Operating Systems
(1DT020 & 1TT802)

Lecture 6
Process synchronisation : Hardware support , Semaphores, Monitors, and Condition Variables

April 22, 2008

Léon Mugwaneza

http://www.it.uu.se/edu/course/homepage/os/vt08

Review: Synchronization problem with Threads

- **One thread per transaction, each running:**

```
Deposit(acctId, amount) {
  acct = GetAccount(actId); /* May use disk I/O */
  acct->balance += amount;
  StoreAccount(acct);      /* Involves disk I/O */
}
```

- **Unfortunately, shared state can get corrupted:**

```
        Thread 1                          Thread 2
load r1, acct->balance
                             load r1, acct->balance
                             add r1, amount2
                             store r1, acct->balance
add r1, amount1
store r1, acct->balance
```

- **Atomic Operation: an operation that always runs to completion or not at all**
  - **It is *indivisible:* it cannot be stopped in the middle and state cannot be modified by someone else in the middle**
- **Race Condition: outcome depends on process interleaving**

Review: Too Much Milk Solution #3

- **Here is a possible two-note solution:**

Thread A

```
leave note A;
while (note B) {//X
   do nothing;
}
if (noMilk) {
   buy milk;
}
remove note A;
```

Thread B

```
leave note B;
if (noNote A) {//Y
   if (noMilk) {
      buy milk;
   }
}
remove note B;
```

- **Does this work? Yes. Both can guarantee that:**
  - It is safe to buy, or
  - Other will buy, ok to quit
- **At X:**
  - if no note B, safe for A to buy,
  - otherwise wait to find out what will happen
- **At Y:**
  - if no note A, safe for B to buy
  - Otherwise, A is either buying or waiting for B to quit

Review: Solution #3 discussion

- Our solution protects a single "Critical-Section" piece of code for each thread:

```
if (noMilk) {
   buy milk;
}
```

- Solution #3 works, but it's really unsatisfactory
  - Really complex – even for this simple an example
    - Hard to convince yourself that this really works
  - A's code is different from B's – what if lots of threads?
    - Code would have to be slightly different for each thread
  - While A is waiting, it is consuming CPU time
    - This is called "busy-waiting"
- There's a better way
  - Have hardware provide better (higher-level) primitives than atomic load and store
  - Build even higher-level programming abstractions on this new hardware support

Too Much Milk: Solution #4

- **Suppose we have some sort of implementation of a lock (more in the next lecture)**
  - `Lock.Acquire()` **– wait until lock is free, then grab**
  - `Lock.Release()` **– Unlock, waking up anyone waiting**
  - **These must be atomic operations – if two threads are waiting for the lock and both see it's free, only one succeeds to grab the lock**
- **Then, our milk problem is easy:**

```
milkLock.Acquire();
if (noMilk)
   buy milk;
milkLock.Release();
```

- **Once again, section of code between `Acquire()` and `Release()` called a "Critical Section"**
- **Of course, you can make this even simpler: suppose you are out of ice cream instead of milk**
  - **Skip the test since you always need more ice cream**
  - **Only Critical Sections need to be accessed in a synchronized way**

Where are we going with synchronization?

Programs	Cooperating Processes
Higher-level API	Locks Semaphores Monitors Send/Receive
Hardware	Load/Store Disable Ints Test&Set Comp&Swap

- We are going to implement various higher-level synchronization primitives using atomic operations
  - Everything is pretty painful if only atomic primitives are load and store
  - Need to provide primitives useful at user-level

Goals for Today

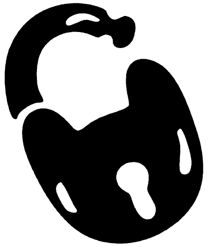
- Hardware Support for Synchronization
- Higher-level Synchronization Abstractions
  - Semaphores, monitors, and condition variables
- Programming paradigms for concurrent programs

Note: Some slides and/or pictures in the following are adapted from slides ©2005 Silberschatz, Galvin, and Gagne, others from Kubiatowicz - CS162 ©UCB Fall 2007 (University of California at Berkeley)

High-Level Picture

- The abstraction of threads is good:
  - Maintains sequential execution model
  - Allows simple parallelism to overlap I/O and computation
- Unfortunately, still too complicated to access state shared between threads
  - Consider “too much milk” example
  - Implementing a concurrent program with only loads and stores would be tricky and error-prone
- Today, we’ll implement higher-level operations on top of atomic operations provided by hardware
  - Develop a “synchronization toolbox”
  - Explore some common programming paradigms

How to implement Locks?



- **Lock**: prevents someone from doing something
  - Lock before entering critical section and before accessing shared data
  - Unlock when leaving, after accessing shared data
  - Wait if locked
    - Important idea: all synchronization involves waiting
    - Should *sleep* if waiting for a long time
- Atomic Load/Store: get solution like Too Much Milk #3
  - Pretty complex and error prone
- Hardware Lock instruction
  - Is this a good idea?
  - Complexity?
    - Done in the Intel 432
    - Each feature makes hardware more complex and slow
  - What about putting a task to sleep?
    - How do you handle the interface between the hardware and scheduler?

Naïve use of Interrupt Enable/Disable

- How can we build multi-instruction atomic operations?
  - Recall: dispatcher gets control in two ways.
    - » Internal : Thread does something to relinquish the CPU (what?)
    - » External: Interrupts cause dispatcher to take CPU
  - On a uniprocessor, can avoid context-switching by:
    - » Avoiding internal events (although virtual memory tricky)
    - » Preventing external events by disabling interrupts
- Consequently, naïve Implementation of locks:

```
LockAcquire() { disable Ints; }
LockRelease() { enable Ints; }
```

☛LockAcquire and LockRelease are system calls (in OS kernel)

- Why ?

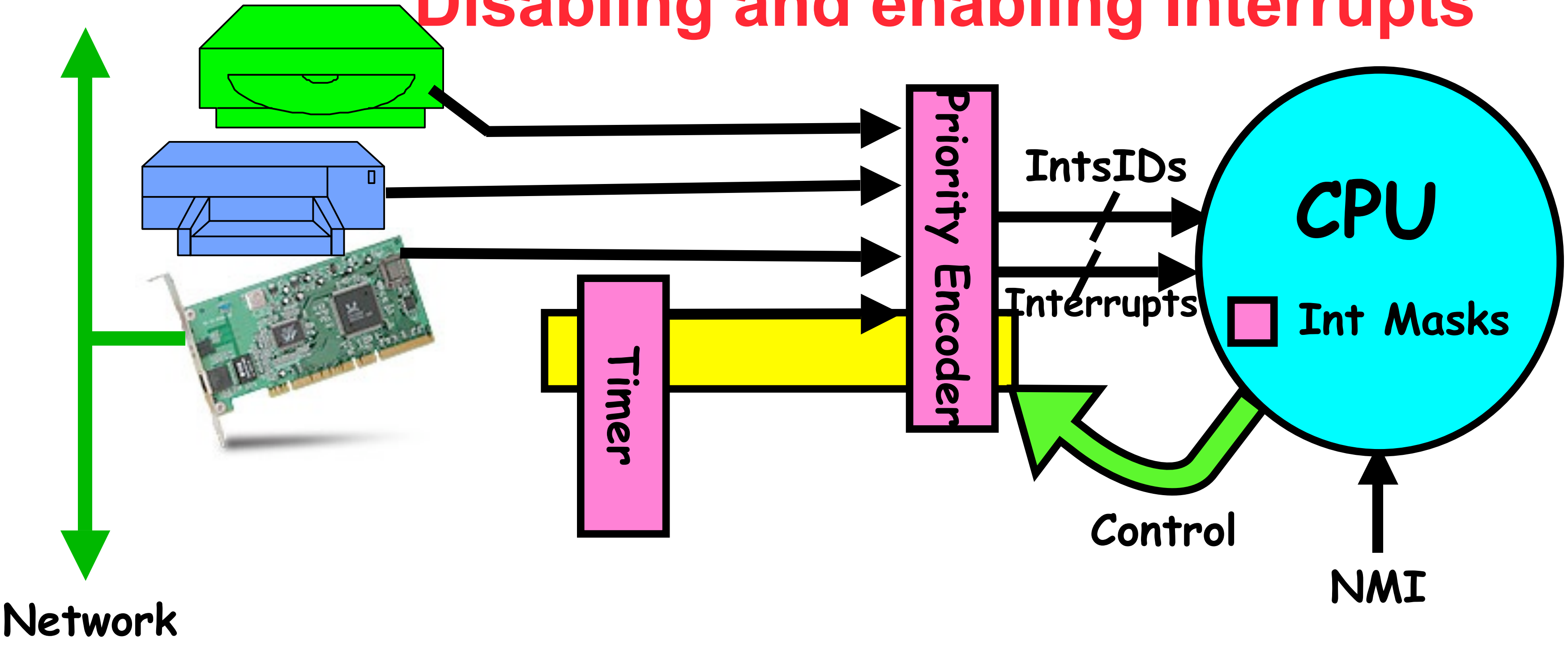
✏Remember how a process calls a procedure in OS kernel ?

- Problems with this approach:
  - Can't let user do this! Consider following:

```
LockAcquire();
While(TRUE) {;}
```

  - Real-Time system—no guarantees on timing!
    - » Critical Sections might be arbitrarily long
  - What happens with I/O or other important events?
    - » "Reactor about to meltdown. Help?"

Disabling and enabling Interrupts



- Interrupts invoked with interrupt lines from devices
- CPU interrupt controller chooses interrupt request to honor
  - Priority encoder picks highest enabled interrupt
  - Interrupt identity specified with ID line
  - Internal Mask flags enable/disable interrupts (mask set/cleared only in kernel mode)
- CPU can configure some devices so as they do not generate interrupts (devices controlled by polling)
- Non-maskable interrupt line (NMI) can't be disabled

Better Implementation of Locks by Disabling Interrupts

- **Key idea: maintain a lock variable and impose mutual exclusion only during operations on that variable**

```
int value = FREE;

Acquire() {
   disable interrupts;
   if (value == BUSY) {
      put thread on wait queue;
      //calling thread sleeping
      update thread state
      call scheduler
      } else {
      value = BUSY;
   }
   enable interrupts;
}
```

```
Release() {
   disable interrupts;
   if (anyone on wait queue) {
      take thread off wait queue
      place thread on ready queue
                } else {
      value = FREE;
   }
   enable interrupts;
}
```

- Locks are provided by the OS (like unix pipes - see lab1)
  - Acquire and Release are system calls, we also need calls to create and close (“kill”) locks
  - Have 1 wait queue and 1 lock variable per lock

New Lock Implementation: Discussion

- **Why do we need to disable interrupts at all?**
  - Avoid interruption between checking and setting lock value
  - Otherwise two threads could think that they both have lock

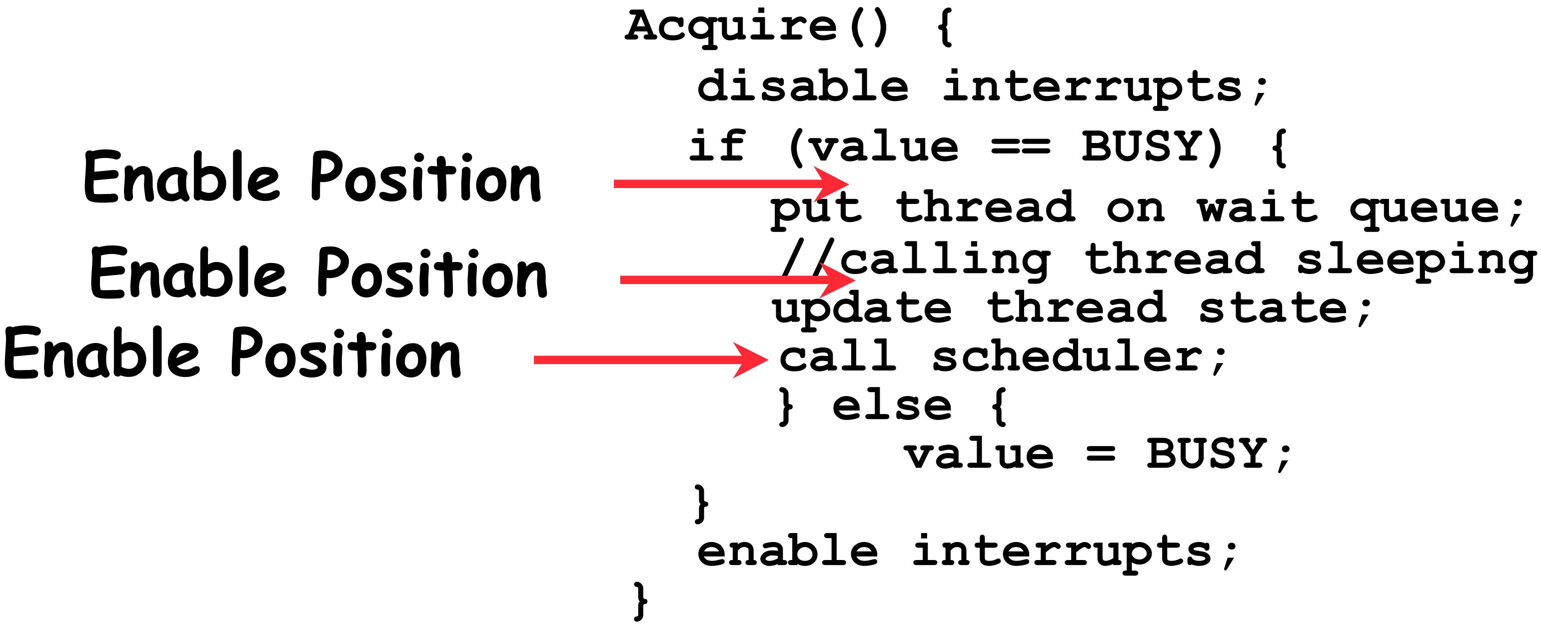
```
Acquire() {
   disable interrupts;
   if (value == BUSY) {
      put thread on wait queue;
      //calling thread sleeping
      update thread state
      call scheduler
      } else {
      value = BUSY;
   }
   enable interrupts;
}
```

Critical Section

- **Note: unlike previous solution, the critical section (inside `Acquire()`) is very short**
  - Users of lock can take as long as they like in their own critical section: doesn’t impact global machine behavior
  - Critical interrupts taken in time!

Interrupt re-enable in “going” to sleep

- **What about re-enabling ints when going to sleep?**



Enable Position

Enable Position

Enable Position

```
Acquire() {
   disable interrupts;
   if (value == BUSY) {
      put thread on wait queue;
      //calling thread sleeping
      update thread state;
      call scheduler;
      } else {
           value = BUSY;
   }
   enable interrupts;
}
```

- **Before putting thread on the wait queue?**
  - **Release can check the queue and not wake up thread**
- **After putting the thread on the wait queue**
  - **Release puts the thread on the ready queue, but Acquire still “thinks” thread needs to go to sleep**
  - **Misses wakeup and still holds lock (deadlock!)**

☛A non-issue : kernel handles this very often.

  - **scheduler will enable interrupts while launching the next thread**

What if a thread does not release lock ?

- What about exceptions that occur after lock is acquired?

```
mylock.acquire();
a = b / 0;
mylock.release()
```

  - Who releases the lock?
- What if thread terminates without releasing the lock?
  - Who releases the lock?
- Os releases lock when threads terminates or error causes thread abortion
  - Are there errors which do not cause process abortion ?
    - » Yes (eg. Page fault - see later)

Atomic Read-Modify-Write instructions

- **Problems with previous solution:**
  - Can't give lock implementation to users
  - Doesn't work well on multiprocessor or multi-core CPU
    - » Disabling interrupts on all processors/cores requires messages and would be very time consuming
- **Alternative: atomic instruction sequences**
  - These instructions read a value from memory and write a new value atomically
  - Hardware is responsible for implementing this correctly
    - » on both uniprocessors (not too hard)
    - » and multiprocessors/multi-core (requires help from cache coherence protocol)
  - Unlike disabling interrupts, can be used on uniprocessors, multiprocessors, and multi-core CPUs

Examples of Read-Modify-Write

- test&set
```
test&set (address, register){ /* most architectures */
     register = M[address];    /* actual inst. slightly ≠*/
     M[address] = 1;           /* 68000: TAS, INTEL : BTS*/ }
```
- swap
```
swap (address, register)
     temp = M[address];
     M[address] = register;
     register = temp;
}
```
- compare&swap
```
compare&swap (address, reg1, reg2) { /* 68000, Sparc */
     if (reg1 == M[address]) {
         M[address] = reg2;
         set bit of CCR=1    //CCR is Condition Code Register
         } else {
         set bit of CCR=0
        }
}
```
  - ☞ compare and exchange on X86
- load-linked AND store conditional
```
load-linked AND store conditional /* R4000, alpha */
   loop:
     ll r1, addr; // Load-linked - read lock : r1<-M[addr]
                  //Remember addr
     movi r2, 1;    // Try to set lock (movi+sc)
     sc r2, addr; //Store conditional(try to do M[addr]<-r2)
     //store only if addr saved by ll not written meanwhile
     beqz r1, loop; // loop if lock read by ll was busy
```

Implementing Locks with test&set

- **Another flawed, but simple solution:**

```
int value = 0; // Free
Acquire() {
   loop: test&set(&value, reg);
         if reg==1 goto loop;   // while (busy) loop;
}
Release() {
   value = 0;
}
```

- **Explanation:**
  - **If lock is free, test&set reads 0 and sets value=1, so lock is now busy. It returns 0 so while exits.**
  - **If lock is busy, test&set reads 1 and sets value=1 (no change). It returns 1, so while loop continues**
  - **When we set value = 0, someone else can get lock**
- **Busy-Waiting: thread consumes cycles while waiting**
  - **Also called spinlocking**

Problem: Busy-Waiting for Lock

- Positives for this solution
  - Machine can receive interrupts
  - No system call (remember system calls have overhead)
  - Works on a multiprocessor/multi-core
- Negatives
  - This is very inefficient because the busy-waiting thread will consume cycles waiting
  - Waiting thread may take cycles away from thread holding lock (no one wins!)
  - Priority Inversion: If busy-waiting thread has higher priority than thread holding lock ⇒ no progress!
- For semaphores and monitors, waiting thread may wait for an arbitrary length of time!
  - Thus even if busy-waiting was OK for locks, definitely not ok for other primitives
  - Good solutions (exams!) should not have busy-waiting!

Better Locks using test&set

- Can we build test&set locks without busy-waiting?
  - Can't entirely, but can minimize!
  - Idea: only busy-wait to atomically check lock value

```
int guard = 0;
int value = FREE;
```

```
Acquire() {
   // Short busy-wait time
   loop : test&set &guard, reg
          if reg==1 goto loop;
   if (value == BUSY) {
       perfom syscall to put
       thread on wait queue,
       And guard = 0
   } else {
      value = BUSY;
      guard = 0;
   }
}
```

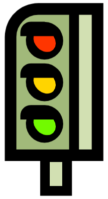
```
Release() {
   // Short busy-wait time
   loop : test&set &guard, reg
          if reg==1 goto loop;
   if anyone on wait queue {
      perfom call to OS to take
      thread off wait queue
      and place it on ready
      queue;
   } else {
      value = FREE;
   }
   guard = 0;
```

- Note: guard variable reset by OS when thread go sleep
  - Why can't we do it just before or just after the syst call to go sleep?

Higher-level Primitives than Locks

- What is the right abstraction for synchronizing threads that share memory?
  - Want as high a level primitive as possible
- Good primitives and practices important!
  - Since execution is not entirely sequential, really hard to find bugs, since they happen rarely
  - UNIX is pretty stable now, but up until about mid-80s (10 years after started), systems running UNIX would crash every week or so – concurrency bugs
- Synchronization is a way of coordinating multiple concurrent activities that are using shared state
  - We will see a couple of ways of structuring the sharing

Semaphores



- Semaphores are a kind of generalized lock
  - First defined by Dijkstra in late 60s
  - Main synchronization primitive used in original UNIX
- Definition: a Semaphore has a non-negative integer value, a wait queue and supports the following two operations (apart from initialization):
  - P(): <u>an atomic operation</u> that does the following:

    ```
    if value = 0 then sleep
    else decrement value by 1
    ```

    - » Course book calls this operation wait()
  - V(): <u>an atomic operation</u> that does the following:

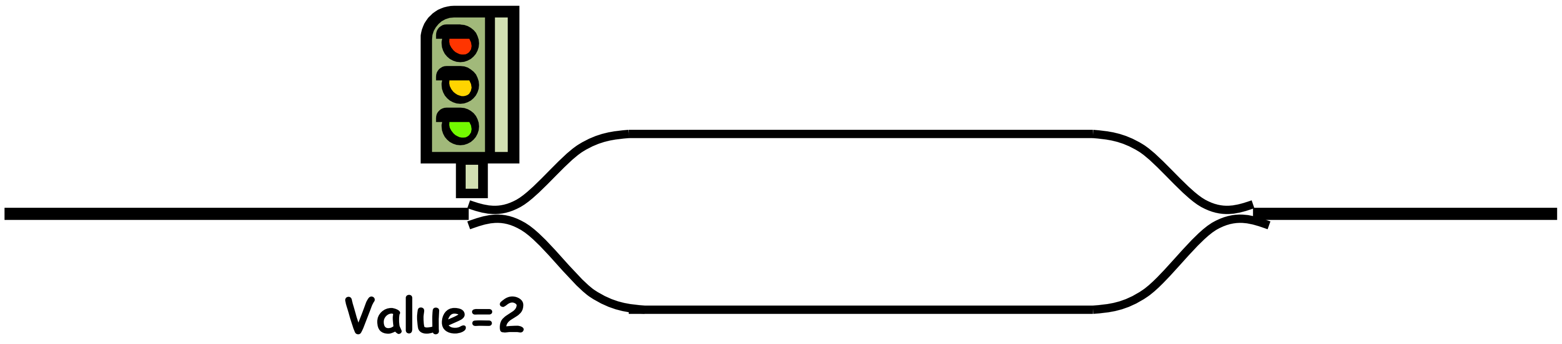
    ```
    if there are any threads sleeping on that
       semaphore, wakeup 1 thread (at random)
    else increment value by 1
    ```

    - » Course book calls this operation signal()
  - Note that P() stands for "*proberen*" (to test) and V() stands for "*verhogen*" (to increment) in Dutch
  - DOWN() sometimes used for P(), and UP() for V()

☛ Some implementations allow negative values (P always decrements value by one, and V always increments value by one)

Semaphores are not integers!

- Semaphores are like integers, except
  - No negative values
  - Only operations allowed are P and V – can't read or write value, except to set it initially
  - Operations must be atomic
    - » Two P's together can't decrement value below zero
    - » Similarly, thread going to sleep in P won't miss wakeup from V – even if they both happen at same time
- Semaphore from railway analogy
  - Here is a semaphore initialized to 2 for resource control:



Two uses of Semaphores

- Mutual Exclusion (initial value = 1)
  - Also called “Binary Semaphore”.
  - Can be used for mutual exclusion:

```
semaphore.P();
// Critical section goes here
semaphore.V();
```

- Scheduling Constraints (initial value = 0)
  - Locks are fine for mutual exclusion, but what if you want a thread to wait for something?
  - Example: suppose you had to implement ThreadJoin which must wait for thread to terminate:

```
Initial value of semaphore = 0
ThreadJoin {
   semaphore.P();
}
ThreadFinish {
   semaphore.V();
}
```

- What if initial value > 1?
  - Counting semaphore : consider a resource with N copies
    - » request a copy using P(), release copy using V()
    - » Scheduling constraints on resource utilization

Producer-consumer with a bounded buffer

- **Problem Definition**
  - Producer puts things into a shared buffer
  - Consumer takes them out
  - Need synchronization to coordinate producer/consumer
- **Don’t want producer and consumer to have to work in lockstep, so put a fixed-size buffer between them**
  - Need to synchronize access to this buffer
  - Producer needs to wait if buffer is full
  - Consumer needs to wait if buffer is empty
- **Example 1: GCC compiler**
  - cpp | cc1 | cc2 | as | ld
- **Example 2: Coke machine**
  - Producer can put limited number of cokes in machine
  - Consumer can’t take cokes out if machine is empty

Correctness constraints for solution

- Correctness Constraints:
  - Consumer must wait for producer to fill buffers, if none full (scheduling constraint)
  - Producer must wait for consumer to empty buffers, if all full (scheduling constraint)
  - Only one thread can manipulate buffer queue at a time (mutual exclusion)
- Remember why we need mutual exclusion
  - Because computers are “not very clever”
  - Imagine if in real life: the delivery person is filling the machine and somebody comes up and tries to stick their money into the machine
- General rule of thumb:
  Use a separate semaphore for each constraint
  - `Semaphore fullBuffers; // consumer's constraint`
  - `Semaphore emptyBuffers;// producer's constraint`
  - `Semaphore mutex;        // mutual exclusion`

Full Solution to Bounded Buffer

```
Semaphore fullBuffer = 0; // Initially, no coke
Semaphore emptyBuffers = num; // Initially, num empty slots
Semaphore mutex = 1;     // No one using machine

Producer(item) {
 While(True){
   do something else;   // incuding producing item
   emptyBuffers.P();   // Wait until space
   mutex.P();          // Wait until buffer free
   Enqueue(item);
   mutex.V();
   fullBuffers.V();   // Tell consumers there is more coke
 }
}

Consumer() {
 While(True){
   fullBuffers.P();     // Check if there's a coke
   mutex.P();           // Wait until machine free
   item = Dequeue();
   mutex.V();
   emptyBuffers.V();    // tell producer a slot is free
   do something else;   // including using item
  }
}
```

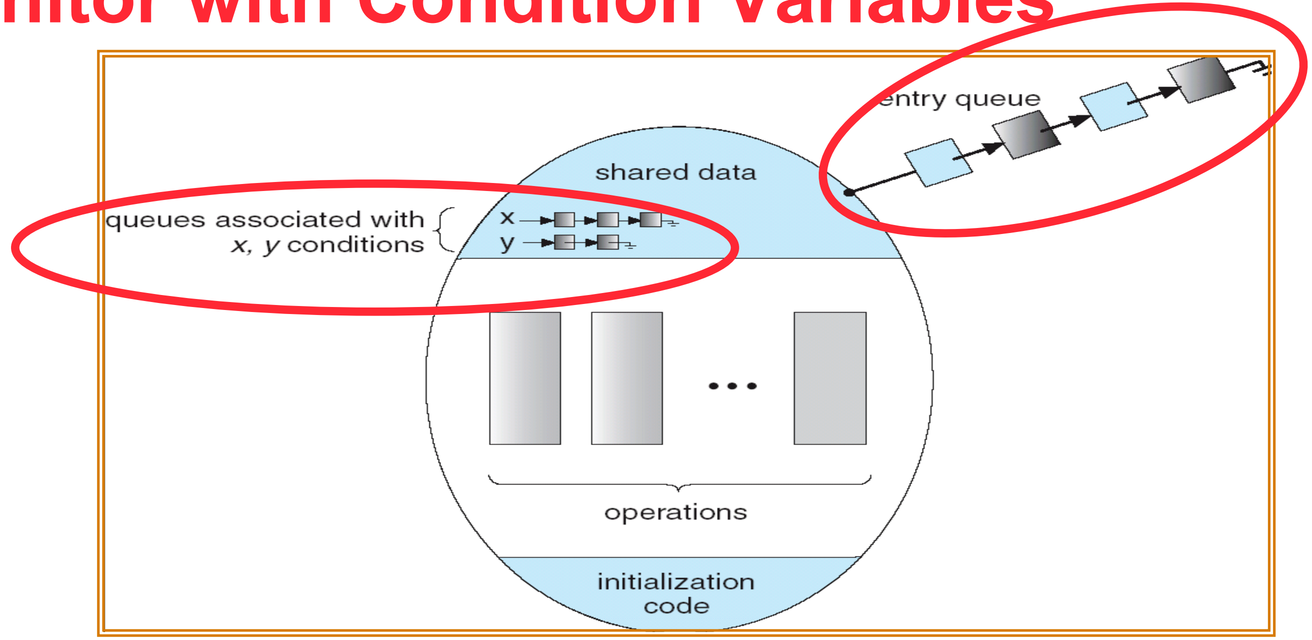
Discussion about Solution

- Why asymmetry?
  - Producer does: `emptyBuffer.P(), fullBuffer.V()`
  - Consumer does: `fullBuffer.P(), emptyBuffer.V()`
- Is order of P’s important?
  - Yes! Can cause deadlock
- Is order of V’s important?
  - No, except that it might affect scheduling efficiency
- What if we have 2 producers or 2 consumers?
  - Do we need to change anything?

Motivation for Monitors and Condition Variables

- **Semaphores are a huge step up; just think of trying to do the bounded buffer with only loads and stores**
  - **Problem is that semaphores are dual purpose:**
    - **They are used for both mutex and scheduling constraints**
    - **Example: the fact that flipping of P's in bounded buffer gives deadlock is not immediately obvious. How do you prove correctness to someone?**
- **Cleaner idea: Use *locks* for mutual exclusion and *condition variables* for scheduling constraints**
- **Monitor: zero or more condition variables for managing concurrent access to shared data, together with operations that are guaranteed to be mutual exclusive**
  - **Monitors are language constructs**
  - **Some languages like Java provide this natively**
  - **Most others use actual locks and condition variables**

Monitor with Condition Variables



- **Lock**: the lock provides mutual exclusion to shared data
    - Always acquire before accessing shared data structure
    - Always release after finishing with shared data
    - Lock initially free
- **Condition Variable**: a queue of threads waiting for something *inside* a critical section
    - Two operations on conditions : condition.wait() and condition.signal()
    - Key idea: make it possible to go to sleep inside critical section by atomically releasing lock at time we go to sleep
    - Contrast to semaphores: Can't wait inside critical section

Simple Monitor Example

- **Here is an (infinite) synchronized queue**

```
Lock lock;
Condition dataready;
Queue queue;

AddToQueue(item) {
   lock.Acquire();              // Get Lock
   queue.enqueue(item);         // Add item
   dataready.signal();          // Signal any waiters
   lock.Release();              // Release Lock
}

RemoveFromQueue() {
   lock.Acquire();              // Get Lock
   while (queue.isEmpty()) {
      dataready.wait(&lock); // If nothing, sleep
                                // and release lock
                                         }
   item = queue.dequeue();      // Get next item
   lock.Release();              // Release Lock
   return(item);
}
```

- **Note: lock, lock.Acquire(), and lock.Release() inserted by compiler**

Summary

- Important concept: Atomic Operations
  - An operation that runs to completion or not at all
  - These are the primitives on which to construct various synchronization primitives
- hardware atomicity primitives:
  - Disabling of Interrupts, test&set, swap, comp&swap, load-linked/store conditional
- Several constructions of Locks
  - Must be very careful not to waste/tie up machine resources
    - » Shouldn’t disable interrupts for long
    - » Shouldn’t spin wait for long
  - Key idea: Separate lock variable, use hardware mechanisms to protect modifications of that variable
- Semaphores, Monitors, and Condition Variables
  - Higher level constructs that are harder to “screw up”