CS 4284
Systems Capstone
Concurrency & Synchronization
Godmar Back
Concurrency & Synchronization
Overview

• Will talk about locks, semaphores, and monitors/condition variables
• For each, will talk about:
  – What abstraction they represent
  – How to implement them
  – How and when to use them
• Two major issues:
  – Mutual exclusion
  – Scheduling constraints
• Project note: Pintos implements its locks on top of semaphores
/* Define a mutex and initialize it. */
static pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

static int counter = 0; /* A global variable to protect. */

/* Function executed by each thread. */
static void *
increment(void *__) 
{
    int i;
    for (i = 0; i < 1000000; i++) {
        pthread_mutex_lock(&lock);
        counter++;
        pthread_mutex_unlock(&lock);
    }
}

movl  counter, %eax
incl  %eax
movl  %eax, counter
A Race Condition

Thread 1:

```
movl counter, %eax
incl %eax
movl %eax, counter
```

Thread 2:

```
movl counter, %eax
incl %eax
movl %eax, counter
```

Assume counter == 0 initially

Final result: counter is 1, should be 2
Race Conditions

- Definition: *two or more threads read and write a shared variable, and final result depends on the order of the execution of those threads*
- Usually timing-dependent and intermittent
  - Hard to debug
- Not a race condition if all execution orderings lead to same result
  - Chances are high that you misjudge this
- How to deal with race conditions:
  - Ignore (!?)
    • Can be ok if final result does not need to be accurate
    • Never an option in CS 3204, 3214, or 4284
  - Don’t share: duplicate or partition state
  - Avoid “bad interleavings” that can lead to wrong result
Not Sharing: Duplication or Partitioning

• Undisputedly best way to avoid race conditions
  – Always consider it first
  – Usually faster than alternative of sharing + protecting
  – But duplicating has space cost; partitioning can have management cost
  – Sometimes must share (B depends on A’s result)

• Examples:
  – Each thread has its own counter (then sum counters up after join())
  – Every CPU has its own ready queue
  – Each thread has its own memory region from which to allocate objects

• Truly ingenious solutions to concurrency involve a way to partition things people originally thought you couldn’t
Aside: Thread-Local Storage

- A concept that helps to avoid race conditions by giving each thread a copy of a certain piece of state

Recall:
- All local variables are already thread-local
  - But their extent is only one function invocation
- All function arguments are also thread-local
  - But must pass them along call-chain

- TLS creates variables of which there’s a separate value for each thread.

- In PThreads/C (compiler or library-supported)
  - Dynamic: pthread_create_key(), pthread_get_key(), pthread_set_key()
    - E.g. myvalue = keytable(key_a)→get(pthread_self());
  - Static: using __thread storage class
    - E.g.: __thread int x;

- Java: java.lang.ThreadLocal

In Pintos:
Add member to struct thread
Race Condition & Execution Order

• Prevent race conditions by imposing constraints on execution order so the final result is the same regardless of actual execution order
  – That is, exclude “bad” interleavings
  – *Specifically*: disallow other threads to start updating shared variables while one thread is in the middle of doing so; make those updates *atomic* – threads either see old or new value, but none in between
Atomicity & Critical Sections

• Atomic: indivisible
  – Certain machine instructions are atomic
  – But need to create larger atomic sections

• Critical Section
  – A synchronization technique to ensure atomic execution of a segment of code
    • Requires entry() and exit() operations

```c
pthread_mutex_lock(&lock);    /* entry() */
counter++;
pthread_mutex_unlock(&lock); /* exit() */
```
Critical Sections (cont’d)

- Critical Section Problem also known as mutual exclusion problem
- Only one thread can be inside critical section; others attempting to enter CS must wait until thread that's inside CS leaves it.
- Solutions can be entirely software, or entirely hardware
  - Usually combined
  - Different solutions for uniprocessor vs multicore/processor scenarios
Critical Section Problem

- A solution for the CS Problem must
  1) Provide mutual exclusion: at most one thread can be inside CS
  2) Guarantee Progress: (no deadlock)
     - if more than one threads attempt to enter, one will succeed
     - ability to enter should not depend on activity of other threads not currently in CS
  3) Bounded Waiting: (no starvation)
     - A thread attempting to enter critical section eventually will (assuming no thread spends unbounded amount of time inside CS)

- A solution for CS problem should be
  - Fair (make sure waiting times are balanced)
  - Efficient (not waste resources)
  - Simple
Myths about CS

• A thread in a CS executes entire section without being preempted
  – No – not usually true
• A thread in a CS executes the entire section
  – No – may exit or crash
• There can be only one critical section in a program
  – No – as many as the programmer decides to use
• Critical sections protect blocks of code
  – No – they protect data accesses (but note role of encapsulation!)
Implementing Critical Sections

• Will look at:
  – Disabling interrupts approach
  – Semaphores
  – Locks
Disabling Interrupts

• All asynchronous context switches start with interrupts
  – So disable interrupts to avoid them!

```c
intr_level old = intr_disable();
/* modify shared data */
intr_set_level(old);

void intr_set_level(intr_level to)
{
    if (to == INTR_ON)
        intr_enable();
    else
        intr_disable();
}
```
Implementing CS by avoiding context switches: Variation (1)

- Variation of “disabling-interrupts” technique
  - That doesn’t actually disable interrupts
  - If IRQ happens, ignore it
- Assumes writes to “taking_interrupts” are atomic and sequential wrt reads

```c
taking_interrupts = false;
/* modify shared data */
taking_interrupts = true;

intr_entry()
{
    if (!taking_interrupts)
        iret
    intr_handle();
}
```
Implementing CS by avoiding context switches: Variation (2)

• Code on previous slide could lose interrupts
  – Remember pending interrupts and check when leaving critical section

• This technique can be used with Unix signal handlers (which are like “interrupts” sent to a Unix process)
  – but tricky to get right

```c
taking_interrupts = false; /* modify shared data */
if (irq_pending)
  intr_handle();
taking_interrupts = true;

intr_entry()
{
  if (!taking_interrupts) {
    irq_pending = true;
    iret
  }
  intr_handle();
}
```
Avoiding context switches: Variation (3)

• Instead of setting flag, have irq handler examine PC where thread was interrupted

• See Bershad ’92: **Fast Mutual Exclusion on Uniprocessors**

```c
critical_section_start:
  /* modify shared data */
critical_section_end:

intr_entry()
{
  if (PC in (critical_section_start, critical_end_end)) {
    iret
  }
  intr_handle();
}
```
Disabling Interrupts: Summary

• (this applies to all variations)
• Sledgehammer solution
• Infinite loop means machine locks up
• Use this to protect data structures from concurrent access by interrupt handlers
  – Keep sections of code where irqs are disabled minimal (nothing else can happen until irqs are reenabled – latency penalty!)
  – If you block (give up CPU) mutual exclusion with other threads is not guaranteed
    • Any function that transitively calls thread_block() may block
• Want something more fine-grained
  – Key insight: don’t exclude everybody else, only those contending for the same critical section
Locks

- Thread that enters CS locks it
  - Others can’t get in and have to wait
- Thread unlocks CS when leaving it
  - Lets in next thread
  - which one?
    - FIFO guarantees bounded waiting
    - Highest priority in Proj1
- Can view Lock as an abstract data type
  - Provides (at least) init, acquire, release
Locks, Take 2

Locks don’t work if threads don’t agree on which locks (doors) to use to get to shared data, or if threads don’t use locks (doors) at all.

Shared Data
Implementing Locks

• Locks can be implemented directly, or – among other options - on top of semaphores
  – If implemented on top of semaphores, then semaphores must be implemented directly
  – Will explain this layered approach first to help in understanding project code
  – Issues in direct implementation of locks apply to direct implementation of semaphores as well
Semaphores

• Invented by Edsger Dijkstra in 1960s
• Counter S, initialized to some value, with two operations:
  – P(S) or “down” or “wait” – if counter greater than zero, decrement. Else wait until greater than zero, then decrement
  – V(S) or “up” or “signal” – increment counter, wake up any threads stuck in P.
• Semaphores don’t go negative:
  – \#V + InitialValue - \#P >= 0
• Note: direct access to counter value after initialization is not allowed
• Counting vs Binary Semaphores
  – Binary: counter can only be 0 or 1
• Simple to implement, yet powerful
  – Can be used for many synchronization problems

Source: inter.scoutnet.org
Semaphores as Locks

- Semaphores can be used to build locks
  - Pintos does just that
- Must initialize semaphore with 1 to allow one thread to enter critical section

- Easily generalized to allow at most N simultaneous threads: multiplex pattern (i.e., a resource can be accessed by at most N threads)

```c
semaphore S(1); // allows initial down

lock_acquire()
{
    // try to decrement, wait if 0
    sema_down(S);
}

lock_release()
{
    // increment (wake up waiters if any)
    sema_up(S);
}
```
Implementing Locks Directly

- NB: Same technique applies to implementing semaphores directly (as in done in Pintos)
  - Will see two applications of the same technique
- Different solutions exist to implement locks for uniprocessor and multiprocessors
- Will talk about how to implement locks for uniprocessors first – next slides all assume uniprocessor
lock_acquire(struct lock *l)
{
    while (l->state == LOCKED)
        continue;
    l->state = LOCKED;
}

lock_release(struct lock *l)
{
    l->state = UNLOCKED;
}

• Does this work?

No – does not guarantee mutual exclusion property – more than one thread may see “state” in UNLOCKED state and break out of while loop. This implementation has itself a race condition.
Implementing Locks, Take 2

lock_acquire(struct lock *l)
{
    disable_preemption();
    while (l->state == LOCKED)
        continue;
    l->state = LOCKED;
    enable_preemption();
}

lock_release(struct lock *l)
{
    l->state = UNLOCKED;
}

• Does this work?

No – does not guarantee progress property. If one thread enters the while loop, no other thread will ever be scheduled since preemption is disabled – in particular, no thread that would call lock_release will ever be scheduled.
Implementing Locks, Take 3

lock_acquire(struct lock *l)
{
    while (true) {
        disable_preemption();
        if (l->state == UNLOCKED) {
            l->state = LOCKED;
            enable_preemption();
            return;
        }
        enable_preemption();
    }
    enable_preemption();
}

lock_release(struct lock *l)
{
    l->state = UNLOCKED;
}

• Does this work?

Yes, this works – but is grossly inefficient. A thread that encounters the lock in the LOCKED state will busy wait until it is unlocked, needlessly using up CPU time.
Implementing Locks, Take 4

```c
lock_acquire(struct lock *l) {
    disable_preemption();
    while (l->state == LOCKED) {
        list_push_back(l->waiters, &current->elem);
        thread_block(current);
    }
    l->state = LOCKED;
    enable_preemption();
}

lock_release(struct lock *l) {
    disable_preemption();
    l->state = UNLOCKED;
    if (list_size(l->waiters) > 0)
        thread_unblock(
            list_entry(list_pop_front(l->waiters),
                        struct thread, elem));
    enable_preemption();
}
```

Correct & uses proper blocking.
Note that thread doing the unlock performs the work of unblocking the first waiting thread.
Multiprocessor Locks

• Can’t stop threads running on other processors
  – too expensive (interprocessor irq)
  – also would create conflict with protection (locking = unprivileged op, stopping = privileged op), involving the kernel in *every* acquire/release

• Instead: use atomic instructions provided by hardware
  – E.g.: test-and-set, atomic-swap, compare-and-exchange, fetch-and-add
  – All variations of “read-and-modify” theme

• Locks are built on top of these
// In C, an atomic swap instruction would look like this
void atomic_swap(int *memory1, int *memory2)
{
    // disable interrupts in CPU;
    // lock cache line(s) for other processors
    int tmp = *memory1;
    *memory1 = *memory2;
    *memory2 = tmp;
    // unlock cache line(s); reenable interrupts
}

Spinlocks

```c
lock_acquire(struct lock *l)
{
    int lockstate = LOCKED;
    while (lockstate == LOCKED) {
        atomic_swap(&lockstate, &l->state);
    }
}
```

```c
lock_release(struct lock *l)
{
    l->state = UNLOCKED;
}
```

- Thread spins until it acquires lock
  - Q1: when should it block instead?
  - Q2: what if spin lock holder is preempted?
Spinning vs Blocking

• Some lock implementations combine spinning and blocking locks
• Blocking has a cost
  – Shouldn’t block if lock becomes available in less time than it takes to block
• Strategy: spin for time it would take to block
  – Even in worst case, total cost for lock_acquire is less than 2*block time
Spinlocks vs Disabling Preemption

• What if spinlocks were used on single CPU? Consider:
  – thread 1 takes spinlock
  – thread 1 is preempted
  – thread 2 with higher priority runs
  – thread 2 tries to take spinlock, finds it taken
  – thread 2 spins forever → deadlock!

• Thus in practice, usually combine spinlocks with disabling preemption
  – E.g., spin_lock_irqsave() in Linux
    • UP kernel: reduces to disable_preemption()
    • SMP kernel: disable_preemption() + spinlock

• Spinlocks are used when holding resources for small periods of time (same rule as for when it’s ok to disable irqs)
Critical Section Efficiency

lock acquisition ($T_a$)

critical section ($T_c$)

lock release ($T_r$)

Efficiency = \frac{T_c}{T_c + T_a + T_r}

- As processors get faster, CSE decreases because atomic instructions become relatively more expensive.

Source: McKenney, 2005
Spinlocks (Faster)

lock_acquire(struct lock *l)
{
    int lockstate = LOCKED;
    while (lockstate == LOCKED) {
        while (l->state == LOCKED)
            continue;
        atomic_swap(&lockstate, &l->state);
    }
}

lock_release(struct lock *l)
{
    l->state = UNLOCKED;
}

• Only try “expensive” atomic_swap instruction if you’ve seen lock in unlocked state
Locks: Ownership & Recursion

• Locks typically (not always) have notion of ownership
  – Only lock holder is allowed to unlock
  – See Pintos lock_held_by_current_thread()

• What if lock holder tries to acquire locks it already holds?
  – Nonrecursive locks: deadlock!
  – Recursive locks:
    • inc counter
    • dec counter on lock_release
    • release when zero
Concurrency & Synchronization

Semaphores
Infinite Buffer Problem

```
producer(item)
{
    lock_acquire(buffer);
    buffer[head++] = item;
    lock_release(buffer);
}

consumer()
{
    lock_acquire(buffer);
    while (buffer is empty) {
        lock_release(buffer);
        thread_yield();
        lock_acquire(buffer);
    }
    item = buffer[tail++];
    lock_release(buffer);
    return item
}
```

- Trying to implement infinite buffer problem with locks alone leads to a very inefficient solution (busy waiting!)
- Locks cannot express precedence constraint: A must happen before B.
Infinite Buffer Problem, Take 2

```c
producer(item)
{
  lock_acquire(buffer);
  buffer[head++] = item;
  if (#consumers > 0)
    for c in consumers {
      thread_unblock(c);
    }
  lock_release(buffer);
}
```

```c
consumer()
{
  lock_acquire(buffer);
  while (buffer is empty) {
    lock_release(buffer);
    consumers.add(current);
    thread_block(current);
    lock_acquire(buffer);
  }
  item = buffer[tail++];
  lock_release(buffer);
  return item
}
```

• Q: Why does this not work?
Infinite Buffer Problem, Take 2

Problem 1: Context switch here would cause *Lost Wakeup* problem: producer will put item in buffer, but won’t unblock consumer thread (since consumer thread isn’t in consumers yet)

Problem 2: consumers is accessed without lock
producer(item)
{
    lock_acquire(buffer);
    buffer[head++] = item;
    if (#consumers > 0)
        for c in consumers {
            thread_unblock(c);
        }
    lock_release(buffer);
}

consumer()
{
    lock_acquire(buffer);
    while (buffer is empty) {
        consumers.add(current);
        lock_release(buffer);
        thread_block(current);
        lock_acquire(buffer);
    }
    item = buffer[tail++];
    lock_release(buffer);
    return item
}
Infinite Buffer Problem, Take 3

**producer(item)**

```c
lock_acquire(buffer);
buffer[head++] = item;
if (#consumers > 0)
    for c in consumers {
        thread_unblock(c);
    }
lock_release(buffer);
```

**consumer()**

```c
lock_acquire(buffer);
while (buffer is empty) {
    consumers.add(current);
    lock_release(buffer);
    thread_block(current);
    lock_acquire(buffer);
}
item = buffer[tail++];
lock_release(buffer);
return item
```

- Idea: move consumers.add before lock_release

Context switch here would allow producer to see a consumer in the queue that is not yet blocked – thread_unblock() will panic. (Or, if thread_unblock() were written to ignore attempts at unblocking threads that aren’t blocked, the wakeup would still be lost.)
Infinite Buffer Problem, Take 4

producer(item)
{
    lock_acquire(buffer);
    buffer[head++] = item;
    if (#consumers > 0)
        for c in consumers {
            thread_unblock(c);
        }
    lock_release(buffer);
}

consumer()
{
    lock_acquire(buffer);
    while (buffer is empty) {
        consumers.add(current);
        disable_preemption();
        lock_release(buffer);
        thread_block(current);
        enable_preemption();
        lock_acquire(buffer);
    }
    item = buffer[tail++];
    lock_release(buffer);
    return item
}
Infinite Buffer Problem, Take 4

```
producer(item)
{
    lock_acquire(buffer);
    buffer[head++] = item;
    if (#consumers > 0)
        for c in consumers {
            thread_unblock(c);
        }
    lock_release(buffer);
}
```

```
c consumer()
{
    lock_acquire(buffer);
    while (buffer is empty) {
        consumers.add(current);
        disable_preemption();
        lock_release(buffer);
        thread_block(current);
        enable_preemption();
        lock_acquire(buffer);
    }
    item = buffer[tail++];
    lock_release(buffer);
    return item
}
```

• Must ensure that releasing the lock and blocking happens atomically.

Final problem: producer always wakes up all consumers, even though at most one will be able to consume the item produced. This is known as a thundering herd problem.
Infinite Buffer Problem, Take 5

```c
producer(item)
{
    lock_acquire(buffer);
    buffer[head++] = item;
    if (#consumers > 0)
        thread_unblock(
            consumers.pop()
        );
    lock_release(buffer);
}
```

```c
consumer()
{
    lock_acquire(buffer);
    while (buffer is empty) {
        consumers.add(current);
        disable_preemption();
        lock_release(buffer);
        thread_block(current);
        enable_preemption();
        lock_acquire(buffer);
    }
    item = buffer[tail++];
    lock_release(buffer);
    return item
}
```

- This is correct, but complicated and very easy to get wrong
  - Want abstraction that does not require direct block/unblock call
Low-level vs. High-level Synchronization

- Low-level synchronization primitives:
  - Disabling preemption, (Blocking) Locks, Spinlocks
  - Implement mutual exclusion

- Implementing precedence constraints directly via thread_unblock/thread_block is problematic because
  - It’s complicated (see last slides)
  - It may violate encapsulation from a software engineering perspective
  - You may not have that access at all (unprivileged code!)

- We need well-understood higher-level constructs that have support for waiting/signaling “built-in”
  - Semaphores
  - Monitors
Semaphores

- Invented by Edsger Dijkstra in 1965s
- Counter S, initialized to some value, with two operations:
  - \( P(S) \) or “down” or “wait” – if counter greater than zero, decrement. Else wait until greater than zero, then decrement
  - \( V(S) \) or “up” or “signal” – increment counter, wake up any threads stuck in \( P \).
- Semaphores don’t go negative:
  - \( \#V + \text{InitialValue} - \#P \geq 0 \)
- Note: direct access to counter value after initialization is not allowed
- Counting vs Binary Semaphores
  - Binary: counter can only be 0 or 1
- Simple to implement, yet powerful
  - Can be used for many synchronization problems
Infinite Buffer w/ Semaphores (1)

```c
semaphore items_avail(0);

producer()
{
    lock_acquire(buffer);
    buffer[head++] = item;
    lock_release(buffer);
    sema_up(items_avail);
}

c consumer()
{
    sema_down(items_avail);
    lock_acquire(buffer);
    item = buffer[tail++];
    lock_release(buffer);
    return item;
}
```

- Semaphore “remembers” items put into queue (no updates are lost)
Infinite Buffer w/ Semaphores (2)

```
semaphore items_avail(0);
semaphore buffer_access(1);

producer()
{
    sema_down(buffer_access);
    buffer[head++] = item;
    sema_up(buffer_access);
    sema_up(items_avail);
}

consumer()
{
    sema_down(items_avail);
    sema_down(buffer_access);
    item = buffer[tail++];
    sema_up(buffer_access);
    return item;
}
```

- Can use semaphore instead of lock to protect buffer access
Bounded Buffer w/ Semaphores

```c
semaphore items_avail(0);
semaphore buffer_access(1);
semaphore slots_avail(CAPACITY);

producer()
{
    sema_down(slots_avail);
    sema_down(buffer_access);
    buffer[head++] = item;
    sema_up(buffer_access);
    sema_up(items_avail);
}

consumer()
{
    sema_down(items_avail);
    sema_down(buffer_access);
    item = buffer[tail++];
    sema_up(buffer_access);
    sema_up(slots_avail);
    return item;
}
```

- Semaphores allow for scheduling of resources
Rendezvous

- A needs to be sure B has advanced to point L, B needs to be sure A has advanced to L

```c
semaphore A_madeit(0);

A_rendezvous_with_B()
{
    sema_up(A_madeit);
    sema_down(B_madeit);
}
```

```c
semaphore B_madeit(0);

B_rendezvous_with_A()
{
    sema_up(B_madeit);
    sema_down(A_madeit);
}
```
Waiting for an activity to finish

- Works no matter which thread is scheduled first after `thread_create` (parent or child)
- Elegant solution that avoids the need to share a “have done task” flag between parent & child
- Two applications of this technique in Pintos Project 2
  - signal successful process startup (“exec”) to parent
  - signal process completion (“exit”) to parent

```c
semaphore done_with_task(0);
thread_create(
  do_task,
  (void*)&done_with_task);
sema_down(done_with_task);
// safely access task’s results

void
do_task(void *arg)
{
  semaphore *s = arg;
  /* do the task */
  sema_up(*s);
}
```
Dining Philosophers (Dijkstra)

- A classic
- 5 Philosophers, 1 bowl of spaghetti
- Philosophers (threads) think & eat ad infinitum
  - Need left & right fork to eat (!?)
- Want solution that prevents starvation & does not delay hungry philosophers unnecessarily
Dining Philosophers (1)

```c
semaphore fork[0..4](1);
philosopher(int i) // i is 0..4
{
    while (true) {
        /* think … finally */
        sema_down(fork[i]); // get left fork
        sema_down(fork[(i+1)%5]); // get right fork
        /* eat */
        sema_up(fork[i]); // put down left fork
        sema_up(fork[(i+1)%5]); // put down right fork
    }
}
```

- What is the problem with this solution?
- Deadlock if all pick up left fork
Dining Philosophers (2)

semaphore fork[0..4](1);
semaphore at_table(4); // allow at most 4 to fight for forks
philosopher(int i) // i is 0..4
{
    while (true) {
        /* think … finally */
        sema_down(at_table); // sit down at table
        sema_down(fork[i]); // get left fork
        sema_down(fork[(i+1)%5]); // get right fork
        /* eat … finally */
        sema_up(fork[i]); // put down left fork
        sema_up(fork[(i+1)%5]); // put down right fork
        sema_up(at_table); // get up
    }
}
Concurrency & Synchronization

continued
Recap: Disabling IRQs

• Disabling IRQs
  – (1) When used as a strategy for achieving mutual exclusion between threads on a uniprocessor
  – Or (2) when used to protect against concurrent access by IRQ handler
    • must not block (e.g., call thread_block())
    • must not loop for long/infinitely
    • (1) typically implemented not actually as cli, but by postponing interrupts that could lead to context switches
  – Insufficient for multiprocessor
  – Traditionally used to avoid losing wakeups on uniprocessor
    • allow interrupts only after thread is prepared to be woken up
Recap: Implementing Locks/Semaphores

• Both locks and semaphores can be implemented directly on uniprocessor
  – Requires disable_preemption
  – Involves state change of thread if contended

• On multiprocessor, build implementations from atomic instructions such as compare-and-swap
  – Must guard against both accesses by other CPUs and accesses by threads on own CPU

• Spinning vs. Blocking

• Locks are simpler than semaphores
  – Can be implemented when semaphores are present
Implementing Locks: Practical Issues

• How expensive are locks?
• Two considerations:
  – Cost to acquire uncontended lock
    • UP Kernel: disable/enable irq + memory access
    • In other scenarios: needs atomic instruction (relatively expensive in terms of processor cycles, especially if executed often)
  – Cost to acquire contended lock
    • Spinlock: blocks current CPU entirely (if no blocking is employed)
    • Regular lock: cost at least two context switches, plus associated management overhead

• Conclusions
  – Optimizing uncontended case is important
  – “Hot locks” can sack performance easily
Using Locks

- Associate each shared variable with lock L
  - “lock L protects that variable”

```c
static struct list usedlist; /* List of used blocks */
static struct list freelist;  /* List of free blocks */
static struct lock listlock; /* Protects usedlist & freelist */
```

```c
void *mem_alloc(...) {
    block *b;
    lock_acquire(&listlock);
    b = alloc_block_from_freelist();
    insert_into_usedlist(&usedlist, b);
    lock_release(&listlock);
    return b->data;
}
```

```c
void mem_free(block *b) {
    lock_acquire(&listlock);
    list_remove(&b->elem);
    coalesce_into_freelist(&freelist, b);
    lock_release(&listlock);
}
```
How many locks should I use?

• Could use one lock for all shared variables
  – Disadvantage: if a thread holding the lock blocks, no other thread can access any shared variable, even unrelated ones
  – Sometimes used when retrofitting non-threaded code into threaded framework
  – Examples:
    • “BKL” Big Kernel Lock in Linux
    • fslock in Pintos Project 2

• Ideally, want fine-grained locking
  – One lock only protects one (or a small set of) variables – how to pick that set?
Multiple locks, the wrong way

static struct list usedlist; /* List of used blocks */
static struct list freelist; /* List of free blocks */

static struct lock alloclock; /* Protects allocations */
static struct lock freelock; /* Protects deallocations */

Wrong: locks protect data structures, not code blocks! Allocating thread & deallocating thread could collide
Multiple locks, 2\textsuperscript{nd} try

```c
static struct list usedlist; /* List of used blocks */
static struct list freelist; /* List of free blocks */

static struct lock usedlock; /* Protects usedlist */
static struct lock freelock; /* Protects freelist */

void *mem_alloc(...)
{
    block *b;
    lock_acquire(&freelock);
    b = alloc_block_from_freelist();
    lock_acquire(&usedlock);
    insert_into_usedlist(&usedlist, b);
    lock_release(&freelock);
    lock_release(&usedlock);
    return b->data;
}

void mem_free(block *b)
{
    lock_acquire(&usedlock);
    list_remove(&b->elem);
    lock_acquire(&freelock);
    coalesce_into_freelist(&freelist, b);
    lock_release(&usedlock);
    lock_release(&freelock);
}
```

Also wrong: deadlock!
Always acquire multiple locks in same order -
Or don’t hold them simultaneously
Multiple locks, correct (1)

static struct list usedlist; /* List of used blocks */
static struct list freelist; /* List of free blocks */

static struct lock usedlock; /* Protects usedlist */
static struct lock freelock; /* Protects freelist */

void *mem_alloc(...)
{
    block *b;
    lock_acquire(&usedlock);
    lock_acquire(&freelock);
    b = alloc_block_from_freelist();
    insert_into_usedlist(&usedlist, b);
    lock_release(&freelock);
    lock_release(&usedlock);
    return b->data;
}

void mem_free(block *b)
{
    lock_acquire(&usedlock);
    lock_acquire(&freelock);
    list_remove(&b->elem);
    coalesce_into_freelist(&freelist, b);
    lock_release(&freelock);
    lock_release(&usedlock);
}

Correct, but inefficient!
Locks are always held simultaneously, one lock would suffice
Multiple locks, correct (2)

Correct, but not necessarily better!

**On uniprocessor:**
No throughput from fine-grained locking, since no blocking inside critical sections – but pay twice the price compared to one-lock solution

**On multiprocessor:**
Gain from being able to manipulate free & used lists in parallel, but
increased risk of contended locks
critical section efficiency may be low (particularly for O(1) operations)

```c
void *mem_alloc(…)
{
    block *b;
    lock_acquire(&freelock);
    b = alloc_block_from_freelist();
    lock_release(&freelock);
    lock_acquire(&usedlock);
    insert_into_usedlist(&usedlist, b);
    lock_release(&usedlock);
    return b->data;
}

void mem_free(block *b)
{
    lock_acquire(&usedlock);
    list_remove(&b->elem);
    lock_release(&usedlock);
    lock_acquire(&freelock);
    coalesce_into_freelist(&freelist, b);
    lock_release(&freelock);
}
```
Conclusion

• Choosing which lock should protect which shared variable(s) is not easy – must weigh:
  – Whether all variables are always accessed together (use one lock if so)
  – Whether code inside critical section may block (if not, no throughput gain from fine-grained locking on uniprocessor)
  – Whether there is a consistency requirement if multiple variables are accessed in related sequence (must hold single lock if so)
    • See “Subtle race condition in Java” later this lecture
  – Cost of multiple calls to lock/unlock (increasing parallelism advantages may be offset by those costs)
Rules for Easy Locking

• Every shared variable must be protected by a lock
  – Establish this relationship with code comments
    • /* protected by … <lock>*/
  – Acquire lock before touching (reading or writing) variable
  – Release when done, on all paths
  – One lock may protect more than one variable, but not too many
    • If in doubt, use fewer locks (may lead to worse efficiency, but less likely to lead to race conditions or deadlock)

• If manipulating multiple variables, acquire locks assigned to protecting each
  – Acquire locks always in same order (doesn’t matter which order, but must be same)
  – Release in opposite order
  – Don’t release any locks before all have been acquired (two-phase locking)
Locks in Java/C#

- Every object can function as lock – no need to declare & initialize them!
- `synchronized` (locked in C#) brackets code in lock/unlock pairs – either entire method or block `{}`
- `finally` clause ensures `unlock()` is always called

```
synchronized void method() {
    code;
    synchronized (obj) {
        more code;
    }
    even more code;
}
```

```
void method() {
    try {
        lock(this);
        code;
        try {
            lock(obj);
            more code;
        } finally { unlock(obj); }
        even more code;
    } finally { unlock(this); }
}
```
Subtle Race Condition

- Race condition even though individual accesses to “sb” are synchronized (protected by a lock)
  - But “len” may no longer be equal to “sb.length” in call to getChars()
- This means simply slapping lock()/unlock() around every access to a shared variable does not thread-safe code make
- Found by Flanagan/Freund

```java
public synchronized StringBuffer append(StringBuffer sb) {
    int len = sb.length(); // note: StringBuffer.length() is synchronized
    int newcount = count + len;
    if (newcount > value.length) {
        expandCapacity(newcount);
        sb.getChars(0, len, value, count); // StringBuffer.getChars() is synchronized
    }
    count = newcount;
    return this;
}
```

Not holding lock on ‘sb’ – other Thread may change its length
Generalization: Atomicity Constraints

- Previous example shows that locking, by itself, may not provide desired atomicity.
- Information read in critical section A must not be used in a critical section B.

```plaintext
lock();
var x = read_var();
unlock();
...
lock();
use(x);
unlock();
```

atomicity required to maintain consistency
Concurrency & Synchronization

Monitors
Monitors

• A monitor combines a set of shared variables & operations to access them
  – Think of an enhanced C++ class with no public fields
• A monitor provides implicit synchronization (only one thread can access private variables simultaneously)
  – Single lock is used to ensure all code associated with monitor is within critical section
• A monitor provides a general signaling facility
  – Wait/Signal pattern (similar to, but different from semaphores)
  – May declare & maintain multiple signaling queues
Monitors (cont’d)

• Classic monitors are embedded in programming languages
  – Invented by Hoare & Brinch-Hansen 1972/73
  – First used in Mesa/Cedar System @ Xerox PARC 1978
  – Adapted version available in Java/C#

• (Classic) Monitors are safer than semaphores
  – can’t forget to lock data – compiler checks this

• In contemporary C, monitors are a *synchronization pattern* that is achieved using locks & condition variables
  – Must understand monitor abstraction to use it
Infinite Buffer w/ Monitor

monitor buffer {
    /* implied: struct lock mlock;*/
private:
    char buffer[];
    int head, tail;
public:
    produce(item);
    item consume();
}

buffer::produce(item i)
{
    buffer[head++] = i;
}
buffer::consume()
{
    return buffer[tail++];
}

• Monitors provide implicit protection for their internal variables
  – Still need to add the signaling part
Condition Variables

- Variables used by a monitor for signaling a condition
  - a general (programmer-defined) condition, not just integer increment as with semaphores
  - The actual condition is typically some boolean predicate of monitor variables, e.g. “buffer.size > 0”

- Monitor can have more than one condition variable

- Three operations:
  - Wait(): leave monitor, wait for condition to be signaled, reenter monitor
  - Signal(): signal one thread waiting on condition
  - Broadcast(): signal all threads waiting on condition
Bounded Buffer w/ Monitor

```cpp
monitor buffer {
    condition items_avail;
    condition slots_avail;
private:
    char buffer[];
    int head, tail;
public:
    produce(item);
    item consume();
}

buffer::produce(item i)
{
    while ((tail+1–head)%CAPACITY==0)
        slots_avail.wait();
    buffer[head++] = i;
    items_avail.signal();
}

buffer::consume()
{
    while (head == tail)
        items_avail.wait();
    item i = buffer[tail++];
    slots_avail.signal();
    return i;
}
```
monitor buffer {
    condition items_avail;
    condition slots_avail;
private:
    char buffer[];
    int head, tail;
public:
    produce(item);
    item consume();
}

buffer::produce(item i) 
{
    while ((tail+1–head)%CAPACITY==0)
        slots_avail.wait();
    buffer[head++] = i;
    items_avail.signal();
}
buffer::consume()
{
    while (head == tail)
        items_avail.wait();
    item i = buffer[tail++];
    slots_avail.signal();
    return i;
}

Q1.: How is lost update problem avoided?
Q2.: Why while() and not if()?
Recall: Infinite Buffer Problem, Take 5

producer(item)
{
    lock_acquire(buffer);
    buffer[head++] = item;
    if (#consumers > 0)
        thread_unblock(
            consumers.pop()
        );
    lock_release(buffer);
}

consumer()
{
    lock_acquire(buffer);
    while (buffer is empty) {
        consumers.add(current);
        disable_preemption();
        lock_release(buffer);
        thread_block(current);
        enable_preemption();
        lock_acquire(buffer);
    }
    item = buffer[tail++];
    lock_release(buffer);
    return item
}

• Aside: this solution to the infinite buffer problem essentially reinvented monitors!
Implementing Condition Variables

- A condition variable’s state is just a queue of waiters:
  - Wait(): adds current thread to (end of queue) & block
  - Signal(): pick one thread from queue & unblock it
  - Broadcast(): unblock all threads
Mesa vs Hoare Style

- Mesa-style monitors signaler keeps lock
  - `cond_signal` keeps lock, so it leaves signaling thread in monitor
  - waiter is made READY, but can’t enter until signaler gives up lock
  - There is no guarantee whether signaled thread will enter monitor next (or some other thread) - so must always use “while()” when checking condition – cannot assume that condition set by signaling thread will still hold when monitor is reentered
  - POSIX Threads & Pintos are Mesa-style (and so are C# & Java)
- Alternative is Hoare-style (after C.A.R. Hoare)
  - `cond_signal` leads to signaling thread’s exit from monitor and immediate reentry of waiter (e.g., monitor lock is passed from signaler to signalee)
  - not commonly used
Condition Variables vs. Semaphores

• Condition Variables
  – Signals are lost if nobody’s on the queue (e.g., nothing happens)
  – Wait() always blocks

• Semaphores
  – Signals (calls to V() or sema_up()) are remembered even if nobody’s current waiting
  – Wait (e.g., P() or sema_down()) may or may not block
Monitors in C

- POSIX Threads & Pintos
- No compiler support, must do it manually
  - must declare locks & condition vars
  - must call lock_acquire/lock_release when entering&leaving the monitor
  - must use cond_wait/cond_signal to wait for/signal condition
- Note: cond_wait(&c, &m) takes monitor lock as parameter
  - necessary so monitor can be left & reentered without losing signals
- Pintos cond_signal() takes lock as well
  - only as debugging help/assertion to check lock is held when signaling
  - pthread_cond_signal() does not
Locks in Java/C#

- Every object can function as lock – no need to declare & initialize them!
- `synchronized (locked in C#)` brackets code in lock/unlock pairs – either entire method or block `{ }`
- finally clause ensures `unlock()` is always called

```java
synchronized void method() {
    code;
    synchronized (obj) {
        more code;
    }
    even more code;
}
```

is transformed to

```java
void method() {
    try {
        lock(this);
        code;
        try {
            lock(obj);
            more code;
        } finally { unlock(obj); }
        even more code;
    } finally { unlock(this); }
}
```
Monitors in Java

• synchronized block means
  – enter monitor
  – execute block
  – leave monitor

• wait()/notify() use condition variable associated with receiver
  – Every object in Java can function as a condition var

```java
class buffer {
    private char buffer[];
    private int head, tail;
    public synchronized produce(item i) {
        while (buffer_full())
            this.wait();
        buffer[head++] = i;
        this.notify();
    }
    public synchronized item consume() {
        while (buffer_empty())
            this.wait();
        buffer[tail++] = i;
        this.notify();
    }
}
```
Per Brinch Hansen’s Criticism

- See *Java’s Insecure Parallelism* [Brinch Hansen 1999]
- Says Java abused concept of monitors because Java does not *require* all accesses to shared variables to be within monitors
- Why did designers of Java not follow his lead?
  - Performance: compiler can’t easily decide if object is local or not - conservatively, would have to make all public methods synchronized – pay at least cost of atomic instruction on entering every time
Readers/Writer w/ Monitor

```c
struct lock mlock; // protects rdrs & wrtrs
int readers = 0, writers = 0;
struct condvar canread, canwrite;
void read_lock_acquire() {
    lock_acquire(&mlock);
    while (writers > 0)
        cond_wait(&canread, &mlock);
    readers++;
    lock_release(&mlock);
}
void read_lock_release() {
    lock_acquire(&mlock);
    if (--readers == 0)
        cond_signal(&canwrite, &mlock);
    lock_release(&mlock);
}
void write_lock_acquire() {
    lock_acquire(&mlock);
    while (readers > 0 || writers > 0)
        cond_wait(&canwrite, &mlock);
    writers++;
    lock_release(&mlock);
}
void write_lock_release() {
    lock_acquire(&mlock);
    writers--;
    ASSERT(writers == 0);
    cond_broadcast(&canread, &mlock);
    cond_signal(&canwrite, &mlock);
    lock_release(&mlock);
}

Q.: does this implementation prevent starvation?
```
Summary

• Semaphores & Monitors are both higher-level constructs
• Monitors can be included in a language (Mesa, Java)
  – in C, however, they are just a programming pattern that involves a structured way of using mutex+condition variables
• When should you use which?
Semaphores vs. Monitors

- Semaphores & Monitors are both higher-level constructs
- Use semaphores where updates must be remembered – where # of signals must match # of waits
- Otherwise, use monitors.
- Prefer semaphore if they are applicable
High vs Low Level Synchronization

• The bounded buffer problem (and many others) can be solved with higher-level synchronization primitives
  – semaphores and monitors
• In Pintos kernel, one could also use thread_block/unblock directly
  – this is not always efficiently possible in other concurrent environments
• Q.: when should you use low-level synchronization (a la thread_block/thread_unblock) and when should you prefer higher-level synchronization?
• A.: Except for the simplest scenarios, higher-level synchronization abstractions are always preferable
  – They’re well understood; make it possible to reason about code.
Nonblocking Synchronization
Nonblocking Synchronization

- Alternative to locks: instead of serializing access, detect when bad interleaving occurred, retry if so

```c
void increment_counter(int *counter) {
    do {
        int oldvalue = *counter;
        int newvalue = oldvalue + 1;
        [ BEGIN ATOMIC COMPARE-AND-AND-SWAP INSTRUCTION ]
        if (*counter == oldvalue) { *counter = newvalue; success = true; }
        else { success = false; }
        [ END CAS ]
    } while (!success);
}
```
Nonblocking Synchronization (2)

• Also referred to a “optimistic concurrency control”
• x86 supports this model via cmpxchg instruction
• Advantages:
  – Less overhead for uncontended locks (can be faster, and need no storage for lock queue)
  – Synchronizes with IRQ handler automatically
  – Can be easier to clean up when killing a thread
  – No priority inversion or deadlock
• Disadvantages
  – Can require lots of retries, leading to wasted CPU cycles
  – Requires careful memory/ownership management – must ensure that memory is not reclaimed while a thread may hold reference to it (this can lead to blocking, indirectly, when exhausting memory – real implementations need to worry about this!)
Aside: Nonblocking Properties

• Different NBS algorithms can be analyzed with respect to their progress guarantees
• Lock-freedom:
  – One thread will eventually make progress
• Wait-freedom guarantee: (strongest)
  – All threads will eventually make progress
• Obstruction-freedom: (weakest)
  – Thread will make progress if it is unobstructed
Recent Developments (1)

• As multi- and manycore architectures become abundant, need for better programming models becomes stronger
  – See “The Landscape of Parallel Computing Research: A View From Berkeley”

• Distinguish programming models along 5 categories (each explicit or implicit):
  – Task identification
  – Task mapping
  – Data distribution
  – Communication Mapping
  – Synchronization
Transactional Memory

- Software (STM) or hardware-based
- Idea:
  - Break computations into pieces called transactions
    - Transaction must have the same “atomicity” semantics as locks
    - NB: not as in a database transaction (no persistence on stable storage!)
  - Don’t use locks to prevent bad interleavings, and occur cost of serialization, rather: focus on the results of the computation: “Could they have been obtained in some serial order (i.e., if locks had been used?)”
    - if so, allow them. Otherwise, undo computations
- Many approaches possible – goal is to relieve the programmer from having to use locks explicitly (and avoid their pitfalls such as forgetting them, potential for deadlock, and potential for low CPU utilization)
  - Challenge is to implement this efficiently and in a manner that integrates with existing languages
  - See [Larus 2007] for a survey of approaches
Closing Thoughts on Concurrency and Synchronization

• Have covered most frequently used concepts and models today:
  – Locks, semaphores, monitors/condition variables
• Have looked at them from both users’ and implementers’ perspective
  – And considered both correctness and performance perspective
• Will use these concepts in projects 2, 3, and 4 (Pintos is a fully preemptive kernel!)
• Yet overall, have barely scratched the surface
  – Many exciting developments may lie ahead in coming years