More Concurrency Problems

Computer Operating Systems, Fall 2023

Instructor: Travis McGaha

Head TAs: Nate Hoaglund & Seungmin Han

TAs:

Andy Jiang	Haoyun Qin	Kevin Bernat	Ryoma Harris
Audrey Yang	Jason hom	Leon Hertzberg	Shyam Mehta
August Fu	Jeff Yang	Maxi Liu	Tina Kokoshvili
Daniel Da	Jerry Wang	Ria Sharma	Zhiyan Lu
Ernest Ng	Jinghao Zhang	Rohan Verma	

Administrivia

- **Full PennOS is due ~~Mon~~ Nov ~~27~~ 29**
  - You will schedule a time to meet with your TA to demonstrate your working code
  - Some info will be posted on the demo & testing functionality soon
- Check-in due before Lecture next week
- Recitation after class is open OH
  No Thursday Lecture
  Calendar is updated as of now for OH this week

pollev.com/tqm

- Any questions, comments or concerns from last lecture?

Lecture Outline

- **Condition Variables**
- Monitors
- Reader/Writer Problem
- Dining Philosophers
- Deadlock Prevention
- Deadlock Handling (start)

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

- Due to 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`

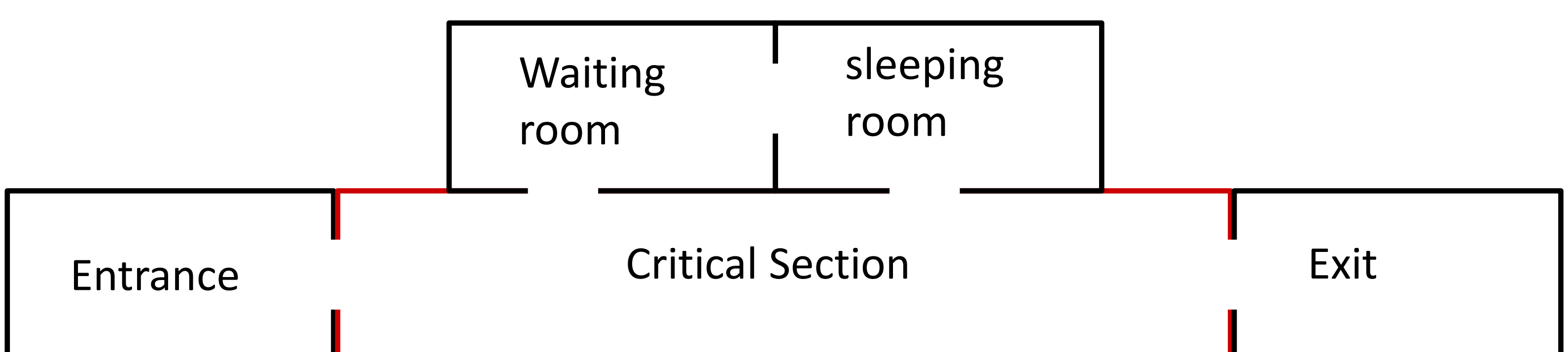
- ```
  int pthread_cond_wait(pthread_cond_t* cond,
                        pthread_mutex_t* mutex);
  ```
  - Atomically releases the mutex and blocks on the condition variable. Once unblocked (by one of the functions below), function will return and calling thread will have the mutex locked

- ```
  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
  ```

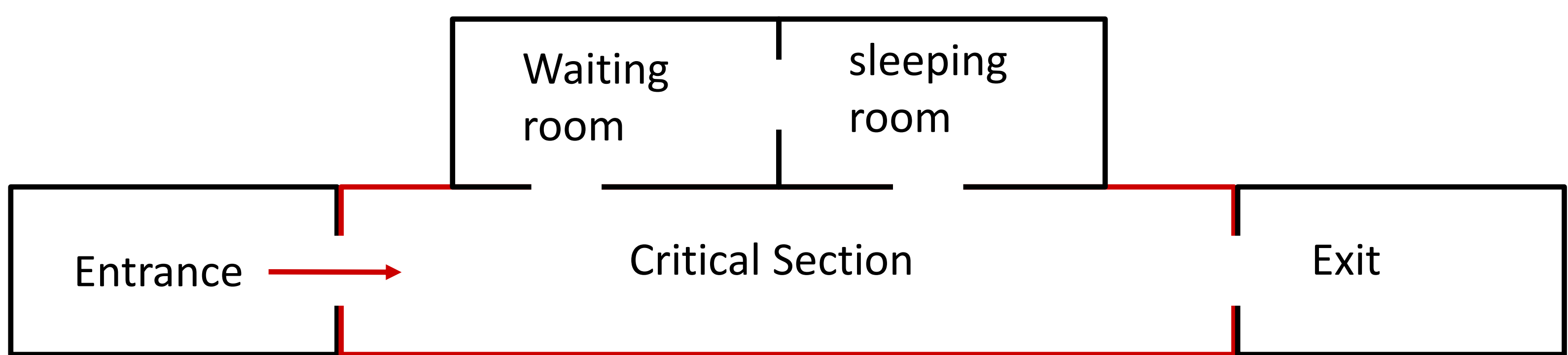
Condition Variable & Mutex Visualization

- 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

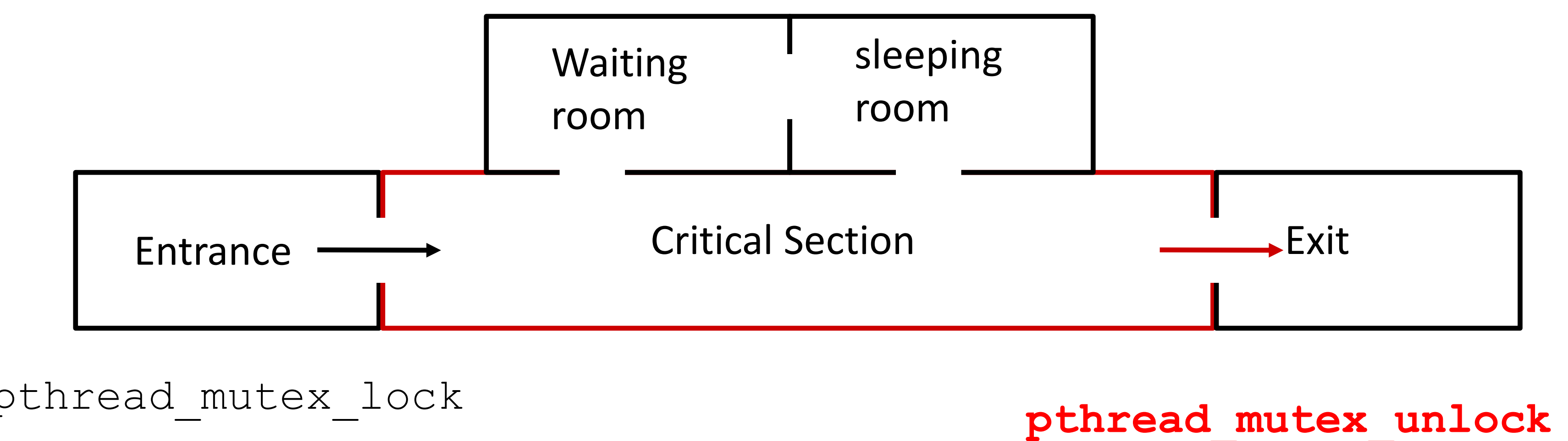


`pthread_mutex_lock`

A thread enters the critical section by acquiring a lock

Condition Variable & Mutex Visualization

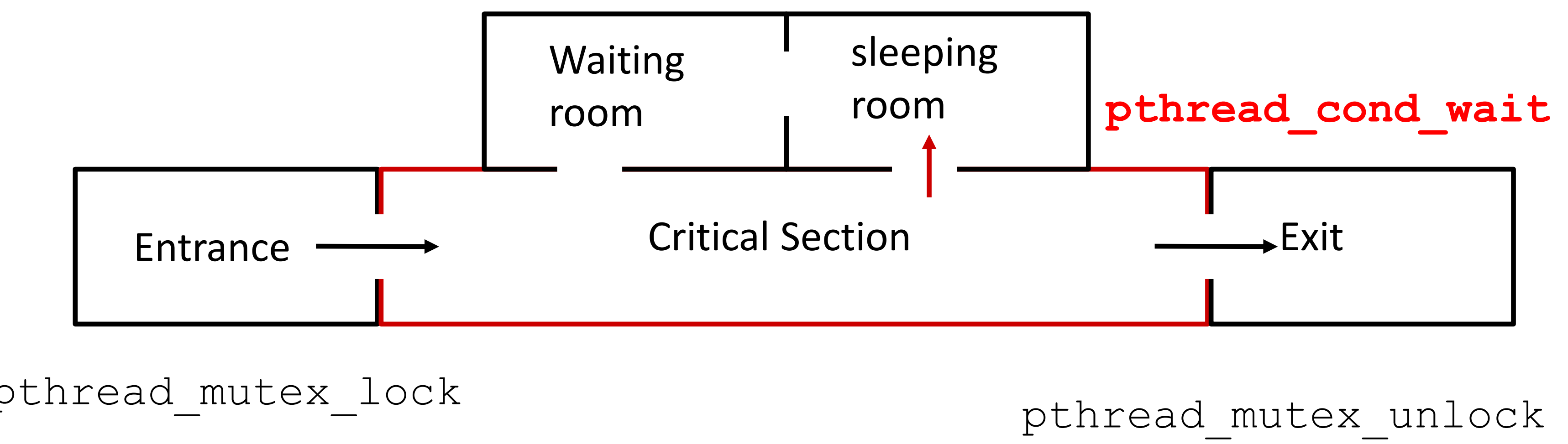
- This is to visualize how we are using condition variables in this example



A thread can exit the critical section by acquiring a lock

Condition Variable & Mutex Visualization

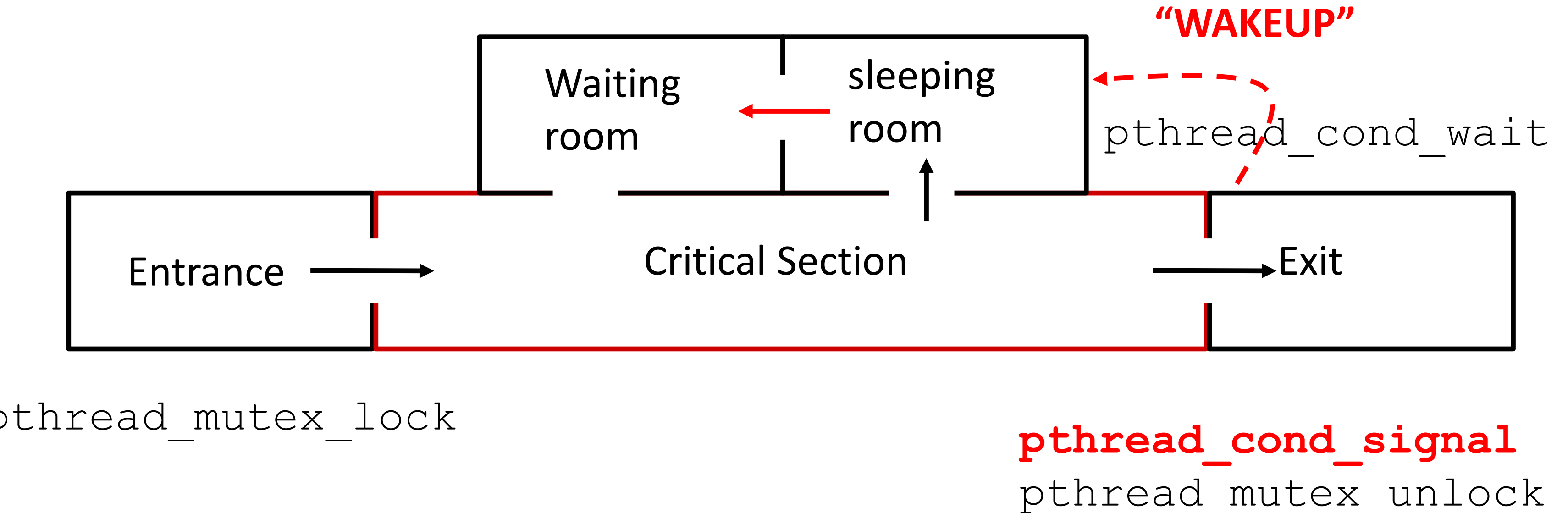
- This is to visualize how we are using condition variables in this example



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

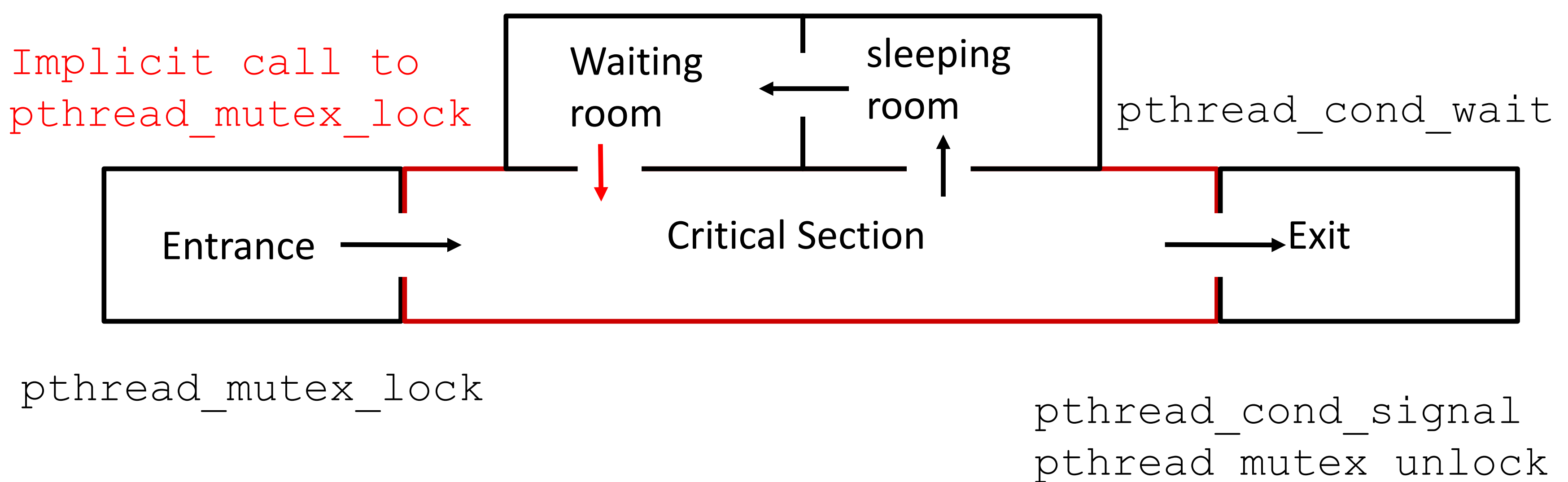
- This is to visualize how we are using condition variables in this example



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);
  }
}
```

Poll Everywhere

pollev.com/tqm

- Why is this needed? Why may our code be incorrect if we don't have one?

- By the time a thread wakes up, the shared state (the queue maybe empty again)

```
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

- Condition Variables
- **Monitors**
- Reader/Writer Problem
- Dining Philosophers
- Deadlock Prevention
- Deadlock Handling (start)

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, a condition variable is an implementation detail**

Lecture Outline

- Condition Variables
- Monitors
- **Reader/Writer Problem**
- Dining Philosophers
- Deadlock Prevention
- Deadlock Handling (start)

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

Poll Everywhere

pollev.com/tqm

- There are (at least 2) issues with liveness in this solution, what are they?

```
void read_lock() {
   pthread_mutex_lock(&lock);
   while (writer_active || writers_waiting > 0) {
     pthread_cond_wait(&cond, &lock);
   }
   num_readers++;
   pthread_mutex_unlock(&lock);
   // after we return, the caller is safe to read
}

void read_unlock() {
  pthread_mutex_lock(&lock);
  num_readers--;
  if (num_readers == 0) {
     pthread_cond_broadcast(&cond);
     // why do we need to broadcast here?
  }
  pthread_mutex_unlock(&lock);
}
```

```
void write_lock() {
  pthread_mutex_lock(&lock);
  writers_waiting++;
  while(num_readers > 0 || writer_active) {
    pthread_cond_wait(&cond, &lock);
  }
  writer_active = true;
  writers_waiting--;
  pthread_mutex_unlock(&lock);

  // after we return, caller thread
  // is ok to write to data
}

void write_unlock() {
  pthread_mutex_lock(&lock);
  writer_active = false;
  pthread_cond_broadcast(&cond, &lock);
  pthread_mutex_unlock(&lock);
}
```

pollev.com/tqm

- There are (at least 2) issues with liveness in this solution, what are they?

```
void read_lock() {
   pthread_mutex_lock(&lock);
   while (writer_active || writers_waiting > 0) {
     pthread_cond_wait(&cond, &lock);
   }
   num_readers++;
   pthread_mutex_unlock(&lock);
   // after we return, the caller is safe to read
}

void read_unlock() {
  pthread_mutex_lock(&lock);
  num_readers--;
  if (num_readers == 0) {
     pthread_cond_broadcast(&cond);
     // why do we need to broadcast here?
  }
  pthread_mutex_unlock(&lock);
}
```

```
void write_lock() {
  pthread_mutex_lock(&lock);
  writers_waiting++;
  while(num_readers > 0 || writer_active) {
    pthread_cond_wait(&cond, &lock);
  }
  writer_active = true;
  writers_waiting--;
  pthread_mutex_unlock(&lock);

  // after we return, caller thread
  // is ok to write to data
}

void write_unlock() {
  pthread_mutex_lock(&lock);
  writer_active = false;
  pthread_cond_broadcast(&cond, &lock);
  pthread_mutex_unlock(&lock);
}
```

Starvation on readers if enough writers come in.

More minor: Condition broadcast may unnecessarily awake readers if there are waiting writers

pthread_rwlock

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

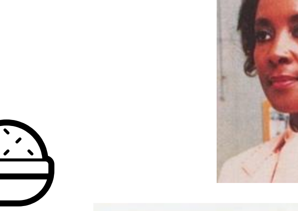
- Very similar to pthread_mutex, but two types of locking
  - `pthread_rwlock_rdlock(…);` // lock as a reader
  - `pthread_rwlock_wrlock(…);` // lock as a writer

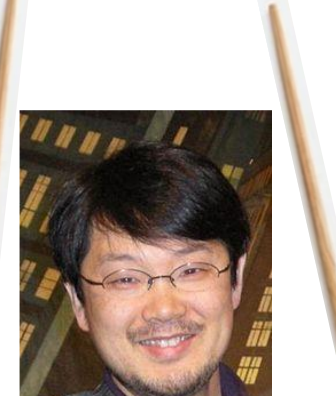
Lecture Outline

- Condition Variables
- Monitors
- Reader/Writer Problem
- **Dining Philosophers**
- Deadlock Prevention
- Deadlock Handling (start)

Dining Philosophers

- Assume the following situation
  - There are N philosophers (computer scientists) that are trying to eat rice.
  - They only have one chopstick each!
    - Need two chopsticks to eat ☹
  - Alternate between two states:
    - Thinking
    - Eating
  - They are arranged in a circle with a chopstick between each of them





Dining Philosophers

- Philosophers have good table manners
  - Must acquire two chopsticks to eat
- Useful abstraction / "standard problem":
  - Soundness
    - Every chopstick is held by <= 1 philosopher at a time
  - Deadlock Free
    - No state where no one gets to eat
  - Starvation Free
    - Solution guarantees that all philosophers occasionally eat

First Solution Attempt

- If we number each philosopher 0 – N and the each chopstick is also 0 – N, we can model the problem with mutexes, each chopstick is a mutex and each philosopher is a thread
  - To eat, thread I must acquire lock **`I`** and **`I + 1`**
  - This ensures that each chopstick is only in use by one philosopher at a time

```
while (true) {
  pthread_mutex_lock(&chopstick[i]);
  pthread_mutex_lock(&chopstick[(i + 1) % N]);
  eat();
  pthread_mutex_unlock(&chopstick[(i + 1) % N]);
  pthread_mutex_unlock(&chopstick[i]);
  think();
}
```

pollev.com/tqm

- What's wrong with this? Any Ideas on how to fix it?

```
while (true) {
  pthread_mutex_lock(&chopstick[i]);
  pthread_mutex_lock(&chopstick[(i + 1) % N]);
  eat();
  pthread_mutex_unlock(&chopstick[(i + 1) % N]);
  pthread_mutex_unlock(&chopstick[i]);
  think();
}
```

pollev.com/tqm

- What's wrong with this? Any Ideas on how to fix it?

```
while (true) {
  pthread_mutex_lock(&chopstick[i]);
  pthread_mutex_lock(&chopstick[(i + 1) % N]);
  eat();
  pthread_mutex_unlock(&chopstick[(i + 1) % N]);
  pthread_mutex_unlock(&chopstick[i]);
  think();
}
```

Deadlock is possible: what happens if all threads pickup their left at the same time?

Second Attempt: Round Robin

- Our first attempt deadlocks.
- What if we instead we tried doing this "round robin", we pass around a token that says "it is your turn to eat"
- Can this deadlock?

  No

- What issues arise with this solution?

  Not parallel, just sequential eating ☹
  Everyone guaranteed gets to eat though ☺

Third Attempt: Global Mutex

- What if instead, we add another "global" mutex that controls permission to pick up chopsticks. Once a philosopher has chopsticks, they can release the lock before they eat

- In our metaphor, this means that each philosopher "waits in line" to pick up chopsticks

- Can this deadlock?

  No

- What issues arise with this solution?

  Not the most parallel, could result in sequential

  Not everyone guarantee gets to eat

Fourth Attempt: More Human Approach

- What if instead, if a philosopher fails to get a chopstick, it puts down any chopsticks it has, waits for a little bit and then tries again?

- Can we do this in code?
  - `pthread_mutex_trylock`: if the lock can't be acquired, return immediately
  - `pthread_mutex_timedlock`: timeout after trying to get a mutex for some specified amount of time

- Can this deadlock? No

- What issues arise with this solution?

Possible spinning and starvation

Fifth Attempt: Break the Symmetry

- What if the even numbered philosophers and odd numbered philosophers do things differently?
  - Even Numbered: Grab chopstick on their left and then right
  - Odd Numbered: Grab chopstick on their right and then left

- Can this deadlock?

  No

- What issues arise with this solution?

threads may still possibly starve ☹

Lecture Outline

- Condition Variables
- Monitors
- Reader/Writer Problem
- Dining Philosophers
- **Deadlock Prevention**
- Deadlock Handling (start)

Previously: 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 ☹

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 thread can acquire resources:
  - All at once
  - Accumulate them over time
  - If it fails to acquire a resource, it will (by default) wait until it is available before doing anything

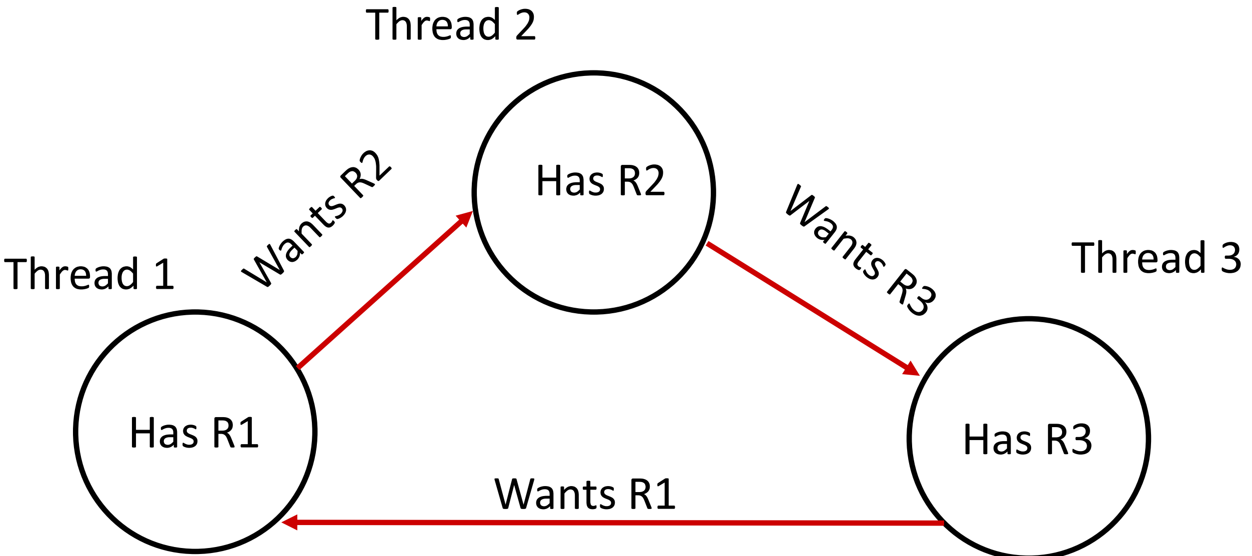
- Deadlock: Cyclical dependency on resource acquisition so that none of them can proceed
  - Even if all unblocked processes release, deadlock will continue

Preconditions for Deadlock

- Deadlock can only happen if these occur simultaneously:
  - **Mutual Exclusion**: at least one resource must be held exclusively by one thread
  - **Hold and Wait**: a thread must be holding a resource, requesting a resource that is held by a thread, and then waiting for it.
  - **No preemption**: A resource is held by a thread until it explicitly releases it. It cannot be preempted by the OS or something else to force it to release the resource
  - **Circular Wait**:
    Can be a chain of more than 2 threads
    Each thread must be waiting for a resource that is held by another thread. That other thread must waiting on a resource that forms a chain of dependency

Circular Wait Example

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



pollev.com/tqm

- Can a thread deadlock if there is only one thread?

Deadlock Prevention

- If we can remove the conditions for deadlock, we could avoid prevent deadlock from every happening

Deadlock Prevention: Mutual Exclusion

- **Mutual Exclusion**: at least one resource must be held exclusively by one thread

- You usually need mutual exclusion or you don't, so it is hard to avoid.
- Some resources require exclusive access

- A lot of work done related to this
  - called: Lock-free programming, Lock-less programming, or Non-blocking algorithms
  - General idea is to take advantage of operations that are atomic at the hardware level when sharing is needed

Deadlock Prevention: Hold and Wait

- **Hold and Wait**: a thread must be holding a resource, requesting a resource that is held by a thread, and then waiting for it.

- What if we had each thread acquire all resources it needs in the begging "at once"
  - This is like one of our dining philosophers implementations
  - Not always practical, a thread may not know ahead of time all the resources it will need

Deadlock Prevention: No Preemption

- **No preemption**: A resource is held by a thread until it explicitly releases it. It cannot be preempted by the OS or something else to force it to release the resource

- If we force a thread to release a resource, how do we ensure it is in a valid state?
  - Undoing actions and recovering valid state is complex (more on this next lecture)

Deadlock Prevention: Circular Wait

- **Circular Wait**: Each thread must be waiting for a resource that is held by another thread. That other thread must waiting on a resource that forms a chain of dependency

- Break cycles in resource acquisition.
- We could enforce an ordering to resource acquisition.
  - Consider dining philosophers, what if each thread was required to get the lowest numbered chopstick it wants first?

- Challenge: Still we may not know all resources we need ahead of time

Deadlock Prevention Summary

- Prevent deadlocks by removing any one of the four deadlock preconditions
- But eliminating even one of the preconditions is often hard/impossible
  - Mutual Exclusion is necessary in a lot of situations
  - Forcing a lower priority process to release resources early requires rollback of execution
  - Not always possible to know all resources that an operating system or process will use upfront

Lecture Outline

- Condition Variables
- Monitors
- Reader/Writer Problem
- Dining Philosophers
- Deadlock Prevention
- **Deadlock Handling (start)**

Deadlock Handling: Ostrich Algorithm

Deadlock Handling: Ostrich Algorithm

Ostriches don't actually do this, but it is an old myth

Deadlock Handling: Ostrich Algorithm

- Ignoring potential problems
  - Usually under the assumption that it is either rare, too expensive to handle, and/or not a fatal error

- Used in real world contexts, there is a real cost to tracking down every possible deadlock case and trying to fix it
  - Cost on the developer side: more time to develop
  - Cost on the software side: more computation for these things to do, slows things down

Deadlock Handling: Prevention

- Ad Hoc Approach
  - Key insights into application logic allow you to write code that avoids cycles/deadlock
  - Example: Dining Philosophers breaking symmetry with even/odd philosophers

- Exhaustive Search Approach
  - Static analysis on source code to detect deadlocks
  - Formal verification: model checking
  - Unable to scale beyond small programs in practice
    Impossible to prove for any arbitrary program (without restrictions)

Next Lecture:

- More On Deadlock
  - Detection
  - Handling
  - Avoidance