Implementing Synchronization

Synchronization Summary

- Use consistent structure
- Always use locks and condition variables when accessing shared data
- Always acquire lock at beginning of procedure, release at end
- Always hold lock when using a condition variable
- Always wait in while loop
- Never spin in sleep()

Main Points

- Implementing locks and CV's using atomic read-modify-write instructions
- Hansen vs. Hoare semantics
  - How to implement one with the other
- Semaphores
  - How to implement condition variables using semaphores

Big Picture: Linux

Concurrent Applications

Semaphores | Locks | Condition Variables

Interrupt Disable | Atomic Read/Modify/Write Instructions

Multiple Processors | Hardware Interrupts

Big Picture: Pintos

Concurrent Kernel Data Structures

Locks and Condition Variables

Semaphores

Interrupt Disable

Hardware Interrupts, Uniprocessor

Implementing Synchronization

Take 1: using memory load/store

- See too much milk solution/Peterson's algorithm

Take 2:

lock.acquire() { disable interrupts }

lock.release() { enable interrupts }

Pintos: how we protect the ready list!

Lock Implementation, Uniprocessor

```
LockAcquire(){
  disableInterrupts ();
  if(value == BUSY){
     waiting.add(current TCB);
     suspend();
  } else {
     value = BUSY;
  }
  enableInterrupts ();
}
```

```
LockRelease() {
  disableInterrupts ();
  if (!waiting.Empty()){
    thread = waiting.Remove();
     readyList.Append(thread);
  } else {
     value = FREE;
  }
  enableInterrupts ();
}
```

Multiprocessor

- Read-modify-write instructions
  - Atomically read a value from memory, operate on it, and then write it back to memory
  - Intervening instructions prevented in hardware
- Examples
  - Test and set
  - Intel: xchgb, lock prefix
  - Compare and swap
- Does it matter?
  - Not for implementing locks and condition variables!

Spinlocks

Lock where the processor waits in a loop for the lock to become free

- Assumes lock will be held for a short time
- Used to protect ready list to implement locks

```
SpinlockAcquire() {
  while (testAndSet(&lockValue) == BUSY)
    ;
}
SpinlockRelease() {
  lockValue = FREE;
}
```

Lock Implementation, Multiprocessor

```
LockAcquire(){
  spinLock.Acquire();
  disableInterrupts ();
  if(value == BUSY){
    waiting.add(current TCB);
     suspend();
  } else {
     value = BUSY;
  }
  enableInterrupts ();
  spinLock.Release();
}
```

```
LockRelease() {
  spinLock.Acquire();
  disableInterrupts ();
  if (!waiting.Empty()){
    thread = waiting.Remove();
     readyList.Append(thread);
  } else {
     value = FREE;
  }
  enableInterrupts ();
  spinLock.Release();
}
```

Lock Implementation, Linux

- Fast path
  - If lock is FREE, and no one is waiting, test&set
- Slow path
  - If lock is BUSY or someone is waiting, see previous slide

- User-level locks
  - Fast path: acquire lock using test&set
  - Slow path: system call to kernel, to use kernel lock

Synchronization Equivalence

- Can we implement Hansen condition variables using Hoare semantics?
- Hoare using Hansen?
- Can we implement semaphores using condition variables?
- Can we implement condition variables using semaphores?

Hansen vs. Hoare semantics

- Hansen
  - Signal puts waiter on ready list
  - Signaller keeps lock and processor
- Hoare
  - Signal gives processor and lock to waiter
  - When waiter finishes, processor/lock given back to signaller
  - Nested signals possible!

Bounded Buffer (Hansen)

```
get() {
 lock.acquire();
 while (front == last)
    empty.wait(lock);
 item = buf[front % size]
 front++;
 full.signal(lock);
 lock.release();
 return item;
}
```

```
put(item) {
  lock.acquire();
  while ((last – front) == size)
    full.wait(lock);
  buf[last % size] = item;
  last++;
  empty.signal(lock);
  lock.release();
}
```

Initially: front = last = 0; size is buffer capacity

empty/full are condition variables

Pre/Post Conditions

- What is state of the bounded buffer at lock acquire?
  - front <= last
  - front + buffer size >= last
  - (also true on return from wait)
- Also true at lock release!
- Allows for proof of correctness

FIFO Bounded Buffer (Hoare semantics)

```
get() {
  lock.acquire();
  if (front == last)
      empty.wait(lock);
  item = buf[front % size];
  front++;
  full.signal(lock);
  lock.release();
  return item;
}
```

```
put(item) {
  lock.acquire();
  if ((last – front) == size)
    full.wait(lock);
  buf[last % size] = item;
  last++;
  empty.signal(lock);
// CAREFUL: someone else ran
  lock.release();
}
```

Initially: front = last = 0; size is buffer capacity

empty/full are condition variables

FIFO Bounded Buffer (Hansen semantics)

- Create a condition variable for every waiter
- Queue condition variables (in FIFO order)
- Signal picks the front of the queue to wake up
- CAREFUL if spurious wakeups!

- Easily extends to case where queue is LIFO, priority, priority donation, …
  - With Hoare semantics, not as easy

FIFO Bounded Buffer
(Hansen, put() is similar)

```
get() {
 lock.acquire();
 if (front == last) or
    nextGet.notEmpty() {
    self = new Condition;
    nextGet.Append(self);
    while (front == last)
       self.wait(lock);
    nextGet.Remove(self);
    delete self;
 }
 item = buf[front % size]
 front++;
 if (!nextPut.empty())
    nextPut.first()->signal(lock);
 lock.release();
 return item;
}
```

Initially: front = last = 0; size is buffer capacity

nextGet, nextPut are queues of Condition Variables

Semaphores

- Semaphore has a non-negative integer value
  - P() atomically waits for value to become > 0, then decrements
  - V() atomically increments value (waking up waiter if needed)
- Semaphores are like integers except:
  - Only operations are P and V
  - Operations are atomic
    - If value is 1, two P's will result in value 0 and one waiter
- Semaphores are useful for
  - Unlocked wait: interrupt handler, fork/join

Semaphore Bounded Buffer

```
get() {
  empty.P();
  mutex.P();
  item = buf[front % size]
  front++;
  mutex.V();
  full.V();
  return item;
}
```

```
put(item) {
  full.P();
  mutex.P();
  buf[last % size] = item;
  last++;
  mutex.V();
  empty.V();
}
```

Initially: front = last = 0; size is buffer capacity
empty/full are semaphores

Implementing Condition Variables using Semaphores (Take 1)

```
wait(lock) {
  lock.release();
  sem.P();
  lock.acquire();
}
signal() {
  sem.V();
}
```

Implementing Condition Variables using Semaphores (Take 2)

```
wait(lock) {
  lock.release();
  sem.P();
  lock.acquire();
}
signal() {
  if semaphore is not empty
     sem.V();
}
```

Implementing Condition Variables using Semaphores (Take 3)

```
wait(lock) {
  sem = new Semaphore;
  queue.Append(sem);   // queue of waiting threads
  lock.release();
  sem.P();
  lock.acquire();
}
signal() {
  if !queue.Empty()
     sem = queue.Remove();
     sem.V();            // wake up waiter
}
```