CSE 451: Operating Systems
Spring 2022

Module 7
Synchronization (cont.)

John Zahorjan

Synchronization Variable Interfaces

- (spin) lock
  - acquire() / release() [lock()/unlock()]

- (blocking) lock [mutex]
  - acquire() / release() [lock()/unlock()]

- Semaphore(int n)
  - deprecated
  - P – if value <= 0 then block; decrement value
  - V – increment value; if there is a waiter, wake one up
  - *binary semaphore (semaphore(1)) is a lock*

- Condition variable(lock)
  - wait() - suspend this thread and release lock
  - signal() - wake up one waiting thread, if there is one, and regain its lock
  - broadcast() - wake up all waiting threads, if any, and let them battle for lock

Part I: Locks

- xk implementation
- locking granularity

Spinlock Implementation in xk

```
void acquire(struct spinlock *lk) {
  pushcli(); // disable interrupts to avoid deadlock.
  if (holding(lk)) {
      cprintf("name=%s",lk->name);
      panic("acquire");
  }

  // The xchg is atomic.
  while (xchg(&lk->locked, 1) != 0)
    ;
  __sync_synchronize();

  // Record info about lock acquisition for
  //  debugging.
  lk->cpu = mycpu();
  getcallerpcs(&lk, lk->pcs);
}
```

```
void release(struct spinlock *lk) {
  if (!holding(lk))
    panic("release");

  lk->pcs[0] = 0;
  lk->cpu = 0;

  __sync_synchronize();

  // Release the lock, equivalent to lk->locked = 0.
  // This code can't use a C assignment, since it might
  // not be atomic. A real OS would use C atomics here.
  asm volatile("movl $0, %0" : "+m"(lk->locked) :);

  popcli();
}
```

Side Issue: How many spinlocks?

- Locking Granularity

- Various data structures
  - Queue of waiting threads on lock X
  - Queue of waiting threads on lock Y
  - List of threads ready to run

- One spinlock per kernel?
  - Bottleneck...

- Instead:
  - One spinlock per blocking lock
  - One spinlock for the scheduler ready list
    - Per-core ready list: one spinlock per core

Blocking / Controlling Core Usage

- What should code do when it detects it can't make useful progress right now?
  - E.g., a lock it needs is in use
  - E.g., it needs a message from the network but there isn't one right now
- The code obviously is using a hardware core
  - The instructions that detected the issue were executed…

- Should it give up the core or should it spin?
  - How long will it have to spin? On average? Worst case?
  - How long does it take to block this thread and resume a different one?
  - *Aside: spin for the context switch time then block (forcing a context switch) is within a factor of 2 of optimal in all cases*

- A mutex has the semantics of a lock, but differs from spinlocks by blocking a thread that invokes acquire() when the lock is already held
  - The thread is put on a queue of threads blocked waiting to acquire that particular lock

Yielding vs. Blocking

- A call to yield() relinquishes the core to a different, runnable thread, if there is one
  - yield'ing thread is put on a runnable queue, which isn't the same as blocked
- The yielding thread resumes execution when it happens to be allocated a core by the OS CPU scheduler
  - If a thread calls yield() because it can't make progress right now, when it resumes has nothing to do with whether or not whatever it needed has become available
  - Programmer controls when to relinquish core, but not when to resume

Mutex Implementation, Unicore

```
Mutex::acquire() {
    disableInterrupts();
    if (value == BUSY) {
        waiting.add(myTCB);
        myTCB->state = WAITING;
        next = readyList.remove();
        switch(myTCB, next);
        myTCB->state = RUNNING;
    } else {
        value = BUSY;
    }
    enableInterrupts();
}
```

```
Mutex::release() {
    disableInterrupts();
    if (!waiting.Empty()) {
        next = waiting.remove();
        next->state = READY;
        readyList.add(next);
    } else {
        value = FREE;
    }
    enableInterrupts();
}
```

Mutex Implementation, Multicore

```
Mutex::acquire() {
    disableInterrupts();
    spinLock.acquire();
    if (value == BUSY) {
        waiting.add(myTCB);
        suspend(&spinlock);
    } else {
        value = BUSY;
    }
    spinLock.release();
   enableInterrupts();
}
```

```
Mutex::release() {
    disableInterrupts();
    spinLock.acquire();
    if (!waiting.Empty()) {
        next = waiting.remove();
        scheduler->makeReady(next);
    } else {
        value = FREE;
    }
    spinLock.release();
    enableInterrupts();
}
```

Linux Mutex Implementation

- Guess that most locks are free most of the time
  - Why?
  - Linux implementation takes advantage of this “fact”
- **Fast path**
  - If lock is FREE, and no one is waiting, two instructions to acquire the lock
  - If no one is waiting, two instructions to release the lock
- **Slow path**
  - If lock is BUSY or someone is waiting, use multicore implementation

Linux Mutex Implementation

```
struct mutex {
 /* 1: unlocked
    0: locked
    negative : locked, possible
    waiters
*/
 atomic_t count;
 spinlock_t wait_lock;
 struct list_head wait_list;
};
```

```
// acquire()
// atomic decrement
// %eax is pointer to count
lock decl (%eax)
jns 1 // jump if not signed
      // (if value is now 0)
call slowpath_acquire
1:
```

Unlock is similar – atomic increment and test for result greater than zero.

Part 2: Condition Variables

- Condition variables don't provide mutual exclusion
- They address a race condition having to do with blocking and locks
- *They defer the policy of when to block to application code*

  (rather than make it part of the semantics of the synchronization variable)
- *They defer the policy of when to unblock the thread to application code*

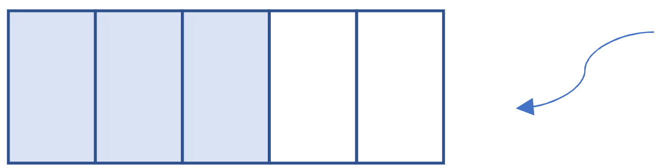
```
get() {
  lock(buffer);
  if (full slot) {
    retval = contents of slot;
    mark slot empty;
    unlock(buffer);
     return retval;
  } else {
     ???
  }
}
```

```
put(item) {
  lock(buffer);
  if (empty slot) {
    empty slot = item;
     mark slot full;
     unlock(buffer);
  } else {
     ???
  }
}
```

Classic example: bounded buffer

Bounded Buffer: Naive Fix 1

Idea: Don't acquire the lock until you're sure there's an empty slot.

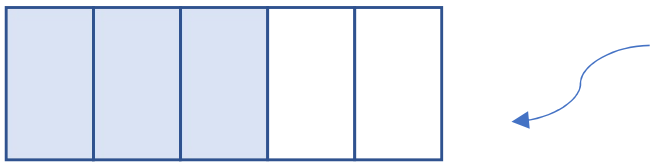


```
put(item) {
  if (empty slot) {
    lock(buffer);
   empty slot = item;
    mark slot full;
    unlock(buffer);
  } else {
    while (no empty slot) {};
     lock(buffer):
     assign item to empty slot;
     mark empty slot full;
     unlock(buffer);
  }
```

Why doesn't this work?

Bounded Buffer: Naive Fix 2

Idea: Try, try again.



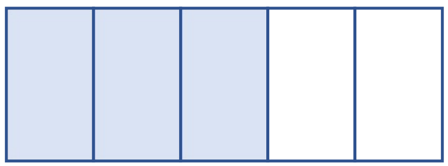
```
put(item) {
   done = false;
   while (!done) {
         lock(buffer);
          if ( empty buffer) {
             assign item to empty slot;
              mark empty slot full;
               done = true;
          }
          unlock(buffer);
    }
}
```

Does this work?

How long will the thread spin?

Bounded Buffer: Naive Fix 3

Idea: Not my problem.



```
put(item) {
    result = false;
    lock(buffer);
    if ( empty slot) {
        assign item to empty slot;
        mark empty slot full;
        result = true;
    }
    unlock(buffer);
    return result;
}
```

Does this work?

Why Can't I Get This Right?

- The thread needs to hold a lock while it's checking for some condition
  - Otherwise, checking is basically useless
    - E.g., is there a free slot?

- If the condition doesn't hold, the thread can't proceed

- So, it needs to block

- It also needs to release the lock
  - otherwise the condition can't be changed (by any other thread)

Why Can’t I Get This Right?

- Situation: A thread holding a lock needs to wait until some condition holds
- “Wait” by blocking, not spinning
- Ideally, when it blocks it will be woken up only when there’s reason to believe the condition holds
  - So, it’s woken up by some other thread that thinks this would be a good time to wake up
    - e.g., other thread observes the condition holds
    - (it would be very unusual for the decision to wake up to be based on time)

Why Can’t I Get This Right?

- Situation: A thread holding a lock needs to
  - wait until some condition holds
  - unlock the lock
- Thread can’t block then release the lock
  - Why?
- Thread also can’t release the lock then block
  - Why?
- We need a single, atomic action that both suspends the thread and releases the lock it has
  - condition variables!

Condition Variables (CVs)

- Condition variables solve the “can’t block then unlock and can’t unlock then block” problem

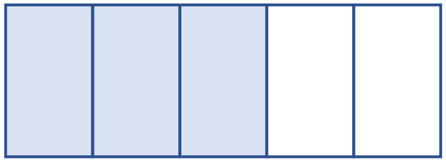
- Condition variables have two *(three)* operations
  - wait(cv, lock)
  - signal(cv)
  - *broadcast(cv)*

- wait(cv,lock): atomically
  - blocks the calling thread and puts it on a queue associated with the CV
  - unlocks the lock

- signal(cv):
  - wakes up a single thread blocked on the CV, if there is one, and otherwise does nothing
  - If a thread is woken up, it reacquires the lock then returns from the wait(cv,lock) call that had blocked it

- *broadcast(cv) wakes up all blocked threads, if any, but only one can gain the lock at a time*

Bounded Buffer: Condition Variables

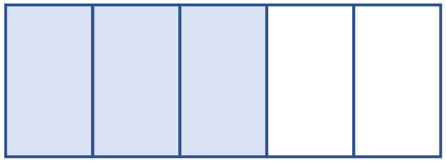


```
Lock               bufferlock;
ConditionVariable  emptyCV;
ConditionVariable  fullCV;
```

```
get() {
    lock(bufferlock);
    while ( buffer is empty ) {
        wait(fullCV, bufferlock);
    }
    take item;
    mark slot empty;
    signal(emptyCV);
    unlock(bufferlock);
    return item;
}
```

```
put(item) {
    lock(bufferlock);
    while ( buffer is full) {
        wait(emptyCV, bufferlock);
    }
    assign item to empty slot;
    mark empty slot full;
    signal(fullCV);
    unlock(bufferlock);
}
```

Bounded Buffer: Condition Variables



```
Lock               bufferlock;
ConditionVariable  emptyCV;
ConditionVariable  fullCV;
```

```
get() {
    lock(bufferlock);
    while ( buffer is empty ) {
        wait(fullCV, bufferlock);
    }
    take item;
    mark slot empty;
    signal(emptyCV);
    unlock(bufferlock);
    return item;
}
```

```
put(item) {
    lock(bufferlock);
    while ( buffer is full) {
        wait(emptyCV, bufferlock);
    }
    assign item to empty slot;
    mark empty slot full;
    signal(fullCV);
    unlock(bufferlock);
}
```

What if no thread is waiting when signal() is called?

Why the while loops?

```
get() {
    lock(bufferlock);
    while ( buffer is empty ) {
        wait(fullCV, bufferlock);
    }
    take item;
    mark slot empty;
    signal(emptyCV);
    unlock(bufferlock);
    return item;
}
```

```
put(item) {
    lock(bufferlock);
    while ( buffer is full) {
        wait(emptyCV, bufferlock);
    }
    assign item to empty slot;
    mark empty slot full;
    signal(fullCV);
    unlock(bufferlock);
}
```

- No one said the thread woken up by signal() must be the thread that next acquires the lock!
  - Some other thread could run after the signal and before the awoken thread, and invalidate the condition
  - *signal'ing just puts a blocked thread on the ready queue…*
- Okay, Tony Hoare said the awoken thread gets the lock, but it's too restrictive to implement that so everyone uses Mesa semantics

Condition Variable Use Correctness

1. What if your code forgets to signal?
2. What if your code signals before some thread waits (and not again after)?
3. What if your code signals when the condition a blocked thread is waiting for doesn't hold?
4. What if you just signal every other instruction ("because you feel like it")?

“Rules” for Using Synchronization

- Use consistent structure
- Always use locks and condition variables
- Always acquire lock at beginning of procedure and release at end
- Always hold lock when using a condition variable
- Always wait() in while loop
- Never spin in sleep()