Today

- Concurrency Problems
  - Producer Consumer
- Synchronization Mechanisms
  - Condition Variables

Review

- What is the API for locks/mutexes?
- How are mutexes implemented?
- What are some rules for using locks in programs?
- What is the difference between a spinlock and a blocking lock?
  - When would you use one vs the other?
- Not losing the forest for the trees:
  - Why do we cover concurrency and synchronization in an OS class?

New Problem: Ping-Pong

Alternate threads working, in pseudocode:

```
void
PingPong() {
   while(not done) {
      …
      if (blue is running)
            switch to purple;
      if (purple is running)
            switch to blue;
   }
}
```

How would we implement using locks?

Incorrect Solution

```
lock = new Lock();

void PingPong() {
      lock.acquire();
      if ( blue ) {
            switch to purple
            lock.release();
      }
      if( purple ) {
            switch to blue
            lock.release();
      }
}
```

We (at the program level) can't literally do the switch/handoff.

It's what we want to do, but we only have locks (so far) …

(Also, need to be careful about where the release is placed.)

Revised Incorrect Solution

```
lock = new Lock();
whichThread = 0;

void PingPong() {
      lock.acquire();
      if ( whichThread == blue ) {
            whichThread = purple;
            switch to purple
      }
      if( whichThread == purple ) {
            whichThread = blue;
            switch to blue

      }
      lock.release();
}
```

Made a flag that says whichThread is running

But, we (still) can't literally do the switch/handoff.

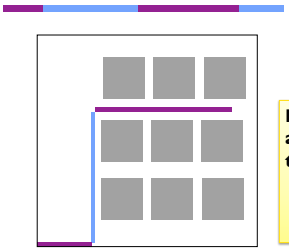
Conclusion: A thread wants to check if some condition is true before continuing execution

Ping-Pong with Mutexes?

```
void
PingPong() {
   while(not done) {
      mx.Acquire();
      …
      mx.Release();
   }
}
```

This solution doesn't work. Why?

Mutexes don't work for ping-pong

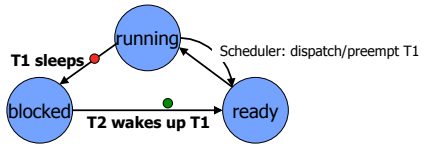


Mutexes can't ensure alternating between the threads.
Ex: Blue could take two turns before Purple gets a turn.

Waiting for conditions

- Need more general synchronization primitives.
- Need some way for a thread to sleep until some other thread wakes it up.
  - Enables explicit signaling over any kind of condition
  - e.g., changes in the program state or state of a shared resource.
- Ideally, threads don't have to know about each other explicitly. They should be able to coordinate around shared objects.

States and transitions for thread T1



Condition variables

- ***Condition variable (CV):*** Data structure to allow thread to check if some condition is true before continuing execution
  - Allows waiting inside a critical section
- CV API
  - `wait`: block until condition becomes true
  - `signal`: signal that the condition is true
    - also called `notify`
    - Wake up one waiting thread
  - May also define a `broadcast` (`notifyAll`)
    - Signal all waiting threads

Condition variables' Mutex

- Every CV is bound to exactly one mutex, which is necessary for safe use of the CV.
  - The mutex protects shared state associated with the condition
  - Mutex is locked when `wait()` is called

(A mutex may have any # of CVs bound to it.)

Condition variable operations

Lock always held

```
wait ( lock ) {
  release lock
  put thread on wait queue
  go to sleep
  // after wake up
  acquire lock
}
```

Atomic

Lock always held

Lock usually held

```
signal () {
  wakeup one waiter (if any)
}
```

Atomic

Lock usually held

```
broadcast () {
  wakeup all waiters (if any)
}
```

Atomic

Ping-Pong using a condition variable

```
void
PingPong() {
   mx.acquire();
   while(not done) {
      while(!myTurn)
          cv.wait(mx);
      do stuff;
      cv.signal();
    }
    mx.release();
}
```

```
wait (lock){
  release lock
  put thread on wait queue
  go to sleep
  // after wake up
  acquire lock
}
```

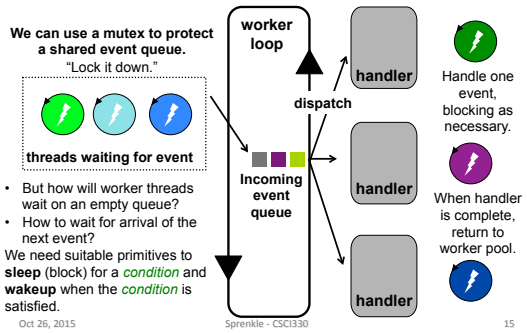
```
signal (){
  wakeup one waiter (if any)
}
```

Waiting for conditions

- Use condition variables (CVs) to represent any condition in your program
  - Q empty, buffer full, op complete, resource ready…
- Associate the condition variable with the mutex that protects the state relating to that condition.
  - CVs are not variables. But you can associate them with whatever data you want, i.e, the state protected by its mutex.
- A caller of CV `wait` must hold its mutex
  - Crucial: a waiter waits on a logical condition and knows that it won't change until the waiter is safely asleep.
  - Otherwise, due to nondeterminism, another thread could change the condition and signal before the waiter is asleep.
    - The waiter would sleep forever: the missed wakeup or wake-up waiter problem.
- `wait` ***atomically*** releases the mutex to sleep, and reacquires it before returning.

CLASSIC PROBLEMS

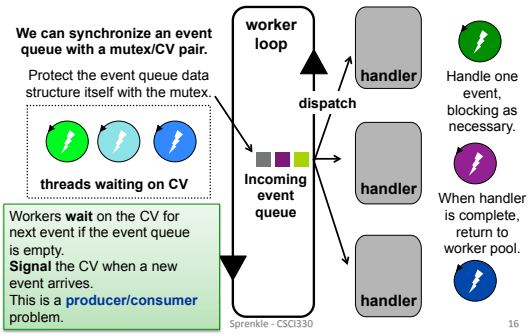
Example: event/request queue



- But how will worker threads wait on an empty queue?
- How to wait for arrival of the next event?

We need suitable primitives to **sleep** (block) for a *condition* and **wakeup** when the *condition* is satisfied.

Example: event/request queue



Workers **wait** on the CV for next event if the event queue is empty.
Signal the CV when a new event arrives.
This is a **producer/consumer** problem.

Bounded-Buffer Problem

- Have a producer thread creating the items
- Have a consumer thread consuming the items
- Example: Soda machine
  - Producer adds a soda
  - Consumer removes a soda

```
consumer () {
  take a soda from machine
}
```

```
producer () {
  add one soda to machine
}
```


Solving producer-consumer

- What are the variables/shared state?
- Locks?
- Mutual exclusion?
- Ordering constraints?

Solving producer-consumer

- What are the variables/shared state?
  - Soda machine buffer
  - Number of sodas in machine (≤ MaxSodas)
- Locks?
  - 1 to protect all shared state (sodaLock)
- Mutual exclusion?
  - Only one thread can manipulate machine at a time
- Ordering constraints?
  - Consumer must wait if machine is empty (CV hasSoda)
  - Producer must wait if machine is full (CV hasRoom)

Producer-consumer code

```
consumer () {
  lock

  take a soda from machine

  unlock
}
```

```
producer () {
  lock

  add one soda to machine

  unlock
}
```


Producer-consumer code

```
consumer () {
  lock
  wait if empty

  take a soda from machine

  notify (not full)
  unlock
}
```

```
producer () {
  lock
  wait if full

  add one soda to machine

  notify (not empty)
  unlock
}
```

Iterating towards a solution!

More Wednesday!

Looking Ahead

- More synchronization mechanisms
  - Semaphores
- More synchronization problems
  - Dining Philosophers
- Still working through Chapter 5 of text book
- Project 3: next Wednesday