Condition Variables

Main Points

- Definition
  - Condition wait/signal/broadcast
- Design pattern
- Example: bounded buffer

Last Time

- lock_acquire
  - wait until lock is free, then take it
- lock_release
  - release lock, waking up anyone waiting for it

1. At most one lock holder at a time (safety)
2. If no one holding, acquire gets lock (progress)
3. If all lock holders finish and no higher priority waiters, waiter eventually gets lock (progress)

Rules for Using Locks

- Lock is initially free
- Always acquire before accessing shared data structure
  - Beginning of procedure!
- Always release after finishing with shared data
  - End of procedure!
  - DO NOT throw lock for someone else to release
- Never access shared data without lock
  - Danger!

Will this code work?

```
if (p == NULL) {
   lock_acquire(lock);
   if (p == NULL) {
      p = newP();
   }
   release_lock(lock);
}
use p->field1
```

```
newP() {
    p = malloc(sizeof(p));
    p->field1 = …
    p->field2 = …
    return p;
}
```

Example: Bounded Buffer

```
tryget(item) {
  lock.acquire();
  if (front < last) {
    item = buf[front % size]
    front++;
  }
  lock.release();
  return item;
}
```

```
tryput(item) {
  lock.acquire();
  if ((last – front) < size) {
    buf[last % size] = item;
    last++;
  }
  lock.release();
}
```

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

Condition Variables

- Called only when holding a lock

- Wait: atomically release lock and relinquish processor until signalled
- Signal: wake up a waiter, if any
- Broadcast: wake up all waiters, if any

Example: Bounded Buffer

```
get(item) {
 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

Condition Variables

- ALWAYS hold lock when calling wait, signal, broadcast
  - Condition variable is sync FOR shared state
  - ALWAYS hold lock when accessing shared state
- Condition variable is memoryless
  - If signal when no one is waiting, no op
  - If wait before signal, waiter wakes up
- Wait atomically releases lock
  - What if wait, then release?
  - What if release, then wait?

Condition Variables, cont'd

- When a thread is woken up from wait, it may not run immediately
  - Signal/broadcast put thread on ready list
  - When lock is released, anyone might acquire it
- Wait MUST be in a loop

```
while (needToWait())
    condition.Wait(lock);
```

- Simplifies implementation
  - Of condition variables and locks
  - Of code that uses condition variables and locks

Java Manual

When waiting upon a Condition, a “spurious wakeup” is permitted to occur, in general, as a concession to the underlying platform semantics. This has little practical impact on most application programs as a Condition should always be waited upon in a loop, testing the state predicate that is being waited for.

Structured Synchronization

- Identify objects or data structures that can be accessed by multiple threads concurrently
  - In Pintos kernel, everything!
- Add locks to object/module
  - Grab lock on start to every method/procedure
  - Release lock on finish
- If need to wait
  - while(needToWait()) condition.Wait(lock);
  - Do not assume when you wake up, signaller just ran
- If do something that might wake someone up
  - Signal or Broadcast
- Always leave shared state variables in a consistent state
  - When lock is released, or when waiting

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!

FIFO Bounded Buffer (Hoare semantics)

```
get(item) {
 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);
  lock.release();
}
```

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

FIFO Bounded Buffer (Mesa semantics)

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

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

FIFO Bounded Buffer (Mesa semantics)

```
get(item) {
 lock.acquire();
 if (front == last) {
    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

Synchronization Summary

- Use consistent structure
- Always use locks and condition variables
- 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()