CS162
Operating Systems and Systems Programming
Lecture 8

Synchronization 3:
Atomic Instructions (Con't), Monitors, Readers/Writers

September 23rd, 2020

Prof. John Kubiatowicz
http://cs162.eecs.Berkeley.edu

Recall: 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

Recall: Too Much Milk: Solution #4

- Solution #3 really complex and undesirable as a general solution
- Recall our target lock interface:
  - `acquire(&milklock)` – wait until lock is free, then grab
  - `release(&milklock)` – 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:

```
acquire(&milklock);
if (nomilk)
   buy milk;
release(&milklock);
```


Recall: Implement Locks by Disabling Interrupts

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

```
int mylock = FREE; // acquire(&mylock) – wait until lock is free, then grab
                   // release(&mylock) – Unlock, waking up anyone waiting

acquire(int *thelock) {
   disable interrupts;
   if (*thelock == BUSY) {
      put thread on wait queue;
      Go to sleep() && Enab ints!
      // Ints disabled on wakeup
   } else {
      *thelock = BUSY;
   }
   enable interrupts;
}
```

```
release(int *thelock) {
   disable interrupts;
   if (anyone on wait queue) {
      take thread off wait queue
      Place on ready queue;
   } else {
      *thelock = FREE;
   }
   enable interrupts;
}
```

- Really only works in kernel – why?

Recall: In-Kernel Lock: Simulation

mylock: 0 | waiters | owner | READY

Running: Thread A — Ready: Thread B

```
acquire(&mylock);
 …
 critical section;
 …
release(&mylock);
```

```
INIT
   int mylock = 0;

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

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

```
acquire(&mylock);
 …
 critical section;
 …
release(&mylock);
```


Recall: In-Kernel Lock: Simulation

mylock: 1 | waiters | owner | READY

Running: Thread A — Ready: Thread B

```
acquire(&mylock);
 …
 critical section;
 …
release(&mylock);
```

```
INIT
   int mylock = 0;

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

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

```
acquire(&mylock);
 …
 critical section;
 …
release(&mylock);
```


Recall: In-Kernel Lock: Simulation

mylock: 1 | waiters | owner | READY

Running / Ready: Thread A — Ready / Running: Thread B

```
acquire(&mylock);
 …
 critical section;
 …
release(&mylock);
```

```
INIT
   int mylock = 0;

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

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

```
acquire(&mylock);
 …
 critical section;
 …
release(&mylock);
```


Recall: In-Kernel Lock: Simulation

mylock: 1 | waiters | owner | READY

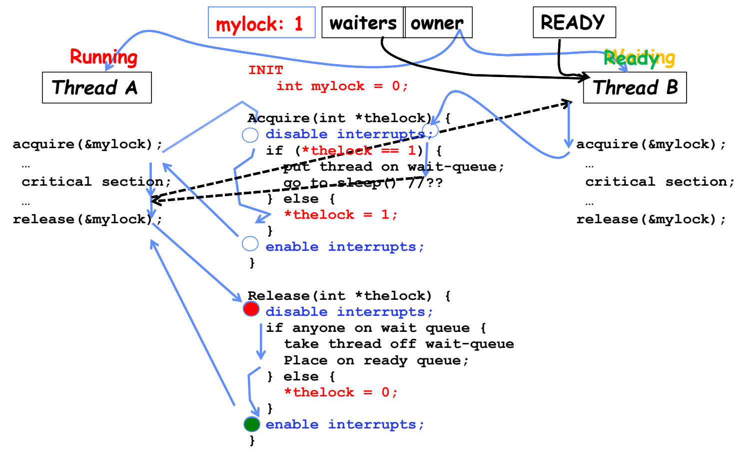
Running / Ready: Thread A — Ready / Running / Waiting: Thread B

```
acquire(&mylock);
 …
 critical section;
 …
release(&mylock);
```

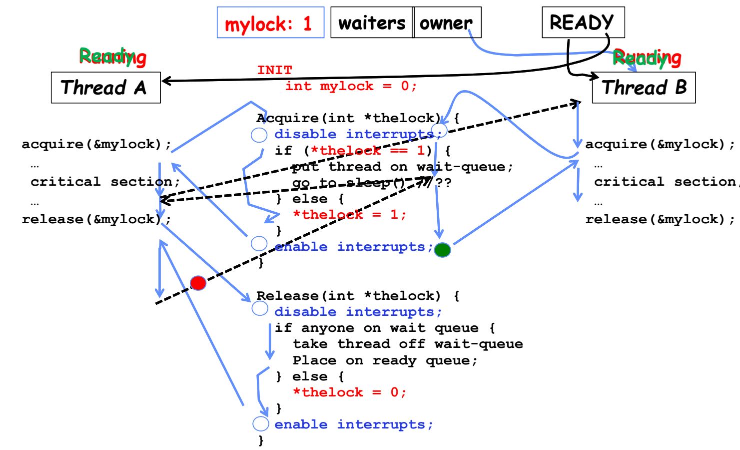
```
INIT
   int mylock = 0;

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

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

```
acquire(&mylock);
 …
 critical section;
 …
release(&mylock);
```

Recall: In-Kernel Lock: Simulation



Recall: In-Kernel Lock: Simulation



Recall: Atomic Read-Modify-Write Instructions

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

Examples of Read-Modify-Write

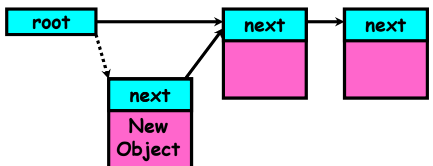
```
• test&set (&address) {           /* most architectures */
      result = M[address];         // return result from "address" and
      M[address] = 1;              // set value at "address" to 1
      return result;
  }
• swap (&address, register) {     /* x86 */
      temp = M[address];           // swap register's value to
      M[address] = register;       // value at "address"
      register = temp;
  }
• compare&swap (&address, reg1, reg2) { /* x86 (returns old value), 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;
  }
```

Using of Compare&Swap for queues

```
• compare&swap (&address, reg1, reg2) { /* x86, 68000 */
      if (reg1 == M[address]) {
         M[address] = reg2;
         return success;
      } else {
         return failure;
      }
  }
```

Here is an atomic add to linked-list function:

```
addToQueue(&object) {
   do {                   // repeat until no conflict
      ld r1, M[root]      // Get ptr to current head
      st r1, M[object]  // Save link in new object
   } until (compare&swap(&root,r1,object));
}
```



Implementing Locks with test&set

- Simple lock that doesn't require entry into the kernel:

```
// (Free) Can access this memory location from user space!
int mylock = 0; // Interface: acquire(&mylock);
                //            release(&mylock);
acquire(int *thelock) {
   while (test&set(thelock)); // Atomic operation!
}
release(int *thelock) {
   *thelock = 0;             // Atomic operation!
}
```

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

```
// (Free) Can access this memory location from user space!
int mylock = 0; // Interface: acquire(&mylock);
                //            release(&mylock);
acquire(int *thelock) {
   do {
      while(*thelock);        // Wait until might be free (quick check/test!)
   } while(test&set(thelock)); // Atomic grab of lock (exit if succeeded)
}
release(int *thelock) {
   *thelock = 0;           // Atomic release of lock
}
```

- 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
- 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?
  - Mostly. Idea: only busy-wait to atomically check lock value
  -
```
int guard = 0; // Global Variable!
int mylock = FREE; // Interface: acquire(&mylock);
                   //            release(&mylock);
```

```
acquire(int *thelock) {
   // Short busy-wait time
   while (test&set(guard));
   if (*thelock == BUSY) {
      put thread on wait queue;
      go to sleep() & guard = 0;
      // guard == 0 on wakup!
   } else {
      *thelock = BUSY;
      guard = 0;
   }
}

release(int *thelock) {
   // Short busy-wait time
   while (test&set(guard));
   if anyone on wait queue {
      take thread off wait queue
      Place on ready queue;
   } else {
      *thelock = 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; // Interface: acquire(&mylock);
                  //            release(&mylock);
```

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

release(int *thelock) {
   disable interrupts;
   if (anyone on wait queue) {
      take thread off wait queue
      Place on ready queue;
   } else {
      *thelock = FREE;
   }
   enable interrupts;
}
```

Basically we replaced:
- **disable interrupts → while (test&set(guard));**
- **enable interrupts → guard = 0;**

Recap: Locks using interrupts

```
int mylock=0;

acquire(&mylock);
 …
 critical section;
 …
release(&mylock);
```

```
acquire(int *thelock) {
  disable interrupts;
}

release(int *thelock)
{
  enable interrupts;
}
```

```
acquire(int *thelock) {
  // Short busy-wait time
  disable interrupts;
  if (*thelock == 1) {
    put thread on wait-queue;
    go to sleep() //??
  } else {
    *thelock = 1;
    enable interrupts;
  }
}

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

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

Lock argument not used!

Recap: Locks using test & set

```
int mylock=0;

acquire(&mylock);
 …
 critical section;
 …
release(&mylock);
```

```
int mylock = 0;
acquire(int *thelock) {
  while(test&set(thelock));
}

release(int *thelock) {
  *thelock = 0;
}
```

```
int guard = 0; // global!

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

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

Threads waiting to enter critical section busy-wait

Linux `futex`: Fast Userspace Mutex

```
#include <linux/futex.h>
#include <sys/time.h>

int futex(int *uaddr, int futex_op, int val,
         const struct timespec *timeout );
```

`uaddr` points to a 32-bit value in user space

`futex_op`
- `FUTEX_WAIT` – if `val == *uaddr` sleep till `FUTEX_WAIT`
  - » ***Atomic*** check that condition still holds after we disable interrupts (in kernel!)
- `FUTEX_WAKE` – wake up at most `val` waiting threads
- `FUTEX_FD`, `FUTEX_WAKE_OP`, `FUTEX_CMP_REQUEUE`: More interesting operations!

`timeout`
- ptr to a *timespec* structure that specifies a timeout for the op

- `Interface to the kernel sleep() functionality!`
  - `Let thread put themselves to sleep – conditionally!`
- futex is not exposed in libc; it is used within the implementation of pthreads
  - Can be used to implement locks, semaphores, monitors, etc…

Example: First try: T&S and `futex`

```
int mylock = 0; // Interface: acquire(&mylock);
                //            release(&mylock);

acquire(int *thelock) {
   while (test&set(thelock)) {
      futex(thelock, FUTEX_WAIT, 1);
   }
}

release(int *thelock) {
   thelock = 0; // unlock
   futex(&thelock, FUTEX_WAKE, 1);
}
```

- Properties:
  - Sleep interface by using futex – no busywaiting
- No overhead to acquire lock
  - Good!
- Every unlock has to call kernel to potentially wake someone up – even if none
  - Doesn't quite give us no-kernel crossings when uncontended…!

Example: Try #2: T&S and `futex`

```
bool maybe_waiters = false;
int mylock = 0; // Interface: acquire(&mylock,&maybe_waiters);
                //            release(&mylock,&maybe_waiters);

acquire(int *thelock, bool *maybe) {
   while (test&set(thelock)) {
      // Sleep, since lock busy!
      *maybe = true;
      futex(thelock, FUTEX_WAIT, 1);

      // Make sure other sleepers not stuck
      *maybe = true;
   }
}

release(int*thelock, bool *maybe) {
   value = 0;
   if (*maybe) {
      *maybe = false;
      // Try to wake up someone
      futex(&value, FUTEX_WAKE, 1);
   }
}
```

- This is syscall-free in the uncontended case
  - Temporarily falls back to syscalls if multiple waiters, or concurrent acquire/release
- But it can be considerably optimized!
  - See "Futexes are Tricky" by Ulrich Drepper

Try #3: Better, using more atomics

- Much better: Three (3) states:
  - UNLOCKED: No one has lock
  - LOCKED: One thread has lock
  - CONTESTED: Possibly more than one (with someone sleeping)
- Clean interface!
- Lock grabbed cleanly by either
  - `compare_and_swap()`
  - First `swap()`
- No overhead if uncontested!
- Could build semaphores in a similar way!

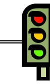
```
typedef enum { UNLOCKED,LOCKED,CONTESTED } Lock;
Lock mylock = UNLOCKED; // Interface: acquire(&mylock);
                        //            release(&mylock);

acquire(Lock *thelock) {
   // If unlocked, grab lock!
   if (compare&swap(thelock,UNLOCKED,LOCKED))
      return;

   // Keep trying to grab lock, sleep in futex
   while (swap(mylock,CONTESTED) != UNLOCKED))
      // Sleep unless someone releases hear!
      futex(thelock, FUTEX_WAIT, CONTESTED);
}

release(Lock *thelock) {
   // If someone sleeping,
   if (swap(thelock,UNLOCKED) == CONTESTED)
      futex(thelock,FUTEX_WAKE,1);
}
```

Recall: Where are we going with synchronization?

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

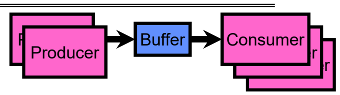
- We are going to implement various higher-level synchronization primitives using atomic operations
  - Everything is pretty painful if only atomic primitives are load and store
  - Need to provide primitives useful at user-level

Recall: 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 operations:
  - Set value when you initialize
  - `Down()` or `P()`: an atomic operation that waits for semaphore to become positive, then decrements it by 1
    - » Think of this as the wait() operation
  - `Up()` or `V()`: an atomic operation that increments the semaphore by 1, waking up a waiting P, if any
    - » This of this as the signal() operation
- Technically examining value after initialization is not allowed.

Recall 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
  - To ensure correctness of the queue/buffer implementation!
- 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`

Recall: Full Solution to Bounded Buffer (coke machine)


```
Semaphore fullSlots = 0;     // Initially, no coke
Semaphore emptySlots = bufSize;
                             // Initially, num empty slots
Semaphore mutex = 1;         // No one using machine

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

fullSlots signals coke

emptySlots signals space

Critical sections using mutex protect integrity of the queue

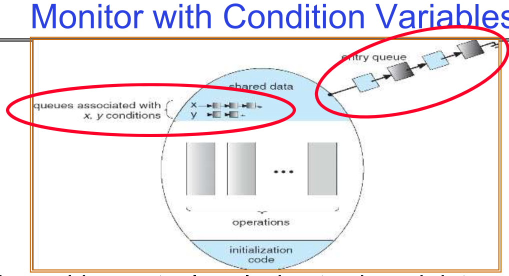
Semaphores are good but…Monitors are better!

- Semaphores are a huge step up; just think of trying to do the bounded buffer with only loads and stores or even with locks!
- 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);
acquire(&buf_lock)             …
…                              while (isEmpty(&queue)) {
cond_signal(&buf_CV);            cond_wait(&buf_CV,&buf_lock);
…                              }
release(&buf_lock));           …
                               lock.Release();
```

Put waiting thread on ready queue

schedule thread (sometime later!)

- 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)

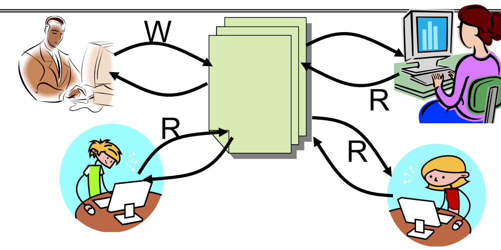
```
lock buf_lock = <initially unlocked>
condition producer_CV = <initially empty>
condition consumer_CV = <initially empty>

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?

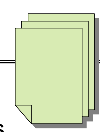
- MESA semantics
- For most operating systems, when a thread is woken up by `signal()`, it is simply put on the ready queue
- It may or may not reacquire the lock immediately!
  - Another thread could be scheduled first and "sneak in" to empty the queue
  - Need a loop to re-check condition on wakeup

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);
}
```


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
- Showed \primitive for constructing user-level locks
  - Packages up functionality of sleeping

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
- Next time: Continue on Readers/Writers example