



CS 3204 Operating Systems

Lecture 8 Godmar Back



Announcements


- Project 1 due Monday Sep 25, 11:59pm
 - Only 11 days left
- Announcements:
 - CS Open House this afternoon 1pm-4pm
 - Free Food
 - CS Career Reception
- Reading assignments:
 - Read carefully Chapter 6.1-6.4



CS 3204 Spring 2006 9/18/2006 2


Project 1 Suggested Timeline

- Today Sep 14:
 - Should pass all alarm & basic priority tests
- Priority Inheritance & Advanced Scheduler take the most time, start them in parallel – will take the most time to implement & debug
 - Read forum for fixed-point implementation
 - Read forum for MLFQS ambiguities
- Due date Sep 25




CS 3204 Spring 2006 9/18/2006 3


Concurrency & Synchronization



Semaphores

 Source: inter.scoutnet.org

- 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 + 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




CS 3204 Spring 2006 9/18/2006 5

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

```
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);
}
```

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



CS 3204 Spring 2006 9/18/2006 6

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.
- Note: different from "all-or-nothing" meaning atomic has in database theory & practice
 - Does not necessarily imply that thread executes section without interruption, or even that thread completes section – just that other threads can't enter it while one thread is inside it
- Solutions can be entirely software, or entirely hardware
 - Usually combined
 - Different solutions for uniprocessor vs multiprocessor scenarios

Disabling Interrupts

- All asynchronous context switches start with interrupts

- So disable interrupts to avoid them!

```
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

```
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

```
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](#)

```
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

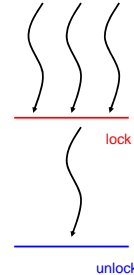
- (this applies to all variations)
- Sledgehammer solution
- Infinite loop means machine locks up
- Use this to protect data structures from concurrent access by interrupt handlers
 - Keep sections of code where irqs are disabled minimal (nothing else can happen until irqs are reenabled – latency penalty!)
 - 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

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

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



Implementing Locks

- Let's discuss how to implement locks to solve CS problem
- Later talk about semaphores
 - Same ideas apply – get to see two views of the same issues
- 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

Implementing Locks, Take 1

```
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();
    }
}
```

```
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

```
lock_acquire(struct lock *)
{
    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.



CS 3204 Spring 2006

9/18/2006

19

Using Locks

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

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

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

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



CS 3204 Spring 2006

9/18/2006

20

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?



CS 3204 Spring 2006

9/18/2006

21

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 freeblock; /* Protects deallocations */
```

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

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

Wrong: locks protect data structures, not code blocks! Allocating thread & deallocating thread could collide



CS 3204 Spring 2006

9/18/2006

22

Multiple locks, 2nd try

```
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 freeblock; /* Protects freelist */
```

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

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

Also wrong: deadlock!
Always acquire multiple locks in same order - Or don't hold them simultaneously



CS 3204 Spring 2006

9/18/2006

23

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 freeblock; /* Protects freelist */
```

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

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

Correct, but inefficient!
Locks are always held simultaneously, one lock would suffice



CS 3204 Spring 2006

9/18/2006

24

Multiple locks, correct (2)

```
static struct t
static struct t
static struct t
static struct t

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;
}

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

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



CS 3204 Spring 2006

9/18/2006

25

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 can 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)
 - Cost of multiple calls to lock/unlock (increasing parallelism advantages may be offset by those costs)



CS 3204 Spring 2006

9/18/2006

26

Rules for easy locking

- Every shared variable must be protected by a 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 manipulating multiple variables, acquire locks protecting each
 - Acquire locks always in same order (doesn't matter which order, but must be same)
 - Release in opposite order
 - Don't mix acquires & release (two-phase locking)



CS 3204 Spring 2006

9/18/2006

27