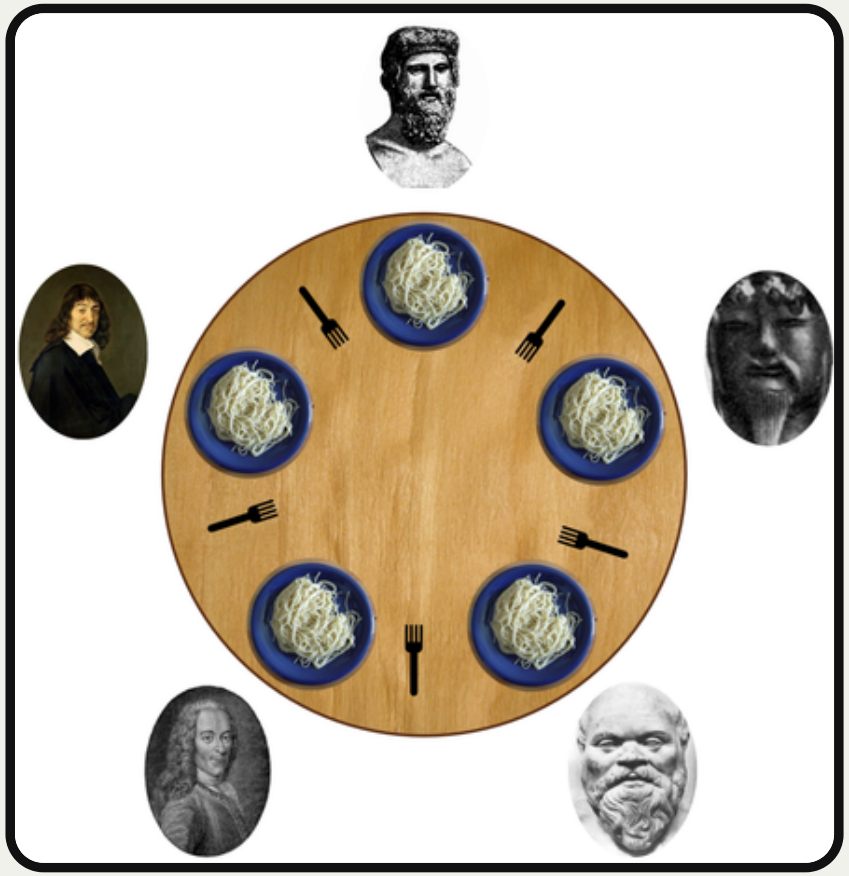
CS110 Lecture 16: Condition Variables and Semaphores

CS110: Principles of Computer Systems

Winter 2021-2022

Stanford University

Instructors: Nick Troccoli and Jerry Cain

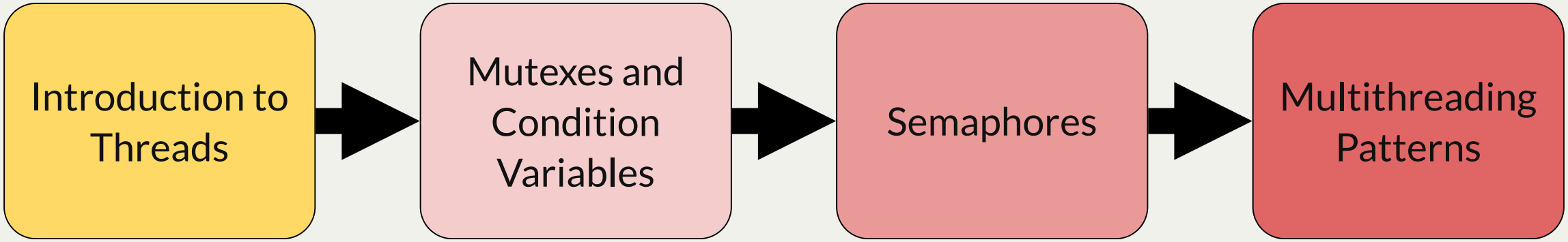


https://commons.wikimedia.org/wiki/File:An_illustration_of_the_dining_philosophers_problem.png

PDF of this presentation

CS110 Topic 3: How can we have concurrency within a single process?

Learning About Multithreading



Lecture 13 | **Lectures 14/15** | **This Lecture** | **Lectures 17/18**

assign5: implement your own multithreaded news aggregator to quickly fetch news from the web!

Learning Goals

- Learn how condition variables can let threads signal to each other
- Get practice with the "available permits" resource model
- Understand how semaphores combine a mutex, counter and condition variable

Plan For Today

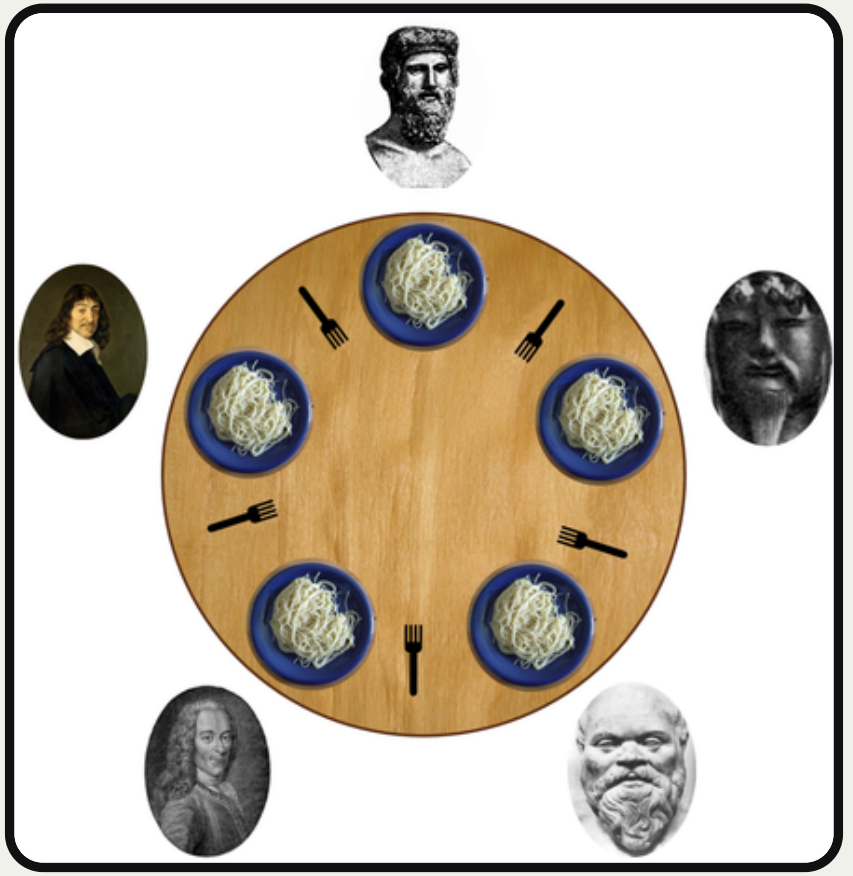
- **Recap:** Dining With Philosophers
- Condition Variables
- Semaphores

Plan For Today

- **Recap: Dining With Philosophers**
- Condition Variables
- Semaphores

Dining Philosophers Problem

- This is a canonical multithreading example of the potential for deadlock and how to avoid it.
- 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**, the a philosopher keeps to themselves for some amount of time. Sometimes they think for a long time, and sometimes they barely think at all.



https://commons.wikimedia.org/wiki/File:An_illustration_of_the_dining_philosophers_problem.png

`dining-philosophers-with-deadlock.cc`

Dining Philosophers Problem

Goal: we must encode resource constraints into our program.

Example: how many philosophers can hold the same fork at the same time? **One.**

How can we encode this into our program? Let's make a mutex for each fork.

- Each philosopher either holds a fork or doesn't.
- A philosopher grabs a fork by locking that mutex. If the fork is available, the philosopher continues. Otherwise, it blocks until the fork becomes available and it can have it.
- A philosopher puts down a fork by unlocking that mutex.

```
1  static void philosopher(size_t id, mutex& left, mutex& right) {
2    ...
3  }
4
5  int main(int argc, const char *argv[]) {
6    mutex forks[kNumForks];
7    thread philosophers[kNumPhilosophers];
8    for (size_t i = 0; i < kNumPhilosophers; i++) {
9      mutex& left = forks[i];
10     mutex& right = forks[(i + 1) % kNumPhilosophers];
11     philosophers[i] = thread(philosopher, i, ref(left), ref(right));
12   }
13   for (thread& p: philosophers) p.join();
14   return 0;
15 }
```

Dining Philosophers Problem

A philosopher thinks, then eats, and repeats this three times.

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

```
1 static void eat(size_t id, mutex& left, mutex& right) {
2   left.lock();
3   right.lock();
4   cout << oslock << id << " starts eating om nom nom nom." << endl << osunlock;
5   sleep_for(getEatTime());
6   cout << oslock << id << " all done eating." << endl << osunlock;
7   left.unlock();
8   right.unlock();
9 }
```

Spoiler: there is a race condition here that leads to deadlock.

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 on line 3, between getting left fork and right fork
- We should be able to insert a **sleep_for** call anywhere in a thread routine and have no concurrency issues.

```
1  static void eat(size_t id, mutex& left, mutex& right) {
2    left.lock();
3    sleep_for(5000);  // artificially force off the processor
4    right.lock();
5    cout << oslock << id << " starts eating om nom nom nom." << endl << osunlock;
6    sleep_for(getEatTime());
7    cout << oslock << id << " all done eating." << endl << osunlock;
8    left.unlock();
9    right.unlock();
10 }
```

Race Conditions and Deadlock

Goal: we must encode resource constraints into our program.

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

- *Alternative:* 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?

- let's add another shared variable representing a count of "permits" or "tickets" available.
- In order to try to eat (aka grab forks at all) a philosopher must get a permit
- Once done eating, a philosopher must return their permit

Tickets, Please...

- Let's add a new variable in main called **permits**, and a lock for it called **permitsLock**, so that we can update it without race conditions.
- We pass these to each philosopher by reference.

```
1  int main(int argc, const char *argv[]) {
2    // NEW
3    size_t permits = 4;
4    mutex permitsLock;
5
6    mutex forks[5];
7    thread philosophers[5];
8    for (size_t i = 0; i < 5; i++) {
9      mutex& left = forks[i];
10     mutex& right = forks[(i + 1) % 5];
11     philosophers[i] = thread(philosopher, i, ref(left), ref(right), ref(permits), ref(permitsLock));
12   }
13   for (thread& p: philosophers) p.join();
14   return 0;
15 }
```

dining-philosophers-with-busy-waiting.cc

Tickets, Please...

- The implementation of **eat** changes:
  - Before eating, the philosopher must get a permit
  - After eating, the philosopher must return their permit.

```
1  static void eat(size_t id, mutex& left, mutex& right, size_t& permits, mutex& permitsLock) {
2    // NEW
3    waitForPermission(permits, permitsLock);
4
5    left.lock();
6    right.lock();
7    cout << oslock << id << " starts eating om nom nom nom." << endl << osunlock;
8    sleep_for(getEatTime());
9    cout << oslock << id << " all done eating." << endl << osunlock;
10
11   // NEW
12   grantPermission(permits, permitsLock);
13
14   left.unlock();
15   right.unlock();
16 }
```

dining-philosophers-with-busy-waiting.cc

grantPermission

- How do we implement **grantPermission**?
- Recall: "To return a permit, increment by 1 and continue"

```
1 static void grantPermission(size_t& permits, mutex& permitsLock) {
2   permitsLock.lock();
3   permits++;
4   permitsLock.unlock();
5 }
```

dining-philosophers-with-busy-waiting.cc

waitForPermission

- How do we implement **waitForPermission**?
- Recall:
  - "If there are permits available (count > 0) then decrement by 1 and continue"
  - "If there are no permits available (count == 0) then block until a permit is available"

```
1  static void waitForPermission(size_t& permits, mutex& permitsLock) {
2    while (true) {
3      permitsLock.lock();
4      if (permits > 0) break;
5      permitsLock.unlock();
6      sleep_for(10);
7    }
8    permits--;
9    permitsLock.unlock();
10 }
```

Problem: this is busy waiting! 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**: Dining With Philosophers
- **Condition Variables**
- Semaphores

Condition Variables

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()** to sleep until another thread signals this condition variable.
- We can call **notify_all()** to send a signal to waiting threads.

```
condition_variable_any myConditionVariable;
...

// thread A waits until another thread signals
myConditionVariable.wait();

...

// thread B signals, waking up thread A
myConditionVariable.notify_all();
```

*not final prototype for **wait**

waitForPermission

How do we implement **waitForPermission?** Recall:

- "If there are permits available (count > 0) then decrement by 1 and continue"
- "If there are no permits available (count == 0) then block until a permit is available"

Idea:

- when someone returns a permit and it is the only one now available, notify all.
- if we need a permit but there are none available, wait.

Condition Variables

Now we must create a condition variable to pass by reference to all threads.

```
1  int main(int argc, const char *argv[]) {
2    mutex forks[kNumForks];
3
4    size_t permits = kNumForks - 1;
5    mutex permitsLock;
6
7    // NEW
8    condition_variable_any permitsCV;
9
10   thread philosophers[kNumPhilosophers];
11   for (size_t i = 0; i < kNumPhilosophers; i++) {
12     mutex& left = forks[i];
13     mutex& right = forks[(i + 1) % kNumForks];
14     philosophers[i] = thread(philosopher, i, ref(left), ref(right),
15       ref(permits), ref(permitsCV), ref(permitsLock));
16   }
17   for (thread& p: philosophers) p.join();
18   return 0;
19 }
```

dining-philosophers-with-cv-wait.cc

grantPermission

For **grantPermission**, we must signal when we make permits go from 0 to 1.

```
1 static void grantPermission(size_t& permits, condition_variable_any& permitsCV, mutex& permitsLock) {
2   permitsLock.lock();
3   permits++;
4   if (permits == 1) permitsCV.notify_all();
5   permitsLock.unlock();
6 }
```

dining-philosophers-with-cv-wait.cc

waitForPermission

For **waitForPermission**, if no permits are available we must wait until one becomes available.

```
1  static void waitForPermission(size_t& permits, condition_variable_any& permitsCV, mutex& permitsLock) {
2    permitsLock.lock();
3    if (permits == 0) {
4      permitsLock.unlock();
5      permitsCV.wait();
6      permitsLock.lock();
7    }
8    permits--;
9    permitsLock.unlock();
10 }
```

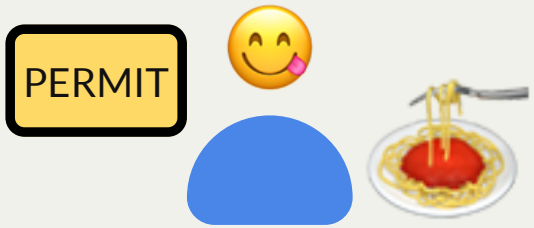
Key Idea: we must give up ownership of the lock when we wait, so that someone else can put a permit back.

Spoiler: there is a race condition here that could lead to deadlock. (**HINT:** notifications aren't queued)

waitForPermission

```
1  static void waitForPermission(size_t& permits, condition_variable_any& permitsCV, mutex& permitsLock) {
2    permitsLock.lock();
3    if (permits == 0) {
4      permitsLock.unlock();
5      permitsCV.wait();
6      permitsLock.lock();
7    }
8    permits--;
9    permitsLock.unlock();
10 }
```

permits = 0



thread #1

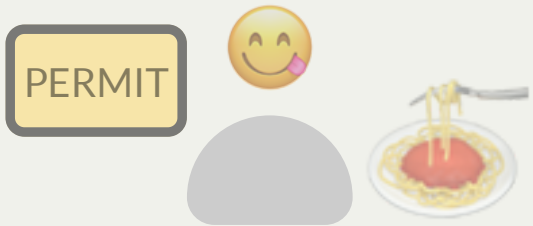
thread #2

waitForPermission

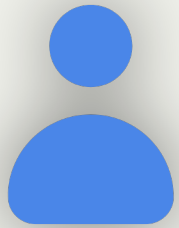
```
1  static void waitForPermission(size_t& permits, condition_variable_any& permitsCV, mutex& permitsLock) {
2    permitsLock.lock();
3    if (permits == 0) {
4      permitsLock.unlock();
5      permitsCV.wait();
6      permitsLock.lock();
7    }
8    permits--;
9    permitsLock.unlock();
10 }
```

permits = 0

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



thread #1



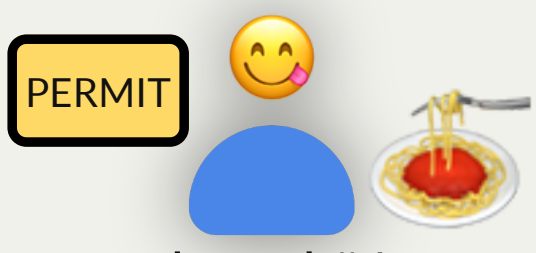
thread #2

waitForPermission

```
1  static void waitForPermission(size_t& permits, condition_variable_any& permitsCV, mutex& permitsLock) {
2    permitsLock.lock();
3    if (permits == 0) {
4      permitsLock.unlock();
5      permitsCV.wait();
6      permitsLock.lock();
7    }
8    permits--;
9    permitsLock.unlock();
10 }
```

permits = 0

All done eating! I will return my permit.



thread #1

thread #2

waitForPermission

```
1  static void waitForPermission(size_t& permits, condition_variable_any& permitsCV, mutex& permitsLock) {
2    permitsLock.lock();
3    if (permits == 0) {
4      permitsLock.unlock();
5      permitsCV.wait();
6      permitsLock.lock();
7    }
8    permits--;
9    permitsLock.unlock();
10 }
```

permits = 1

All done eating! I will return my permit.

thread #1

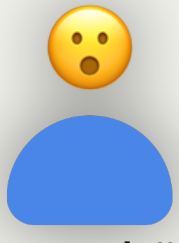
thread #2

waitForPermission

```
1  static void waitForPermission(size_t& permits, condition_variable_any& permitsCV, mutex& permitsLock) {
2    permitsLock.lock();
3    if (permits == 0) {
4      permitsLock.unlock();
5      permitsCV.wait();
6      permitsLock.lock();
7    }
8    permits--;
9    permitsLock.unlock();
10 }
```

permits = 1

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



thread #1

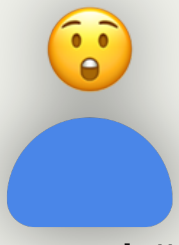
thread #2

waitForPermission

```
1  static void waitForPermission(size_t& permits, condition_variable_any& permitsCV, mutex& permitsLock) {
2    permitsLock.lock();
3    if (permits == 0) {
4      permitsLock.unlock();
5      permitsCV.wait();
6      permitsLock.lock();
7    }
8    permits--;
9    permitsLock.unlock();
10 }
```

permits = 1

"attention waiting threads, a permit is available!"



thread #1

thread #2

waitForPermission

```
1  static void waitForPermission(size_t& permits, condition_variable_any& permitsCV, mutex& permitsLock) {
2    permitsLock.lock();
3    if (permits == 0) {
4      permitsLock.unlock();
5      permitsCV.wait();
6      permitsLock.lock();
7    }
8    permits--;
9    permitsLock.unlock();
10 }
```

permits = 1

thread #1

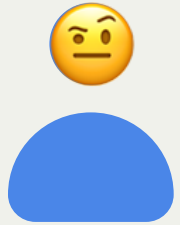
thread #2

waitForPermission

```
1  static void waitForPermission(size_t& permits, condition_variable_any& permitsCV, mutex& permitsLock) {
2    permitsLock.lock();
3    if (permits == 0) {
4      permitsLock.unlock();
5      permitsCV.wait();
6      permitsLock.lock();
7    }
8    permits--;
9    permitsLock.unlock();
10 }
```

permits = 1

thread #1



thread #2

100 years later

waitForPermission

For **waitForPermission**, if no permits are available we must wait until one becomes available.

```
1  static void waitForPermission(size_t& permits, condition_variable_any& permitsCV, mutex& permitsLock) {
2    permitsLock.lock();
3    if (permits == 0) {
4      permitsLock.unlock();
5      permitsCV.wait();
6      permitsLock.lock();
7    }
8    permits--;
9    permitsLock.unlock();
10 }
```

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

Key Problem: 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.

waitForPermission

For **waitForPermission**, if no permits are available we must wait until one becomes available.

```
1  static void waitForPermission(size_t& permits, condition_variable_any& permitsCV, mutex& permitsLock) {
2    permitsLock.lock();
3    if (permits == 0) {
4      permitsLock.unlock();
5      permitsCV.wait();
6      permitsLock.lock();
7    }
8    permits--;
9    permitsLock.unlock();
10 }
```

Solution: condition variables are meant for these situations. They are able to unlock the lock *for us* after we are put to sleep.

waitForPermission

For **waitForPermission**, if no permits are available we must wait until one becomes available.

```
1 static void waitForPermission(size_t& permits, condition_variable_any& permitsCV, mutex& permitsLock) {
2   permitsLock.lock();
3   if (permits == 0) {
4     permitsCV.wait(permitsLock);
5   }
6   permits--;
7   permitsLock.unlock();
8 }
```

Solution: a condition variable is meant for these situations.

- It will unlock the lock *for us* after we are put to sleep.
- When we are notified, it will only return once it has reacquired the lock for us.

waitForPermission

For **waitForPermission**, if no permits are available we must wait until one becomes available.

```
1 static void waitForPermission(size_t& permits, condition_variable_any& permitsCV, mutex& permitsLock) {
2   permitsLock.lock();
3   if (permits == 0) {
4     permitsCV.wait(permitsLock);
5   }
6   permits--;
7   permitsLock.unlock();
8 }
```

Here's what **cv.wait** does:

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

For **waitForPermission**, if no permits are available we must wait until one becomes available.

```
1 static void waitForPermission(size_t& permits, condition_variable_any& permitsCV, mutex& permitsLock) {
2   permitsLock.lock();
3   if (permits == 0) {
4     permitsCV.wait(permitsLock);
5   }
6   permits--;
7   permitsLock.unlock();
8 }
```

Spoiler: there is a race condition here that could lead to negative permits.
(**HINT:** consider when one permit becomes available for multiple waiting threads.)

waitForPermission

```
1 static void waitForPermission(size_t& permits, condition_variable_any& permitsCV, mutex& permitsLock) {
2   permitsLock.lock();
3   if (permits == 0) {
4     permitsCV.wait(permitsLock);
5   }
6   permits--;
7   permitsLock.unlock();
8 }
```

permits = 0

thread #1

thread #2

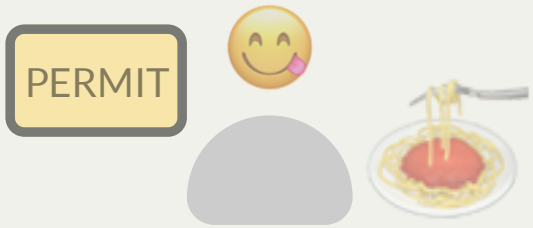
thread #3

waitForPermission

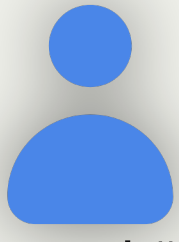
```
1 static void waitForPermission(size_t& permits, condition_variable_any& permitsCV, mutex& permitsLock) {
2   permitsLock.lock();
3   if (permits == 0) {
4     permitsCV.wait(permitsLock);
5   }
6   permits--;
7   permitsLock.unlock();
8 }
```

permits = 0

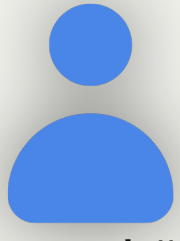
We need to wait for a permit in order to eat.



thread #1



thread #2



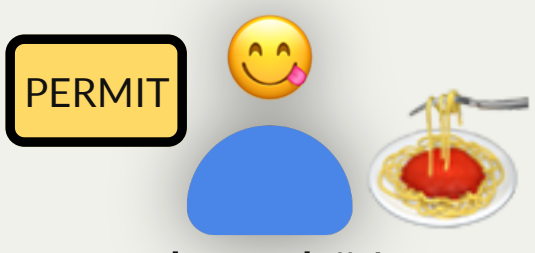
thread #3

waitForPermission

```
1 static void waitForPermission(size_t& permits, condition_variable_any& permitsCV, mutex& permitsLock) {
2   permitsLock.lock();
3   if (permits == 0) {
4     permitsCV.wait(permitsLock);
5   }
6   permits--;
7   permitsLock.unlock();
8 }
```

permits = 0

All done eating! I will return my permit.



thread #1

thread #2

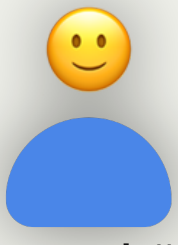
thread #3

waitForPermission

```
1 static void waitForPermission(size_t& permits, condition_variable_any& permitsCV, mutex& permitsLock) {
2   permitsLock.lock();
3   if (permits == 0) {
4     permitsCV.wait(permitsLock);
5   }
6   permits--;
7   permitsLock.unlock();
8 }
```

permits = 1

All done eating! I will return my permit.



thread #1

thread #2

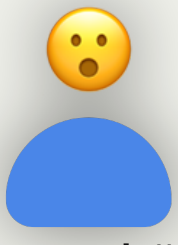
thread #3

waitForPermission

```
1 static void waitForPermission(size_t& permits, condition_variable_any& permitsCV, mutex& permitsLock) {
2   permitsLock.lock();
3   if (permits == 0) {
4     permitsCV.wait(permitsLock);
5   }
6   permits--;
7   permitsLock.unlock();
8 }
```

permits = 1

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



thread #1

thread #2

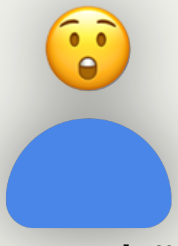
thread #3

waitForPermission

```
1 static void waitForPermission(size_t& permits, condition_variable_any& permitsCV, mutex& permitsLock) {
2   permitsLock.lock();
3   if (permits == 0) {
4     permitsCV.wait(permitsLock);
5   }
6   permits--;
7   permitsLock.unlock();
8 }
```

permits = 1

"attention waiting threads, a permit is available!"



thread #1

thread #2

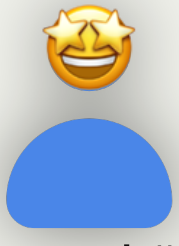
thread #3

waitForPermission

```
1 static void waitForPermission(size_t& permits, condition_variable_any& permitsCV, mutex& permitsLock) {
2   permitsLock.lock();
3   if (permits == 0) {
4     permitsCV.wait(permitsLock);
5   }
6   permits--;
7   permitsLock.unlock();
8 }
```

permits = 1

thread #1



thread #2

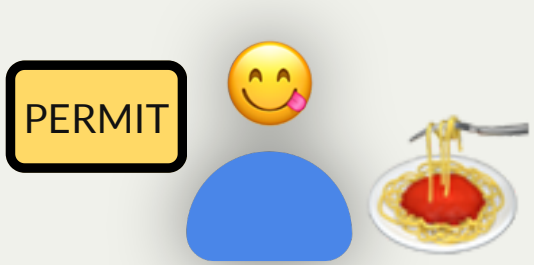
thread #3

waitForPermission

```
1 static void waitForPermission(size_t& permits, condition_variable_any& permitsCV, mutex& permitsLock) {
2   permitsLock.lock();
3   if (permits == 0) {
4     permitsCV.wait(permitsLock);
5   }
6   permits--;
7   permitsLock.unlock();
8 }
```

permits = 0

thread #1



thread #2

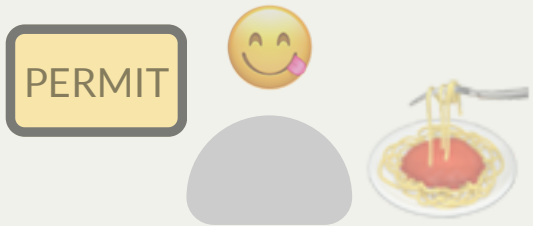
thread #3

waitForPermission

```
1 static void waitForPermission(size_t& permits, condition_variable_any& permitsCV, mutex& permitsLock) {
2   permitsLock.lock();
3   if (permits == 0) {
4     permitsCV.wait(permitsLock);
5   }
6   permits--;
7   permitsLock.unlock();
8 }
```

permits = 0

thread #1



thread #2

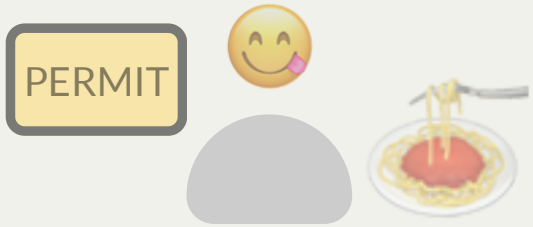
thread #3

waitForPermission

```
1 static void waitForPermission(size_t& permits, condition_variable_any& permitsCV, mutex& permitsLock) {
2   permitsLock.lock();
3   if (permits == 0) {
4     permitsCV.wait(permitsLock);
5   }
6   permits--;
7   permitsLock.unlock();
8 }
```

permits = <very large #>

thread #1



thread #2

thread #3

waitForPermission

For **waitForPermission**, if no permits are available we must wait until one becomes available.

```
1 static void waitForPermission(size_t& permits, condition_variable_any& permitsCV, mutex& permitsLock) {
2   permitsLock.lock();
3   if (permits == 0) {
4     permitsCV.wait(permitsLock);
5   }
6   permits--;
7   permitsLock.unlock();
8 }
```

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.

waitForPermission

For **waitForPermission**, if no permits are available we must wait until one becomes available.

```
1 static void waitForPermission(size_t& permits, condition_variable_any& permitsCV, mutex& permitsLock) {
2   permitsLock.lock();
3   while (permits == 0) {
4     permitsCV.wait(permitsLock);
5   }
6   permits--;
7   permitsLock.unlock();
8 }
```

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

Condition Variables

There is a **notify_one()** method to notify just one thread instead of all.

- however, here, we would then have to notify *every time we put a permit back* (why?)
    - imagine we notify just when permits goes from 0 to 1, but only notify one thread
    - threads A and B have permits and are eating
    - threads C and D are waiting for permits (none currently available)
    - first, thread A returns their permit (0 -> 1) and signals to one thread (e.g. C)
    - then, thread B returns their permit (1 -> 2), but doesn't signal
    - C wakes up and claims the permit
    - D is still waiting! :(

Key Idea: In our approach of notifying just when we go from 0 -> 1, others may return further permits after us; therefore, we should wake up everyone just in case. Perhaps this approach results in fewer notifications.

Other note: we still always need a loop around calls to **wait** - because of spurious wakeups even when a notification wasn't sent! Or if another philosopher grabs the permit before us.

Condition Variables

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.
- We can call ***notify_all()*** to send a signal to waiting threads.
- We call ***wait(lock)*** in a loop in case we are woken up but must wait longer.

Demo: Dining Philosophers with Condition Variables

`dining-philosophers-with-cv-wait.cc`

Condition Variables

```
1 static void waitForPermission(size_t& permits, condition_variable_any& permitsCV, mutex& permitsLock) {
2   permitsLock.lock();
3   while (permits == 0) {
4     permitsCV.wait(permitsLock);
5   }
6   permits--;
7   permitsLock.unlock();
8 }
```

This **while** loop pattern is so common that there is another convenience form of **wait** that also includes the loop.

- There is a second parameter which is a ***lambda function***: it should return **true** when we wish to stop repeatedly waiting for a notification.

dining-philosophers-with-cv-wait-lambda.cc

Condition Variables

```
1 static void waitForPermission(size_t& permits, condition_variable_any& permitsCV, mutex& permitsLock) {
2   permitsLock.lock();
3   permitsCV.wait(permitsLock, [&permits] { return permits > 0; });
4   permits--;
5   permitsLock.unlock();
6 }
```

```
1 // possible implementation of 2-arg wait
2
3 template <Predicate pred>
4 void condition_variable_any::wait(mutex& m, Pred pred) {
5   while (!pred()) wait(m);
6 }
```

dining-philosophers-with-cv-wait-lambda.cc

Semaphore

This "permission slip" pattern with signaling is a very common pattern:

- Have a **counter**, **mutex** and **condition_variable_any** to track some permission slips
- Thread-safe way to grant permission and to wait for permission (aka sleep)
- But, it's cumbersome to need 3 variables to implement this - is there a better way?
- A *semaphore* is a single variable type that encapsulates all this functionality

Plan For Today

- **Recap**: Dining With Philosophers
- Condition Variables
- **Semaphores**

Semaphore

A semaphore is a variable type that lets you manage a count of finite resources.

- You initialize the semaphore with the count of resources to start with
- You can request permission via **semaphore::wait()** - aka waitForPermission
- You can grant permission via **semaphore::signal()** - aka grantPermission
- Note: count can be negative! This allows for some interesting use cases (more later).

```
1  class semaphore {
2   public:
3    semaphore(int value = 0);
4    void wait();
5    void signal();
6
7   private:
8    int value;
9    std::mutex m;
10   std::condition_variable_any cv;
11 }
```

Semaphore

A semaphore is a variable type that lets you manage a count of finite resources.

- You initialize the semaphore with the count of resources to start with

```
semaphore permits(5); // this will allow five permits
```

- When a thread wants to use a permit, it first **`wait`**s for the permit, and then **`signal`**s when it is done using a permit:

```
permits.wait(); // if five other threads currently hold permits, this will block

// only five threads can be here at once

permits.signal(); // if other threads are waiting, a permit will be available
```

- A **`mutex`** is kind of like a special case of a semaphore with one permit, but you should use a **`mutex`** in that case as it is simpler and more efficient. Additionally, the benefit of a mutex is that it can *only* be released by the lock-holder.

Semaphore - signal

A semaphore is a variable type that lets you manage a count of finite resources.

- You can grant permission via **semaphore::signal()** - aka grantPermission

```
1  class semaphore {
2   public:
3    semaphore(int value = 0);
4    void wait();
5    void signal();
6
7   private:
8    int value;
9    std::mutex m;
10   std::condition_variable_any cv;
11 }
```

```
1 void semaphore::signal() {
2   m.lock();
3   value++;
4   if (value == 1) cv.notify_all();
5   m.unlock();
6 }
```

Semaphore - wait

A semaphore is a variable type that lets you manage a count of finite resources.

- You can request permission via **semaphore::wait()** - aka waitForPermission

```
1  class semaphore {
2   public:
3    semaphore(int value = 0);
4    void wait();
5    void signal();
6
7   private:
8    int value;
9    std::mutex m;
10   std::condition_variable_any cv;
11 }
```

```
1 void semaphore::wait() {
2   m.lock();
3   cv.wait(m, [this] { return value > 0; })
4   value--;
5   m.unlock();
6 }
```

Why [this]? To access instance variables in a lambda, we must capture the current object.

And Now....We Eat!

Here's our final version of the **dining-philosophers**, replacing **size_t**, **mutex**, and **condition_variable_any** with a single **semaphore.**

```
1  static void philosopher(size_t id, mutex& left, mutex& right, semaphore& permits) {
2    for (size_t i = 0; i < kNumMeals; i++) {
3      think(id);
4      eat(id, left, right, permits);
5    }
6  }
7
8  int main(int argc, const char *argv[]) {
9    // NEW
10   semaphore permits(kNumForks - 1);
11
12   mutex forks[kNumForks];
13   thread philosophers[kNumPhilosophers];
14   for (size_t i = 0; i < kNumPhilosophers; i++) {
15     mutex& left = forks[i];
16     mutex& right = forks[(i + 1) % kNumPhilosophers];
17     philosophers[i] = thread(philosopher, i, ref(left), ref(right), ref(permits));
18   }
19   for (thread& p: philosophers) p.join();
20   return 0;
21 }
```

dining-philosophers-with-semaphore.cc

Recap

- **Recap**: Dining With Philosophers
- Condition Variables
- Semaphores

Next time: multithreading patterns with mutexes, condition variables and semaphores