Rendezvous

A needs to be sure B has advanced to point L, B needs to be sure A has advanced to L

```
semaphore A_madeit(0);

A_rendezvous_with_B()
{
  sema_up(A_madeit);
  sema_down(B_madeit);
}
```

```
semaphore B_madeit(0);

B_rendezvous_with_A()
{
  sema_up(B_madeit);
  sema_down(A_madeit);
}
```

Waiting for an activity to finish

```
semaphore done_with_task(0);
thread_create(
      do_task,
      (void*)&done_with_task);

sema_down(done_with_task);
// safely access task's results
```

```
void
do_task(void *arg)
{
   semaphore *s = arg;
   /* do the task */
   sema_up(*s);
}
```

Works no matter which thread is scheduled first after thread_create (parent or child)

Elegant solution that avoids the need to share a "have done task" flag between parent & child

Two applications of this technique in Pintos Project 2

- signal successful process startup ("exec") to parent
- signal process completion ("exit") to parent

Condition Variables

Variables used by a monitor for signaling a condition

- a general (programmer-defined) condition, not just integer increment as with semaphores
- The actual condition is typically some boolean predicate of monitor variables, e.g. "buffer.size > 0"

Monitor can have more than one condition variable

Three operations:

- Wait(): leave monitor, wait for condition to be signaled, reenter monitor
- Signal(): signal one thread waiting on condition
- Broadcast(): signal all threads waiting on condition

Implementing Condition Variables

State is just a queue of waiters:

- Wait(): adds current thread to (end of queue) & block
- Signal(): pick one thread from queue & unblock it
  - Hoare-style Monitors: gives lock directly to waiter
  - Mesa-style monitors (C, Pintos, Java): signaler keeps lock – waiter gets READY, but can’t enter until signaler gives up lock
- Broadcast(): unblock all threads

Compare to semaphores:

- Condition variable signals are lost if nobody’s on the queue (semaphore’s V() are remembered)
- Condition variable wait() always blocks (semaphore’s P() may or may not block)

Monitors

A monitor combines a set of shared variables & operations to access them

- Think of an enhanced C++ class with no public fields

A monitor provides implicit synchronization (only one thread can access private variables simultaneously)

- Single lock is used to ensure all code associated with monitor is within critical section

A monitor provides a general signaling facility

- Wait/Signal pattern (similar to, but different from semaphores)
- May declare & maintain multiple signaling queues

Monitors (cont'd)

Classic monitors are embedded in programming language
- Invented by Hoare & Brinch-Hansen 1972/73
- First used in Mesa/Cedar System @ Xerox PARC 1978
- Limited version available in Java/C#

(Classic) Monitors are safer than semaphores
- can't forget to lock data – compiler checks this

In contemporary C, monitors are a *synchronization pattern* that is achieved using locks & condition variables
- Must understand monitor abstraction to use it

Infinite Buffer w/ Monitor

```
monitor buffer {
    /* implied: struct lock mlock;*/
private:
    char buffer[];
    int head, tail;
public:
    produce(item);
    item consume();
}
```

```
buffer::produce(item i)
{  /* try { lock_acquire(&mlock); */
   buffer[head++] = i;
   /* } finally {lock_release(&mlock);} */
}

buffer::consume()
{  /* try { lock_acquire(&mlock); */
   return buffer[tail++];
    /* } finally {lock_release(&mlock);} */
}
```

Monitors provide implicit protection for their internal variables

- Still need to add the signaling part

Bounded Buffer w/ Monitor

```
monitor buffer {
      condition items_avail;
      condition slots_avail;
private:
      char buffer[];
      int head, tail;
public:
      produce(item);
      item consume();
}
```

```
buffer::produce(item i)
{
  while ((tail+1–head)%CAPACITY==0)
     slots_avail.wait();
  buffer[head++] = i;
  items_avail.signal();
}
buffer::consume()
{
   while (head == tail)
     items_avail.wait();
   item i = buffer[tail++];
   slots_avail.signal();
   return i;
}
```

Bounded Buffer w/ Monitor

```
monitor buffer {
      condition items_avail;
      condition slots_avail;
private:
      char buffer[];
      int head, tail;
public:
      produce(item);
      item consume();
}
```

```
buffer::produce(item i)
{
  while ((tail+1–head)%CAPACITY==0)
     slots_avail.wait();
  buffer[head++] = i;
  items_avail.signal();
}
buffer::consume()
{
   while (head == tail)
     items_avail.wait();
   item i = buffer[ta
   slots_avail.si
   return i;
}
```

```
lock_release(&mlock);
block_on(items_avail);
lock_acquire(&mlock);
```

Q1.: How is lost update problem avoided?

Q2.: Why *while()* and not *if()*?

Monitors in C

POSIX Threads & Pintos

No compiler support, must do it manually

- must declare locks & condition vars
- must call lock_acquire/lock_release when entering&leaving the monitor
- must use cond_wait/cond_signal to wait for/signal condition

Note: cond_wait(&c, &m) takes monitor lock as parameter

- necessary so monitor can be left & reentered without losing signals

Pintos cond_signal() takes lock as well

- only as debugging help/assertion to check lock is held when signaling
- pthread_cond_signal() does not

Mesa vs Hoare Style

Mesa-style:

- Cond_signal leaves signaling thread in monitor
- so must always use “while()” when checking loop condition
- POSIX Threads & Pintos are Mesa-style (and so are C# & Java)

Alternative is “Hoare”-style where cond_signal leads to exit from monitor and immediate reentry of waiter

- Not commonly used

Monitors in Java

synchronized *block* means

- enter monitor
- *execute block*
- leave monitor

wait()/notify() use condition variable associated with receiver

- Every object in Java can function as a condition var

```
class buffer {
  private char buffer[];
  private int head, tail;
  public synchronized produce(item i) {
    while (buffer_full())
        this.wait();
     buffer[head++] = i;
     this.notify();
  }
  public synchronized item consume() {
    while (buffer_empty())
        this.wait();
     buffer[tail++] = i;
     this.notify();
  }
}
```

Per Brinch Hansen's Criticism

See *Java's Insecure Parallelism* [Brinch Hansen 1999]

Says Java abused concept of monitors because Java does not *require* all accesses to shared variables to be within monitors

Why did designers of Java not follow his lead?

- Performance: compiler can't easily decide if object is local or not - conservatively, would have to make all public methods synchronized – pay at least cost of atomic instruction on entering every time

High vs Low Level Synchronization

As we've seen, bounded buffer can be solved with higher-level synchronization primitives

- semaphores and monitors

In Pintos kernel, one could also use thread_block/unblock directly

- this is not always efficiently possible in other concurrent environments

Q.: when should you use low-level synchronization (a la thread_block/thread_unblock) and when should you prefer higher-level synchronization?

A.: Except for the simplest scenarios, higher-level synchronization abstractions are always preferable

- They're well understood; make it possible to reason about code.