Today

- Synchronization Problem: Taking turns
- Synchronization Mechanisms
  - Condition Variables
  - Semaphores

Project 3 Development Recommendation

- At least after every step, git commit and push your code
- VM issues → don't lose (as much of) your owrk

Review

- Why do we need locks?
- What are 3 different ways to implement locks?
  - How do we evaluate implementations?
  - What are the implementations' tradeoffs?
  - What did we need to be able to implement them?

Review: Evaluating Lock Implementations

- Mutual Exclusion
- Performance
- Fairness

Review: Implementing Locks Summary

- Disabling Interrupts
  - Not practical on multiprocessor systems
- Spin Locks
  - Need: hardware support – atomic RMW operation
  - Useful for locks on short critical sections, won't be preempted/blocked (e.g., in kernel)
    - Good in with multiple processors; context switch may be more expensive than burning CPU in busy/wait
  - No fairness guarantees
- Blocking/Queueing Locks
  - Need: hardware support – atomic RMW operation
  - Need: OS Support – maintaining waiting queue ←overhead
  - Less unproductive use of CPU; closer to fairness

Why do we need a RMW operation?

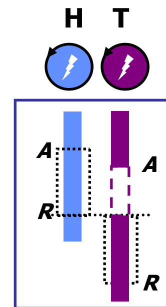
kernel waiting queue may be locked with a spin lock

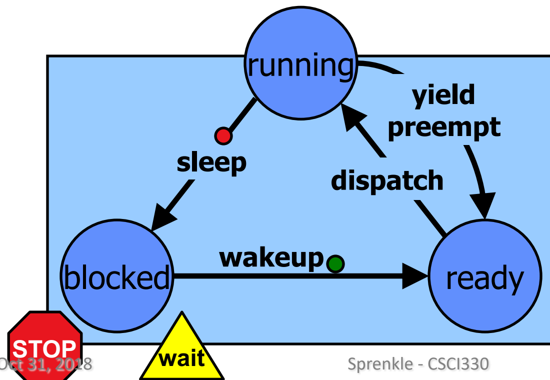
Review: Hardware Support

- To implement mutual exclusion, we need support with a "magic toehold"
  - **Lock primitives themselves have critical sections to test and/or set the lock flags.**
- Safe mutual exclusion on multicore systems requires some hardware support: ***atomic instructions***
  - Examples: test-and-set, compare-and-swap, fetch-and-add.
    - Perform an ***atomic*** *read-modify-write* of a memory location
    - Expensive but necessary
  - If we have any of those atomic instructions, we can build higher-level synchronization objects.

Review: Locking with blocking

T calls acquire and enters the kernel (via syscall) to block because H has the lock.
T sleeps in the kernel to wait for the contended lock.
When the lock holder **H** *releases*, **H** enters the kernel (via syscall) to wakeup a waiting thread (e.g., **T**).





H can block too, perhaps for some other resource.
H doesn't implicitly release the lock just because it blocks.

Lock Implementation Concerns

- What happens if thread dies while holding the lock?
- Priority Inversion Problem
  - A lower priority thread holds the [spin] lock and keeps getting preempted because a higher-priority thread wants the lock
- Even with lock queues, no guarantee that the first waiting thread will get the lock

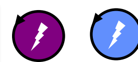
We may return to these issues later…

PING PONG

New Problem: Ping Pong

Alternate threads working, in pseudocode:

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

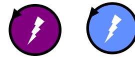


How would we implement using locks?

Note that, at the program level,
we cannot say which thread to switch to

Ping Pong with Mutexes?

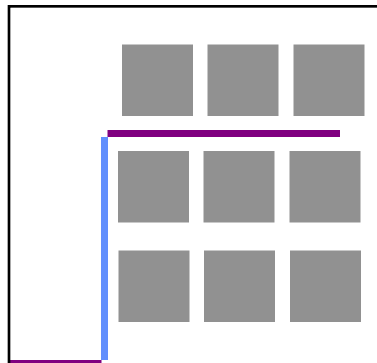
```
void PingPong() {
   while(not done) {
      mx->Acquire();
      …
      mx->Release();
   }
}
```



This solution doesn't work. Why?

Oct 31, 2018 Sprenkle - CSCI330 13

Mutexes Don't Work for Ping Pong



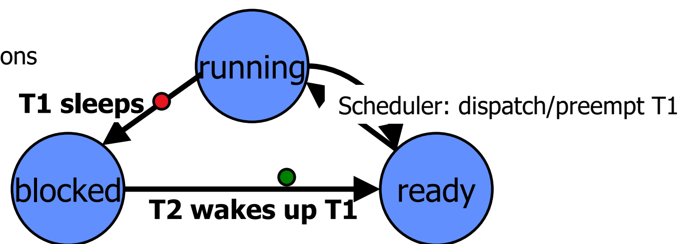
Mutexes can't ensure *alternating* between the threads.

Ex: Blue could take two turns before Purple gets a turn.

Oct 31, 2018 Sprenkle - CSCI330 14

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**



Condition Variables

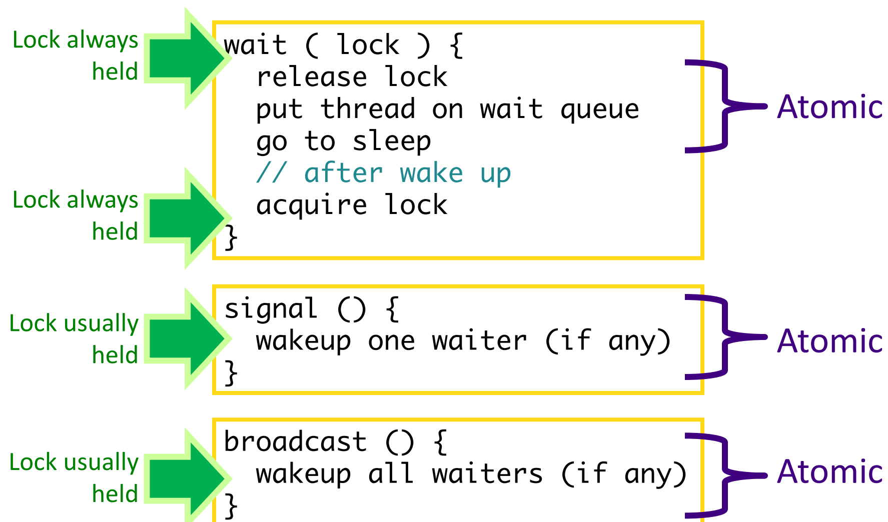
- ***Condition variable (CV):*** Data structure that allows thread to check if some condition is true before continuing execution
  - Allows waiting ***inside*** a critical section
- Condition Variable API
  - **`wait`**: block until condition becomes true
  - **`signal`**: signal that the condition 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.)

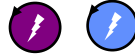
Condition Variable Operations



Ping Pong using a Condition Variable

```
turn = purple; cv = new ConditionVariable();
mx = new Lock();
```

```
void
PingPong() {
   mx.acquire();
   while(not done) {
      while(turn !=
purple)
         cv.wait(mx);
      do stuff;
      turn = blue;
      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)
}
```

Blue's code is similar, with change to turn.

Ping Pong using a Condition Variable

```
turn = purple; cv = new ConditionVariable();
mx = new Lock();
```

```
void
PingPong() {
   mx.acquire();
   while(not done) {
      while(turn !=
purple)
         cv.wait(mx);
      do stuff;
      turn = blue;
      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)
}
```

If blue calls cv.signal(), purple doesn't immediately run. Why?

Waiting for Conditions

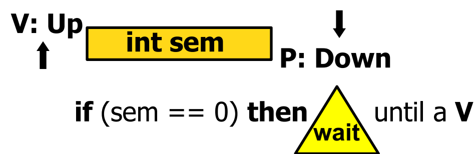
- Use condition variables (CVs) to represent any condition in your program
  - Queue empty, buffer full, op complete, resource ready…
- Associate the condition variable with the mutex that protects the state relating to that condition.
  - CVs are not variables. But you can associate them with whatever 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.

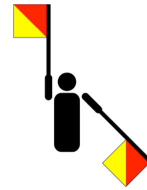
Another synchronization mechanism

SEMAPHORE

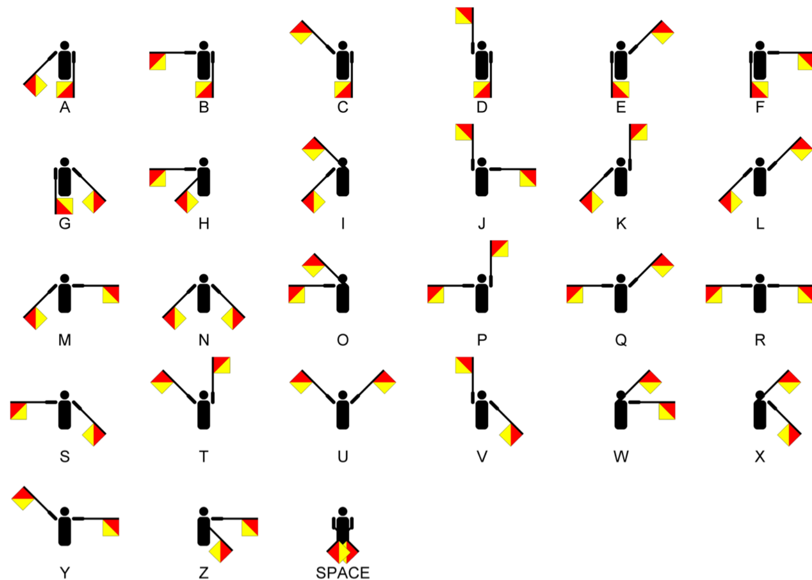
Semaphore

- A ***semaphore*** is a hidden atomic integer counter with only increment/up (V) and decrement/down (P) operations.
  - Book calls V `signal` and P `wait`
- Decrement blocks *iff* the count is zero.
- Semaphores handle all of your synchronization needs with one *elegant* but *confusing* abstraction.



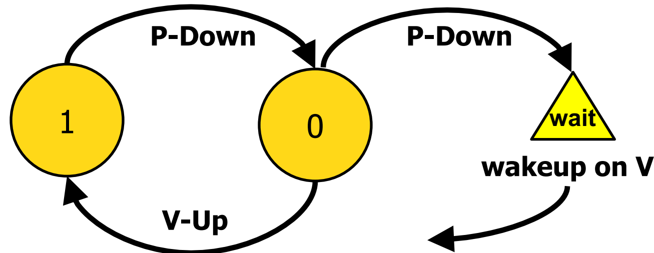


Semaphore - Flag Signals



Example: Binary Semaphore

- A binary semaphore takes only values 0 and 1.
- Requires a usage constraint:
  the set of threads using the semaphore
  call P and V in strict alternation.
  - Never two Vs in a row.

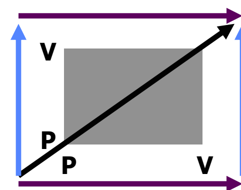


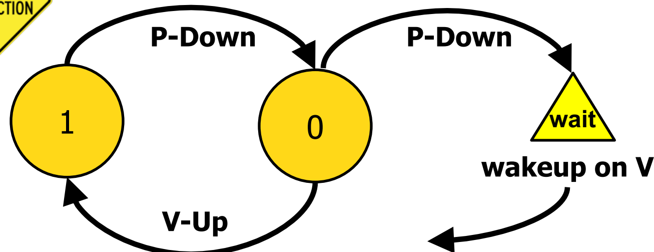
Typical initialization:
```
Semaphore s = new Semaphore(1);
```


A Mutex is a Binary Semaphore

A ***mutex*** is a binary semaphore with an initial value of 1, for which each thread calls **P-V** in strict pairs.

Once a thread A completes its **P**,
no other thread can **P**
until A does a matching **V**.





Semaphores vs. Mutex

- A binary semaphore is similar to a mutex, but …

Semaphores vs. Mutex

- A binary semaphore is similar to a mutex, but …
- Mutex has an *owner*
  - Only the owner can acquire/release the lock
- Semaphores: anyone *could* release the lock

Semaphores vs. Condition Variables

- Semaphores are "prefab CVs" with an atomic integer.
- V(Up) differs from signal (notify) in that …?
- P(Down) differs from wait in that …?

Semaphores vs. Condition Variables

- Semaphores are "prefab CVs" with an atomic integer.
- V(Up) differs from signal (notify) in that:
  - Signal has no effect if no thread is waiting on the condition.
    - Condition variables are not variables! They have no value!
  - Up has the same effect whether or not a thread is waiting.
    - Semaphores retain a "memory" of calls to Up.
- P(Down) differs from wait in that:
  - Down checks the condition and blocks only if necessary.
    - No need to recheck the condition after returning from Down.
    - The wait condition is defined internally, but is limited to a counter.
  - Wait is explicit: it does not check the condition itself, ever.
    - Condition is defined externally and protected by integrated mutex.

Ping Pong with Semaphores

How would we implement Ping Pong with Semaphores?

Ping Pong with Semaphores

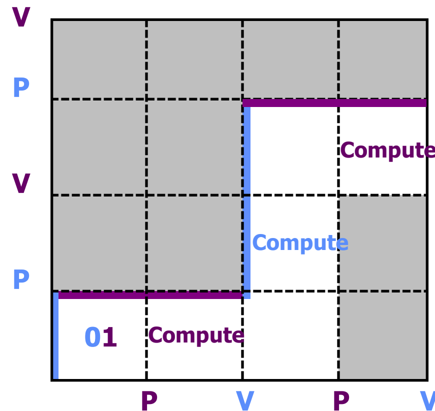
```
blue = Sempahore(0);
purple = Semaphore(1);
```

```
void
PingPong() {
    while(not done) {
        blue.P();
        Compute();
        purple.V();
    }
}
```

```
void
PingPong() {
    while(not done) {
        purple.P();
        Compute();
        blue.V();
    }
}
```

Ping Pong with Semaphores

The threads compute in strict alternation.



Ping Pong with Semaphores

```
blue = Sempahore(0);
purple = Semaphore(1);
```

```
void
PingPong() {
    while(not done) {
        blue.P();
        Compute();
        purple.V();
    }
}
```

```
void
PingPong() {
    while(not done) {
        purple.P();
        Compute();
        blue.V();
    }
}
```

Looking Ahead

- Project 3 due on Friday!
- Synchronization
  - In Java
  - Classic problems