

#### Spring 2016

**Using Semaphores and Condition Variables** 

# **Higher-Level Synchronization**

- We looked at using locks to provide mutual exclusion
- Locks work, but they have limited semantics
  - Just provide mutual exclusion
- Instead, we want synchronization mechanisms that
  - Block waiters, leave interrupts enabled in critical sections
  - Provide semantics beyond mutual exclusion
- Look at three common high-level mechanisms
  - Semaphores: Modelling resource pools
  - Condition Variables: Modelling uncounted events
  - Monitors: Simplifying complex concurrency control policies with mutexes and condition variables
- Use them to solve common synchronization problems

### Semaphores

- Semaphores are an abstract data type that provide mutual exclusion to critical sections
  - Described by Dijkstra in THE system in 1968
- Semaphores can also be used as atomic counters
  - More later
- Semaphores are "integers" that support two operations:
  - Semaphore::Wait(): decrement, block until semaphore is open
    - » Also P(), after the Dutch word for "try to reduce" (also test, down)
  - Semaphore::Signal: increment, allow another thread to enter
    - » Also V() after the Dutch word for increment, up
  - That's it! No other operations not even just reading its value
- Semaphore safety property: the semaphore value is always greater than or equal to 0

# **Blocking in Semaphores**

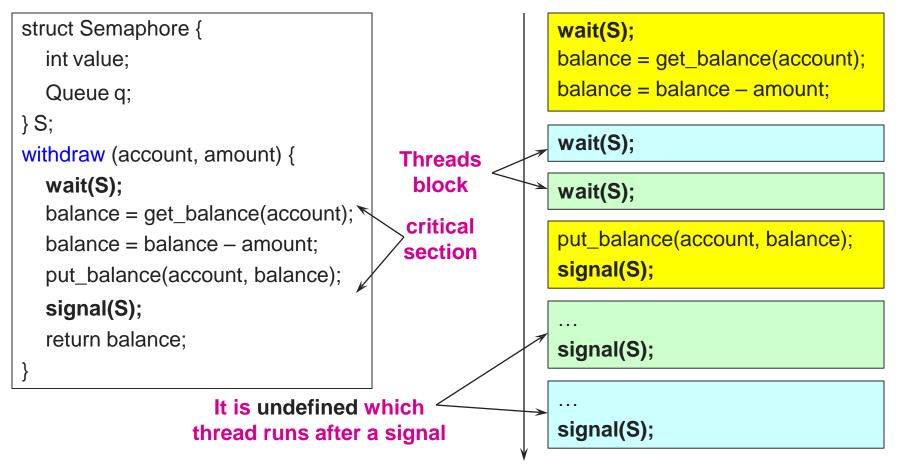
- Associated with each semaphore is a queue of waiting processes
- When wait() is called by a thread:
  - If semaphore is open, thread continues
  - If semaphore is closed, thread blocks on queue
- Then signal() opens the semaphore:
  - If a thread is waiting on the queue, the thread is unblocked
  - If no threads are waiting on the queue, the signal is remembered for the next thread
    - » In other words, signal() has "history" (c.f., condition vars later)
    - » This "history" is a counter

## **Semaphore Types**

- Semaphores come in two types
- Mutex semaphore (or binary semaphore)
  - Represents single access to a resource
  - Guarantees mutual exclusion to a critical section
- Counting semaphore (or general semaphore)
  - Represents a resource with many units available, or a resource that allows certain kinds of unsynchronized concurrent access (e.g., reading)
  - Multiple threads can pass the semaphore
  - Number of threads determined by the semaphore "count"
    - » mutex has count = 1, counting has count = N

# **Using Semaphores**

• Use is similar to our locks, but semantics are different



# **Semaphores in Nachos**

```
P () { // wait
    Disable interrupts;
if (value == 0) {
    add currentThread to waitQueue;
    KThread.sleep(); // currentThread
    }
    value = value - 1;
    Enable interrupts;
}
```

```
V () { // signal

Disable interrupts;

thread = get next on waitQueue;

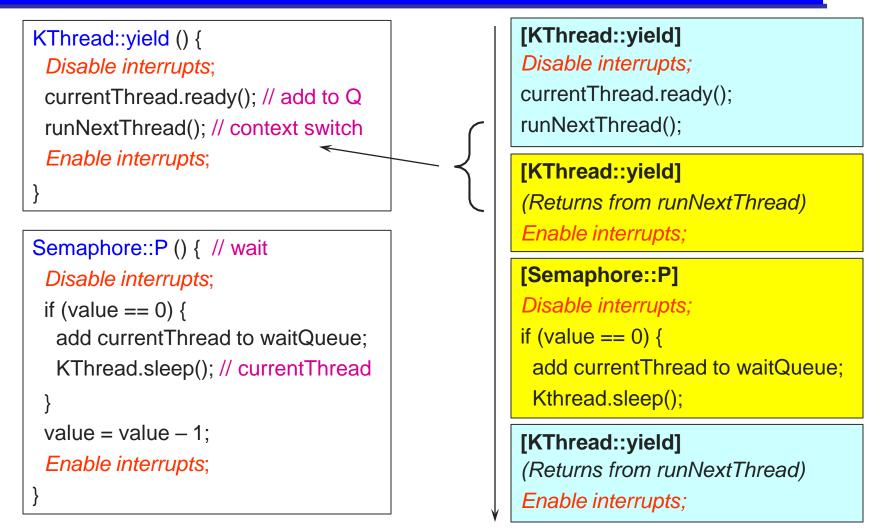
thread.ready();

value = value + 1;

Enable interrupts;
```

- To reference current thread: KThread.currentThread()
- KThread.sleep() assumes interrupts are disabled
  - Note that interrupts are disabled only to enter/leave critical section
  - How can it sleep with interrupts disabled?

## Interrupts Disabled During Context Switch



## **Get Operation – Nope!**

- We can't add a get() to get the value from a semaphore
- It could change between when we get() it and when we try to use what we got
- It can go up via a V() or down via a P()
- It would necessarily be useless and harmful if used.
- Students ask, "What if a stale value is okay?"
  - Just make a call to Random.randInt() instead!
  - Since whatever you could get from the semaphore could get incremented or decremented any number of times before use, a random number within the proper range really, really, really is just as good.

## **Semaphore Questions**

- Are there any problems that can be solved with counting semaphores that cannot be solved with mutex semaphores?
- Does it matter which thread is unblocked by a signal operation?
  - Hint: consider the following three processes sharing a semaphore mutex that is initially 1:

while (1) { wait(mutex); // in critical section signal(mutex); while (1) {

wait(mutex);
// in critical section
signal(mutex);

while (1) }
 wait(mutex);
 // in critical section
 signal(mutex);

#### **Counting vs Binary Semaphores**

- All semaphores are counting
  - If V()ed they will increment
  - If P()ed they will decrement
- When semaphores are intended to move between 0 and 1 we call them binary semaphores
  - But, if we break this discipline they will count.
  - There is no safety built-in
- Binary semaphores can "count" the availability of a mutually exclusive critical section, i.e. from 1 available to 0 available and vice-versa

# Semaphore Summary

- Semaphores can be used to solve any of the traditional synchronization problems
- However, they have some drawbacks
  - They are essentially shared global variables
    - » Can potentially be accessed anywhere in program
  - No connection between the semaphore and the data being controlled by the semaphore
  - Used both for critical sections (mutual exclusion) and coordination (scheduling)
    - » Note that I had to use comments in the code to distinguish
  - No control or guarantee of proper usage
- Sometimes hard to use and prone to bugs
  - Another approach: Use programming language support

## **Condition Variables**

- Condition variables provide a way to wait for events
- Unlike semaphores, they do not count or otherwise track resources
- Combined with mutexes, they can be used to manage resource pools.
  - We'll talk about, for example, using them to construct semaphores and higher-level monitors
- Condition variables are not boolean objects
  - "if (condition\_variable) then" ... does not make sense
  - "if (num\_resources == 0) then wait(resources\_available)" does
  - An example will make this more clear

## **Condition Variables**

- Condition variables support three operations:
- Wait (Mutex m) add calling thread to the condition variable's queue and put the thread to sleep
- Signal (Mutex m) remove a thread, if any, from the condition variable's queue and wake it up
- Broadcast (Mutex m) remove and wake-up all threads in the condition variables queue

## Why the mutex?

```
if (predicate) {
    .----context switch-----> if (predicate) {
        cv.wait()
    }
}
```

The test of the predicate and the wait need to be atomic or two or more threads can end up skipping the wait and entering The critical section

```
mutex_acquire (m)
if (predicate) {
    cv.wait(m)
}
// possible critical section
mutex release(m)
```

*Note*: cv.wait() must atomically block the calling process *and release the mutex* or other threads can't acquire it – including to signal. It must also require it before returning to put things back as found.

Don't make a habit if the *if* quite yet. See next slide.

## About that if...don't do that!

```
if (predicate) { -----→ while (predicate) {
    cv.wait() cv.wait()
}
```

We'll talk about the reason more shortly. But, the short version is that, we might see a sequence such as (i) a thread wait()s for a condition, (ii) a thread gets dispatched and then pre-empted, (iii) another thread signals and wakes up the wait()ing thread, placing it into the runnable list -- behind the pre-empted thread (iii) before the awoken thread gets to run, the pre-empted thread runs and changes the predicate condition, e.g. consumes some available resources.

In this case, a while is needed to ensure the predicate invariant holds.

In some cv implementations or uses an if might be safe. But, it is terrible practice. A change in library, software port, or misunderstanding can break things.

A "while" is ALWAYS safe. An "if" offers no gain – just risk. Just make "while" your habit.



#### Mutex mx;

```
GetLock (condition cv, mutex mx) {
    mutex_acquire (mx);
    while (LOCKED)
        wait (cv,mx)
        ;
    lock=LOCKED;
    mutex_release (mx);
}
```

# **Typical Use (cont.)**

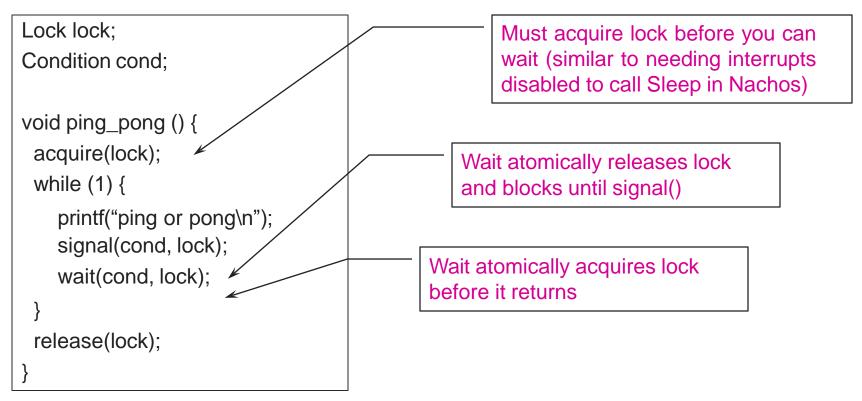
ReleaseLock (condition cv, mutex mx)

```
{
mutex_acquire (mx);
lock = UNLOCKED;
signal (cv);
mutex_release (mx);
```

}

# **Using Cond Vars & Locks**

- Alternation of two threads (ping-pong)
- Each executes the following:



# **Condition Vars != Semaphores**

- Condition variables != semaphores
  - Although their operations have the same names, they have entirely different semantics (such is life, worse yet to come)
  - However, they each can be used to implement the other
- Access to the monitor is controlled by a lock
  - wait() blocks the calling thread, and gives up the lock
    - » To call wait, the thread has to be in the monitor (hence has lock)
    - » Semaphore::wait just blocks the thread on the queue
  - signal() causes a waiting thread to wake up
    - » If there is no waiting thread, the signal is lost
    - » Semaphore::signal increases the semaphore count, allowing future entry even if no thread is waiting
    - » Condition variables have no history



- Semaphores
  - wait()/signal() implement blocking mutual exclusion
  - Model a resource pool
- Condition variables
  - Waits for events
  - Does not count
  - Requires a mutex to protect predicate-wait sequence