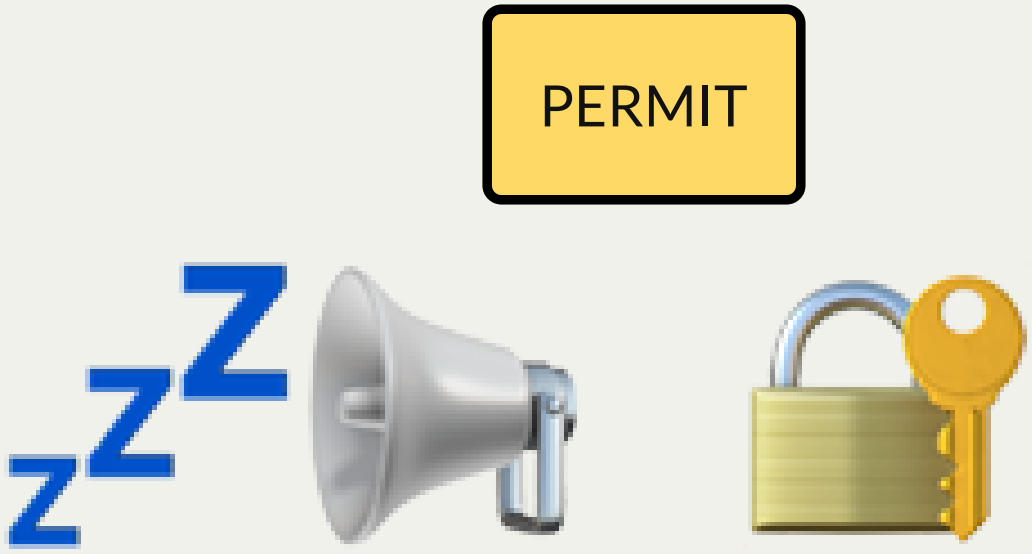
CS110 Lecture 17: Semaphores and Multithreading Patterns

CS110: Principles of Computer Systems

Winter 2021-2022

Stanford University

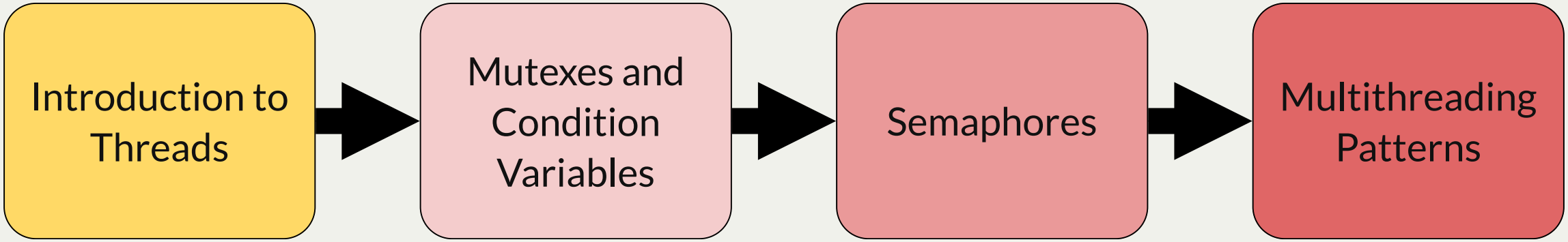
Instructors: Nick Troccoli and Jerry Cain



PDF of this presentation

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

Learning About Multithreading



Lecture 13 | **Lectures 14/15** | **Lecture 16** | **This/next lecture**

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

Learning Goals

- Learn how a semaphore generalizes the "permits pattern" we previously saw
- Learn how to apply our toolbox of concurrency directives (mutexes, condition variables and semaphores) to coordinate threads in different ways

Plan For Today

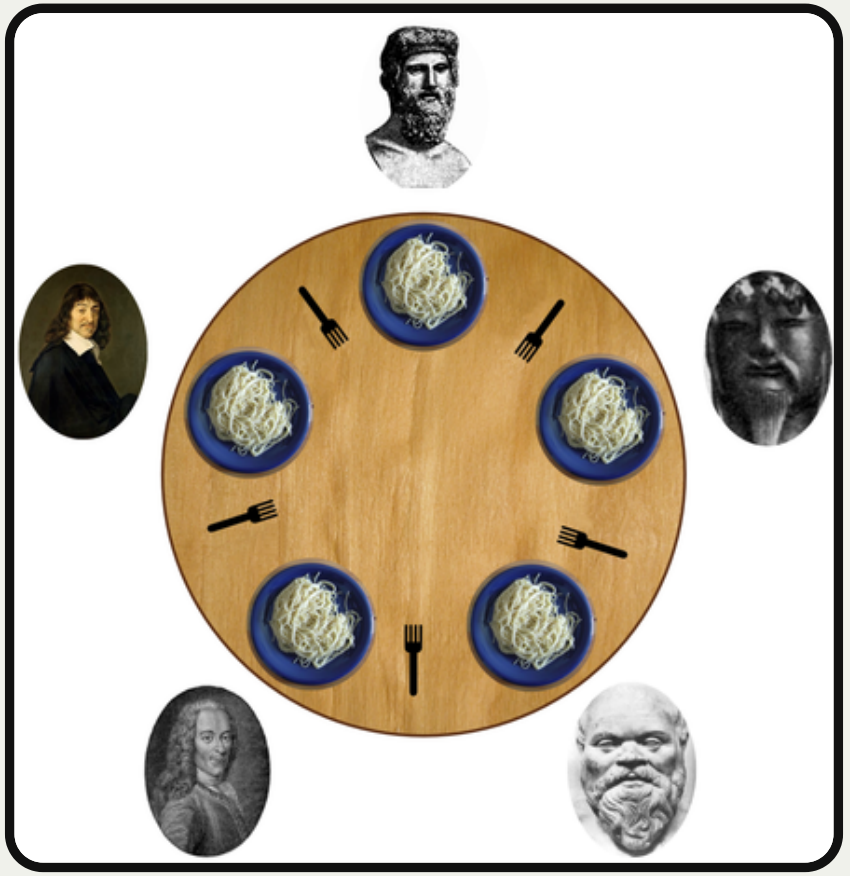
- **Recap:** Dining With Philosophers
- Convenience - *Unique Locks*
- More about Semaphores
- Multithreading Patterns
- **Example**: Reader-Writer

Plan For Today

- **Recap: Dining With Philosophers**
- Convenience - *Unique Locks*
- More about Semaphores
- Multithreading Patterns
- **Example**: Reader-Writer

Dining Philosophers Problem

- Last time we successfully implemented the dining philosophers program!
- We used **mutexes** to model exclusive access to forks.
- We used a **counter** and a **mutex** to model permits that you must have before attempting to eat.
- We used a **condition variable** to allow threads returning permits to tell waiting threads there are permits available.
- We saw how a **semaphore** combines a **counter**, **mutex** and **condition** variable to implement this permits model.

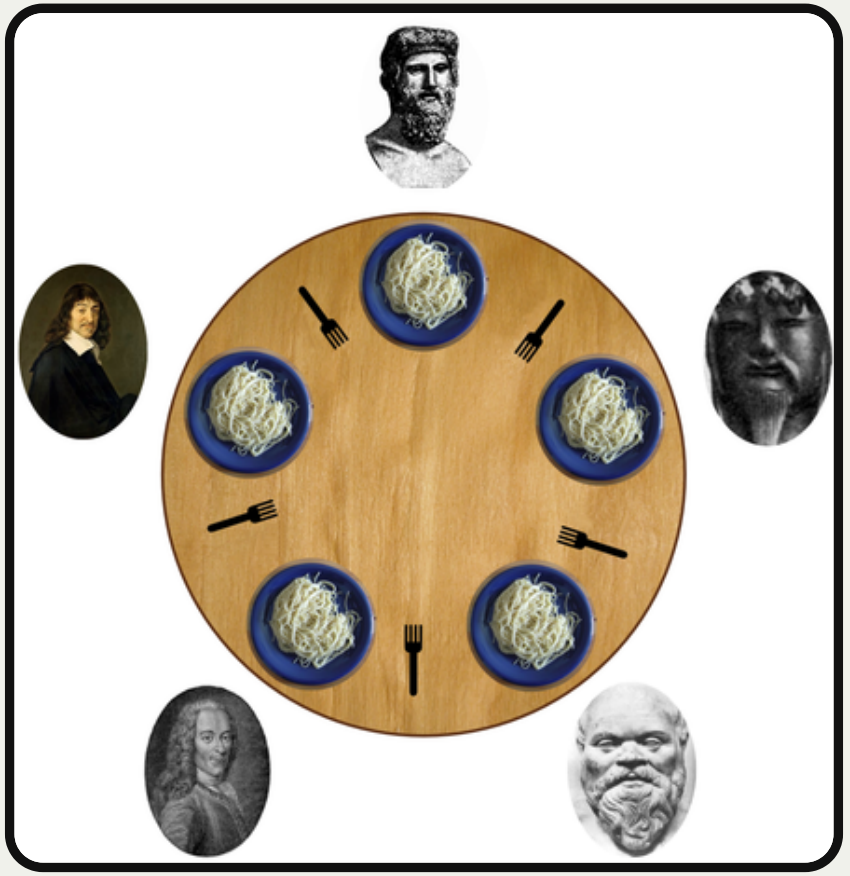


https://commons.wikimedia.org/wiki/File:An_illustration_of_the_dining_philosophers_problem.png

`dining-philosophers-with-semaphore.cc`

Dining Philosophers Problem

- Last time we successfully implemented the dining philosophers program!
- We used **mutexes** to model exclusive access to forks.
- We used a **counter** and a **mutex** to model permits that you must have before attempting to eat.
- **We used a condition variable to allow threads returning permits to tell waiting threads there are permits available.**
- We saw how a **semaphore** combines a **counter**, **mutex** and **condition** variable to implement this permits model.



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

`dining-philosophers-with-semaphore.cc`

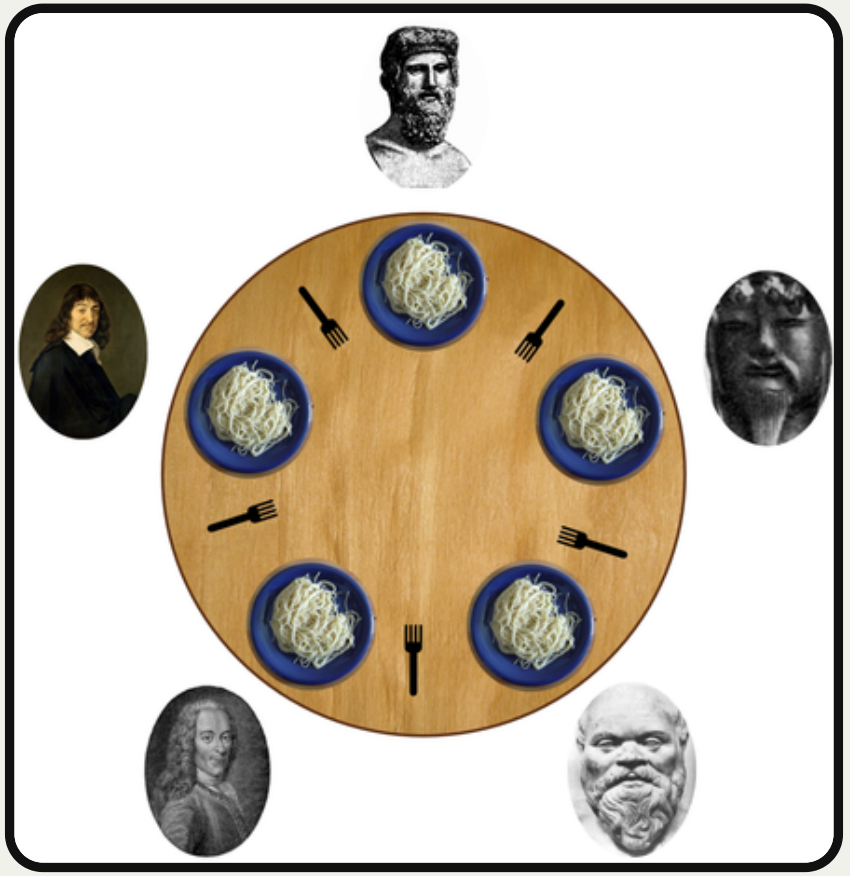
Condition Variable Key Takeaways

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.
- 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.

Dining Philosophers Problem

- Last time we successfully implemented the dining philosophers program!
- We used **mutexes** to model exclusive access to forks.
- We used a **counter** and a **mutex** to model permits that you must have before attempting to eat.
- We used a **condition variable** to allow threads returning permits to tell waiting threads there are permits available.
- **We saw how a semaphore combines a counter, mutex and condition variable to implement this permits model.**



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

`dining-philosophers-with-semaphore.cc`

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 }
```

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

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

// only five threads can be here at once

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

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

And Now....We Eat!

eat now relies on the **semaphore** instead of calling **waitForPermission** and **grantPermission**.

```
1  static void eat(size_t id, mutex& left, mutex& right, semaphore& permits) {
2    // NEW
3    permits.wait();
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   permits.signal();
13
14   left.unlock();
15   right.unlock();
16 }
```

Thought Questions:

- Could/should we switch the order of lines 14-15, so that **right.unlock()** precedes **left.unlock()**?
  - Yes, it is arbitrary

And Now....We Eat!

eat now relies on the **semaphore** instead of calling **waitForPermission** and **grantPermission**.

```
1  static void eat(size_t id, mutex& left, mutex& right, semaphore& permits) {
2    // NEW
3    permits.wait();
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   permits.signal();
13
14   left.unlock();
15   right.unlock();
16 }
```

- Could we call **permits.signal()** in between **right.lock()** and the first **cout** statement?
  - Yes, but others will still have to wait for forks

And Now....We Eat!

eat now relies on the **semaphore** instead of calling **waitForPermission** and **grantPermission**.

```
1  static void eat(size_t id, mutex& left, mutex& right, semaphore& permits) {
2    // NEW
3    permits.wait();
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   permits.signal();
13
14   left.unlock();
15   right.unlock();
16 }
```

- Instead of a semaphore, could we use a **mutex** to bundle the calls to **left.lock()** and **right.lock()** into a critical region?
- - Yes!

Plan For Today

- **Recap:** Dining With Philosophers
- **Convenience - *Unique Locks***
- More about Semaphores
- Multithreading Patterns
- **Example**: Reader-Writer

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 an *adapter* ("wrapper") for mutexes called ***unique_lock*** that when created can automatically lock a mutex, and when destroyed can automatically unlock a mutex.

```
1 void myFunc(mutex& myLock) {
2   myLock.lock();
3   ... // some code here
4   myLock.unlock();
5 }
```

-- same as --

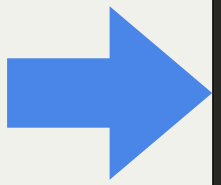
```
1 void myFunc(mutex& myLock) {
2   unique_lock<mutex> myLockAdapter(myLock); // acquires myLock
3   ... // some code here
4   // myLockAdapter goes out of scope here, unlocks myLock
5 }
```

Particularly useful if you have many paths to exit a function and you must unlock in all paths.

Unique Locks

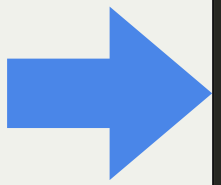
There is an *adapter* ("wrapper") for mutexes called ***unique_lock*** that when created can automatically lock a mutex, and when destroyed can automatically unlock a mutex.

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



```
1 void semaphore::signal() {
2   unique_lock<mutex> lockAdapter(m);
3   value++;
4   if (value == 1) cv.notify_all();
5 }
```

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

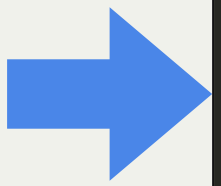


```
1 void semaphore::wait() {
2   unique_lock<mutex> lockAdapter(m);
3   cv.wait(lockAdapter, [this] { return value > 0; })
4   value--;
5 }
```

Unique Locks

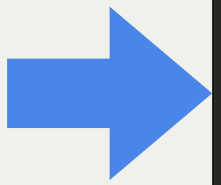
There is an *adapter* ("wrapper") for mutexes called ***unique_lock*** that when created can automatically lock a mutex, and when destroyed can automatically unlock a mutex.

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



```
1 void semaphore::signal() {
2   unique_lock<mutex> lockAdapter(m);
3   value++;
4   if (value == 1) cv.notify_all();
5 }
```

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



```
1 void semaphore::wait() {
2   unique_lock<mutex> lockAdapter(m);
3   cv.wait(lockAdapter, [this] { return value > 0; })
4   value--;
5 }
```

Cool note: unique locks still let us manually lock/unlock if needed via wrapper lock() and unlock() methods.

Plan For Today

- **Recap:** Dining With Philosophers
- Convenience - *Unique Locks*
- **More about Semaphores**
- Multithreading Patterns
- **Example**: Reader-Writer

Semaphore Patterns

semaphores can be used to support **thread coordination**.

- One thread can stall—via **semaphore::wait**—until other thread(s) use **semaphore::signal**, e.g. the signaling thread prepared some data that the waiting thread needs to continue.
- Generalization of **thread::join**

3 core patterns:

- permits
- binary coordination
- general coordination

Semaphore Use Case: Permits

A *semaphore* is a variable type that represents a count of finite resources.

- "Permits" pattern with a **counter**, **mutex** and **condition_variable_any**
- Thread-safe way to grant permission and to wait for permission (aka sleep)

How can we use semaphores to model a finite, discrete number of available resources?

- We initialize the semaphore with the initial resource count. If a thread needs the resource, it calls *wait* on the semaphore. Threads may also *signal* to indicate more resources are available. (dining philosophers)

Semaphore Use Case: Binary Coordination

A *semaphore* is a variable type that represents a count of finite resources.

- "Permits" pattern with a **counter**, **mutex** and **condition_variable_any**
- Thread-safe way to grant permission and to wait for permission (aka sleep)

How can we use semaphores to have thread A to proceed only once thread B completes some work?

- We initialize the semaphore to 0; thread A waits on it, and thread B signals when done. The semaphore thus records the status of one event (0 = not-yet-completed or completed-and-checked and 1 = completed-but-not-yet-checked). This is like a general version of thread::join().

Binary Thread Coordination

```
1  void create(int creationCount, semaphore &s) {
2      for (int i = 0; i < creationCount; i++) {
3          cout << oslock << "Now creating " << i << endl << osunlock;
4          s.signal();
5      }
6  }
7
8  void consume_after_create(int consumeCount, semaphore &s) {
9      for (int i = 0; i < consumeCount; i++) {
10         s.wait();
11         cout << oslock << "Now consuming " << i << endl << osunlock;
12     }
13 }
14
15 int main(int argc, const char *argv[]) {
16     semaphore zeroSemaphore; // can omit (0), since default initializes to 0
17     int numIterations = 5;
18     thread thread_waited_on(create, numIterations, ref(zeroSemaphore));
19     thread waiting_thread(consume_after_create, numIterations, ref(zeroSemaphore));
20     thread_waited_on.join();
21     waiting_thread.join();
22     return 0;
23 }
```

```
$ ./binary-coordination
Now creating 0
Now creating 1
Now creating 2
Now creating 3
Now creating 4
Now consuming 0
Now consuming 1
Now consuming 2
Now consuming 3
Now consuming 4
```

binary-coordination.cc

Semaphore Use Case: General Coordination

A *semaphore* is a variable type that represents a count of finite resources.

- "Permits" pattern with a **counter**, **mutex** and **condition_variable_any**
- Thread-safe way to grant permission and to wait for permission (aka sleep)

How can we use semaphores to have thread A wait for something to happen ***n*** times before proceeding?

- Have a semaphore initialized to ***-n + 1***; thread A waits on it, and other threads signal. Thus, we can imagine that there are missing permits that must be returned for A to advance. Or, initialize to 0; thread A waits ***n*** times on it, and other threads signal. Thus, there are no permits initially, and A must have ***n*** permits to advance. This is like a general version of thread::join().

General Thread Coordination

```
1  void writer(int i, semaphore &s) {
2      cout << oslock << "Sending signal " << i << endl << osunlock;
3      s.signal();
4  }
5
6  void read_after_ten(semaphore &s) {
7      s.wait();
8      cout << oslock << "Got enough signals to continue!" << endl << osunlock;
9  }
10
11 int main(int argc, const char *argv[]) {
12     semaphore negSemaphore(-9);
13     thread writers[10];
14     for (size_t i = 0; i < 10; i++) {
15         writers[i] = thread(writer, i, ref(negSemaphore));
16     }
17     thread r(read_after_ten, ref(negSemaphore));
18     for (thread &t : writers) t.join();
19     r.join();
20     return 0;
21 }
```

```
$ ./general-coordination
Sending signal 0
Sending signal 1
Sending signal 2
Sending signal 3
Sending signal 5
Sending signal 7
Sending signal 8
Sending signal 9
Sending signal 6
Sending signal 4
Got enough signals to continue!
```

general-coordination.cc

Semaphore Configurations

A **`semaphore`** initialized with a positive number means:

- We start with a fixed number of permits.
- Once those permits are taken, further threads must wait for permits to be returned before continuing
- Example: Dining Philosophers

A **`semaphore`** initialized with zero means:

- We don't have *any* permits!
- **`permits.wait()`** *always* initially waits for a signal, and will only stop waiting once that signal is received. E.g. you want to wait until another thread finishes before a thread continues.

A **`semaphore`** initialized with a negative number means:

- The semaphore must reach 1 before the initial wait would end. E.g. you want to wait until other threads finish before a final thread continues

Plan For Today

- **Recap:** Dining With Philosophers
- Convenience - *Unique Locks*
- More about Semaphores
- **Multithreading Patterns**
- **Example**: Reader-Writer

Multithreading Techniques

We have learned about the three tools we will use to write multithreaded programs: **mutexes**, **condition variables** and **semaphores**.

- Mutex: Binary Lock
- Condition Variable: Generalized Wait
- Semaphore: Permits and Thread Coordination

Mutex

A lock with one holder at a time that lets us enforce *mutual exclusion*

Enables multithreading pattern #1: binary lock - we have multiple threads, but we want only one thread at a time to be able to execute some code (e.g. modifying shared data structure).

Optionally paired with a **unique_lock** that locks when created, and unlocks when destroyed.

Condition Variable

Allows thread communication by letting threads wait or notify other waiting threads.

Enables multithreading pattern #2: generalized wait - we want a thread to go to sleep until it's notified by another thread that some condition is true.

Note: a condition variable provides a more general form of waiting not limited to a permits count (e.g. useful where a wait condition is more complex than a counter being > 0).

Semaphore

Combines a condition variable, mutex and int to track a count of something. Threads can increment this count or wait for the count to be > 0.

Enables multithreading pattern #3: permits - model a finite, discrete number of available resources. We initialize the semaphore with the initial resource count. If a thread needs the resource, it calls *wait* on the semaphore. Threads may also *signal* to indicate more resources are available.

Enables multithreading pattern #4: binary thread coordination - we want thread A to proceed only once thread B completes some work. We initialize the semaphore to 0; thread A waits on it, and thread B signals when done. The semaphore thus records the status of one event (0 = not-yet-completed or completed-and-checked and 1 = completed-but-not-yet-checked). This is like a general version of thread::join().

Semaphore

Enables multithreading pattern #5: general thread coordination - Thread A waits for something to happen ***n*** times before proceeding. Have a semaphore initialized to ***-n + 1***; thread A waits on it, and other threads signal. Thus, we can imagine that there are missing permits that must be returned for A to advance. Or, initialize to 0; thread A waits ***n*** times on it, and other threads signal. Thus, there are no permits initially, and A must have ***n*** permits to advance. This is like a general version of thread::join().

Multithreading Patterns

Binary lock (mutex) - e.g. dining philosophers' forks

Generalized wait (condition variable) - e.g. waiting for complex condition

Permits (semaphore) - e.g. dining philosophers permits for eating

Binary coordination (semaphore) - e.g. writer telling reader there is new content

Generalized coordination (semaphore) - e.g. thread waits for **N** others to finish a task

Layered Construction (combo) - combine multiple patterns

Layered Construction

Layered Construction (combo) - combine multiple patterns

Examples:

dining philosophers: mutexes for forks, semaphore for permits

worker threads operate on a shared data structure protected by a mutex. When they're done, they signal a generalized coordination semaphore, and a final thread waits for all workers to signal before continuing.

threads A and B compete to be the first to finish work and signal C to continue via a binary coordination semaphore. To ensure the winner's info is recorded, the winning thread acquires a mutex, updates a shared variable with their information, and then signals C to continue.

Plan For Today

- **Recap:** Dining With Philosophers
- Convenience - *Unique Locks*
- More about Semaphores
- Multithreading Patterns
- **Example**: Reader-Writer

Reader-Writer

Let's implement a program that requires thread coordination with semaphores. First, we'll look at a version **without** semaphores to see why they are necessary.

The **reader-writer** pattern/program spawns 2 threads: one writer (publishes content to a shared buffer) and one reader (reads from shared buffer when content is available)

Common pattern! E.g. web server publishes content over a dedicated communication channel, and the web browser consumes that content.

Optionally consider a more complex version: multiple readers, similar to how a web server handles many incoming requests (puts request in buffer, readers each read and process requests)

Confused Reader-Writer

```
 1 int main(int argc, const char *argv[]) {
 2   // Create an empty buffer
 3   char buffer[kNumBufferSlots];
 4   memset(buffer, ' ', sizeof(buffer));
 5
 6   thread writer(writeToBuffer, buffer, sizeof(buffer), kNumIterations);
 7   thread reader(readFromBuffer, buffer, sizeof(buffer), kNumIterations);
 8   writer.join();
 9   reader.join();
10   return 0;
11 }
```

Both threads share the same buffer, so they agree where content is stored (think of buffer like state for a pipe or a connection between client and server)

confused-reader-writer.cc

Confused Reader-Writer

```
1  static void readFromBuffer(char buffer[], size_t bufferSize, size_t iterations) {
2    cout << oslock << "Reader: ready to read." << endl << osunlock;
3    for (size_t i = 0; i < iterations * bufferSize; i++) {
4
5      // Read and process the data
6      char ch = buffer[i % bufferSize];
7      processData(ch); // sleep to simulate work
8      buffer[i % bufferSize] = ' ';
9
10     cout << oslock << "Reader: consumed data packet "
11       << "with character '" << ch << "'.\t\t" << osunlock;
12     printBuffer(buffer, bufferSize);
13   }
14 }
```

The **reader** consumes the content as it's written. Each thread cycles through the buffer the same number of times, and they both agree that **i % 8** identifies the next slot of interest.

confused-reader-writer.cc

Confused Reader-Writer

```
1  static void writeToBuffer(char buffer[], size_t bufferSize, size_t iterations) {
2    cout << oslock << "Writer: ready to write." << endl << osunlock;
3    for (size_t i = 0; i < iterations * bufferSize; i++) {
4
5      char ch = prepareData();
6      buffer[i % bufferSize] = ch;
7
8      cout << oslock << "Writer: published data packet with character '"
9        << ch << "'.\t\t" << osunlock;
10     printBuffer(buffer, bufferSize);
11   }
12 }
```

The **writer** publishes content to the circular buffer. Each thread cycles through the buffer the same number of times, and they both agree that **i % 8** identifies the next slot of interest.

confused-reader-writer.cc

The Race Condition Checklist

☐ **Identify shared data that may be modified concurrently.** What shared data is used across threads, passed by reference or globally?

☐ **Document and confirm an ordering of events that causes unexpected behavior**. What assumptions are made in the code that can be broken by certain orderings?

☐ **Use concurrency directives to force expected orderings and add constraints**. How can we use mutexes, atomic operations, or other constraints to force the correct ordering(s)?

The Race Condition Checklist

☐ **Identify shared data that may be modified concurrently.** What shared data is used across threads, passed by reference or globally? **The buffer.**

☐ **Document and confirm an ordering of events that causes unexpected behavior.** What assumptions are made in the code that can be broken by certain orderings?

☐ **Use concurrency directives to force expected orderings and add constraints.** How can we use mutexes, atomic operations, or other constraints to force the correct ordering(s)?

The Race Condition Checklist

☑ **Identify shared data that may be modified concurrently.** What shared data is used across threads, passed by reference or globally? **The buffer.**

☐ **Document and confirm an ordering of events that causes unexpected behavior.** What assumptions are made in the code that can be broken by certain orderings? **We assume that the reader will only read data that was written, and the writer will not overwrite unread data.**

☐ **Use concurrency directives to force expected orderings and add constraints.** How can we use mutexes, atomic operations, or other constraints to force the correct ordering(s)?

The Race Condition Checklist

☑ **Identify shared data that may be modified concurrently.** What shared data is used across threads, passed by reference or globally? **The buffer.**

☑ **Document and confirm an ordering of events that causes unexpected behavior.** What assumptions are made in the code that can be broken by certain orderings? **We assume that the reader will only read data that was written, and the writer will not overwrite unread data.**

☐ **Use concurrency directives to force expected orderings and add constraints.** How can we use mutexes, atomic operations, or other constraints to force the correct ordering(s)?

Confused Reader-Writer

Problem: each thread runs independently, without knowing how much progress the other has made.

- Example: no way for the **reader** to know that the slot it wants to read from has meaningful data in it. It's possible the writer hasn't gotten that far yet.
- Example: the **writer** could loop around and overwrite content that the **reader** has not yet consumed.

Reader-Writer Constraints

Goal: we must encode constraints into our program.

What constraint(s) should we add to our program?

A reader should not read until something is available to read

A writer should not write until there is space available to write

How can we model these constraint(s)?

One semaphore to manage empty slots

One semaphore to manage full slots

Reader-Writer

What might this look like in code?

- The **writer** thread waits until at least one buffer slot is empty before writing. Once it writes, it increments the full buffer count by one.
- The **reader** thread waits until at least one buffer slot is full before reading. Once it reads, it increments the empty buffer count by one.

We have two semaphores to permit bidirectional thread coordination: reader can communicate with writer, and writer can communicate with reader.

Demo: Reader-Writer

reader-writer.cc

Recap

- **Recap:** Dining With Philosophers
- Convenience - *Unique Locks*
- More about Semaphores
- Multithreading Patterns
- **Example**: Reader-Writer

Next time: more multithreading examples