CS162
Operating Systems and Systems Programming
Lecture 7

Synchronization (Con't): Semaphores, Monitors, and Readers/Writers

February 13th, 2020

Prof. John Kubiatowicz

http://cs162.eecs.Berkeley.edu

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
- Solution #3 works, but it's really unsatisfactory
  - Really complex – even for this simple of 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"

Recall: What is a Lock?

- 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
- For example: fix the milk problem by putting a key on the refrigerator
  - Lock it and take key if you are going to go buy milk
  - Fixes too much: roommate angry if only wants OJ

  - Of Course – We don't know how to make a lock yet

Recall: Too Much Milk: Solution #4

- Suppose we have some sort of implementation of a lock
  - `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 ;-)

Recall: Implement 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;
      Go to sleep();
      // Enable interrupts?
   } else {
      value = BUSY;
   }
   enable interrupts;
}
```

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

- Note – Can easily have many locks
  - Use an array of values, for instance!

Recall: How to Re-enable After Sleep()?

- In scheduler, since interrupts are disabled when you call sleep:
  - Responsibility of the next thread to re-enable ints
  - When the sleeping thread wakes up, returns to acquire and re-enables interrupts

Thread A | Thread B

```
Thread A                            Thread B
   .
   .
disable ints
   sleep      --context switch-->  sleep return
                                   enable ints
                                        .
                                        .
                                        .
                                   disable int
                                      sleep
sleep return  <--context switch--
enable ints
   .
   .
```


In-Kernel Lock: Simulation

Value: 0 | waiters | owner | READY

Running: Thread A | Ready: Thread B

Thread A:
```
lock.Acquire();
 …
 critical section;
 …
lock.Release();
```

```
INIT
   int value = 0;

Acquire() {
  disable interrupts;
  if (value == 1) {
    put thread on wait-queue;
    go to sleep() //??
  } else {
    value = 1;
  }
  enable interrupts;
}

Release() {
  disable interrupts;
  if anyone on wait queue {
    take thread off wait-queue
    Place on ready queue;
  } else {
    value = 0;
  }
  enable interrupts;
}
```

Thread B:
```
lock.Acquire();
 …
 critical section;
 …
lock.Release();
```


In-Kernel Lock: Simulation

Value: 1 | waiters | owner | READY

Running: Thread A | Ready: Thread B

Thread A:
```
lock.Acquire();
 …
 critical section;
 …
lock.Release();
```

```
INIT
   int value = 0;

Acquire() {
  disable interrupts;
  if (value == 1) {
    put thread on wait-queue;
    go to sleep() //??
  } else {
    value = 1;
  }
  enable interrupts;
}

Release() {
  disable interrupts;
  if anyone on wait queue {
    take thread off wait-queue
    Place on ready queue;
  } else {
    value = 0;
  }
  enable interrupts;
}
```

Thread B:
```
lock.Acquire();
 …
 critical section;
 …
lock.Release();
```

In-Kernel Lock: Simulation

Value: 1 | waiters | owner | READY

Running — Thread A

Ready — Thread B

```
INIT
  int value = 0;

lock.Acquire();
…
critical section;
…
lock.Release();

Acquire() {
  disable interrupts;
  if (value == 1) {
    put thread on wait-queue;
    go to sleep() //??
  } else {
    value = 1;
  }
  enable interrupts;
}

Release() {
  disable interrupts;
  if anyone on wait queue {
    take thread off wait-queue
    Place on ready queue;
  } else {
    value = 0;
  }
  enable interrupts;
}

lock.Acquire();
…
critical section;
…
lock.Release();
```


In-Kernel Lock: Simulation

Value: 1 | waiters | owner | READY

Ready — Thread A

Waiting — Thread B

```
INIT
  int value = 0;

lock.Acquire();
…
critical section;
…
lock.Release();

Acquire() {
  disable interrupts;
  if (value == 1) {
    put thread on wait-queue;
    go to sleep() //??
  } else {
    value = 1;
  }
  enable interrupts;
}

Release() {
  disable interrupts;
  if anyone on wait queue {
    take thread off wait-queue
    Place on ready queue;
  } else {
    value = 0;
  }
  enable interrupts;
}

lock.Acquire();
…
critical section;
…
lock.Release();
```


In-Kernel Lock: Simulation

Value: 1 | waiters | owner | READY

Running — Thread A

Ready — Thread B

```
INIT
  int value = 0;

lock.Acquire();
…
critical section;
…
lock.Release();

Acquire() {
  disable interrupts;
  if (value == 1) {
    put thread on wait-queue;
    go to sleep() //??
  } else {
    value = 1;
  }
  enable interrupts;
}

Release() {
  disable interrupts;
  if anyone on wait queue {
    take thread off wait-queue
    Place on ready queue;
  } else {
    value = 0;
  }
  enable interrupts;
}

lock.Acquire();
…
critical section;
…
lock.Release();
```


In-Kernel Lock: Simulation

Value: 1 | waiters | owner | READY

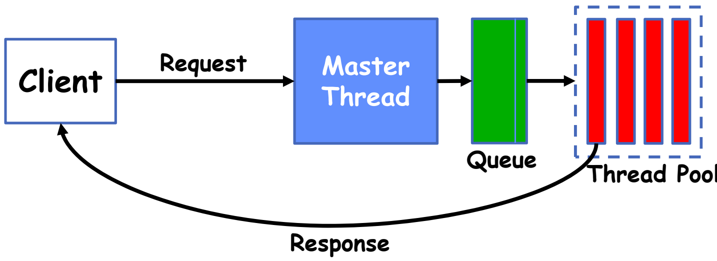
Ready — Thread A

Running — Thread B

```
INIT
  int value = 0;

lock.Acquire();
…
critical section;
…
lock.Release();

Acquire() {
  disable interrupts;
  if (value == 1) {
    put thread on wait-queue;
    go to sleep() //??
  } else {
    value = 1;
  }
  enable interrupts;
}

Release() {
  disable interrupts;
  if anyone on wait queue {
    take thread off wait-queue
    Place on ready queue;
  } else {
    value = 0;
  }
  enable interrupts;
}

lock.Acquire();
…
critical section;
…
lock.Release();
```

Recall: Multithreaded Server

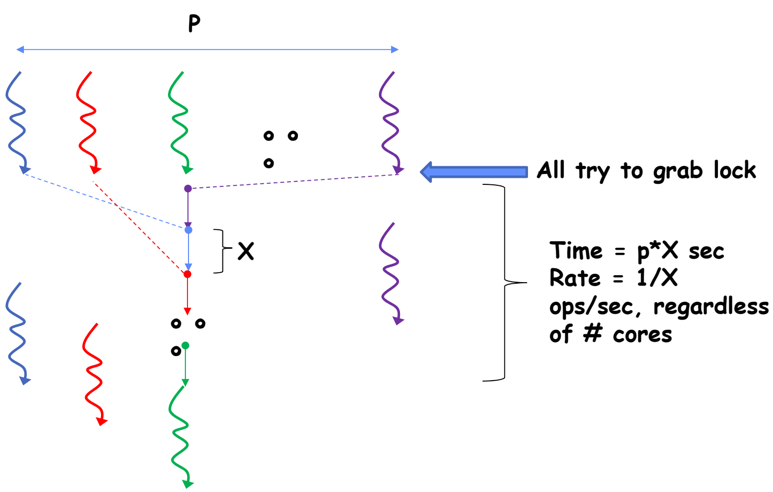
- **Bounded** pool of worker threads
  - Allocated in **advance:** no thread creation overhead
  - **Queue** of pending requests



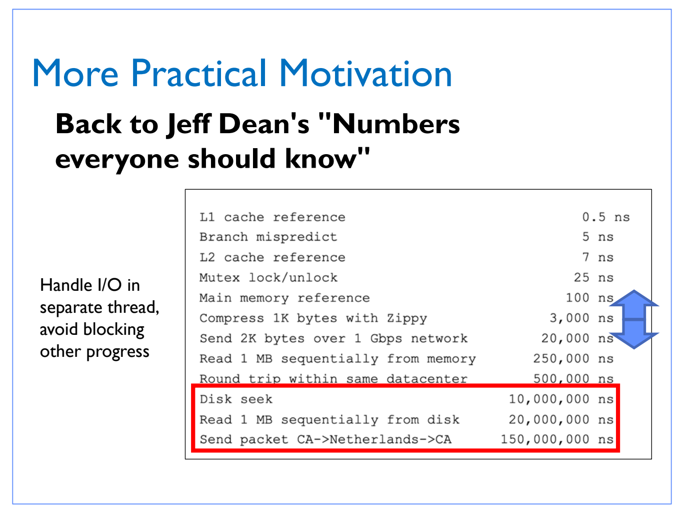
Simple Performance Model

- Given that the overhead of a critical section is X
  - User->Kernel Context Switch
  - Acquire Lock
  - Kernel->User Context Switch
  - <perform exclusive work>
  - User->Kernel Context Switch
  - Release Lock
  - Kernel->User Context Switch
- Even if everything else is infinitely fast, with any number of threads and cores
- What is the maximum rate of operations that involve this overhead?

Highly Contended Case – in a picture



Back to system performance



More Practical Motivation

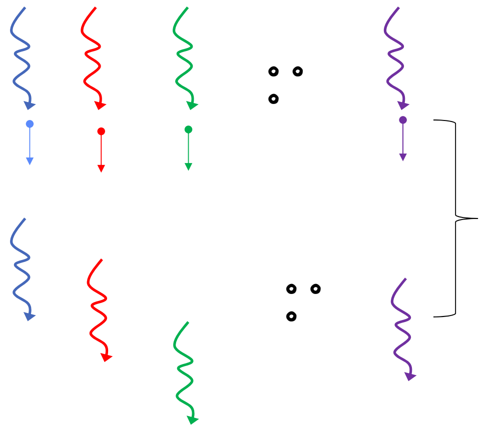
Back to Jeff Dean's "Numbers everyone should know"

Handle I/O in separate thread, avoid blocking other progress

L1 cache reference	0.5 ns
Branch mispredict	5 ns
L2 cache reference	7 ns
Mutex lock/unlock	25 ns
Main memory reference	100 ns
Compress 1K bytes with Zippy	3,000 ns
Send 2K bytes over 1 Gbps network	20,000 ns
Read 1 MB sequentially from memory	250,000 ns
Round trip within same datacenter	500,000 ns
Disk seek	10,000,000 ns
Read 1 MB sequentially from disk	20,000,000 ns
Send packet CA->Netherlands->CA	150,000,000 ns

- X = `1ms` => `1,000` ops/sec

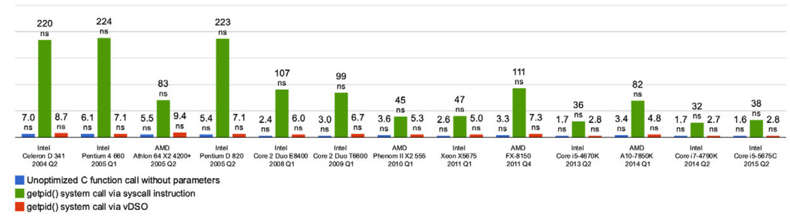
Uncontended Many-Lock Case



What if sys overhead is Y, even when the lock is free?

What if the OS can only handle one lock operation at a time?

Basic cost of a system call



- Min System call ~ 25x cost of function call
- Scheduling could be many times more
- Streamline system processing as much as possible
- Other optimizations seek to process as much of the call in user space as possible (eg, Linux vDSO)

A Better Lock Implementation

- Interrupt-based solution works for single core, but costly
  - Kernel crossings/system calls required for users
  - Disruption of interrupt handling (by disabling interrupts)
- Doesn't work well on multi-core machines
  - Disable intr on all cores?
- Solution: Utilize hardware support for **atomic operations**
  - Operations work on *memory* which is *shared* between cores and doesn't require system calls

Recall: Examples of Read-Modify-Write

```
• test&set (&address) {           /* most architectures */
      result = M[address];        // return result from "address" and
      M[address] = 1;             // set value at "address" to 1
      return result;
  }
• swap (&address, register) {     /* x86 */
      temp = M[address];          // swap register's value to
      M[address] = register;      // value at "address"
      register = temp;
  }
• compare&swap (&address, reg1, reg2) { /* 68000 */
      if (reg1 == M[address]) {   // If memory still == reg1,
          M[address] = reg2;      // then  put reg2 => memory
          return success;
      } else {                    // Otherwise do not change memory
          return failure;
      }
  }
• load-linked&store-conditional(&address) { /* R4000, alpha */
      loop:
          ll r1, M[address];
          movi r2, 1;            // Can do arbitrary computation
          sc r2, M[address];
          beqz r2, loop;
  }
```

Recall: Implementing Locks with test&set

- Our first (simple!) cut at using atomic operations for locking:

```
int value = 0; // Free
Acquire() {
   while (test&set(value)); // while busy
}
Release() {
   value = 0;
}
```

- Simple 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
  – This is not a good implementation for single core
  – For multiprocessors: every test&set() is a write, which makes value ping-pong around in cache (using lots of network BW)

Problem: Busy-Waiting for Lock

- Positives for this solution
  – Machine can receive interrupts
  – User code can use this lock
  – Works on a multiprocessor
- Negatives
  – This is very inefficient as 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!
- Priority Inversion problem with original Martian rover
- Looking forward: For semaphores and monitors, waiting thread may wait for an arbitrary long time!
  – Thus even if busy-waiting was OK for locks, definitely not ok for other primitives
  – Homework/exam solutions should avoid busy-waiting!

Multiprocessor Spin Locks: test&test&set

- A better solution for multiprocessors:

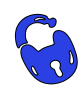
```
int mylock = 0; // Free
Acquire() {
   do {
      while(mylock);    // Wait until might be free
   } while(test&set(&mylock)); // exit if get lock
}

Release() {
   mylock = 0;
}
```

- Simple explanation:
  – Wait until lock might be free (only reading – stays in cache)
  – Then, try to grab lock with test&set
  – Repeat if fail to actually get lock
- Still have issues with this solution:
  – Busy-Waiting: thread still consumes cycles while waiting
    » However, it does not impact other processors!

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
   while (test&set(guard));
   if (value == BUSY) {
      put thread on wait queue;
      go to sleep() & guard = 0;
   } else {
      value = BUSY;
      guard = 0;
   }
}
```

```
Release() {
   // Short busy-wait time
   while (test&set(guard));
   if anyone on wait queue {
      take thread off wait queue
      Place on ready queue;
   } else {
      value = FREE;
   }
   guard = 0;
```

- Note: sleep has to be sure to reset the guard variable
  – Why can't we do it just before or just after the sleep?

Recall: Locks using Interrupts vs. test&set

Compare to “disable interrupt” solution

```
int value = FREE;
```

```
Acquire() {
   disable interrupts;
   if (value == BUSY) {
      put thread on wait queue;
      Go to sleep();
      // Enable interrupts?
   } else {
      value = BUSY;
   }
   enable interrupts;
}
```

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

Basically we replaced:

- disable interrupts → while (test&set(guard));
- enable interrupts → guard = 0;

Recap: Locks using interrupts

```
lock.Acquire();
 …
 critical section;
 …
lock.Release();
```

```
Acquire() {
  disable interrupts;
}
```

```
Release() {
  enable interrupts;
}
```

If one thread in critical section, no other activity (including OS) can run!

```
int value = 0;
Acquire() {
  // Short busy-wait time
  disable interrupts;
  if (value == 1) {
    put thread on wait-queue;
    go to sleep() && Enab Ints
  } else {
    value = 1;
    enable interrupts;
  }
}
```

```
Release() {
  // Short busy-wait time
  disable interrupts;
  if anyone on wait queue {
    take thread off wait-queue
    Place on ready queue;
  } else {
    value = 0;
  }
  enable interrupts;
}
```


Recap: Locks using test & set

```
lock.Acquire();
 …
 critical section;
 …
lock.Release();
```

```
int value = 0;
Acquire() {
  while(test&set(value));
}
```

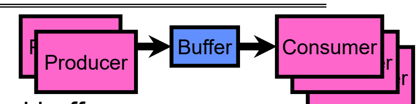
```
Release() {
  value = 0;
}
```

Threads waiting to enter critical section busy-wait

```
int guard = 0;
int value = 0;
Acquire() {
  // Short busy-wait time
  while(test&set(guard));
  if (value == 1) {
    put thread on wait-queue;
    go to sleep()& guard = 0;
  } else {
    value = 1;
    guard = 0;
  }
}
```

```
Release() {
  // Short busy-wait time
  while (test&set(guard));
  if anyone on wait queue {
    take thread off wait-queue
    Place on ready queue;
  } else {
    value = 0;
  }
  guard = 0;
}
```


Producer-Consumer with a Bounded Buffer

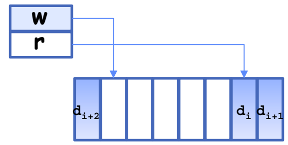


- Problem Definition
  - Producer(s) put things into a shared buffer
  - Consumer(s) take them out
  - Need synchronization to coordinate producer/consumer
- Don’t want producer and consumer to have to work in lockstep, so put a fixed-size buffer between them
  - Need to synchronize access to this buffer
  - 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
- Others: Web servers, Routers, ….

Circular Buffer Data Structure (sequential case)

```
typedef struct buf {
  int write_index;
  int read_index;
  <type> *entries[BUFSIZE];
} buf_t;
```



- Insert: write & bump write ptr (enqueue)
- Remove: read & bump read ptr (dequeue)
- *How to tell if Full (on insert) Empty (on remove)?*
- *And what do you do if it is?*
- *What needs to be atomic?*

Circular Buffer – first cut

```
mutex buf_lock = <initially unlocked>

Producer(item) {
  acquire(&buf_lock);
  while (buffer full) {}; // Wait for a free slot
  enqueue(item);
  release(&buf_lock);
}

Consumer() {
  acquire(&buf_lock);
  while (buffer empty) {}; // Wait for arrival
  item = dequeue();
  release(&buf_lock);
  return item
}
```

Will we ever come out of the wait loop?

Circular Buffer – 2nd cut

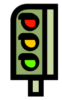
```
mutex buf_lock = <initially unlocked>

Producer(item) {
  acquire(&buf_lock);
  while (buffer full) {release(&buf_lock); acquire(&buf_lock);}
  enqueue(item);
  release(&buf_lock);
}

Consumer() {
  acquire(&buf_lock);
  while (buffer empty) {release(&buf_lock); acquire(&buf_lock);}
  item = dequeue();
  release(&buf_lock);
  return item
}
```

What happens when one is waiting for the other?
- Multiple cores ?
- Single core ?

Higher-level Primitives than Locks

- Goal of last couple of lectures:
  - What is 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
  - This lecture and the next presents a some ways of structuring sharing

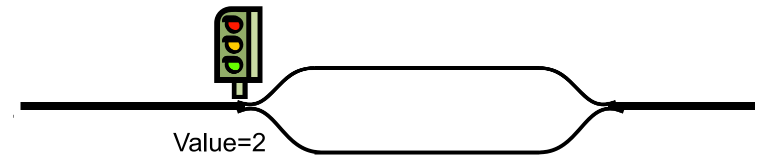
Semaphores



- 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 and supports the following two operations:
  - P(): an atomic operation that waits for semaphore to become positive, then decrements it by 1
    - » Think of this as the wait() operation
  - V(): an atomic operation that increments the semaphore by 1, waking up a waiting P, if any
    - » This of this as the signal() operation
  - Note that P() stands for "*proberen"* (to test) and V() stands for "*verhogen"* (to increment) in Dutch

Semaphores Like Integers Except

- 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)

- Allow thread 1 to wait for a signal from thread 2
  - thread 2 schedules thread 1 when a given event occurs
- 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();
}
```


Revisit Bounded Buffer: 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 stupid
  - 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 fullSlots = 0;    // Initially, no coke
Semaphore emptySlots = bufSize;
                            // Initially, num empty slots
Semaphore mutex = 1;        // No one using machine

Producer(item) {
    emptySlots.P();         // Wait until space
    mutex.P();              // Wait until machine free
    Enqueue(item);
    mutex.V();
    fullSlots.V();          // Tell consumers there is
                            // more coke
}
Consumer() {
    fullSlots.P();          // Check if there's a coke
    mutex.P();              // Wait until machine free
    item = Dequeue();
    mutex.V();
    emptySlots.V();         // tell producer need more
    return item;
}
```


Discussion about Solution

- Why asymmetry?
  - Producer does: **`emptyBuffer.P(), fullBuffer.V()`**
  - Consumer does: **`fullBuffer.P(), emptyBuffer.V()`**

Decrease # of empty slots

Increase # of occupied slots

Decrease # of occupied slots

Increase # of empty slots

- Is order of P's important?
- Is order of V's important?
- What if we have 2 producers or 2 consumers?

```
Producer(item) {
    mutex.P();
    emptySlots.P();
    Enqueue(item);
    mutex.V();
    fullSlots.V();
}
Consumer() {
    fullSlots.P();
    mutex.P();
    item = Dequeue();
    mutex.V();
    emptySlots.V();
    return item;
}
```

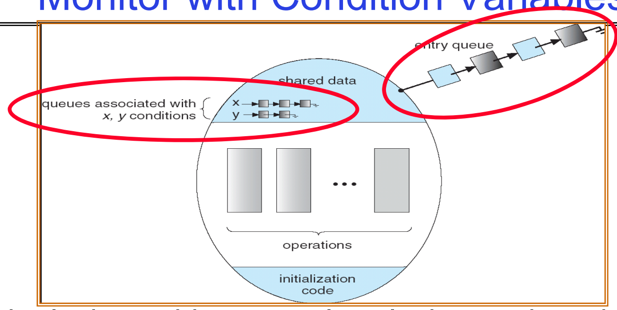

Semaphores are good but…Monitors are better!

- 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
- Definition: Monitor: a lock and zero or more condition variables for managing concurrent access to shared data
  - Some languages like Java provide this natively
  - Most others use actual locks and condition variables
- A "Monitor" is a paradigm for concurrent programming!
  - Some languages support monitors explicitly

Condition Variables

- How do we change the consumer() routine to wait until something is on the queue?
  - Could do this by keeping a count of the number of things on the queue (with semaphores), but error prone
- Condition Variable: a queue of threads waiting for something *inside* a critical section
  - Key idea: allow sleeping inside critical section by atomically releasing lock at time we go to sleep
  - Contrast to semaphores: Can't wait inside critical section
- Operations:
  - `Wait(&lock)`: Atomically release lock and go to sleep. Re-acquire lock later, before returning.
  - `Signal()`: Wake up one waiter, if any
  - `Broadcast()`: Wake up all waiters
- Rule: Must hold lock when doing condition variable ops!

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
  - 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

Synchronized Buffer (with condition variable)

- Here is an (infinite) synchronized queue:

```
lock buf_lock;                  // Initially unlocked
condition buf_CV;               // Initially empty
queue queue;

Producer(item) {
   acquire(&buf_lock);          // Get Lock
   enqueue(&queue,item);        // Add item
   cond_signal(&buf_CV);        // Signal any waiters
   release(&buf_lock);          // Release Lock
}

Consumer() {
   acquire(&buf_lock);          // Get Lock
   while (isEmpty(&queue)) {
      cond_wait(&buf_CV, &buf_lock); // If empty, sleep
   }
   item = dequeue(&queue);      // Get next item
   release(&buf_lock);          // Release Lock
   return(item);
}
```


Mesa vs. Hoare monitors

- Need to be careful about precise definition of signal and wait. Consider a piece of our dequeue code:

```
while (isEmpty(&queue)) {
   cond_wait(&buf_CV,&buf_lock); // If nothing, sleep
}
item = dequeue(&queue);   // Get next item
```

  - Why didn't we do this?

```
if (isEmpty(&queue)) {
   cond_wait(&buf_CV,&buf_lock); // If nothing, sleep
}
item = dequeue(&queue);   // Get next item
```

- Answer: depends on the type of scheduling
  - Mesa-style: Named after Xerox-Park Mesa Operating System
    - » Most OSes use Mesa Scheduling!
  - Hoare-style: Named after British logician Tony Hoare

Hoare monitors

- Signaler gives up lock, CPU to waiter; waiter runs immediately
- Then, Waiter gives up lock, processor back to signaler when it exits critical section or if it waits again

```
…                                                  acquire(&buf_lock);
acquire(&buf_lock);                                …
…                           Lock, CPU              if (isEmpty(&queue)) {
cond_signal(&buf_CV);   ------------------------>     cond_wait(&buf_CV,&buf_lock);
…                       <----- Lock, CPU           }
release(&buf_lock);                                …
                                                   release(&buf_lock);
```

- On first glance, this seems like good semantics
  - Waiter gets to run immediately, condition is still correct!
- Most textbooks talk about Hoare scheduling
  - However, hard to do, not really necessary!
  - Forces a lot of context switching (inefficient!)

Mesa monitors

- Signaler keeps lock and processor
- Waiter placed on ready queue with no special priority

```
…
acquire(&buf_lock);
…
cond_signal(&buf_CV);
…
release(&buf_lock));
```

Put waiting thread on ready queue

schedule thread (sometime later!)

```
acquire(&buf_lock);
…
while (isEmpty(&queue)) {
  cond_wait(&buf_CV,&buf_lock);
}
…
lock.Release();
```

- Practically, need to check condition again after wait
  - By the time the waiter gets scheduled, condition may be false again – so, just check again with the “while” loop
- Most real operating systems do this!
  - More efficient, easier to implement
  - Signaler’s cache state, etc still good

Circular Buffer – 3rd cut (Monitors, pthread-like)

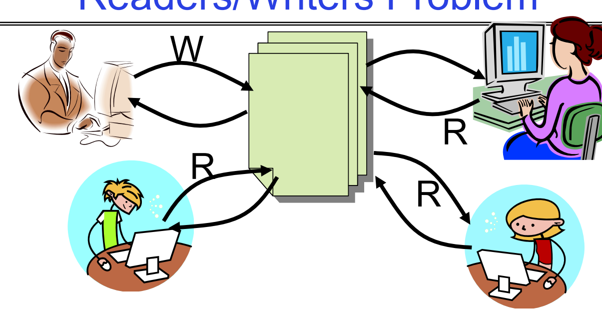
```
lock buf_lock = <initially unlocked>
condition producer_CV = <initially empty>
condition consumer_CV = <initially empty>

Producer(item) {
  acquire(&buf_lock);
  while (buffer full) { cond_wait(&producer_CV, &buf_lock); }
  enqueue(item);
  cond_signal(&consumer_CV);
  release(&buf_lock);
}

Consumer() {
  acquire(buf_lock);
  while (buffer empty) { cond_wait(&consumer_CV, &buf_lock); }
  item = dequeue();
  cond_signal(&producer_CV);
  release(buf_lock);
  return item
}
```

What does thread do when it is waiting?
- Sleep, not busywait!

Again: Why the `while` Loop?

- MESA semantics
- For most operating systems, when a thread is woken up by `signal()`, it is simply put on the ready queue
- It may or may not reacquire the lock immediately!
  - Another thread could be scheduled first and "sneak in" to empty the queue
  - Need a loop to re-check condition on wakeup

Readers/Writers Problem



- Motivation: Consider a shared database
  - Two classes of users:
    - » Readers – never modify database
    - » Writers – read and modify database
  - Is using a single lock on the whole database sufficient?
    - » Like to have many readers at the same time
    - » Only one writer at a time

Basic Readers/Writers Solution

- Correctness Constraints:
  - Readers can access database when no writers
  - Writers can access database when no readers or writers
  - Only one thread manipulates state variables at a time
- Basic structure of a solution:
  - **Reader()**
    ```
    Wait until no writers
    Access data base
    Check out – wake up a waiting writer
    ```
  - **Writer()**
    ```
    Wait until no active readers or writers
    Access database
    Check out – wake up waiting readers or writer
    ```
  - State variables (Protected by a lock called "lock"):
    - » int AR: Number of active readers; initially = 0
    - » int WR: Number of waiting readers; initially = 0
    - » int AW: Number of active writers; initially = 0
    - » int WW: Number of waiting writers; initially = 0
    - » Condition okToRead = NIL
    - » Condition okToWrite = NIL

Code for a Reader

```
Reader() {
  // First check self into system
  acquire(&lock);
  while ((AW + WW) > 0) { // Is it safe to read?
    WR++;                 // No. Writers exist
    cond_wait(&okToRead,&lock);// Sleep on cond var
    WR--;                 // No longer waiting
  }
  AR++;                   // Now we are active!
  release(&lock);
  // Perform actual read-only access
  AccessDatabase(ReadOnly);
  // Now, check out of system
  acquire(&lock);
  AR--;                   // No longer active
  if (AR == 0 && WW > 0)  // No other active readers
    cond_signal(&okToWrite);// Wake up one writer
  release(&lock);
}
```


Code for a Writer

```
Writer() {
  // First check self into system
  acquire(&lock);
  while ((AW + AR) > 0) { // Is it safe to write?
    WW++;                 // No. Active users exist
    cond_wait(&okToWrite,&lock);  // Sleep on cond var
    WW--;                 // No longer waiting
  }
  AW++;                   // Now we are active!
  release(&lock);
  // Perform actual read/write access
  AccessDatabase(ReadWrite);
  // Now, check out of system
  acquire(&lock);
  AW--;                   // No longer active
  if (WW > 0){            // Give priority to writers
    cond_signal(&okToWrite);// Wake up one writer
  } else if (WR > 0) {    // Otherwise, wake reader
    cond_broadcast(&okToRead); // Wake all readers
  }
  release(&lock);
}
```


Simulation of Readers/Writers Solution

- Use an example to simulate the solution
- Consider the following sequence of operators:
  - R1, R2, W1, R3
- Initially: AR = 0, WR = 0, AW = 0, WW = 0

Simulation of Readers/Writers Solution

- R1 comes along (no waiting threads)
- AR = 0, WR = 0, AW = 0, WW = 0

```
Reader() {
    acquire(&lock)
    while ((AW + WW) > 0) { // Is it safe to read?
      WR++;                 // No. Writers exist
      cond_wait(&okToRead,&lock);// Sleep on cond var
      WR--;                 // No longer waiting
    }
    AR++;                   // Now we are active!
    release(&lock);

    AccessDBase(ReadOnly);

    acquire(&lock);
    AR--;
    if (AR == 0 && WW > 0)
      cond_signal(&okToWrite);
    release(&lock);
  }
```


Simulation of Readers/Writers Solution

- R1 comes along (no waiting threads)
- AR = 0, WR = 0, AW = 0, WW = 0

```
Reader() {
    acquire(&lock);
    while ((AW + WW) > 0) { // Is it safe to read?
      WR++;                 // No. Writers exist
      cond_wait(&okToRead,&lock);// Sleep on cond var
      WR--;                 // No longer waiting
    }
    AR++;                   // Now we are active!
    release(&lock);

    AccessDBase(ReadOnly);

    acquire(&lock);
    AR--;
    if (AR == 0 && WW > 0)
      cond_signal(&okToWrite);
    release(&lock);
  }
```


Simulation of Readers/Writers Solution

- R1 comes along (no waiting threads)
- AR = 1, WR = 0, AW = 0, WW = 0

```
Reader() {
    acquire(&lock);
    while ((AW + WW) > 0) { // Is it safe to read?
      WR++;                 // No. Writers exist
      cond_wait(&okToRead,&lock);// Sleep on cond var
      WR--;                 // No longer waiting
    }
    AR++;                   // Now we are active!
    release(&lock);

    AccessDBase(ReadOnly);

    acquire(&lock);
    AR--;
    if (AR == 0 && WW > 0)
      cond_signal(&okToWrite);
    release(&lock);
  }
```


Simulation of Readers/Writers Solution

- R1 comes along (no waiting threads)
- AR = 1, WR = 0, AW = 0, WW = 0

```
Reader() {
    acquire(&lock);
    while ((AW + WW) > 0) { // Is it safe to read?
      WR++;                 // No. Writers exist
      cond_wait(&okToRead,&lock);// Sleep on cond var
      WR--;                 // No longer waiting
    }
    AR++;                   // Now we are active!
    release(&lock);

    AccessDBase(ReadOnly);

    acquire(&lock);
    AR--;
    if (AR == 0 && WW > 0)
      cond_signal(&okToWrite);
    release(&lock);
  }
```

Simulation of Readers/Writers Solution

- R1 accessing dbase (no other threads)
- AR = 1, WR = 0, AW = 0, WW = 0

```
Reader() {
    acquire(&lock);
    while ((AW + WW) > 0) { // Is it safe to read?
      WR++;                // No. Writers exist
      cond_wait(&okToRead,&lock);// Sleep on cond var
      WR--;                // No longer waiting
    }
    AR++;                  // Now we are active!
    release(&lock);

    AccessDBase(ReadOnly);

    acquire(&lock);
    AR--;
    if (AR == 0 && WW > 0)
      cond_signal(&okToWrite);
    release(&lock);
  }
```


Simulation of Readers/Writers Solution

- R2 comes along (R1 accessing dbase)
- AR = 1, WR = 0, AW = 0, WW = 0

```
Reader() {
    acquire(&lock);
    while ((AW + WW) > 0) { // Is it safe to read?
      WR++;                // No. Writers exist
      cond_wait(&okToRead,&lock);// Sleep on cond var
      WR--;                // No longer waiting
    }
    AR++;                  // Now we are active!
    release(&lock);

    AccessDBase(ReadOnly);

    acquire(&lock);
    AR--;
    if (AR == 0 && WW > 0)
      cond_signal(&okToWrite);
    release(&lock);
  }
```


Simulation of Readers/Writers Solution

- R2 comes along (R1 accessing dbase)
- AR = 1, WR = 0, AW = 0, WW = 0

```
Reader() {
    acquire(&lock);
    while ((AW + WW) > 0) { // Is it safe to read?
      WR++;                // No. Writers exist
      cond_wait(&okToRead,&lock);// Sleep on cond var
      WR--;                // No longer waiting
    }
    AR++;                  // Now we are active!
    release(&lock);

    AccessDBase(ReadOnly);

    acquire(&lock);
    AR--;
    if (AR == 0 && WW > 0)
      cond_signal(&okToWrite);
    release(&lock);
  }
```


Simulation of Readers/Writers Solution

- R2 comes along (R1 accessing dbase)
- AR = 2, WR = 0, AW = 0, WW = 0

```
Reader() {
    acquire(&lock);
    while ((AW + WW) > 0) { // Is it safe to read?
      WR++;                // No. Writers exist
      cond_wait(&okToRead,&lock);// Sleep on cond var
      WR--;                // No longer waiting
    }
    AR++;                  // Now we are active!
    release(&lock);

    AccessDBase(ReadOnly);

    acquire(&lock);
    AR--;
    if (AR == 0 && WW > 0)
      cond_signal(&okToWrite);
    release(&lock);
  }
```

Simulation of Readers/Writers Solution

- R2 comes along (R1 accessing dbase)
- AR = 2, WR = 0, AW = 0, WW = 0

```
Reader() {
    acquire(&lock);
    while ((AW + WW) > 0) { // Is it safe to read?
      WR++;                 // No. Writers exist
      cond_wait(&okToRead,&lock);// Sleep on cond var
      WR--;                 // No longer waiting
    }
    AR++;                   // Now we are active!
    release(&lock);

    AccessDBase(ReadOnly);

    acquire(&lock);
    AR--;
    if (AR == 0 && WW > 0)
      cond_signal(&okToWrite);
    release(&lock);
  }
```


Simulation of Readers/Writers Solution

- R1 and R2 accessing dbase
- AR = 2, WR = 0, AW = 0, WW = 0

```
Reader() {
    acquire(&lock);
    while ((AW + WW) > 0) { // Is it safe to read?
      WR++;                 // No. Writers exist
      cond_wait(&okToRead,&lock);// Sleep on cond var
      WR--;                 // No longer waiting
    }
    AR++;                   // Now we are active!
    release(&lock);

    AccessDBase(ReadOnly);

    acquire(&lock);
    AR--;
    if (AR == 0 && WW > 0)
```

Assume readers take a while to access database
Situation: Locks released, only AR is non-zero

Simulation of Readers/Writers Solution

- W1 comes along (R1 and R2 are still accessing dbase)
- AR = 2, WR = 0, AW = 0, WW = 0

```
Writer() {
    acquire(&lock);
    while ((AW + AR) > 0) {  // Is it safe to write?
      WW++;                  // No. Active users exist
      cond_wait(&okToWrite,&lock);// Sleep on cond var
      WW--;                  // No longer waiting
    }
    AW++;
    release(&lock);

    AccessDBase(ReadWrite);

    acquire(&lock);
    AW--;
    if (WW > 0){
      cond_signal(&okToWrite);
    } else if (WR > 0) {
      cond_broadcast(&okToRead);
    }
    release(&lock);
  }
```


Simulation of Readers/Writers Solution

- W1 comes along (R1 and R2 are still accessing dbase)
- AR = 2, WR = 0, AW = 0, WW = 0

```
Writer() {
    acquire(&lock);
    while ((AW + AR) > 0) {  // Is it safe to write?
      WW++;                  // No. Active users exist
      cond_wait(&okToWrite,&lock);// Sleep on cond var
      WW--;                  // No longer waiting
    }
    AW++;
    release(&lock);

    AccessDBase(ReadWrite);

    acquire(&lock);
    AW--;
    if (WW > 0){
      cond_signal(&okToWrite);
    } else if (WR > 0) {
      cond_broadcast(&okToRead);
    }
    release(&lock);
  }
```

Simulation of Readers/Writers Solution

- W1 comes along (R1 and R2 are still accessing dbase)
- AR = 2, WR = 0, AW = 0, WW = 1

```
Writer() {
    acquire(&lock);
    while ((AW + AR) > 0) {  // Is it safe to write?
      WW++;                  // No. Active users exist
      cond_wait(&okToWrite,&lock);// Sleep on cond var
      WW--;                  // No longer waiting
    }
    AW++;
    release(&lock);

    AccessDBase(ReadWrite);

    acquire(&lock);
    AW--;
    if (WW > 0){
      cond_signal(&okToWrite);
    } else if (WR > 0) {
      cond_broadcast(&okToRead);
    }
    release(&lock);
}
```


Simulation of Readers/Writers Solution

- R3 comes along (R1 and R2 accessing dbase, W1 waiting)
- AR = 2, WR = 0, AW = 0, WW = 1

```
Reader() {
    acquire(&lock);
    while ((AW + WW) > 0) { // Is it safe to read?
      WR++;                 // No. Writers exist
      cond_wait(&okToRead,&lock);// Sleep on cond var
      WR--;                 // No longer waiting
    }
    AR++;                   // Now we are active!
    release(&lock);

    AccessDBase(ReadOnly);

    acquire(&lock);
    AR--;
    if (AR == 0 && WW > 0)
      cond_signal(&okToWrite);
    release(&lock);
}
```


Simulation of Readers/Writers Solution

- R3 comes along (R1 and R2 accessing dbase, W1 waiting)
- AR = 2, WR = 0, AW = 0, WW = 1

```
Reader() {
    acquire(&lock);
    while ((AW + WW) > 0) { // Is it safe to read?
      WR++;                 // No. Writers exist
      cond_wait(&okToRead,&lock);// Sleep on cond var
      WR--;                 // No longer waiting
    }
    AR++;                   // Now we are active!
    release(&lock);

    AccessDBase(ReadOnly);

    acquire(&lock);
    AR--;
    if (AR == 0 && WW > 0)
      cond_signal(&okToWrite);
    release(&lock);
}
```


Simulation of Readers/Writers Solution

- R3 comes along (R1 and R2 accessing dbase, W1 waiting)
- AR = 2, WR = 1, AW = 0, WW = 1

```
Reader() {
    acquire(&lock);
    while ((AW + WW) > 0) { // Is it safe to read?
      WR++;                 // No. Writers exist
      cond_wait(&okToRead,&lock);// Sleep on cond var
      WR--;                 // No longer waiting
    }
    AR++;                   // Now we are active!
    lock.release();

    AccessDBase(ReadOnly);

    lock.Acquire();
    AR--;
    if (AR == 0 && WW > 0)
      okToWrite.signal();
    lock.Release();
}
```

Simulation of Readers/Writers Solution

- R3 comes along (R1, R2 accessing dbase, W1 waiting)
- AR = 2, WR = 1, AW = 0, WW = 1

```
Reader() {
    lock.Acquire();
    while ((AW + WW) > 0) { // Is it safe to read?
      WR++;                 // No. Writers exist
      cond_wait(&okToRead,&lock);// Sleep on cond var
      WR--;                 // No longer waiting
    }
    AR++;                   // Now we are active!
    lock.release();

    AccessDBase(ReadOnly);

    lock.Acquire();
    AR--;
    if (AR == 0 && WW > 0)
      okToWrite.signal();
    lock.Release();
  }
```


Simulation of Readers/Writers Solution

- R1 and R2 accessing dbase, W1 and R3 waiting
- AR = 2, WR = 1, AW = 0, WW = 1

```
Reader() {
    acquire(&lock);
    while ((AW + WW) > 0) { // Is it safe to read?
      WR++;                 // No. Writers exist
      cond_wait(&okToRead,&lock);// Sleep on cond var
      WR--;                 // No longer waiting
    }
    AR++;                   // Now we are active!
    release(&lock);

    AccessDBase(ReadOnly);

    acquire(&lock);
    AR--;
    if (AR == 0 && WW > 0)
```

Status:
- R1 and R2 still reading
- W1 and R3 waiting on okToWrite and okToRead, respectively

Simulation of Readers/Writers Solution

- R2 finishes (R1 accessing dbase, W1 and R3 waiting)
- AR = 2, WR = 1, AW = 0, WW = 1

```
Reader() {
    acquire(&lock);
    while ((AW + WW) > 0) { // Is it safe to read?
      WR++;                 // No. Writers exist
      cond_wait(&okToRead,&lock);// Sleep on cond var
      WR--;                 // No longer waiting
    }
    AR++;                   // Now we are active!
    release(&lock);

    AccessDBase(ReadOnly);

    acquire(&lock);
    AR--;
    if (AR == 0 && WW > 0)
      cond_signal(&okToWrite);
    release(&lock);
  }
```


Simulation of Readers/Writers Solution

- R2 finishes (R1 accessing dbase, W1 and R3 waiting)
- AR = 1, WR = 1, AW = 0, WW = 1

```
Reader() {
    acquire(&lock);
    while ((AW + WW) > 0) { // Is it safe to read?
      WR++;                 // No. Writers exist
      cond_wait(&okToRead,&lock);// Sleep on cond var
      WR--;                 // No longer waiting
    }
    AR++;                   // Now we are active!
    release(&lock);

    AccessDBase(ReadOnly);

    acquire(&lock);
    AR--;
    if (AR == 0 && WW > 0)
      cond_signal(&okToWrite);
    release(&lock);
  }
```

Simulation of Readers/Writers Solution

- R2 finishes (R1 accessing dbase, W1 and R3 waiting)
- AR = 1, WR = 1, AW = 0, WW = 1

```
Reader() {
    acquire(&lock);
    while ((AW + WW) > 0) { // Is it safe to read?
      WR++;               // No. Writers exist
      cond_wait(&okToRead,&lock);// Sleep on cond var
      WR--;               // No longer waiting
    }
    AR++;                 // Now we are active!
    release(&lock);

    AccessDBase(ReadOnly);

    acquire(&lock);
    AR--;
    if (AR == 0 && WW > 0)
      cond_signal(&okToWrite);
    release(&lock);
  }
```


Simulation of Readers/Writers Solution

- R2 finishes (R1 accessing dbase, W1 and R3 waiting)
- AR = 1, WR = 1, AW = 0, WW = 1

```
Reader() {
    acquire(&lock);
    while ((AW + WW) > 0) { // Is it safe to read?
      WR++;               // No. Writers exist
      cond_wait(&okToRead,&lock);// Sleep on cond var
      WR--;               // No longer waiting
    }
    AR++;                 // Now we are active!
    release(&lock);

    AccessDBase(ReadOnly);

    acquire(&lock);
    AR--;
    if (AR == 0 && WW > 0)
      cond_signal(&okToWrite);
    release(&lock);
  }
```


Simulation of Readers/Writers Solution

- R1 finishes (W1 and R3 waiting)
- AR = 1, WR = 1, AW = 0, WW = 1

```
Reader() {
    acquire(&lock);
    while ((AW + WW) > 0) { // Is it safe to read?
      WR++;               // No. Writers exist
      cond_wait(&okToRead,&lock);// Sleep on cond var
      WR--;               // No longer waiting
    }
    AR++;                 // Now we are active!
    release(&lock);

    AccessDBase(ReadOnly);

    acquire(&lock);
    AR--;
    if (AR == 0 && WW > 0)
      cond_signal(&okToWrite);
    release(&lock);
  }
```


Simulation of Readers/Writers Solution

- R1 finishes (W1, R3 waiting)
- AR = 0, WR = 1, AW = 0, WW = 1

```
Reader() {
    acquire(&lock);
    while ((AW + WW) > 0) { // Is it safe to read?
      WR++;               // No. Writers exist
      cond_wait(&okToRead,&lock);// Sleep on cond var
      WR--;               // No longer waiting
    }
    AR++;                 // Now we are active!
    release(&lock);

    AccessDBase(ReadOnly);

    acquire(&lock);
    AR--;
    if (AR == 0 && WW > 0)
      cond_signal(&okToWrite);
    release(&lock);
  }
```

Simulation of Readers/Writers Solution

- R1 finishes (W1, R3 waiting)
- AR = 0, WR = 1, AW = 0, WW = 1

```
Reader() {
    acquire(&lock);
    while ((AW + WW) > 0) { // Is it safe to read?
      WR++;                 // No. Writers exist
      cond_wait(&okToRead,&lock);// Sleep on cond var
      WR--;                 // No longer waiting
    }
    AR++;                   // Now we are active!
    release(&lock);

    AccessDBase(ReadOnly);

    acquire(&lock);
    AR--;
    if (AR == 0 && WW > 0)
      cond_signal(&okToWrite);
    release(&lock);
  }
```


Simulation of Readers/Writers Solution

- R1 signals a writer (W1 and R3 waiting)
- AR = 0, WR = 1, AW = 0, WW = 1

```
Reader() {
    acquire(&lock);
    while ((AW + WW) > 0) { // Is it safe to read?
      WR++;                 // No. Writers exist
      cond_wait(&okToRead,&lock);// Sleep on cond var
      WR--;                 // No longer waiting
    }
    AR++;                   // Now we are active!
    release(&lock);

    AccessDBase(ReadOnly);

    acquire(&lock);
    AR--;
    if (AR == 0 && WW > 0)
      cond_signal(&okToWrite);
    lock.Release();
  }
```


Simulation of Readers/Writers Solution

- W1 gets signal (R3 still waiting)
- AR = 0, WR = 1, AW = 0, WW = 1

```
Writer() {
    acquire(&lock);
    while ((AW + AR) > 0) {  // Is it safe to write?
      WW++;                  // No. Active users exist
      cond_wait(&okToWrite,&lock);// Sleep on cond var
      WW--;                  // No longer waiting
    }
    AW++;
    release(&lock);

    AccessDBase(ReadWrite);

    acquire(&lock);
    AW--;
    if (WW > 0){
      cond_signal(&okToWrite);
    } else if (WR > 0) {
      cond_broadcast(&okToRead);
    }
    release(&lock);
  }
```


Simulation of Readers/Writers Solution

- W1 gets signal (R3 still waiting)
- AR = 0, WR = 1, AW = 0, WW = 0

```
Writer() {
    acquire(&lock);
    while ((AW + AR) > 0) {  // Is it safe to write?
      WW++;                  // No. Active users exist
      cond_wait(&okToWrite,&lock);// Sleep on cond var
      WW--;                  // No longer waiting
    }
    AW++;
    release(&lock);

    AccessDBase(ReadWrite);

    acquire(&lock);
    AW--;
    if (WW > 0){
      cond_signal(&okToWrite);
    } else if (WR > 0) {
      cond_broadcast(&okToRead);
    }
    release(&lock);
  }
```

Simulation of Readers/Writers Solution

- W1 gets signal (R3 still waiting)
- AR = 0, WR = 1, AW = 1, WW = 0

```
Writer() {
    acquire(&lock);
    while ((AW + AR) > 0) {  // Is it safe to write?
      WW++;                  // No. Active users exist
      cond_wait(&okToWrite,&lock);// Sleep on cond var
      WW--;                  // No longer waiting
    }
    AW++;
    release(&lock);

    AccessDBase(ReadWrite);

    acquire(&lock);
    AW--;
    if (WW > 0){
      cond_signal(&okToWrite);
    } else if (WR > 0) {
      cond_broadcast(&okToRead);
    }
    release(&lock);
  }
```


Simulation of Readers/Writers Solution

- W1 accessing dbase (R3 still waiting)
- AR = 0, WR = 1, AW = 1, WW = 0

```
Writer() {
    acquire(&lock);
    while ((AW + AR) > 0) {  // Is it safe to write?
      WW++;                  // No. Active users exist
      cond_wait(&okToWrite,&lock);// Sleep on cond var
      WW--;                  // No longer waiting
    }
    AW++;
    release(&lock);

    AccessDBase(ReadWrite);

    acquire(&lock);
    AW--;
    if (WW > 0){
      cond_signal(&okToWrite);
    } else if (WR > 0) {
      cond_broadcast(&okToRead);
    }
    release(&lock);
  }
```


Simulation of Readers/Writers Solution

- W1 finishes (R3 still waiting)
- AR = 0, WR = 1, AW = 1, WW = 0

```
Writer() {
    acquire(&lock);
    while ((AW + AR) > 0) {  // Is it safe to write?
      WW++;                  // No. Active users exist
      okToWrite.wait(&lock);// Sleep on cond var
      WW--;                  // No longer waiting
    }
    AW++;
    release(&lock);

    AccessDBase(ReadWrite);

    acquire(&lock);
    AW--;
    if (WW > 0){
      cond_signal(&okToWrite);
    } else if (WR > 0) {
      cond_broadcast(&okToRead);
    }
    release(&lock);
  }
```


Simulation of Readers/Writers Solution

- W1 finishes (R3 still waiting)
- AR = 0, WR = 1, AW = 0, WW = 0

```
Writer() {
    acquire(&lock);
    while ((AW + AR) > 0) {  // Is it safe to write?
      WW++;                  // No. Active users exist
      okToWrite.wait(&lock);// Sleep on cond var
      WW--;                  // No longer waiting
    }
    AW++;
    release(&);

    AccessDBase(ReadWrite);

    acquire(&lock);
    AW--;
    if (WW > 0){
      cond_signal(&okToWrite);
    } else if (WR > 0) {
      cond_broadcast(&okToRead);
    }
    release(&lock);
  }
```

Simulation of Readers/Writers Solution

- W1 finishes (R3 still waiting)
- AR = 0, WR = 1, AW = 0, WW = 0

```
Writer() {
    acquire(&lock);
    while ((AW + AR) > 0) {  // Is it safe to write?
      WW++;                  // No. Active users exist
      cond_wait(&okToWrite,&lock);// Sleep on cond var
      WW--;                  // No longer waiting
    }
    AW++;
    release(&lock);

    AccessDBase(ReadWrite);

    acquire(&lock);
    AW--;
    if (WW > 0){
      cond_signal(&okToWrite);
    } else if (WR > 0) {
      cond_broadcast(&okToRead);
    }
    release(&lock);
  }
```


Simulation of Readers/Writers Solution

- W1 signaling readers (R3 still waiting)
- AR = 0, WR = 1, AW = 0, WW = 0

```
Writer() {
    acquire(&lock);
    while ((AW + AR) > 0) {  // Is it safe to write?
      WW++;                  // No. Active users exist
      cond_wait(&okToWrite,&lock);// Sleep on cond var
      WW--;                  // No longer waiting
    }
    AW++;
    release(&lock);

    AccessDBase(ReadWrite);

    acquire(&lock);
    AW--;
    if (WW > 0){
      cond_signal(&okToWrite);
    } else if (WR > 0) {
      cond_broadcast(&okToRead);
    }
    release(&lock);
  }
```


Simulation of Readers/Writers Solution

- R3 gets signal (no waiting threads)
- AR = 0, WR = 1, AW = 0, WW = 0

```
Reader() {
    acquire(&lock);
    while ((AW + WW) > 0) { // Is it safe to read?
      WR++;                 // No. Writers exist
      cond_wait(&okToRead,&lock);// Sleep on cond var
      WR--;                 // No longer waiting
    }
    AR++;                   // Now we are active!
    release(&lock);

    AccessDBase(ReadOnly);

    acquire(&lock);
    AR--;
    if (AR == 0 && WW > 0)
      cond_signal(&okToWrite);
    release(&lock);
  }
```


Simulation of Readers/Writers Solution

- R3 gets signal (no waiting threads)
- AR = 0, WR = 0, AW = 0, WW = 0

```
Reader() {
    acquire(&lock);
    while ((AW + WW) > 0) { // Is it safe to read?
      WR++;                 // No. Writers exist
      cond_wait(&okToRead,&lock);// Sleep on cond var
      WR--;                 // No longer waiting
    }
    AR++;                   // Now we are active!
    release(&lock);

    AccessDBase(ReadOnly);

    acquire(&lock);
    AR--;
    if (AR == 0 && WW > 0)
      cond_signal(&okToWrite);
    release(&lock);
  }
```

Simulation of Readers/Writers Solution

- R3 accessing dbase (no waiting threads)
- AR = 1, WR = 0, AW = 0, WW = 0

```
Reader() {
    acquire(&lock);
    while ((AW + WW) > 0) { // Is it safe to read?
      WR++;                 // No. Writers exist
      cond_wait(&okToRead,&lock);// Sleep on cond var
      WR--;                 // No longer waiting
    }
    AR++;                   // Now we are active!
    release(&lock);

    AccessDBase(ReadOnly);

    acquire(&lock);
    AR--;
    if (AR == 0 && WW > 0)
      cond_signal(&okToWrite);
    release(&lock);
  }
```


Simulation of Readers/Writers Solution

- R3 finishes (no waiting threads)
- AR = 1, WR = 0, AW = 0, WW = 0

```
Reader() {
    acquire(&lock);
    while ((AW + WW) > 0) { // Is it safe to read?
      WR++;                 // No. Writers exist
      cond_wait(&okToRead,&lock);// Sleep on cond var
      WR--;                 // No longer waiting
    }
    AR++;                   // Now we are active!
    release(&lock);

    AccessDBase(ReadOnly);

    acquire(&lock);
    AR--;
    if (AR == 0 && WW > 0)
      cond_signal(&okToWrite);
    release(&lock);
  }
```


Simulation of Readers/Writers Solution

- R3 finishes (no waiting threads)
- AR = 0, WR = 0, AW = 0, WW = 0

```
Reader() {
    acquire(&lock);
    while ((AW + WW) > 0) { // Is it safe to read?
      WR++;                 // No. Writers exist
      cond_wait(&okToRead,&lock);// Sleep on cond var
      WR--;                 // No longer waiting
    }
    AR++;                   // Now we are active!
    release(&lock);

    AccessDbase(ReadOnly);

    acquire(&lock);
    AR--;
    if (AR == 0 && WW > 0)
      cond_signal(&okToWrite);
    release(&lock);
  }
```


Questions

- Can readers starve? Consider Reader() entry code:

```
while ((AW + WW) > 0) { // Is it safe to read?
  WR++;                 // No. Writers exist
  cond_wait(&okToRead,&lock);// Sleep on cond var
  WR--;                 // No longer waiting
}
AR++;                   // Now we are active!
```

yes, but normally it does not matter.

- What if we erase the condition check in Reader exit?

dost not matter. recheck,but waste cpu

```
AR--;
if (AR == 0 && WW > 0)  // No other active readers
  cond_signal(&okToWrite);// Wake up one writer
```

- Further, what if we turn the signal() into broadcast()

dost not matter. recheck,but waste cpu

```
AR--;
cond_broadcast(&okToWrite); // Wake up sleepers
```

- Finally, what if we use only one condition variable (call it "**okContinue**") instead of two separate ones?
  - Both readers and writers sleep on this variable
  - Must use broadcast() instead of signal()

Use of Single CV: okContinue

```
Reader() {
    // check into system
    acquire(&lock);
    while ((AW + WW) > 0) {
       WR++;
       cond_wait(&okContinue);
       WR--;
    }
    AR++;
    release(&lock);

    // read-only access
    AccessDbase(ReadOnly);

    // check out of system
    acquire(&lock);
    AR--;
    if (AR == 0 && WW > 0)
       cond_signal(&okContinue);
    release(&lock);
  }
```

```
Writer() {
    // check into system
    acquire(&lock);
    while ((AW + AR) > 0) {
       WW++;
       cond_wait(&okContinue);
       WW--;
    }
    AW++;
    release(&lock);

    // read/write access
    AccessDbase(ReadWrite);

    // check out of system
    acquire(&lock);
    AW--;
    if (WW > 0){
       cond_signal(&okContinue);
    } else if (WR > 0) {
       cond_broadcast(&okContinue);
    }
    release(&lock);
  }
```

almost…

What if we turn okToWrite and okToRead into okContinue (i.e. use only one condition variable instead of two)?

Use of Single CV: okContinue

```
Reader() {
    // check into system
    acquire(&lock);
    while ((AW + WW) > 0) {
       WR++;
       cond_wait(&okContinue);
       WR--;
    }
    AR++;
    release(&lock);

    // read-only access
    AccessDbase(ReadOnly);

    // check out of system
    acquire(&lock);
    AR--;
    if (AR == 0 && WW > 0)
       cond_signal(&okContinue);
    release(&lock);
  }
```

```
Writer() {
    // check into system
    acquire(&lock);
    while ((AW + AR) > 0) {
       WW++;
       cond_wait(&okContinue);
       WW--;
    }
    AW++;
    release(&lock);

    // read/write access
    AccessDbase(ReadWrite);

    // check out of system
    acquire(&lock);
    AW--;
    if (WW > 0){
       cond_signal(&okContinue);
    } else if (WR > 0) {
       cond_broadcast(&okContinue);
    }
    release(&lock);
  }
```

Consider this scenario:
- R1 arrives
- W1, R2 arrive while R1 still reading → W1 and R2 wait for R1 to finish
- Assume R1's signal is delivered to R2 (not W1)

R2 awake, but there is a WW, R2 is deadlocked.

Use of Single CV: okContinue

```
Reader() {
    // check into system
    acquire(&lock);
    while ((AW + WW) > 0) {
       WR++;
       okContinue.wait(&lock);
       WR--;
    }
    AR++;
    release(&lock);

    // read-only access
    AccessDbase(ReadOnly);

    // check out of system
    acquire(&lock);
    AR--;
    if (AR == 0 && WW > 0)
       okContinue.broadcast();
    release(&lock);
  }
```

```
Writer() {
    // check into system
    acquire(&lock);
    while ((AW + AR) > 0) {
       WW++;
       okContinue.wait(&lock);
       WW--;
    }
    AW++;
    release(&lock);

    // read/write access
    AccessDbase(ReadWrite);

    // check out of system
    acquire(&lock);
    AW--;
    if (WW > 0 || WR > 0){
       okContinue.broadcast();
    }
    release(&lock);
  }
```

Need to change to broadcast()!

Must broadcast() to sort things out!

Can we construct Monitors from Semaphores?

- Locking aspect is easy: Just use a mutex
- Can we implement condition variables this way?

```
Wait()   { semaphore.P(); }
Signal() { semaphore.V(); }
```

doesn't work. Wait() may sleep with lock held.

- Does this work better?

```
Wait(Lock lock) {
   lock.Release();
   semaphore.P();
   lock.Acquire();
}
Signal() { semaphore.V(); }
```

No. Condition vars have no history, semaphores have history.

Construction of Monitors from Semaphores (con't)

- Problem with previous try:
  - P and V are commutative – result is the same no matter what order they occur
  - Condition variables are NOT commutative
- Does this fix the problem?

```
Wait(Lock lock) {
   lock.Release();
   semaphore.P();
   lock.Acquire();
}
Signal() {
   if semaphore queue is not empty
      semaphore.V();
}
```

  - Not legal to look at contents of semaphore queue
  - There is a race condition – signaler can slip in after lock release and before waiter executes semaphore.P()
- It is actually possible to do this correctly
  - Complex solution for Hoare scheduling in book
  - Can you come up with simpler Mesa-scheduled solution?

Monitor Conclusion

- Monitors represent the logic of the program
  - Wait if necessary
  - Signal when change something so any waiting threads can proceed
- Basic structure of monitor-based program:

```
lock
while (need to wait) {
   condvar.wait();
}
unlock

do something so no need to wait

lock

condvar.signal();

unlock
```

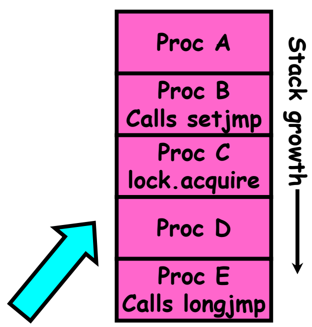
Check and/or update state variables
Wait if necessary

Check and/or update state variables

C-Language Support for Synchronization

- C language: Pretty straightforward synchronization
  - Just make sure you know *all* the code paths out of a critical section

```
int Rtn() {
  lock.acquire();
  …
  if (exception) {
     lock.release();
     return errReturnCode;
  }
  …
  lock.release();
  return OK;
}
```



  - Watch out for setjmp/longjmp!
    - » Can cause a non-local jump out of procedure
    - » In example, procedure E calls longjmp, poping stack back to procedure B
    - » If Procedure C had lock.acquire, problem!

C++ Language Support for Synchronization

- Languages with exceptions like C++
  - Languages that support exceptions are problematic (easy to make a non-local exit without releasing lock)
  - Consider:

```
void Rtn() {
  lock.acquire();
  …
  DoFoo();
  …
  lock.release();
}
void DoFoo() {
  …
  if (exception) throw errException;
  …
}
```

  - Notice that an exception in DoFoo() will exit without releasing the lock!

C++ Language Support for Synchronization (con't)

- Must catch all exceptions in critical sections
  – Catch exceptions, release lock, and re-throw exception:

```
void Rtn() {
  lock.acquire();
  try {
    …
    DoFoo();
    …
  } catch (…) {      // catch exception
    lock.release(); // release lock
    throw;          // re-throw the exception
  }
  lock.release();
}
void DoFoo() {
  …
  if (exception) throw errException;
  …
}
```

  – Even Better: auto_ptr<T> facility. See C++ Spec.
    » Can deallocate/free lock regardless of exit method

Java Language Support for Synchronization

- Java has explicit support for threads and thread synchronization
- Bank Account example:

```
class Account {
  private int balance;
  // object constructor
  public Account (int initialBalance) {
    balance = initialBalance;
  }
  public synchronized int getBalance() {
    return balance;
  }
  public synchronized void deposit(int amount) {
    balance += amount;
  }
}
```

  – Every object has an associated lock which gets automatically acquired and released on entry and exit from a *synchronized* method.

Java Language Support for Synchronization (con't)

- Java also has *synchronized* statements:

```
synchronized (object) {
    …
}
```

  – Since every Java object has an associated lock, this type of statement acquires and releases the object's lock on entry and exit of the body
  – Works properly even with exceptions:

```
synchronized (object) {
  …
  DoFoo();
  …
}
void DoFoo() {
  throw errException;
}
```


Java Language Support for Synchronization (con't 2)

- In addition to a lock, every object has a single condition variable associated with it
  – How to wait inside a synchronization method of block:
    » `void wait(long timeout); // Wait for timeout`
    » `void wait(long timeout, int nanoseconds); //variant`
    » `void wait();`
  – How to signal in a synchronized method or block:
    » `void notify();    // wakes up oldest waiter`
    » `void notifyAll(); // like broadcast, wakes everyone`
  – Condition variables can wait for a bounded length of time. This is useful for handling exception cases:

```
t1 = time.now();
while (!ATMRequest()) {
  wait (CHECKPERIOD);
  t2 = time.new();
  if (t2 – t1 > LONG_TIME) checkMachine();
}
```

  – Not all Java VMs equivalent!
    » Different scheduling policies, not necessarily preemptive!

Summary (1/2)

- 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
- Talked about hardware atomicity primitives:
  - Disabling of Interrupts, test&set, swap, compare&swap, load-locked & store-conditional
- Showed 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

Summary (2/2)

- Semaphores: Like integers with restricted interface
  - Two operations:
    - » `P()`: Wait if zero; decrement when becomes non-zero
    - » `V()`: Increment and wake a sleeping task (if exists)
    - » Can initialize value to any non-negative value
  - Use separate semaphore for each constraint
- Monitors: A lock plus one or more condition variables
  - Always acquire lock before accessing shared data
  - Use condition variables to wait inside critical section
    - » Three Operations: `Wait()`, `Signal()`, and `Broadcast()`
- Monitors represent the logic of the program
  - Wait if necessary
  - Signal when change something so any waiting threads can proceed