Synchronization Part 2
Module 7

Implementing Synchronization

Concurrent Applications

Semaphores | Locks | Condition Variables

Interrupt Disable | Atomic Read/Modify/Write Instructions

Multiple Processors | Hardware Interrupts

Synchronization Variable Interfaces

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

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

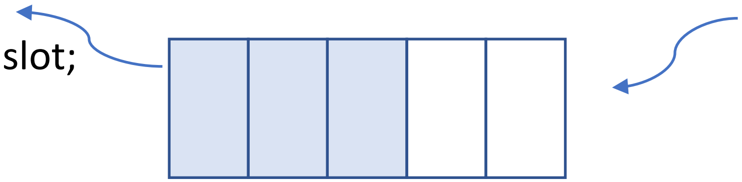
- Semaphore(int n)
  - deprecated
  - P – if value <= 0 then wait; decrement value
  - V – increment value; if there is a waiter, wake one up
  - binary semaphore (semaphore(1)) is a lock

- 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 1: Condition Variables

- Condition variables don't provide mutual exclusion
- They address a problem with using locks
- Classical example: bounded buffer

```
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 {
     ???
  }
```

Bounded Buffer: Naive Fix 1

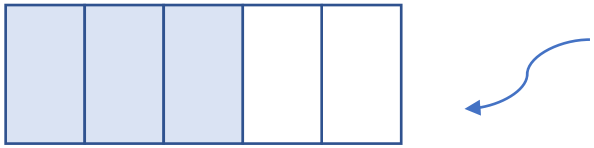
Idea: Don’t acquire the lock until you’re sure there’s an empty slot.

Why doesn’t this work?

```
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);
  }
```

Bounded Buffer: Naive Fix 2

Idea: Try, try again.

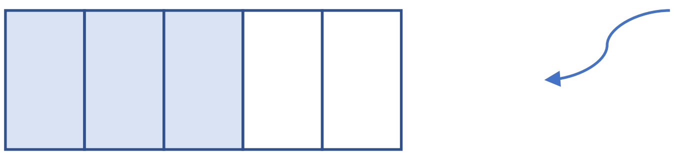


```
put(item) {
  done = false;
  while (!done) {};
        lock(buffer);
         if ( empty buffer) {
            assign item to empty slot;
             mark empty slot full;
              done = true;
         }
         unlock(buffer);
    }
}
```

Does this work?

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 T holding a lock needs to wait until some condition holds

- “Wait” by blocking, not spinning
  - Why not spin
    - as fast as you can?
    - slowly (embed sleep() in loop)?
  - Auto mechanic shop question from midterm...

- 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 observes the condition holds
    - Not 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 ghen block” problem

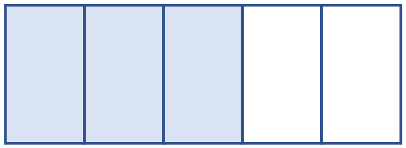
- Condition variables have two operations
  - wait(cv, lock)
  - signal(cv)

- 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(buffer);
}
```

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;
}
```

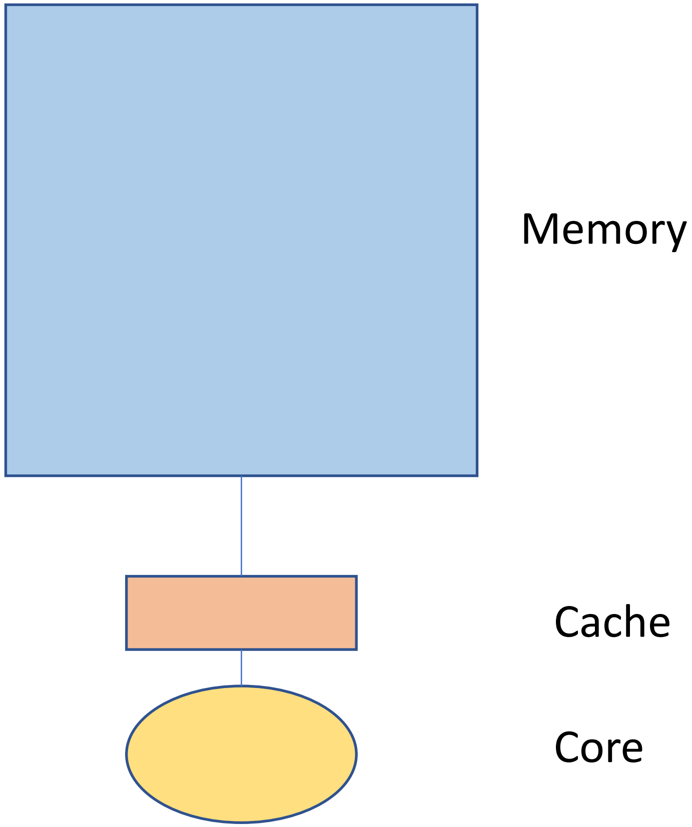
```
put(item) {
    lock(bufferlock);
    while ( buffer is full) {
        wait(emptyCV, bufferlock);
    }
    assign item to empty slot;
    mark empty slot full;
    signal(fullCV);
    unlock(buffer);
}
```

- 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
- 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 or signals before some thread waits (and not again after)?
3. What if your code mistakenly signals a condition when it doesn’t hold?
4. What if you just signal every other instruction (“because you feel like it”)?

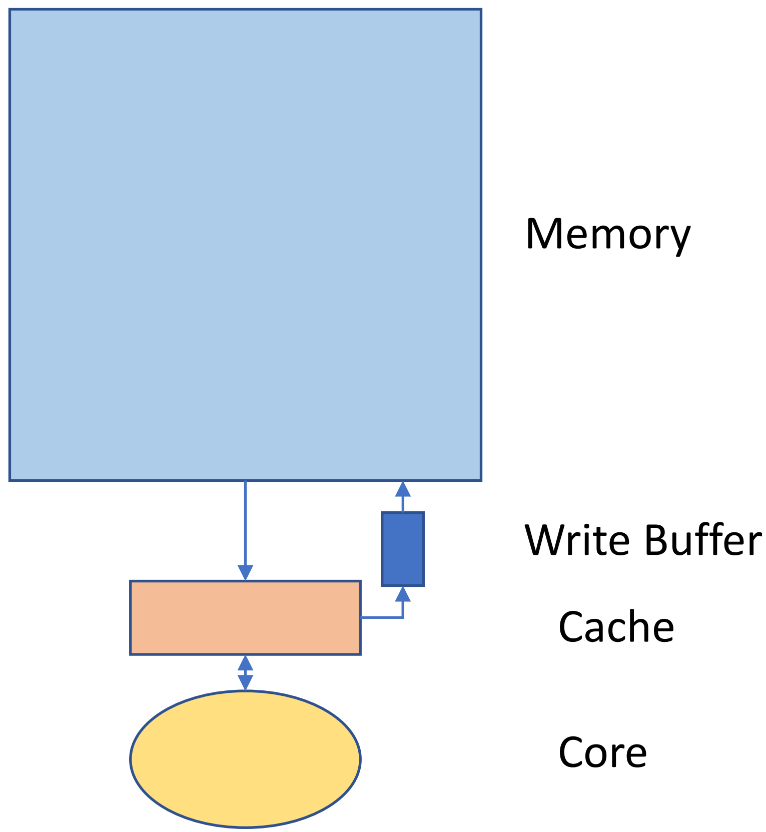
Part II: Memory Semantics



Cache is fast compared to memory.

Cache is slow compared to core.

Write Buffer



Write buffer absorbs (limited) bursts of writes.

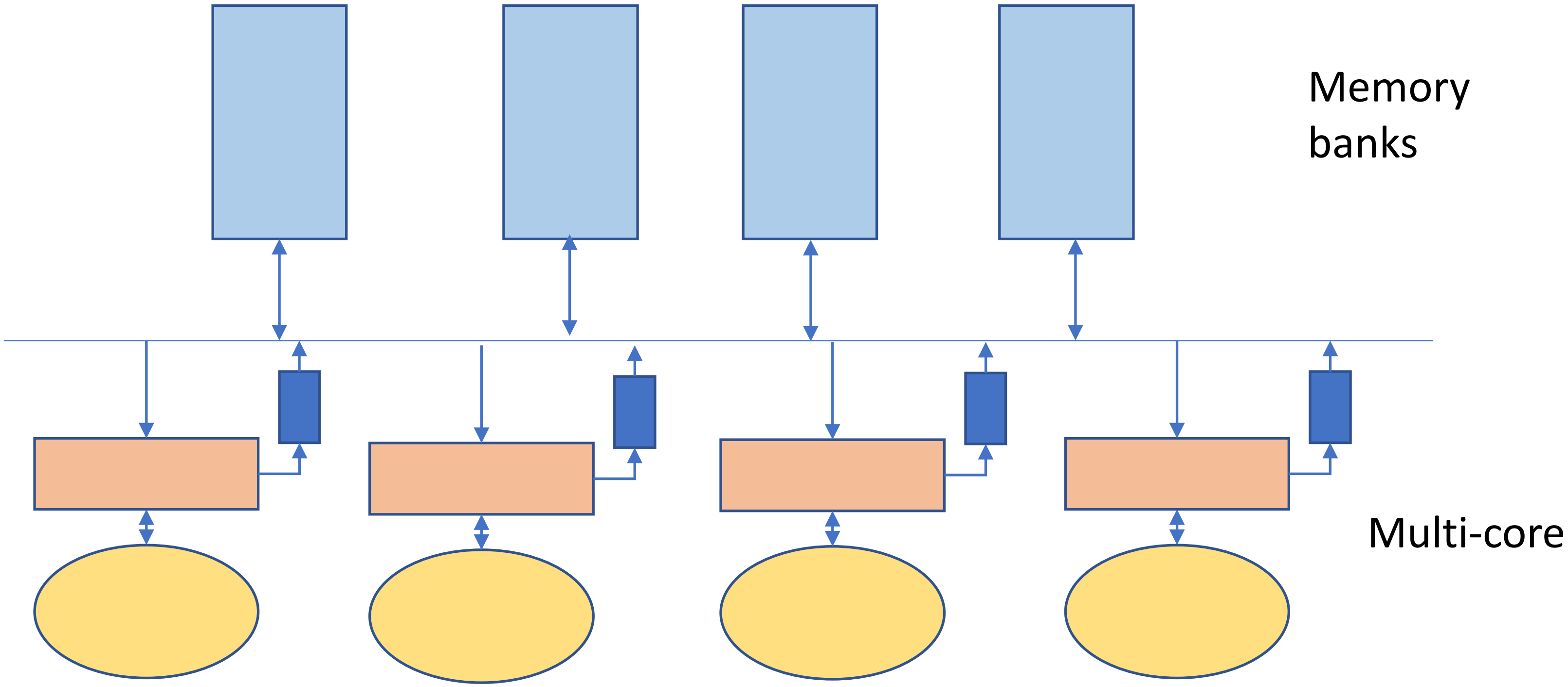
When reading a memory location, most recent value written may be in:

- cache
- write buffer
- memory

So, core always sees "sequential semantics" for its own writes

- the value it reads is the last one it wrote
- as seen by its own reads, writes happen in order

But what about this?



Suppose locations A and B start out with value 0.
Core 0 writes 1 to A and then 1 to B, and no one else writes.

Can core 0 read memory and find B==1 and A==0?
Can core 2 read memory and find B==1 and A==0?

Question: Can this panic?

Thread 1

```
p = someComputation();
pInitialized = true;
```

Thread 2

```
while (!pInitialized)
    ;
q = someFunction(p);
if (q != someFunction(p))
    panic
```

Will this code work?

```
if (p == NULL) {
    lock.acquire();
    if (p == NULL) {
        p = newP();
    }
    lock.release();
}
use p->field1
```

```
newP() {
    p = malloc(sizeof(p));
    p->field1 = …
    p->field2 = …
    return p;
}
```

Why Does Reordering Occur?

Why do compilers reorder instructions?

- Efficient code generation requires analyzing control/data dependency
- If variables can spontaneously change, most compiler optimizations become impossible

Why do CPUs reorder instructions?

- Write buffering: allow next instruction to execute while write is being completed

Fix: **memory barrier**

- Instruction to compiler/CPU
- All ops before barrier complete before barrier returns
- No op after barrier starts until barrier returns

The implementations of synchronization primitives perform memory barriers.

Your code probably doesn’t (except by correctly synchronizing)!

Spinlock Implementation in xk

```
void acquire(struct spinlock *lk) {
  pushcli(); // disable interrupts to avoid deadlock.
  if (holding(lk))
    panic("acquire");

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

  // Tell the C compiler and the processor to not move loads or stores
  // past this point, to ensure that the critical section's memory
  // references happen after the lock is acquired.
  __sync_synchronize();

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

Spinlock Implementation in xk

```
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();
}
```

How many spinlocks?

- 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

Mutex Implementation, Uniprocessor

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

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

Lock Implementation, Multiprocessor

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

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

What thread is currently running?

- Thread scheduler needs to find the TCB of the currently running thread
  - To suspend and switch to a new thread
  - To check if the current thread holds a lock before acquiring or releasing it

- On a uniprocessor, easy: just use a global

- On a multiprocessor, various methods:
  - Compiler dedicates a register (e.g., r31 points to TCB running on the this CPU; each CPU has its own r31)
  - If hardware has a special per-processor register, use it
  - Fixed-size stacks: put a pointer to the TCB at the bottom of its stack
    - Find it by masking the current stack pointer

Lock Implementation, Linux

- 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 multiproc impl.

- User-level locks
  - Fast path: acquire lock using test&set
  - Slow path: system call to kernel, use kernel lock

Lock Implementation, Linux

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

```
// 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:
```

“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()