

# CS111, Lecture 15

## Multithreading Patterns



masks strongly  
recommended

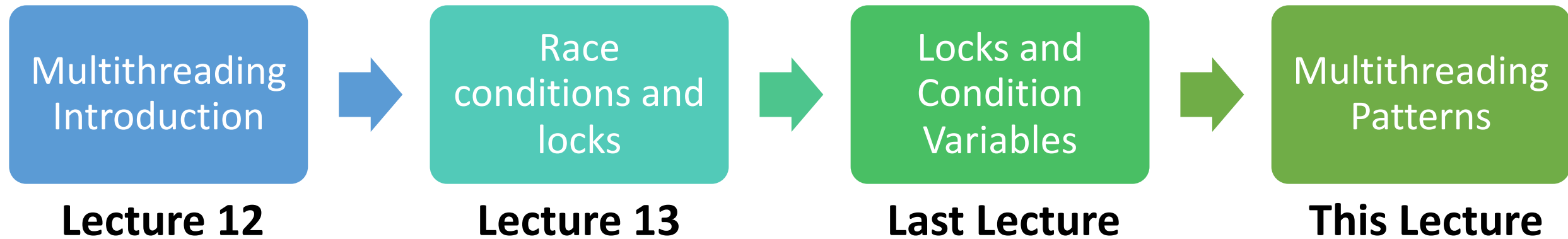
This document is copyright (C) Stanford Computer Science and Nick Troccoli, licensed under Creative Commons Attribution 2.5 License. All rights reserved.

Based on slides and notes created by John Ousterhout, Jerry Cain, Chris Gregg, and others.

NOTICE RE UPLOADING TO WEBSITES: This content is protected and may not be shared, uploaded, or distributed. (without expressed written permission)

# **Topic 3: Multithreading - How can we have concurrency within a single process? How does the operating system support this?**

# CS111 Topic 3: Multithreading, Part 1



**assign4:** implement several multithreaded programs while eliminating race conditions!

# Learning Goals

- Get more practice using both mutexes and condition variables to implement synchronization logic.
- Learn about the **monitor** pattern for designing multithreaded code in the simplest way possible, using classes.

# Plan For Today

- **Recap and continuing:** condition variables and dining philosophers
- Unique locks
- Monitor pattern
- **Example:** Bridge Crossing

```
cp -r /afs/ir/class/cs111/lecture-code/lect15 .
```

# Plan For Today

- **Recap and continuing: condition variables and dining philosophers**
- Unique locks
- Monitor pattern
- **Example: Bridge Crossing**

```
cp -r /afs/ir/class/cs111/lecture-code/lect15 .
```

# Condition Variables

A **condition variable** is a variable type that can be shared across threads and used for one thread to notify other thread(s) when something happens. Conversely, a thread can also use this to wait until it is notified by another thread.

- You make one for each distinct event you need to wait / notify for.
- We can call **wait** on the condition variable to sleep until another thread signals this condition variable.
- You call **notify\_all** on the condition variable to send a notification to all waiting threads and wake them up.
- Analogy: radio station – broadcast and tune in

# Condition Variables

1. Identify a single kind of event that we need to wait / notify for
2. Ensure there is proper state to check if the event has happened
3. Create a condition variable and share it among all threads either waiting for that event to happen or triggering that event
4. Identify who will notify that this happens, and have them notify via the condition variable
5. Identify who will wait for this to happen, and have them wait via the condition variable



# Condition Variables

1. Identify a single kind of event that we need to wait / notify for
2. Ensure there is proper state to check if the event has happened
3. Create a condition variable and share it among all threads either waiting for that event to happen or triggering that event
4. Identify who will notify that this happens, and have them notify via the condition variable
5. Identify who will wait for this to happen, and have them wait via the condition variable

The event here is "some permits are again available".

# Condition Variables

1. Identify a single kind of event that we need to wait / notify for
2. **Ensure there is proper state to check if the event has happened**
3. Create a condition variable and share it among all threads either waiting for that event to happen or triggering that event
4. Identify who will notify that this happens, and have them notify via the condition variable
5. Identify who will wait for this to happen, and have them wait via the condition variable

We can check whether there are permits now available by checking the permits count.

# Condition Variables

1. Identify a single kind of event that we need to wait / notify for
2. Ensure there is proper state to check if the event has happened
- 3. Create a condition variable and share it among all threads either waiting for that event to happen or triggering that event**
4. Identify who will notify that this happens, and have them notify via the condition variable
5. Identify who will wait for this to happen, and have them wait via the condition variable

# Condition Variables

```
int main(int argc, const char *argv[]) {
    mutex forks[kNumForks];
    size_t permits = kNumForks - 1;
    mutex permitsLock;
    condition_variable_any permitsCV;

    thread philosophers[kNumPhilosophers];
    for (size_t i = 0; i < kNumPhilosophers; i++) {
        philosophers[i] = thread(philosopher, i, ref(forks[i]),
                                ref(forks[(i + 1) % kNumPhilosophers]),
                                ref(permits), ref(permitsCV),
                                ref(permitsLock));
    }
    for (thread& p: philosophers) p.join();
    return 0;
}
```

# Condition Variables

1. Identify a single kind of event that we need to wait / notify for
2. Ensure there is proper state to check if the event has happened
3. Create a condition variable and share it among all threads either waiting for that event to happen or triggering that event
- 4. Identify who will notify that this happens, and have them notify via the condition variable**
5. Identify who will wait for this to happen, and have them wait via the condition variable

When someone returns a permit and there were no permits available previously, notify all.

# grantPermission

We must notify all once permits have become available again to wake up waiting threads.

```
static void grantPermission(size_t& permits,  
condition_variable_any& permitsCV, mutex& permitsLock) {  
    permitsLock.lock();  
    permits++;  
    if (permits == 1) permitsCV.notify_all();  
    permitsLock.unlock();  
}
```

When someone returns a permit and there were no permits available previously (meaning some people might be waiting), notify all.

# Condition Variables

1. Identify a single kind of event that we need to wait / notify for
2. Ensure there is proper state to check if the event has happened
3. Create a condition variable and share it among all threads either waiting for that event to happen or triggering that event
4. Identify who will notify that this happens, and have them notify via the condition variable
5. Identify who will wait for this to happen, and have them wait via the condition variable

If we need a permit but there are none available, wait.

# waitForPermission (In progress)

If no permits are available, we must wait until one becomes available.

```
static void waitForPermission(size_t& permits,  
condition_variable_any& permitsCV, mutex& permitsLock) {  
    if (permits == 0) {  
        permitsCV.wait(); // (note: not final form of wait)  
    }  
    permits--;  
}
```

This is the idea for what we want to do – but there are some additional cases/ quirks we need to account for.



# waitForPermission (Final version)

```
static void waitForPermission(size_t& permits, condition_variable_any& permitsCV,
mutex& permitsLock) {
    permitsLock.lock();
    while (permits == 0) {
        permitsCV.wait(permitsLock);
    }
    permits--;
    permitsLock.unlock();
}
```

This is the final implementation with the final version of wait() that takes a mutex parameter and which is called in a while loop. Let's build our way to this solution!

# waitForPermission (In progress)

If no permits are available, we must wait until one becomes available.

```
static void waitForPermission(size_t& permits,
condition_variable_any& permitsCV, mutex& permitsLock) {
    if (permits == 0) {
        permitsCV.wait();    // (note: not final form of wait)
    }
    permits--;
}
```

**Problem:** we are accessing and modifying permits without our lock! This causes race conditions.

# waitForPermission (In progress)

If no permits are available, we must wait until one becomes available.

```
static void waitForPermission(size_t& permits,
condition_variable_any& permitsCV, mutex& permitsLock) {
    permitsLock.lock();
    if (permits == 0) {
        permitsCV.wait();           // (note: not final form of wait)
    }
    permits--;
    permitsLock.unlock();
}
```

**Problem:** we have the lock when we call wait(). Wait puts this thread to sleep and runs others, but no-one will be able to return permits if we keep the lock.

# grantPermission

Other threads need the lock in order to return permits:

```
static void grantPermission(size_t& permits,  
condition_variable_any& permitsCV, mutex& permitsLock) {  
    permitsLock.lock();  
    permits++;  
    if (permits == 1) permitsCV.notify_all();  
    permitsLock.unlock();  
}
```

# waitForPermission (In progress)

If no permits are available, we must wait until one becomes available.

**Solution:** we must give up ownership of the lock when we wait, so that someone else can put a permit back.

```
static void waitForPermission(size_t& permits,
condition_variable_any& permitsCV, mutex& permitsLock) {
    permitsLock.lock();
    if (permits == 0) {
        permitsLock.unlock();
        permitsCV.wait(); // (note: not final form of wait)
        permitsLock.lock();
    }
    permits--;
    permitsLock.unlock();
}
```

# Deadlock, Round 2

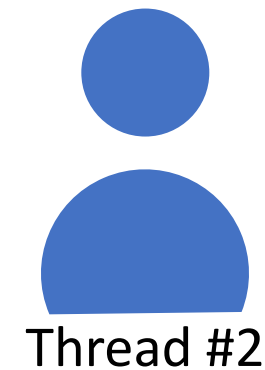
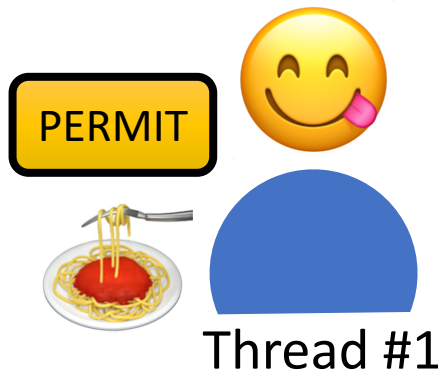
```
static void waitForPermission(size_t& permits, condition_variable_any&
permitsCV, mutex& permitsLock) {
    permitsLock.lock();
    if (permits == 0) {
        permitsLock.unlock();
        permitsCV.wait();    // (note: not final form of wait)
        permitsLock.lock();
    }
    permits--;
    permitsLock.unlock();
}
```

Spoiler: there is a race condition that could lead to deadlock. What ordering of events between threads could cause deadlock here? (Hint: if a thread isn't waiting, it won't get a notification from another thread).

# Deadlock: waitForPermission

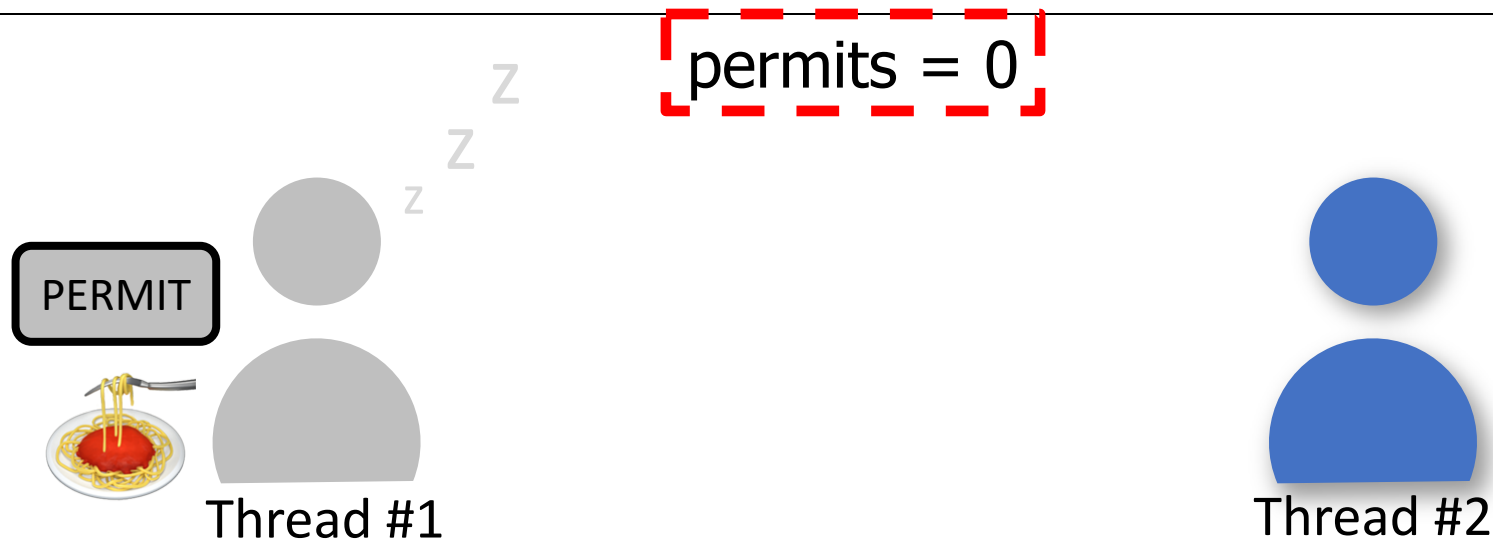
```
static void waitForPermission(size_t& permits, condition_variable_any& permitsCV,
mutex& permitsLock) {
    permitsLock.lock();
    if (permits == 0) {
        permitsLock.unlock();
        permitsCV.wait(); // (note: not final form of wait)
        permitsLock.lock();
    }
    permits--;
    permitsLock.unlock();
}
```

permits = 0



# Deadlock: waitForPermission

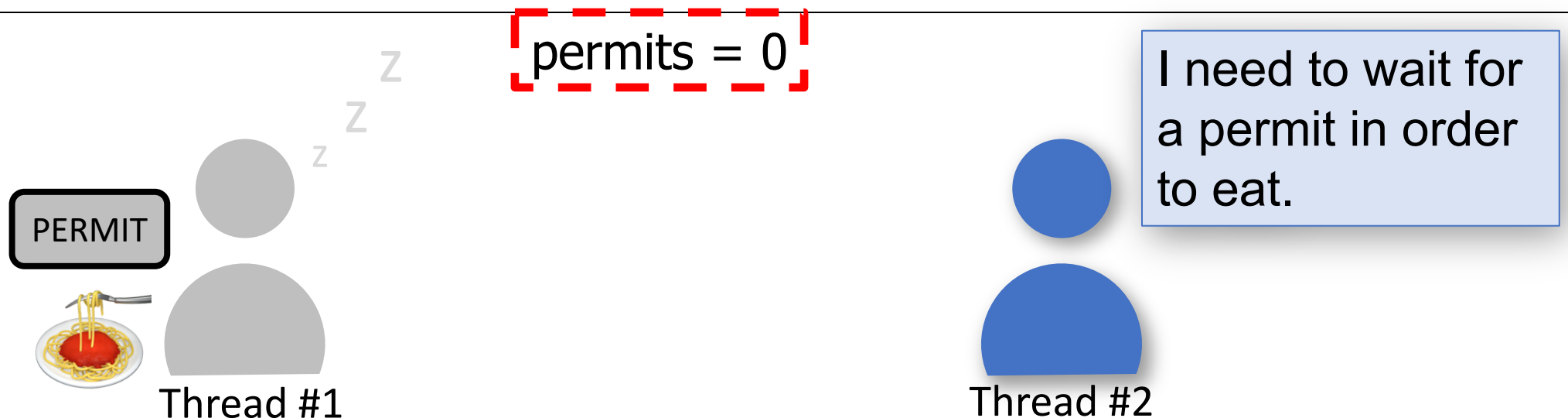
```
static void waitForPermission(size_t& permits, condition_variable_any& permitsCV,  
mutex& permitsLock) {  
    permitsLock.lock();  
    if (permits == 0) {  
        permitsLock.unlock();  
        permitsCV.wait();    // (note: not final form of wait)  
        permitsLock.lock();  
    }  
    permits--;  
    permitsLock.unlock();  
}
```





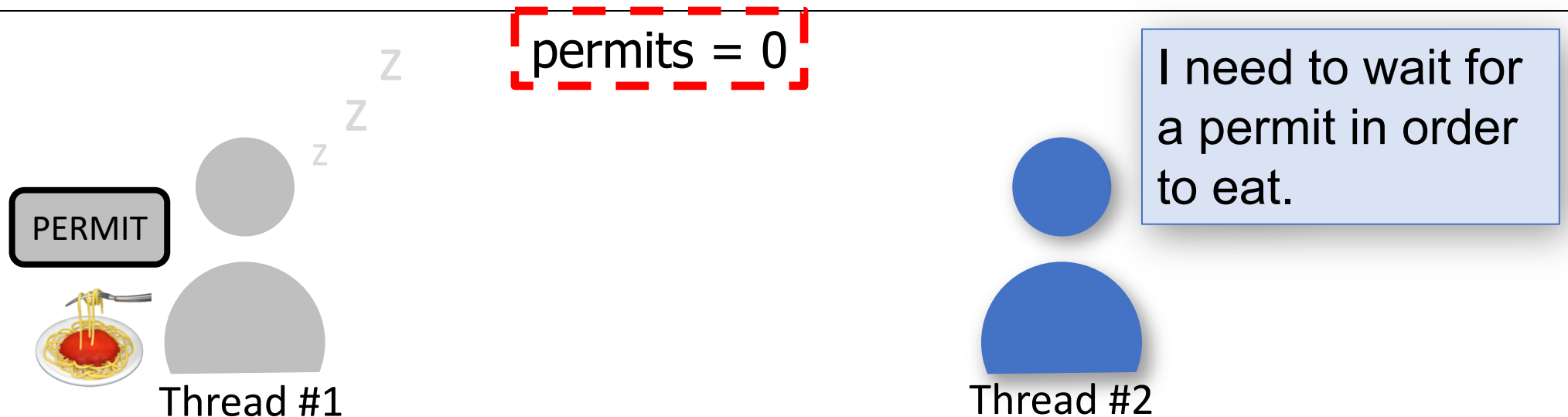
# Deadlock: waitForPermission

```
static void waitForPermission(size_t& permits, condition_variable_any& permitsCV,  
mutex& permitsLock) {  
    permitsLock.lock();  
    if (permits == 0) {  
        permitsLock.unlock();  
        permitsCV.wait();    // (note: not final form of wait)  
        permitsLock.lock();  
    }  
    permits--;  
    permitsLock.unlock();  
}
```



# Deadlock: waitForPermission

```
static void waitForPermission(size_t& permits, condition_variable_any& permitsCV,  
mutex& permitsLock) {  
    permitsLock.lock();  
    if (permits == 0) {  
        permitsLock.unlock();  
        permitsCV.wait();    // (note: not final form of wait)  
        permitsLock.lock();  
    }  
    permits--;  
    permitsLock.unlock();  
}
```



# Deadlock: waitForPermission

```
static void waitForPermission(size_t& permits, condition_variable_any& permitsCV,
mutex& permitsLock) {
    permitsLock.lock();
    if (permits == 0) {
        permitsLock.unlock();
        permitsCV.wait();    // (note: not final form of wait)
        permitsLock.lock();
    }
    permits--;
    permitsLock.unlock();
}
```

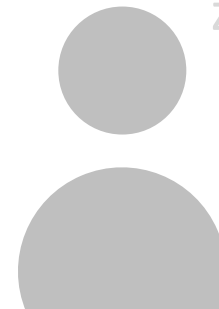
All done eating! I  
will return my permit.

PERMIT



Thread #1

permits = 0



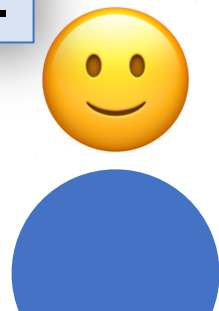
Thread #2

z  
z  
z

# Deadlock: waitForPermission

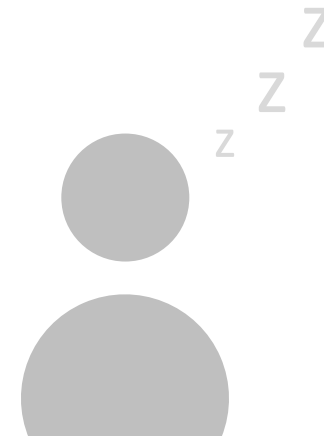
```
static void waitForPermission(size_t& permits, condition_variable_any& permitsCV,
mutex& permitsLock) {
    permitsLock.lock();
    if (permits == 0) {
        permitsLock.unlock();
        permitsCV.wait();    // (note: not final form of wait)
        permitsLock.lock();
    }
    permits--;
    permitsLock.unlock();
}
```

All done eating! I  
will return my permit.



Thread #1

permits = 1

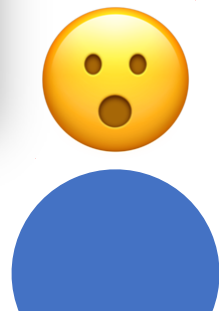


Thread #2

# Deadlock: waitForPermission

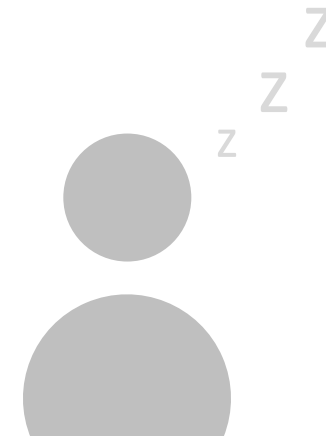
```
static void waitForPermission(size_t& permits, condition_variable_any& permitsCV,
mutex& permitsLock) {
    permitsLock.lock();
    if (permits == 0) {
        permitsLock.unlock();
        permitsCV.wait();    // (note: not final form of wait)
        permitsLock.lock();
    }
    permits--;
    permitsLock.unlock();
}
```

Oh! I should notify  
that there is a  
permit now.



Thread #1

**permits = 1**



Thread #2

# Deadlock: waitForPermission

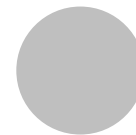
```
static void waitForPermission(size_t& permits, condition_variable_any& permitsCV,
mutex& permitsLock) {
    permitsLock.lock();
    if (permits == 0) {
        permitsLock.unlock();
        permitsCV.wait();    // (note: not final form of wait)
        permitsLock.lock();
    }
    permits--;
    permitsLock.unlock();
}
```

*“Attention waiting threads, a permit is available!”*



Thread #1

**permits = 1**



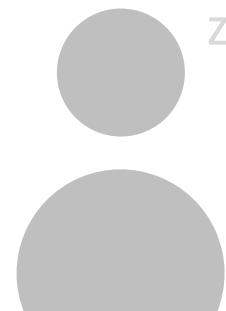
Thread #2

z  
z  
z

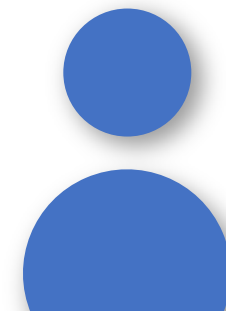
# Deadlock: waitForPermission

```
static void waitForPermission(size_t& permits, condition_variable_any& permitsCV,
mutex& permitsLock) {
    permitsLock.lock();
    if (permits == 0) {
        permitsLock.unlock();
        permitsCV.wait(); // (note: not final form of wait)
        permitsLock.lock();
    }
    permits--;
    permitsLock.unlock();
}
```

**permits = 1**



Thread #1

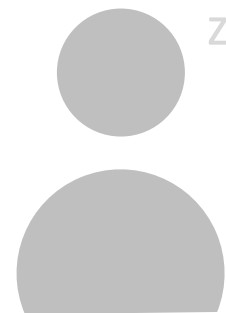


Thread #2

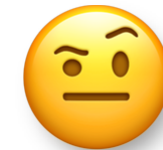
# Deadlock: waitForPermission

```
static void waitForPermission(size_t& permits, condition_variable_any& permitsCV,  
mutex& permitsLock) {  
    permitsLock.lock();  
    if (permits == 0) {  
        permitsLock.unlock();  
        permitsCV.wait(); // (note: not final form of wait)  
        permitsLock.lock();  
    }  
    permits--;  
    permitsLock.unlock();  
}
```

**permits = 1**



Thread #1



Thread #2

**\*100 years later\***



# Deadlock: waitForPermission

```
static void waitForPermission(size_t& permits, condition_variable_any& permitsCV,
mutex& permitsLock) {
    permitsLock.lock();
    if (permits == 0) {
        permitsLock.unlock();
        permitsCV.wait();    // (note: not final form of wait)
        permitsLock.lock();
    }
    permits--;
    permitsLock.unlock();
}
```

If we give up the lock before calling *wait()*, someone could notify before we are ready, because notifications aren't queued! If that is the last notification, we may wait forever.

# Deadlock: waitForPermission

```
static void waitForPermission(size_t& permits, condition_variable_any& permitsCV,
mutex& permitsLock) {
    permitsLock.lock();
    if (permits == 0) {
        permitsCV.wait(permitsLock);
    }
    permits--;
    permitsLock.unlock();
}
```

**Solution:** condition variables are meant for these situations.

- **wait()** takes a mutex as a parameter
- It will unlock the mutex *for us* after we are put to sleep.
- When we are notified, it will only return once it has reacquired the mutex for us.

# Condition Variable Wait

```
static void waitForPermission(size_t& permits, condition_variable_any& permitsCV,
mutex& permitsLock) {
    permitsLock.lock();
    if (permits == 0) {
        permitsCV.wait(permitsLock);
    }
    permits--;
    permitsLock.unlock();
}
```

**cv.wait()** does the following:

1. it puts the caller to sleep *and* unlocks the given lock, all atomically
2. it wakes up when the cv is signaled
3. upon waking up, it tries to acquire the given lock (and blocks until it's able to do so)
4. then, cv.wait returns

# waitForPermission (In progress)

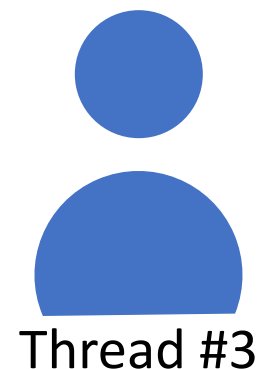
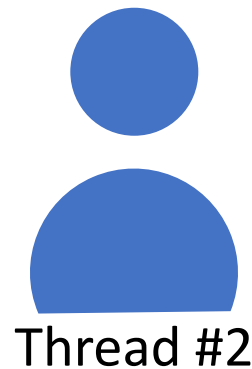
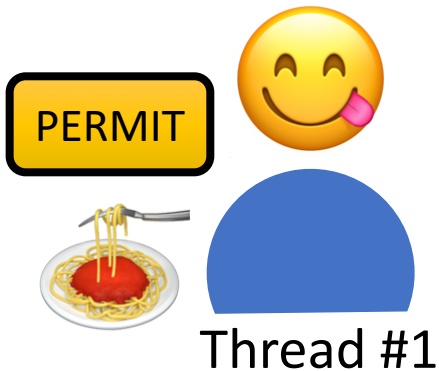
```
static void waitForPermission(size_t& permits, condition_variable_any& permitsCV,
mutex& permitsLock) {
    permitsLock.lock();
    if (permits == 0) {
        permitsCV.wait(permitsLock);
    }
    permits--;
    permitsLock.unlock();
}
```

*Spoiler:* there is a race condition here that could lead to negative permits if multiple threads are waiting on a permit (e.g. say we limit permits to 3).

# waitForPermission Over-permitting

```
static void waitForPermission(size_t& permits, condition_variable_any& permitsCV,  
mutex& permitsLock) {  
    permitsLock.lock();  
    if (permits == 0) {  
        permitsCV.wait(permitsLock);  
    }  
    permits--;  
    permitsLock.unlock();  
}
```

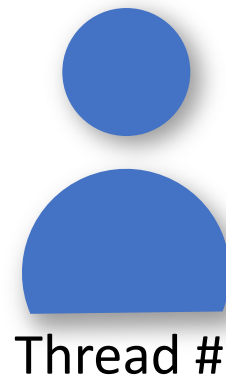
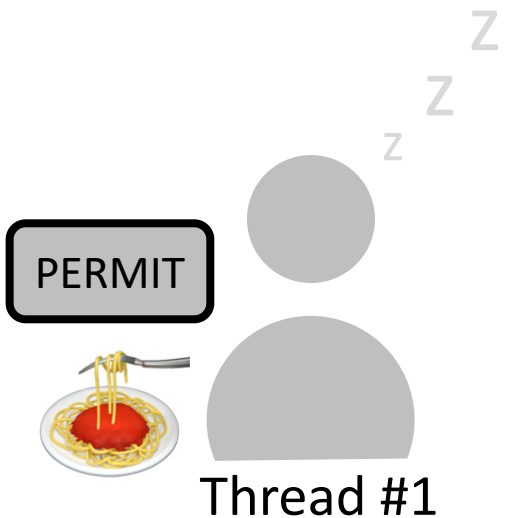
permits = 0



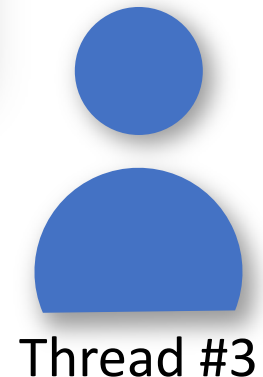
# waitForPermission Over-permitting

```
static void waitForPermission(size_t& permits, condition_variable_any& permitsCV,  
mutex& permitsLock) {  
    permitsLock.lock();  
    if (permits == 0) {  
        permitsCV.wait(permitsLock);  
    }  
    permits--;  
    permitsLock.unlock();  
}
```

**permits = 0**



We need to wait  
for a permit in  
order to eat.

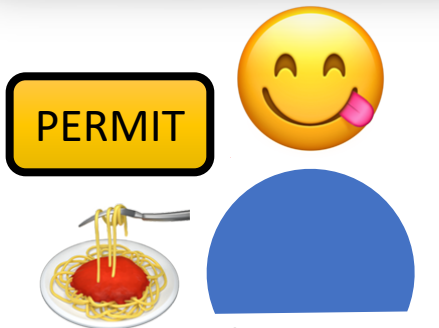


# waitForPermission Over-permitting

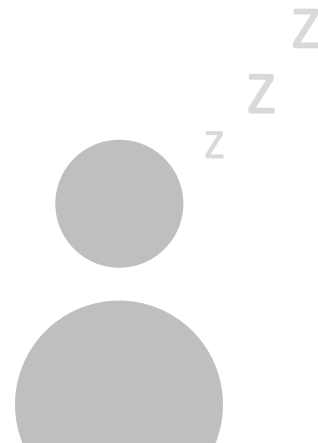
```
static void waitForPermission(size_t& permits, condition_variable_any& permitsCV,  
mutex& permitsLock) {  
    permitsLock.lock();  
    if (permits == 0) {  
        permitsCV.wait(permitsLock);  
    }  
    permits--;  
    permitsLock.unlock();  
}
```

All done eating! I  
will return my permit.

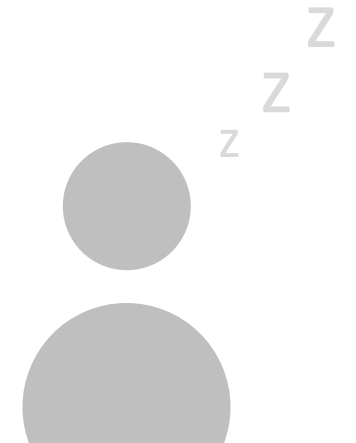
permits = 0



Thread #1



Thread #2

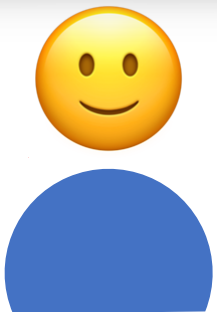


Thread #3

# waitForPermission Over-permitting

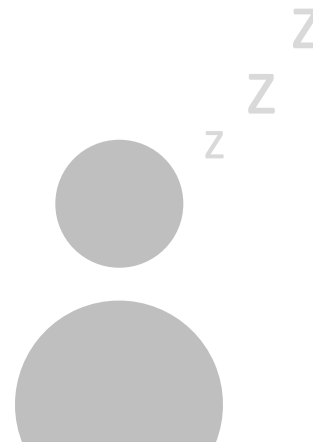
```
static void waitForPermission(size_t& permits, condition_variable_any& permitsCV,  
mutex& permitsLock) {  
    permitsLock.lock();  
    if (permits == 0) {  
        permitsCV.wait(permitsLock);  
    }  
    permits--;  
    permitsLock.unlock();  
}
```

All done eating! I  
will return my permit.

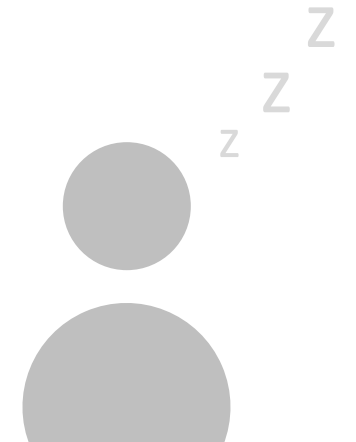


Thread #1

permits = 1



Thread #2



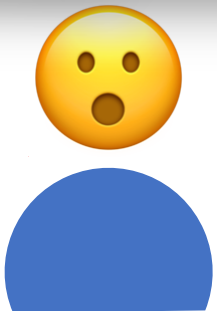
Thread #3



# waitForPermission Over-permitting

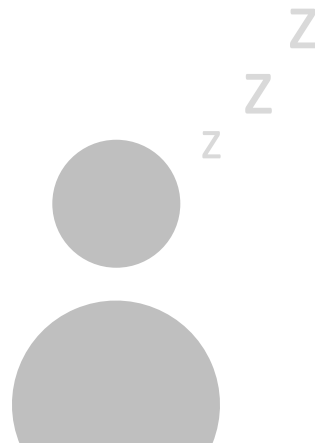
```
static void waitForPermission(size_t& permits, condition_variable_any& permitsCV,  
mutex& permitsLock) {  
    permitsLock.lock();  
    if (permits == 0) {  
        permitsCV.wait(permitsLock);  
    }  
    permits--;  
    permitsLock.unlock();  
}
```

Oh! I should notify  
that there is a  
permit now.

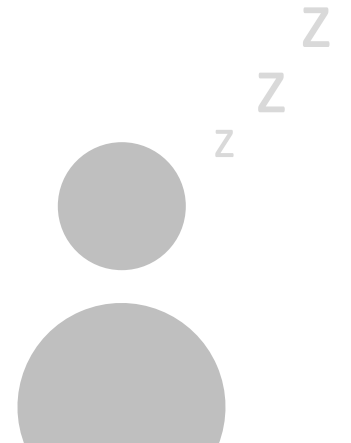


Thread #1

permits = 1



Thread #2



Thread #3

# waitForPermission Over-permitting

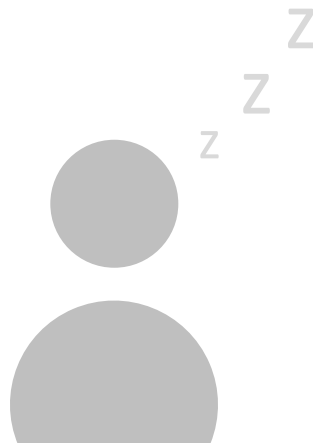
```
static void waitForPermission(size_t& permits, condition_variable_any& permitsCV,  
mutex& permitsLock) {  
    permitsLock.lock();  
    if (permits == 0) {  
        permitsCV.wait(permitsLock);  
    }  
    permits--;  
    permitsLock.unlock();  
}
```

*“Attention waiting threads, a permit is available!”*

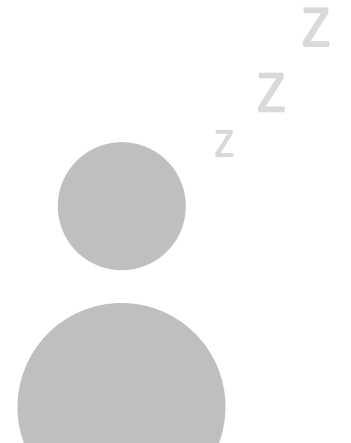


Thread #1

**permits = 1**



Thread #2

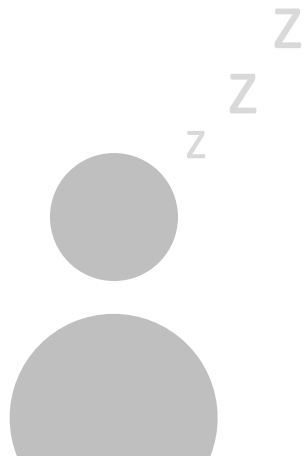


Thread #3

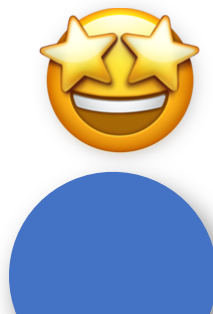
# waitForPermission Over-permitting

```
static void waitForPermission(size_t& permits, condition_variable_any& permitsCV,  
mutex& permitsLock) {  
    permitsLock.lock();  
    if (permits == 0) {  
        permitsCV.wait(permitsLock);  
    }  
    permits--;  
    permitsLock.unlock();  
}
```

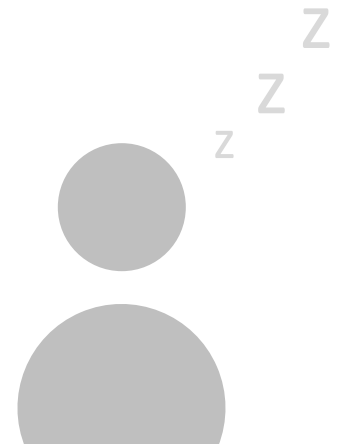
**permits = 1**



Thread #1



Thread #2

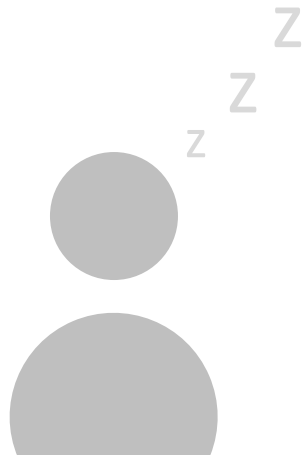


Thread #3

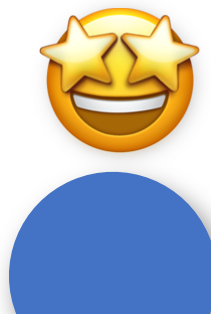
# waitForPermission Over-permitting

```
static void waitForPermission(size_t& permits, condition_variable_any& permitsCV,  
mutex& permitsLock) {  
    permitsLock.lock();  
    if (permits == 0) {  
        permitsCV.wait(permitsLock);  
    }  
    permits--;  
    permitsLock.unlock();  
}
```

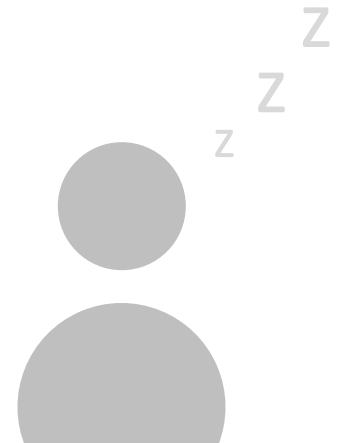
**permits = 1**



Thread #1



Thread #2

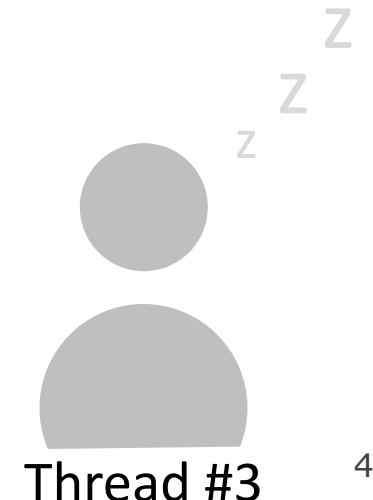
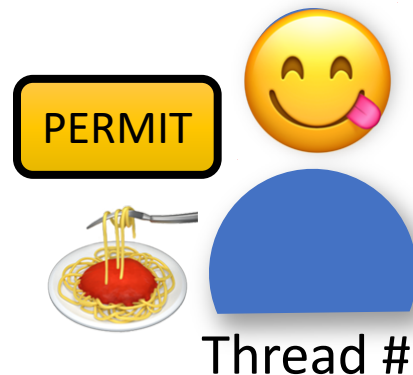
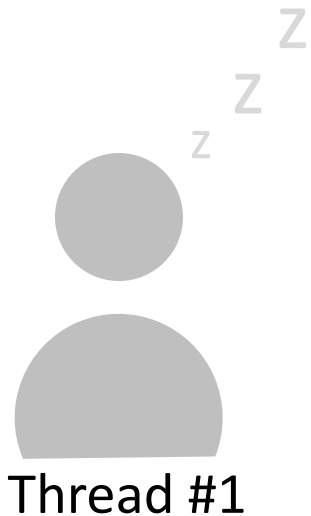


Thread #3

# waitForPermission Over-permitting

```
static void waitForPermission(size_t& permits, condition_variable_any& permitsCV,  
mutex& permitsLock) {  
    permitsLock.lock();  
    if (permits == 0) {  
        permitsCV.wait(permitsLock);  
    }  
    permits--;  
    permitsLock.unlock();  
}
```

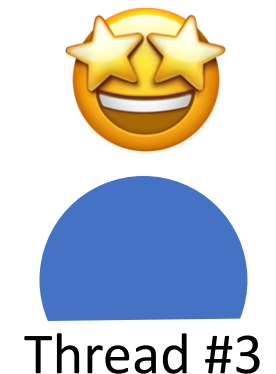
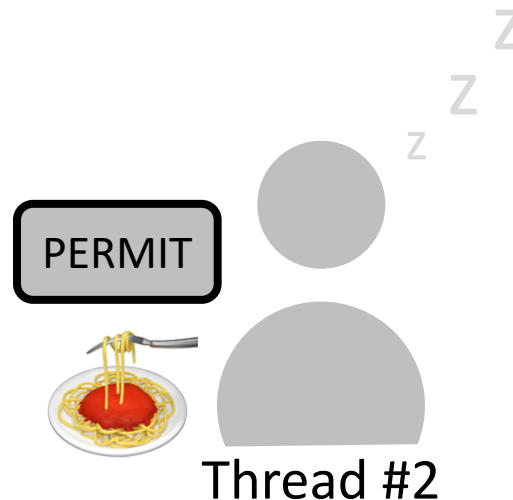
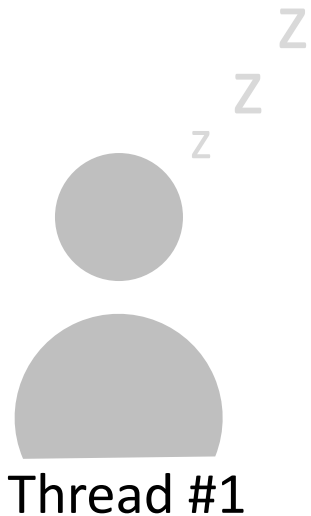
permits = 0



# waitForPermission Over-permitting

```
static void waitForPermission(size_t& permits, condition_variable_any& permitsCV,  
mutex& permitsLock) {  
    permitsLock.lock();  
    if (permits == 0) {  
        permitsCV.wait(permitsLock);  
    }  
    permits--;  
    permitsLock.unlock();  
}
```

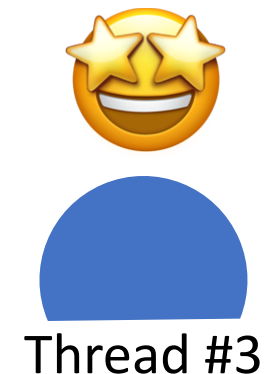
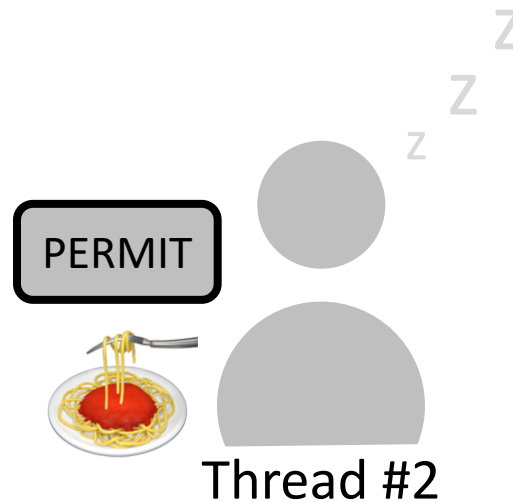
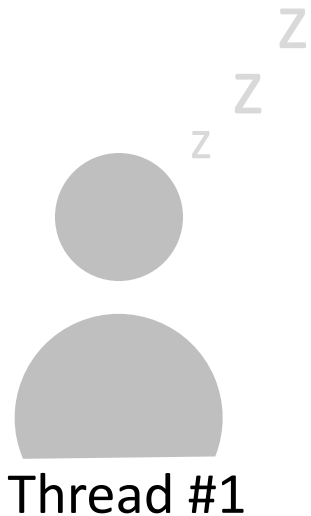
**permits = 0**



# waitForPermission Over-permitting

```
static void waitForPermission(size_t& permits, condition_variable_any& permitsCV,  
mutex& permitsLock) {  
    permitsLock.lock();  
    if (permits == 0) {  
        permitsCV.wait(permitsLock);  
    }  
    permits--;  
    permitsLock.unlock();  
}
```

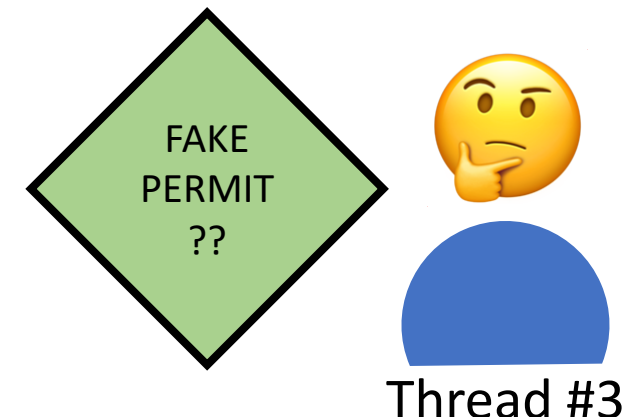
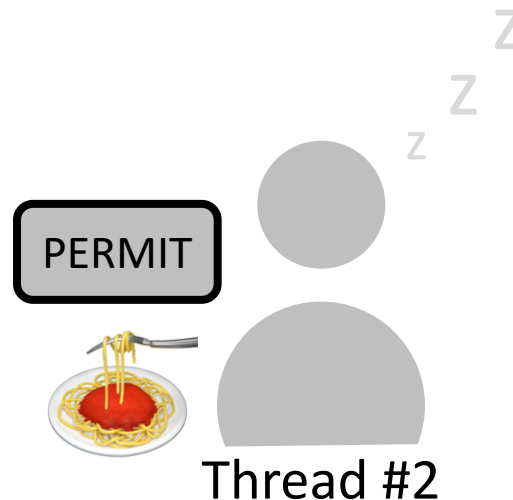
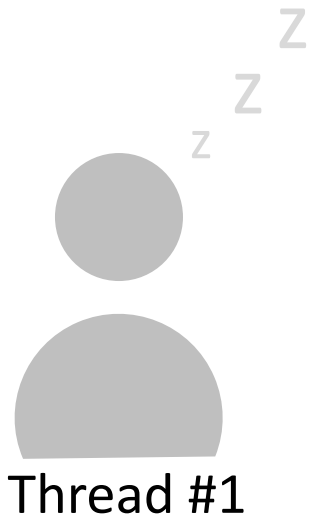
**permits = 0**



# waitForPermission Over-permitting

```
static void waitForPermission(size_t& permits, condition_variable_any& permitsCV,  
mutex& permitsLock) {  
    permitsLock.lock();  
    if (permits == 0) {  
        permitsCV.wait(permitsLock);  
    }  
    permits--;  
    permitsLock.unlock();  
}
```

**permits = <very large number>**





# waitForPermission (In progress)

```
static void waitForPermission(size_t& permits, condition_variable_any& permitsCV,
mutex& permitsLock) {
    permitsLock.lock();
    if (permits == 0) {
        permitsCV.wait(permitsLock);
    }
    permits--;
    permitsLock.unlock();
}
```

**Key Problem:** if multiple threads are woken up for one new permit, it's possible that some of them may have to continue waiting for a permit.

**Solution:** we must call *wait()* in a loop, in case we must call it again to wait longer.

# waitForPermission (Final version)

```
static void waitForPermission(size_t& permits, condition_variable_any& permitsCV,
mutex& permitsLock) {
    permitsLock.lock();
    while (permits == 0) {
        permitsCV.wait(permitsLock);
    }
    permits--;
    permitsLock.unlock();
}
```

**Key Problem:** if multiple threads are woken up for one new permit, it's possible that some of them may have to continue waiting for a permit.

**Solution:** we must call *wait()* in a loop, in case we must call it again to wait longer.



[dining-philosophers-with-cv-wait.cc](#)

# Spurious Wakeups

```
static void waitForPermission(size_t& permits, condition_variable_any& permitsCV,  
mutex& permitsLock) {  
    permitsLock.lock();  
    while (permits == 0) {  
        permitsCV.wait(permitsLock);  
    }  
    permits--;  
    permitsLock.unlock();  
}
```

It turns out that in addition to this reason, condition variables can have *spurious wakeups* – they wake us up even when not being notified by another thread! Thus, we should *always* wrap calls to **wait** in a while loop.

# Condition Variable Key Takeaways

A **condition variable** is a variable that can be shared across threads and used for one thread to notify other threads when something happens. Conversely, a thread can also use this to wait until it is notified by another thread.

- We can call ***wait(lock)*** to sleep until another thread signals this condition variable. The condition variable will unlock and re-lock the specified lock for us.
  - This is necessary because we must give up the lock while waiting so another thread may return a permit, but if we unlock before waiting, there is a race condition.
- We can call ***notify\_all()*** to send a signal to waiting threads and wake them up.
- We call ***wait(lock)*** in a loop in case we are woken up but must wait longer
  - This could happen if multiple threads are woken up for a single new permit, or because of spurious wakeups.

# Condition Variables

1. Identify a single kind of event that we need to wait / notify for
2. Ensure there is proper state to check if the event has happened
3. Create a condition variable and share it among all threads either waiting for that event to happen or triggering that event
4. Identify who will notify that this happens, and have them notify via the condition variable
5. Identify who will wait for this to happen, and have them wait via the condition variable

# Plan For Today

- **Recap and continuing:** condition variables and dining philosophers
- **Unique locks**
- Monitor pattern
- **Example:** Bridge Crossing

```
cp -r /afs/ir/class/cs111/lecture-code/lect15 .
```

# Unique Locks

- It is common to acquire a lock and hold onto it until the end of some scope (e.g. end of function, end of loop, etc.).
- There is a convenient variable type called *unique\_lock* that when created can automatically lock a mutex, and when destroyed (e.g. when it goes out of scope) can automatically unlock a mutex.
- Particularly useful if you have many paths to exit a function and you must unlock in all paths.

# grantPermission

We lock at the beginning of this function and unlock at the end.

```
static void grantPermission(size_t& permits,  
condition_variable_any& permitsCV, mutex& permitsLock) {  
    permitsLock.lock();  
    permits++;  
    if (permits == 1) permitsCV.notify_all();  
    permitsLock.unlock();  
}
```



# grantPermission

We lock at the beginning of this function and unlock at the end.

```
static void grantPermission(size_t& permits,  
condition_variable_any& permitsCV, mutex& permitsLock) {  
    unique_lock<mutex> uniqueLock(permitsLock);  
    permits++;  
    if (permits == 1) permitsCV.notify_all();  
}
```

Auto-locks permitsLock here



# grantPermission

We lock at the beginning of this function and unlock at the end.

```
static void grantPermission(size_t& permits,  
condition_variable_any& permitsCV, mutex& permitsLock) {  
    unique_lock<mutex> uniqueLock(permitsLock);  
    permits++;  
    if (permits == 1) permitsCV.notify_all();  
}
```



Auto-unlocks permitsLock  
here (goes out of scope)

# waitForPermission

```
static void waitForPermission(size_t& permits,  
condition_variable_any& permitsCV, mutex& permitsLock) {  
    permitsLock.lock();  
    while (permits == 0) {  
        permitsCV.wait(permitsLock);  
    }  
    permits--;  
    permitsLock.unlock();  
}
```

# waitForPermission

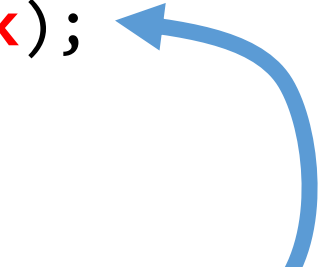
```
static void waitForPermission(size_t& permits,  
condition_variable_any& permitsCV, mutex& permitsLock) {  
    unique_lock<mutex> uniqueLock(permitsLock);  
    while (permits == 0) {  
        permitsCV.wait(uniqueLock);  
    }  
    permits--;  
}
```

Auto-locks permitsLock here



# waitForPermission

```
static void waitForPermission(size_t& permits,  
condition_variable_any& permitsCV, mutex& permitsLock) {  
    unique_lock<mutex> uniqueLock(permitsLock);  
    while (permits == 0) {  
        permitsCV.wait(uniqueLock);  
    }  
    permits--;  
}
```



Use it with CV instead of original lock (it has wrapper methods for manually locking/unlocking!)

# waitForPermission

```
static void waitForPermission(size_t& permits,  
condition_variable_any& permitsCV, mutex& permitsLock) {  
    unique_lock<mutex> uniqueLock(permitsLock);  
    while (permits == 0) {  
        permitsCV.wait(uniqueLock);  
    }  
    permits--;  
}
```



Auto-unlocks permitsLock  
here (goes out of scope)

# Plan For Today

- **Recap and continuing:** condition variables and dining philosophers
- Unique locks
- **Monitor pattern**
- **Example:** Bridge Crossing

```
cp -r /afs/ir/class/cs111/lecture-code/lect15 .
```

# Multithreading Patterns

- Writing synchronization code is *hard* – difficult to reason about, bugs are tricky if they are hard to reproduce
- E.g. how many locks should we use for a given program?
  - Just one? Doesn't allow for much concurrency
  - One lock per shared variable? Very hard to manage, gets complex, inefficient
- Like with dining philosophers, we must consider many scenarios and have lots of state to track and manage
- **One design idea to help:** the “monitor” design pattern - associate a single lock with a collection of related variables, e.g. a **class**



# Monitor Design Pattern

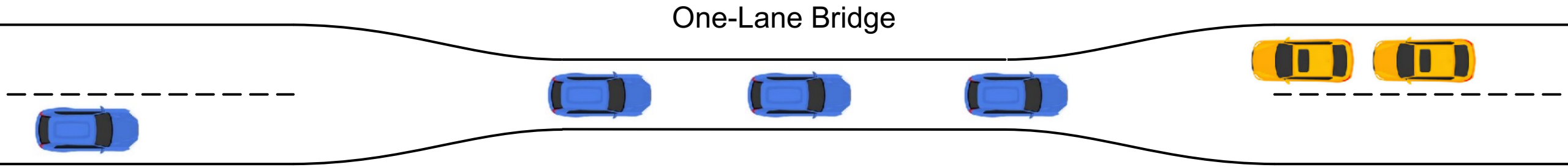
- For a multithreaded program, we can define a class that encapsulates the key multithreading logic and make an instance of it in our program.
- This class will have 1 mutex instance variable, and in all its methods we'll lock and unlock it as needed when accessing our shared state, so multiple threads can call the methods
- We can add any other state or condition variables we need as well – but the key idea is there is **one mutex** protecting access to all shared state, and which is locked/unlocked in the class methods that use the shared state.

# Plan For Today

- **Recap and continuing:** condition variables and dining philosophers
- Unique locks
- Monitor pattern
- **Example: Bridge Crossing**

```
cp -r /afs/ir/class/cs111/lecture-code/lect15 .
```

# Bridge Crossing



Let's write a program that simulates cars crossing a one-lane bridge.

- We will have each car represented by a thread, and they must coordinate as though they all need to cross the bridge.
- A car can be going either east or west
- All cars on bridge must be travelling in the same direction
- Any number of cars can be on the bridge at once
- A car from the other direction can only go once the coast is clear

# Bridge Crossing

```
int main(int argc, const char *argv[]) {
    Bridge bridge;
    thread cars[kNumCars];
    for (size_t i = 0; i < kNumCars; i++) {
        if (flipCoin()) {
            cars[i] = thread(crossBridgeEast, i, ref(bridge));
        } else {
            cars[i] = thread(crossBridgeWest, i, ref(bridge));
        }
    }
    for (thread& car : cars) car.join();
    return 0;
}
```

# Bridge Crossing

```
int main(int argc, const char *argv[]) {  
    Bridge bridge;  
    thread cars[kNumCars];  
    for (size_t i = 0; i < kNumCars; i++) {  
        if (flipCoin()) {  
            cars[i] = thread(crossBridgeEast, i, ref(bridge));  
        } else {  
            cars[i] = thread(crossBridgeWest, i, ref(bridge));  
        }  
    }  
    for (thread& ca  
    return 0;  
}
```

Wouldn't it be cool if, instead of making all these CVs/locks/etc and managing them directly in our program, we had a variable type that would manage them internally?

# Bridge Crossing

```
int main(int argc, const char *argv[]) {  
    Bridge bridge;  
    thread cars[kNumCars];  
    for (size_t i = 0; i < kNumCars; i++) {  
        if (flipCoin()) {  
            cars[i] =  
        } else {  
            cars[i] =  
        }  
    }  
    for (thread& car  
    return 0;  
}
```

Imagine a variable type **Bridge** that you could have manage the following:

- “I need to cross!” – would block for you until you’re able to cross in a given direction.
- “I’m done crossing!” – would automatically manage things to potentially allow cars going the other direction to proceed.

# Bridge Crossing

Each car thread would run a function like this – the concurrency is managed internally inside the bridge variable!

```
static void crossBridgeEast(size_t id, Bridge& bridge) {  
    approachBridge(); // sleep  
    bridge.arrive_eastbound(id);  
    driveAcross(); // sleep  
    bridge.leave_eastbound(id);  
}
```

# Bridge Crossing

Each car thread would run a function like this – the concurrency is managed internally inside the bridge variable!

```
static void crossBridgeWest(size_t id, Bridge& bridge) {  
    approachBridge(); // sleep  
    bridge.arrive_westbound(id);  
    driveAcross(); // sleep  
    bridge.leave_westbound(id);  
}
```

arrive\_westbound/eastbound would need to know if we are able to cross, and either return immediately or block.



# Bridge Crossing

Each car thread would run a function like this – the concurrency is managed internally inside the bridge variable!

```
static void crossBridgeWest(size_t id, Bridge& bridge) {  
    approachBridge(); // sleep  
    bridge.arrive_westbound(id);  
    driveAcross(); // sleep  
    bridge.leave_westbound(id);  
}
```

leave\_westbound/eastbound would need to be able to communicate with other threads who are waiting to cross in the other direction.

# Demo: Starter Code

[car-simulation.cc](http://car-simulation.cc) and [bridge.hh/bridge.cc](http://bridge.hh/bridge.cc)

# Arriving Eastbound

**arrive\_eastbound** needs to wait for it to be clear for the car to cross, and then let it cross.

- If other cars are already crossing eastbound, they can go
- If other cars are already crossing *westbound*, we have to wait

**“Waiting for an event to happen” -> condition variable!**

# Condition Variables

1. Identify a single kind of event that we need to wait / notify for
2. Ensure there is proper state to check if the event has happened
3. Create a condition variable and share it among all threads either waiting for that event to happen or triggering that event
4. Identify who will notify that this happens, and have them notify via the condition variable
5. Identify who will wait for this to happen, and have them wait via the condition variable

# Arriving Westbound

**arrive\_westbound** needs to wait for it to be clear for the car to cross, and then let it cross.

- If other cars are already crossing westbound, they can go
- If other cars are already crossing *eastbound*, we have to wait

**“Waiting for an event to happen” -> condition variable!**

# Plan For Today

- **Recap and continuing:** condition variables and dining philosophers
- Unique locks
- Monitor pattern
- **Example:** Bridge Crossing

**Lecture 15 takeaway:** The monitor pattern combines procedures and state into a class for easier management of synchronization. Then threads can call its thread-safe methods!

```
cp -r /afs/ir/class/cs111/lecture-code/lect15 .
```