CS111, Lecture 14
Condition Variables

Optional reading:

Operating Systems: Principles and Practice (2nd Edition): Sections 5.2-5.4 and Section 6.5

CS111 Topic 3: Multithreading, Part 1

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

Multithreading Introduction	Race conditions and locks	Condition Variables	Multithreading Patterns
Lecture 12	**Last Lecture**	**This Lecture**	**Lecture 15**

assign4: implement several multithreaded programs while eliminating race conditions!

Learning Goals

- Learn about ways to add constraints to our programs to prevent deadlock
- Learn how condition variables can let threads signal to each other and wait for conditions to become true

Plan For Today

- **Recap:** mutexes and dining philosophers
- Encoding resource constraints
- Condition Variables

```
cp -r /afs/ir/class/cs111/lecture-code/lect14 .
```

Plan For Today

- **Recap: mutexes and dining philosophers**
- Encoding resource constraints
- Condition Variables

```
cp -r /afs/ir/class/cs111/lecture-code/lect14 .
```

Mutexes

A **mutex** (”mutual exclusion”) is a variable type that lets us enforce the pattern of only 1 thread having access to something at a time.

- You make a mutex for each distinct thing you need to limit access to.
- You call **lock()** on the mutex to attempt to take the lock
- You call **unlock()** on the mutex when you are done to give the lock back
- A way to add a *constraint* to your program: “only one thread may access or execute this at a time”.

Ticket Agents

```
static void sellTickets(size_t id, size_t& remainingTickets, mutex&
counterLock) {
    while (true) {
        counterLock.lock();  // only 1 thread can proceed at a time
        if (remainingTickets == 0) {
            counterLock.unlock(); // must give up lock before exiting
            break;
        }
        size_t myTicket = remainingTickets;
        remainingTickets--;
        counterLock.unlock(); // once thread passes here, another can go
        sleep_for(500); // simulate "selling a ticket"
        ...
```

Deadlock

Deadlock occurs when multiple threads are all blocked, waiting on a resource owned by one of the other threads. None can make progress! Example:

Thread A	Thread B
`mutex1.lock();`	`mutex2.lock();`
`mutex2.lock();`	`mutex1.lock();`
`...`	`...`

E.g. if thread A executes 1 line, then thread B executes 1 line, deadlock!

One prevention technique - prevent circularities: all threads request resources in the same order (e.g., always lock mutex1 before mutex2.)

Another – limit number of threads competing for a shared resource

Deadlock Example: Dining Philosophers Simulation

- Five philosophers sit around a **circular table**, eating spaghetti
- There is **one fork** for each of them
- Each philosopher **thinks, then eats**, and repeats this **three times** for their three daily meals.
- **To eat**, a philosopher must grab the fork on their left *and* the fork on their right. Then they chow on spaghetti to nourish their big, philosophizing brain.
- When they're full, they put down the forks in the same order they picked them up and return to thinking for a while.
- **To think**, a philosopher keeps to themselves for some amount of time. Sometimes they think for a long time, and sometimes they barely think at all.

Dining Philosophers

eat is modeled as grabbing the two forks, sleeping for some amount of time, and putting the forks down.

```
static void eat(size_t id, mutex& left, mutex& right) {
    left.lock();
    right.lock();
    cout << oslock << id << " starts eating om nom nom
nom." << endl << osu
    sleep_for(getEat
    cout << oslock <
         << osunlock
    left.unlock();
    right.unlock();
}
```

Spoiler: there is a race condition here that leads to **deadlock – **deadlock occurs when multiple threads are all blocked, waiting on a resource owned by one of the other blocked threads. When could this happen?

Food For Thought

What if: all philosophers grab their left fork and then go off the CPU?

- Deadlock! All philosophers will wait on their right fork, which will never become available
- **Testing our hypothesis**: insert a **sleep_for** call in between grabbing the two forks
- We should be able to insert a **sleep_for** call anywhere in a thread routine and have no concurrency issues.
- We (incorrectly) assumed that at least one philosopher is always able to pick up both of their forks. How can we fix this?

dining-philosophers-with-deadlock.cc

Plan For Today

- **Recap:** mutexes and dining philosophers
- **Encoding resource constraints**
- Condition Variables

```
cp -r /afs/ir/class/cs111/lecture-code/lect14 .
```

Encoding Resource Constraints

Goal: we must encode resource constraints into our program.

Example: how many philosophers can *try* to eat at the same time? **Four**.

- *Alternatively:* how many philosophers can *eat* at the same time? **Two**.
- Why might the first one be better? Imposes less bottlenecking while still solving the issue.

How can we encode this into our program?

Have a counter of "permits". Initially 4. A philosopher must have a permit (decrement counter or wait) to try to eat. Once done eating, a philosopher returns its permit (increment counter).

Tickets, Please...

```
int main(int argc, const char *argv[]) {
    mutex forks[kNumForks];

    size_t permits = kNumForks - 1;
    mutex permitsLock;

    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(permitsLock));
    }
    for (thread& p: philosophers) p.join();
    return 0;
}
```

Tickets, Please...

A philosopher thinks and eats, and repeats this 3 times.

```
static void philosopher(size_t id, mutex& left, mutex&
right, size_t& permits, mutex& permitsLock) {
    for (size_t i = 0; i < kNumMeals; i++) {
        think(id);
        eat(id, left, right, permits, permitsLock);
    }
}
```

Tickets, Please...

```
static void eat(size_t id, mutex& left, mutex& right,
size_t& permits, mutex& permitsLock) {

    waitForPermission(permits, permitsLock);
    left.lock();
    right.lock();
    cout << oslock << id << " starts eating om nom nom
nom." << endl << osunlock;
    sleep_for(getEatTime());
    cout << oslock << id << " all done eating." << endl
         << osunlock;
    grantPermission(permits, permitsLock);
    left.unlock();
    right.unlock();
}
```

grantPermission

To put a permit back, increment the counter by 1 and continue.

```
static void grantPermission(size_t& permits, mutex&
permitsLock) {
    permitsLock.lock();
    permits++;
    permitsLock.unlock();
}
```

waitForPermission

- If there are permits, decrement the counter by 1 and continue
- If there aren't permits, wait for a permit, then decrement by 1 and continue

```
static void waitForPermission(size_t& permits, mutex&
permitsLock) {
  while (true) {
    permitsLock.lock();
    if (permits > 0) break;
    permitsLock.unlock();
    // wait a little while (how??)
  }
  permits--;
  permitsLock.unlock();
}
```

waitForPermission

- If there are permits, decrement the counter by 1 and continue
- If there aren't permits, wait for a permit, then decrement by 1 and continue

```
static void waitForPermission(size_t& permits, mutex&
permitsLock) {
  while (true) {
    permitsLock.lock();
    if (permits > 0) break;
    permitsLock.unlock();
    sleep(??);
  }
  permits--;
  permitsLock.unlock();
}
```

This is called busy waiting (bad). We are unnecessarily and arbitrarily using CPU time to check when a permit is available.

It would be nice if someone could let us know when they return their permit. Then, we can sleep until this happens.

Plan For Today

- **Recap:** mutexes and dining philosophers
- Encoding resource constraints
- **Condition Variables**

```
cp -r /afs/ir/class/cs111/lecture-code/lect14 .
```

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 (no busy waiting).
- 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.

Key Idea: 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();
}
```

grantPermission

Other threads need the lock 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.

Key Idea: 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();
}
```

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!

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. Hints:

- As soon as we release a lock, another thread can use it
- if a thread isn't waiting on a CV, it won't get a notification from another thread

What ordering of events between threads could lead to deadlock here?

Nobody has responded yet.

Hang tight! Responses are coming in.

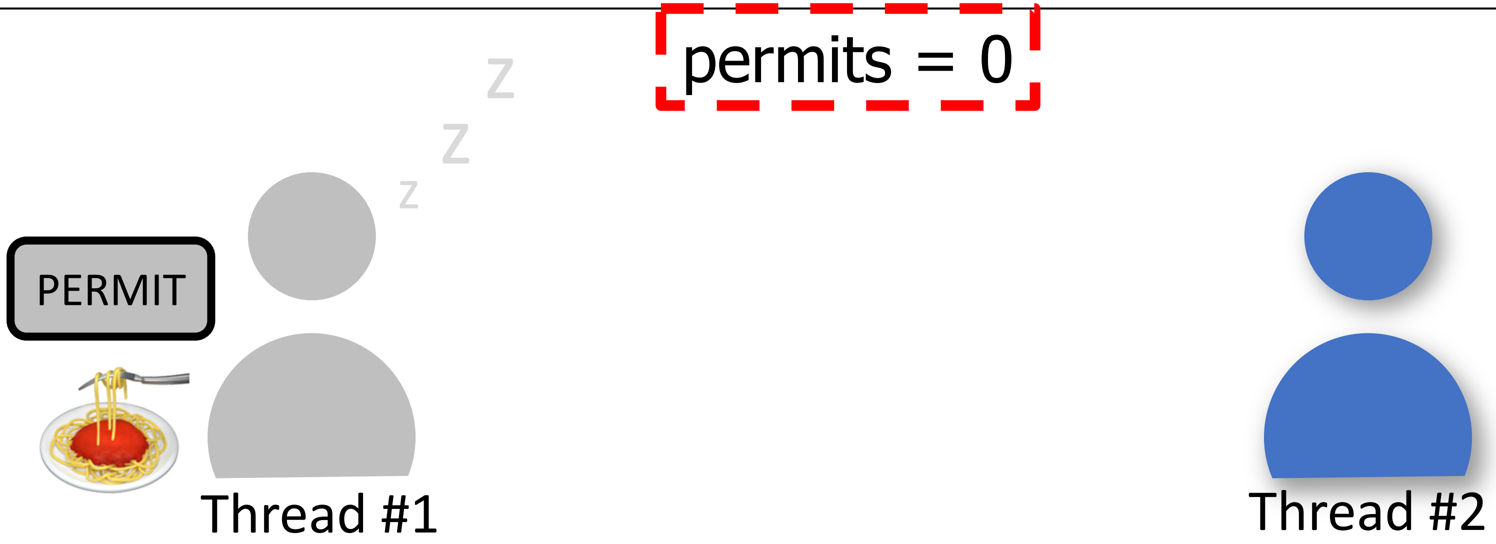
Deadlock: waitForPermission

```
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

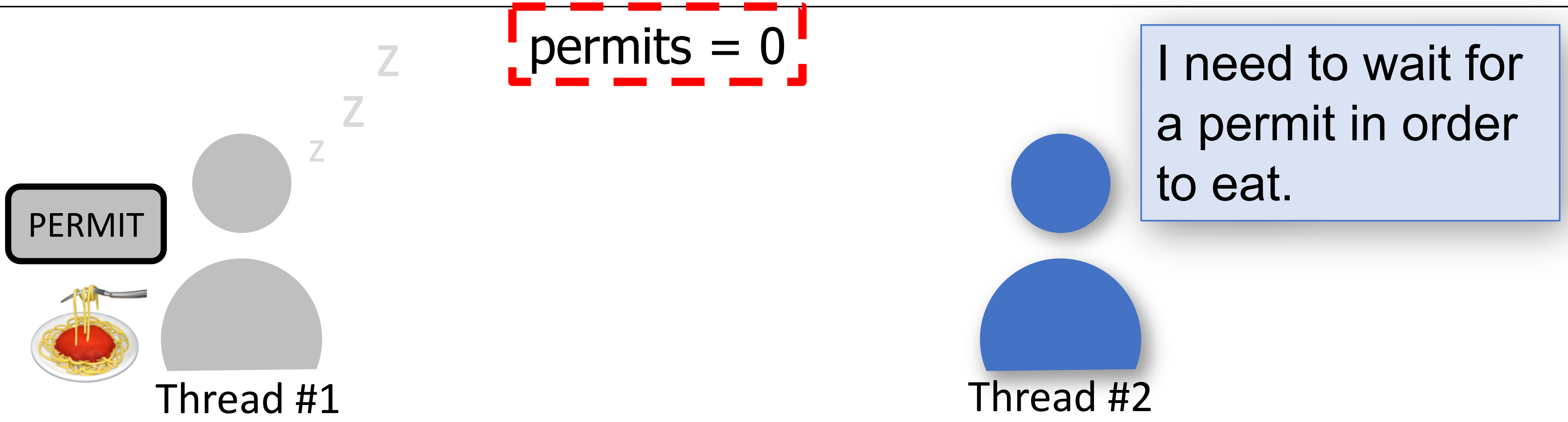
Deadlock: waitForPermission

```
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();
}
```



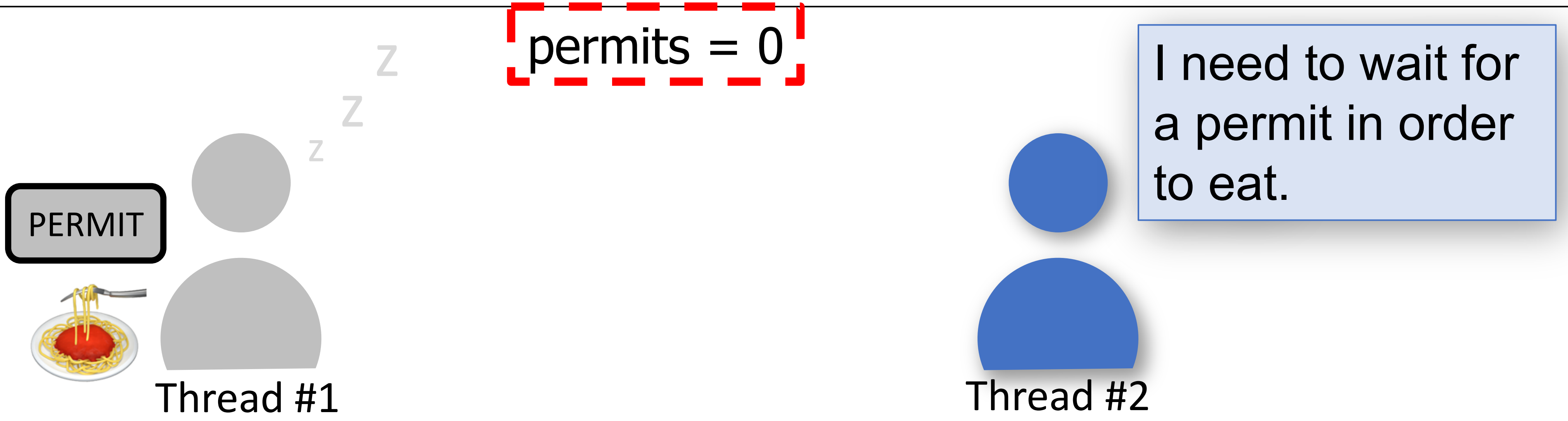
Deadlock: waitForPermission

```
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();
}
```

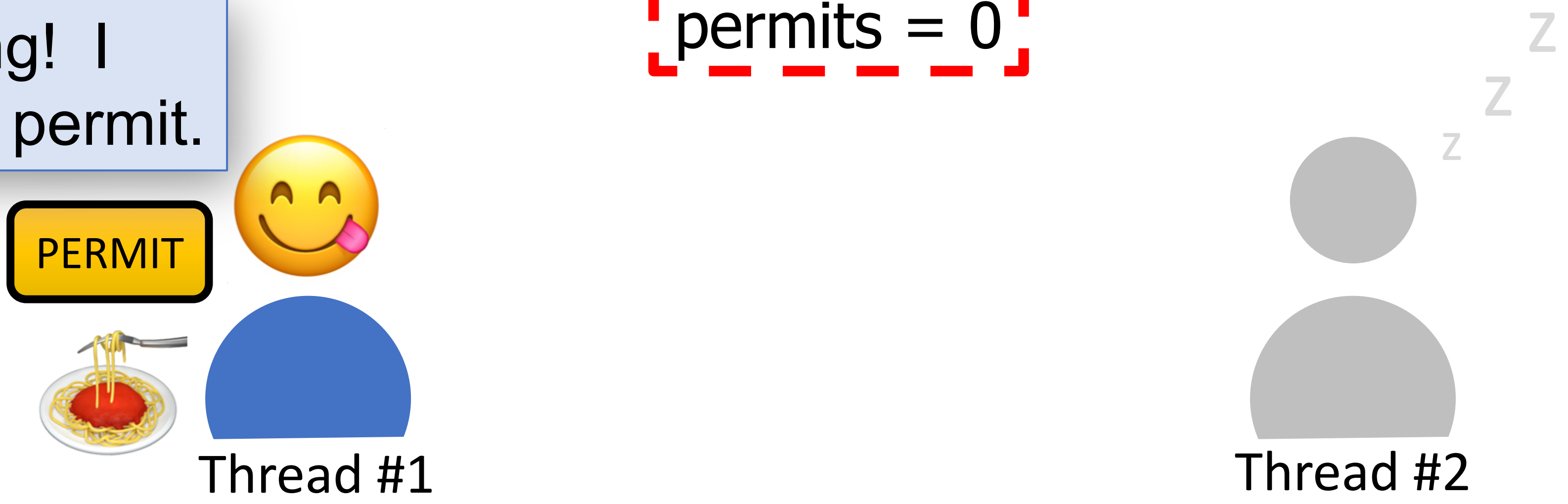


Deadlock: waitForPermission

```
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();
}
```

permits = 0

All done eating! I will return my permit.

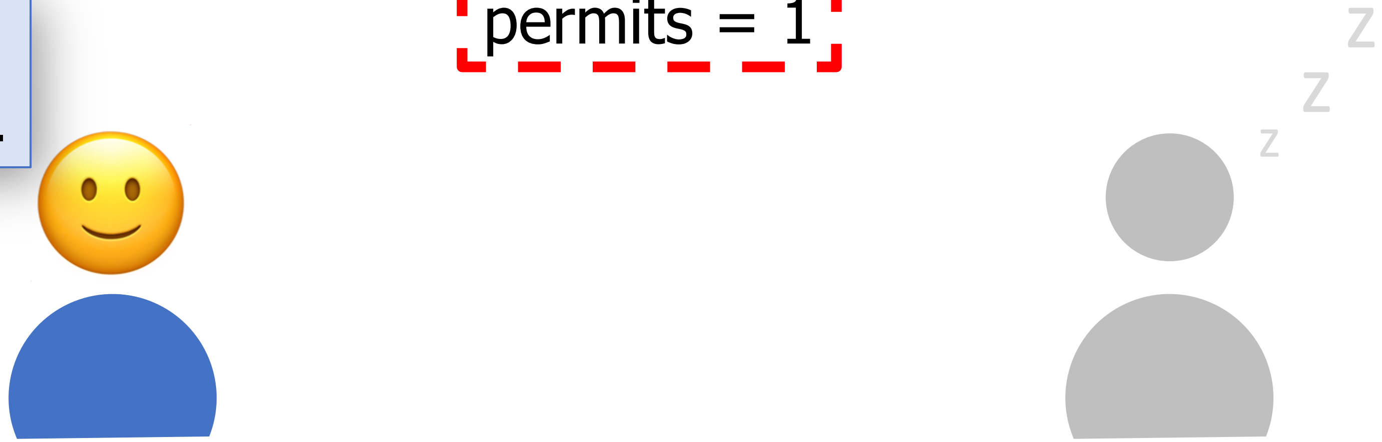


Deadlock: waitForPermission

```
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

All done eating! I will return my permit.



Thread #1

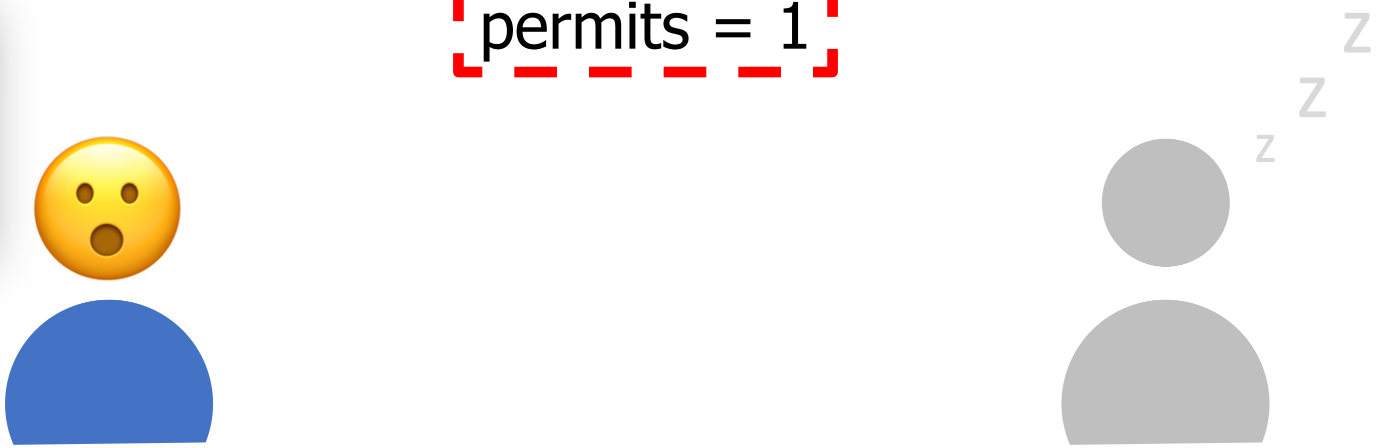
Thread #2

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();
}
```

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

permits = 1



Thread #1

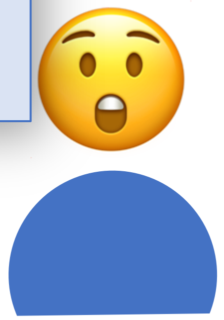
Thread #2

Deadlock: waitForPermission

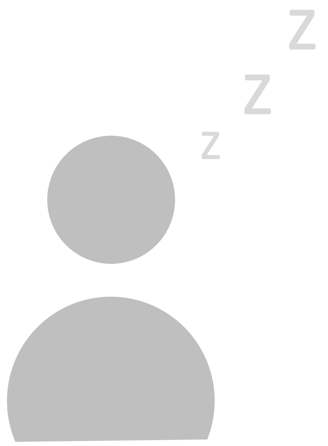
```
static void waitForPermission(size_t& permits, condition_variable_any& permitsCV,
mutex& permitsLock) {
    permitsLock.lock();
    if (permits == 0) {
        permitsLock.unlock();
        permitsCV.wait();     // (note: not final form of wait)
        permitsLock.lock();
    }
    permits--;
 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

permits = 1

"Attention waiting threads, a permit is available!"



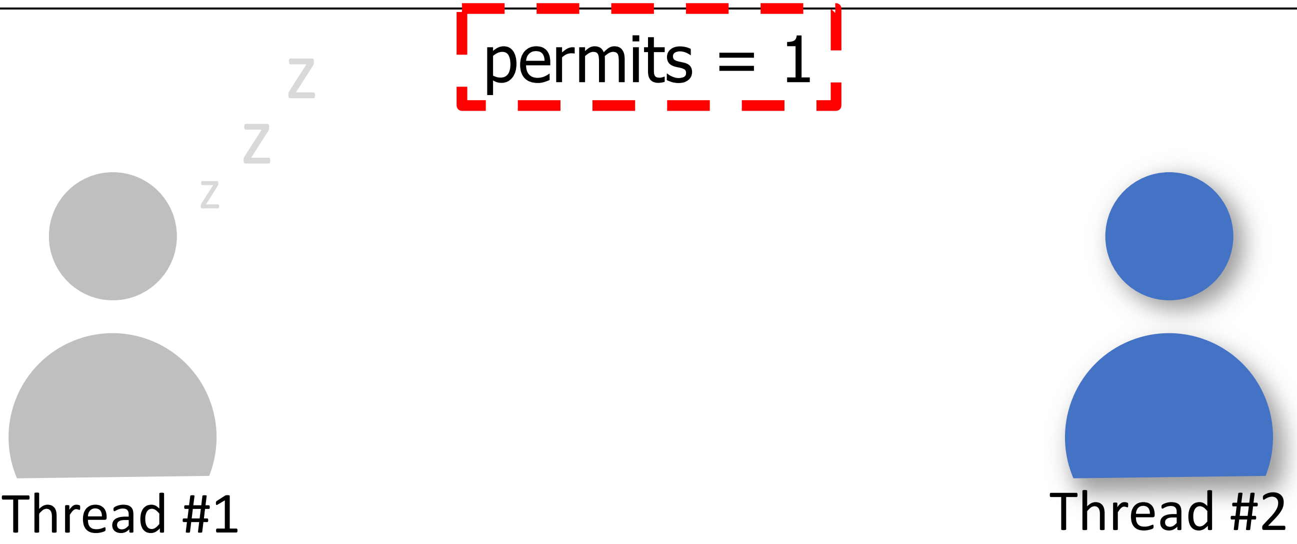
Thread #1



Thread #2

Deadlock: waitForPermission

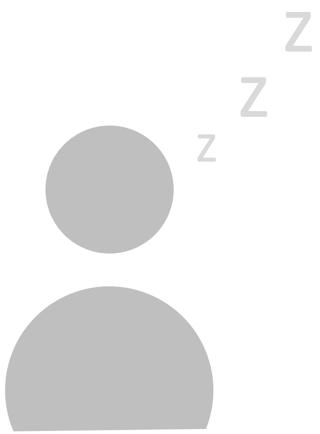
```
static void waitForPermission(size_t& permits, condition_variable_any& permitsCV,
mutex& permitsLock) {
    permitsLock.lock();
    if (permits == 0) {
        permitsLock.unlock();
        permitsCV.wait();      // (note: not final form of wait)
        permitsLock.lock();
    }
    permits--;
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

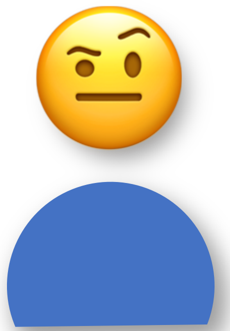


Deadlock: waitForPermission

```
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





100 years later

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();
        // AIR GAP HERE – someone could acquire the lock before we wait
        permitsCV.wait();     // (note: not final form of wait)
        permitsLock.lock();
    }
    ...
```

Key ideas:

- We must release the lock when waiting so someone else can put a permit back (which requires having the lock)
- But if we release the lock before calling wait, someone else could swoop in and put a permit back before we call wait(), meaning we will miss the notification! 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)

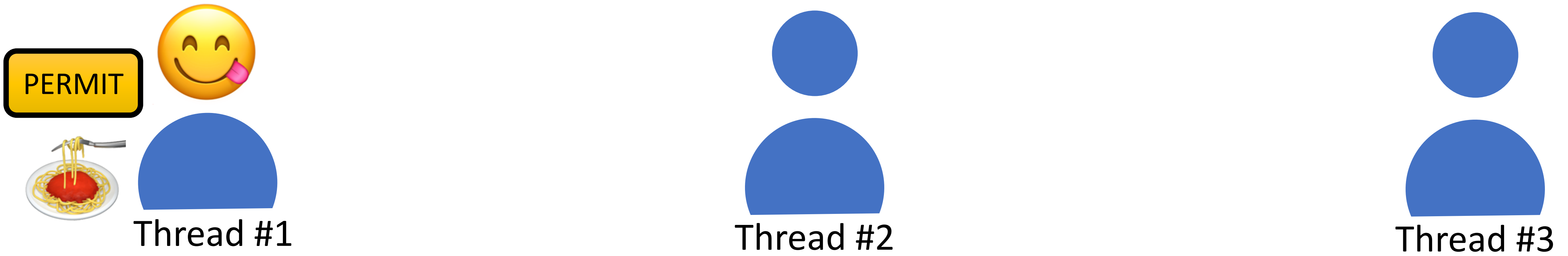
```
static void waitForPermission(size_t& permits, condition_variable_any& permitsCV,
mutex& permitsLock) {
    permitsLock.lock();
    if (permits == 0) {
        permitsCV.wait(permitsLock);
    }
    permits--;
    permitsLock.unlock();
}
```

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) and just 1 is returned.

waitForPermission Over-permitting

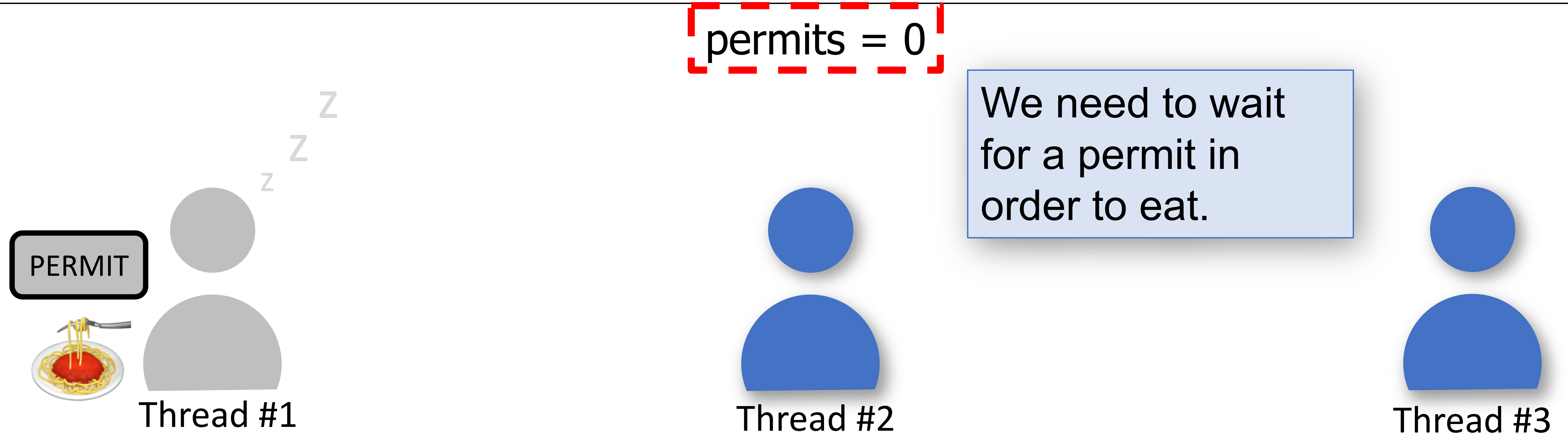
```
static void waitForPermission(size_t& permits, condition_variable_any& permitsCV,
mutex& permitsLock) {
    permitsLock.lock();
    if (permits == 0) {
        permitsCV.wait(permitsLock);
    }
    permits--;
    permitsLock.unlock();
}
```

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();
}
```

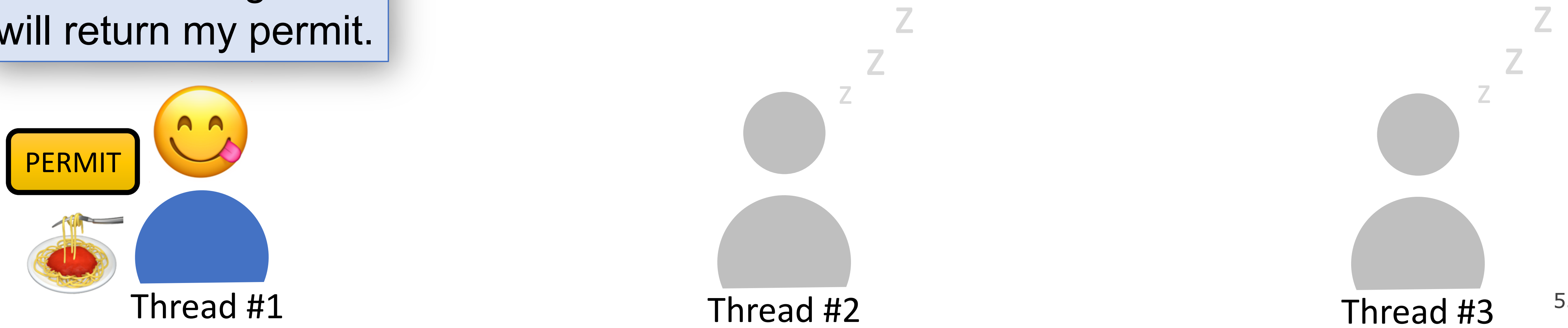


waitForPermission Over-permitting

```
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

All done eating! I will return my permit.

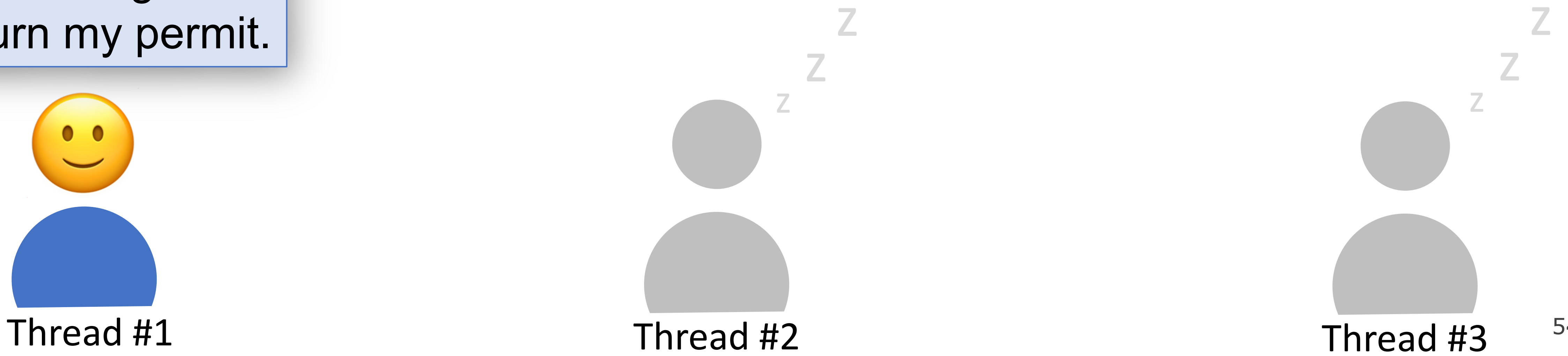


waitForPermission Over-permitting

```
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

All done eating! I will return my permit.

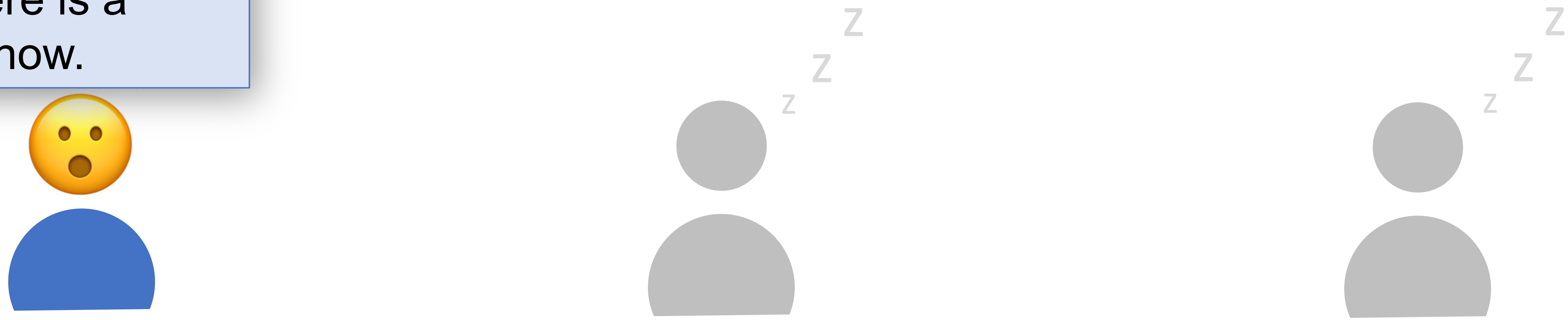


waitForPermission Over-permitting

```
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.

permits = 1



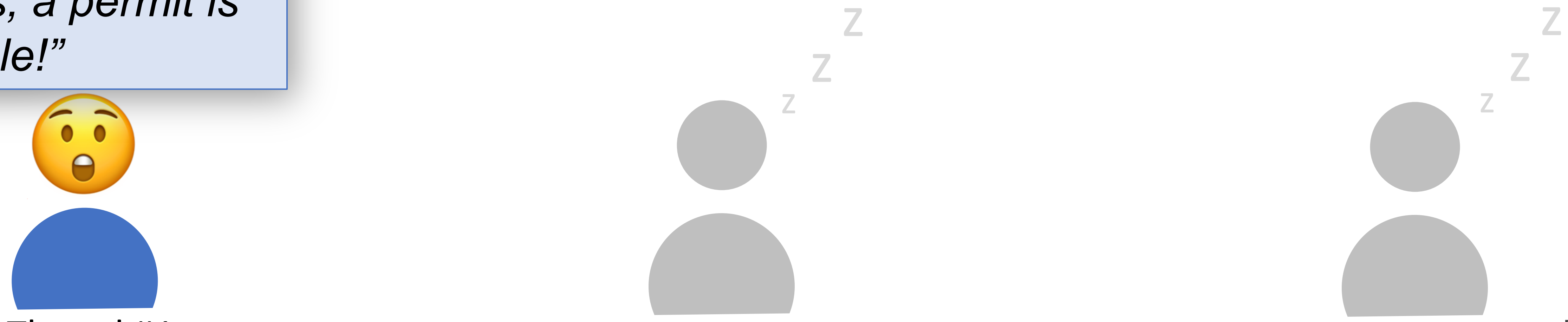
Thread #1 Thread #2 Thread #3

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();
}
```

"Attention waiting threads, a permit is available!"

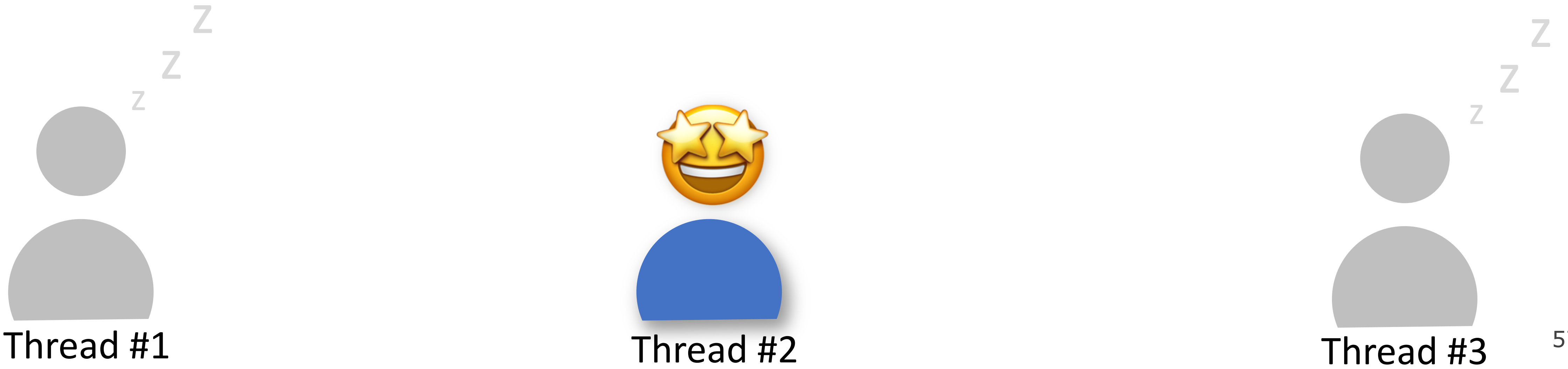
permits = 1



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();
}
```

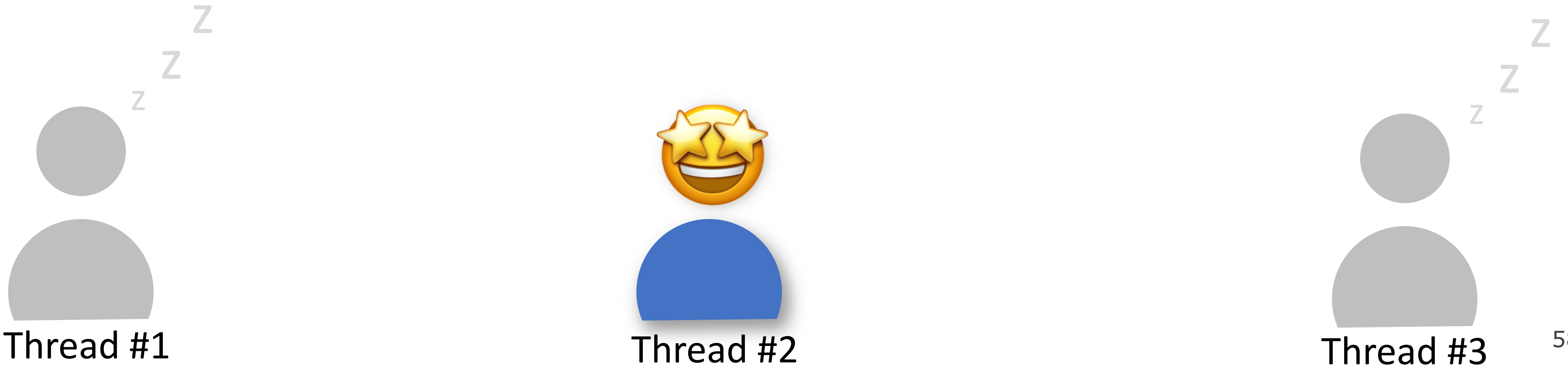
permits = 1



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();
}
```

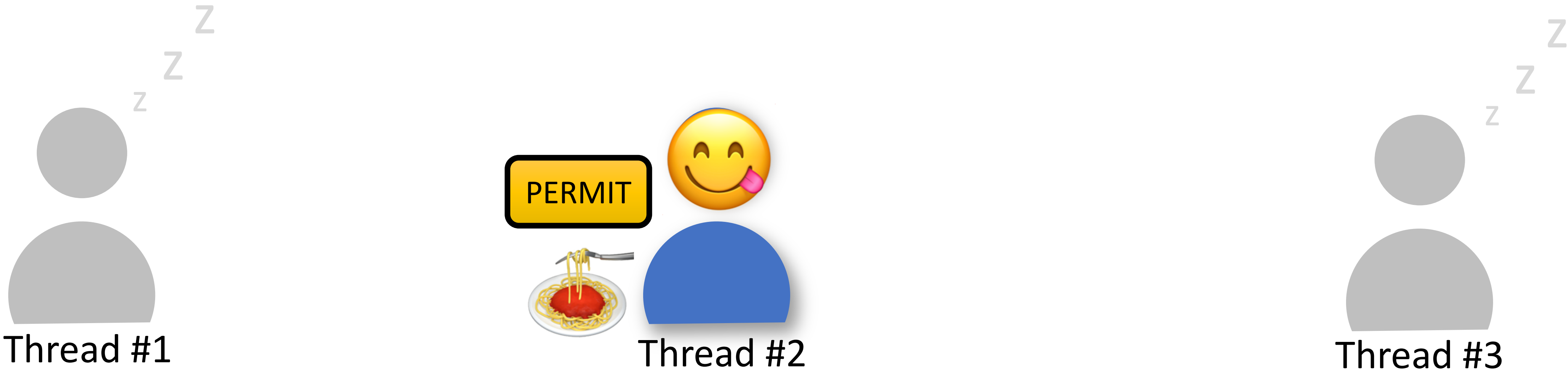
permits = 1



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();
}
```

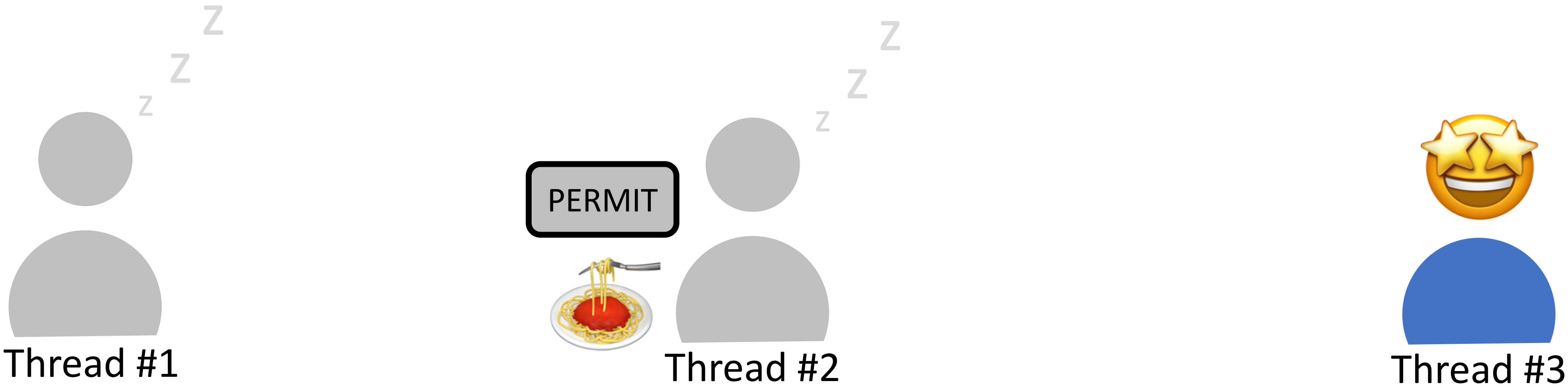
permits = 0



waitForPermission Over-permitting

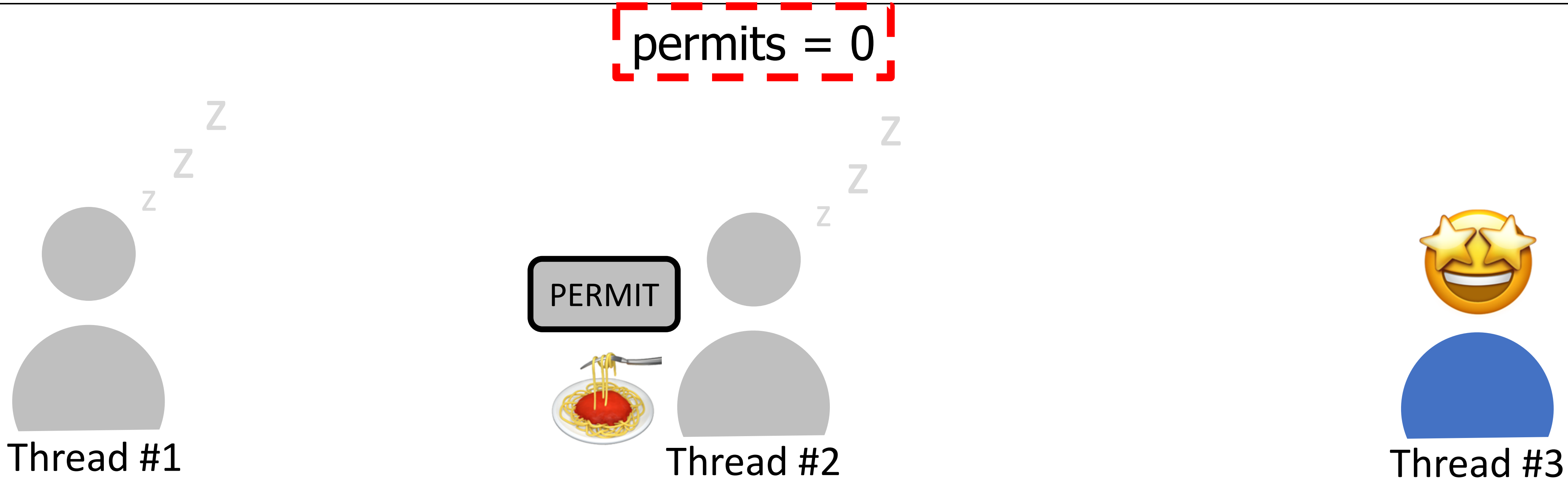
```
static void waitForPermission(size_t& permits, condition_variable_any& permitsCV,
mutex& permitsLock) {
    permitsLock.lock();
    if (permits == 0) {
        permitsCV.wait(permitsLock);
    }
    permits--;
    permitsLock.unlock();
}
```

permits = 0



waitForPermission Over-permitting

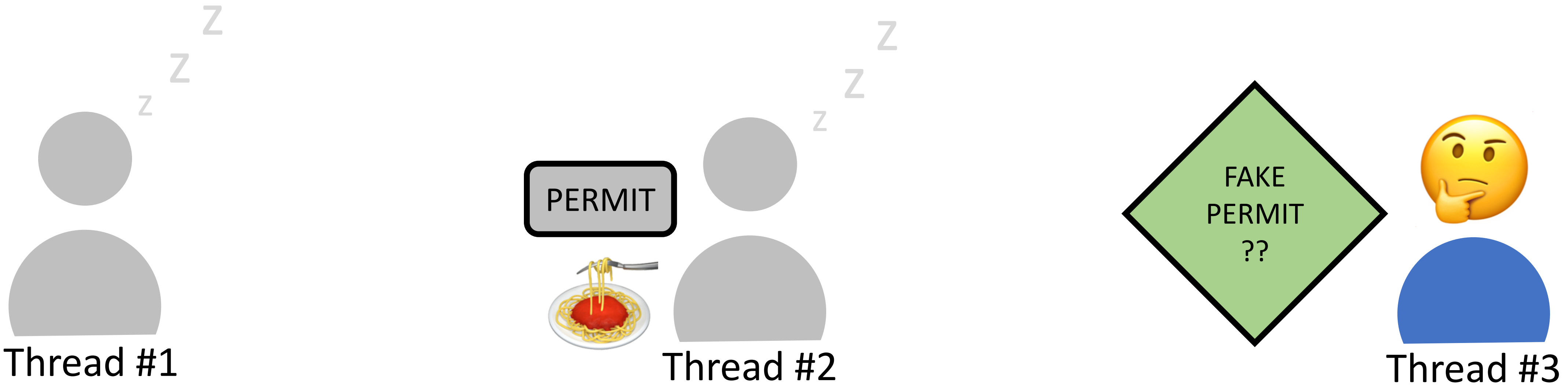
```
static void waitForPermission(size_t& permits, condition_variable_any& permitsCV,
mutex& permitsLock) {
    permitsLock.lock();
    if (permits == 0) {
        permitsCV.wait(permitsLock);
    }
    permits--;
    permitsLock.unlock();
}
```



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 Over-permitting

Key Idea: If we are waiting and then woken up by a notification, it's possible by the time we exit wait(), there are no permits, so we must wait again.

- Note: wait() reacquires the lock before returning

```
static void waitForPermission(size_t& permits, condition_variable_any& permitsCV,
mutex& permitsLock) {
    permitsLock.lock();
    if (permits == 0) {
      permitsCV.wait(permitsLock);
      // by the time we wake up here, all the permits could already be gone!
    }
    permits--;
    permitsLock.unlock();
}
```

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();
}
```

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 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.

- We can call **wait(*lock*)** to sleep (no busy waiting) 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

Recap

- **Recap:** mutexes and dining philosophers
- Encoding resource constraints
- Condition Variables

Lecture 14 takeaway: Condition variables let us wait on an event to occur and notify other threads that an event has occurred, all without busy waiting.

Next time: more about race conditions

```
cp -r /afs/ir/class/cs111/lecture-code/lect14 .
```