```
deque<int> dq;
pthread_mutex_t dq_lock;
pthread_cond_t dq_cond;


void* consumer_thread(void* arg) {
  while (true) {
    pthread_mutex_lock(&dq_lock);
    while (dq.size() == 0) {
      pthread_cond_wait(&dq_cond,
                        &dq_lock);
      // do nothing
    }
    int val = dq.at(0);
    dq.pop_front();
    do_something(val);
    pthread_mutex_unlock(&dq_lock);
  }
}
```

Lecture Outline

- Data Race & Mutex Practice
- Intro to Deadlocks
- Producer & Consumer Problem
- Condition Variables
- **Monitors**
- Reader/Writer Problem

Monitors

- Monitors are a higher-level synchronization concept.
- A Monitor is associated with an object and enforces that only one thread can access data/call the functions of an object at a time.
- A monitor is made up of a mutex and a condition variable.
- Every Object in java is/has a monitor.

Java Monitor Example

```
public class obj {
  private List<String> data;

  public synchronized String get() {
     while (this.data.size() == 0) {
        wait();
        // Ommitted Java exception handling bs
     }
     return this.data.remove(0);
  }

  public synchronized void set(String new_data) {
     this.data.add(new_data);
     notifyAll();
  }

}
```

Monitor vs Condition Variables

- What we implemented with condition variables was essentially a monitor. But condition variables are not restricted to being used in that context.

- Monitors in Java work in a lot of cases and can help abstract away ***some*** of the details with synchronization

- In some cases, a monitor would not make the most sense, but you can still use condition variables to solve the issue.

- **Monitors are a concept, condition variables is an implementation detail**

Lecture Outline

- Data Race & Mutex Practice
- Intro to Deadlocks
- Producer & Consumer Problem
- Condition Variables
- Monitors
- **Reader/Writer Problem**

Readers / Writers Problem

- What if we have some shared data/object and threads can either read or write to the shared data

- How many readers can we have at a time?
  - Any number of readers, as long as no one is writing, we can have an unlimited number of readers.

- How many writers can we have at a time?
  - If a thread is writing to the shared data, then only that thread can have access to the shared data

- How do we support multiple readers but single writer?

Reader/Writers

- We need some metadata, more than just a lock and a cond. Consider the following solutions.

```
// These would normally be put
// into a rdwr_lock structure
int num_readers = 0;    // number of active readers
int writers_waiting = 0;  // number of writers waiting
bool writer_active = false;  // is there a writer active?

// lock to make sure only one thread can access &
// modify the metadata at a time
pthread_mutex_t lock;

// allows a reader/writer to wait until
// it is ok to read/write
pthread_mutex_t cond;
```

Reader/Writers Demo

- Demo: `rw_lock.c`
  - Lots of code for how we grant access to readers & writers

- Any thoughts on how we could make this better?
  - Any issues you notice? It is correct, but are there issues with starvation, wakeups, liveness, etc?
  - Hint: there are issues

pthread_rwlock

- Pthread provides a read/write lock implementation that handles this problem for us and hides many of the dirty implementation details

- Very similar to pthread_mutex, but two types of locking
  - **`pthread_rwlock_rdlock(…);`** // lock as a reader
  - **`pthread_rwlock_wrlock(…);`** // lock as a writer