- Note -

Synchronization 1

Examples on following slides assume a slightly different version of Project 0 (used last semesters) where used blocks were also kept on a list, the "used list."

- mem alloc would add a block to used list
- mem free would remove block from used list

In that case, the code needed to protect both the free and used list

The following slides discuss correct and incorrect way of doing so

Computer Science Dept Va Tech August 200

Computer Science Dept Va Tech August 2007

Operating Systems

©2003-07 Back

Using Locks Synchronization 2 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(...) void mem_free(block *b) block *b; lock_acquire(&listlock); lock_acquire(&listlock); list_remove(&b->elem); b = alloc_block_from_freelist(); coalesce_into_freelist(&freelist, b); insert_into_usedlist(&usedlist, b); lock_release(&listlock); } lock_release(&listlock); return b->data; }

How many locks should I use?

Synchronization 3

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?

Computer Science Dept Va Tech August 200

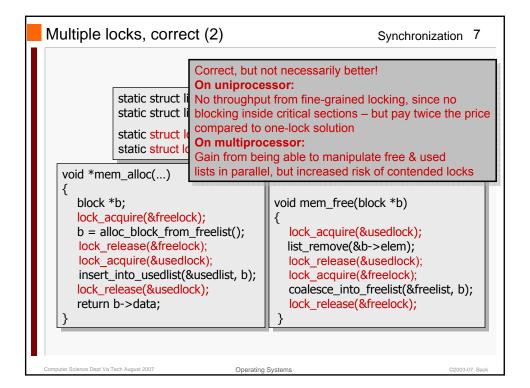
Operating Systems

©2003-07 Back

Multiple locks, the wrong way Synchronization 4 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 */ void *mem_alloc(...) void mem_free(block *b) { { block *b; lock_acquire(&freelock); lock_acquire(&alloclock); list_remove(&b->elem); b = alloc_block_from_freelist(); coalesce_into_freelist(&freelist, b); insert_into_usedlist(&usedlist, b); lock_release(&freelock); } lock_release(&alloclock); return b->data; } Wrong: locks protect data structures, not code blocks! Allocating thread & deallocating thread could collide Computer Science Dept Va Tech August 2007 Operating Systems

```
Multiple locks, 2<sup>nd</sup> try
                                                                Synchronization 5
              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(...)
                                            void mem_free(block *b)
      block *b:
                                               lock_acquire(&usedlock);
      lock acquire(&freelock);
                                               list remove(&b->elem);
      b = alloc block from freelist();
                                               lock acquire(&freelock);
      lock acquire(&usedlock);
                                               coalesce_into_freelist(&freelist, b);
      insert_into_usedlist(&usedlist, b);
                                               lock release(&usedlock);
      lock release(&freelock);
                                               lock_release(&freelock);
      lock_release(&usedlock);
                                            }
      return b->data;
                                  Also wrong: deadlock!
                                  Always acquire multiple locks in same order -
                                  Or don't hold them simultaneously
```

```
Multiple locks, correct (1)
                                                                  Synchronization 6
              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(...)
                                             void mem_free(block *b)
      block *b;
                                                lock_acquire(&usedlock);
      lock_acquire(&usedlock);
                                                lock acquire(&freelock);
      lock acquire(&freelock);
                                                list remove(&b->elem);
      b = alloc block from freelist();
                                                coalesce into freelist(&freelist, b);
      insert into usedlist(&usedlist, b);
                                                lock release(&freelock);
      lock release(&freelock);
                                                lock release(&usedlock);
      lock_release(&usedlock);
                                              }
      return b->data;
                                   Correct, but inefficient!
                                   Locks are always held simultaneously,
                                   one lock would suffice
Computer Science Dept Va Tech August 2007
                                     Operating Systems
```



Conclusion

Synchronization 8

3

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)
 - See "Subtle race condition in Java" below
- Cost of multiple calls to lock/unlock (increasing parallelism advantages may be offset by those costs)

Computer Science Dept Va Tech August 2007

Operating Systems

©2003-07 Bac

Rules for Easy Locking

Synchronization 9

Every shared variable must be protected by a lock

- One lock may protect more than one variable, but not too many
- Acquire lock before touching (reading or writing) variable
- Release when done, on all paths

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)

Computer Science Dept Va Tech August 2007

Operating Systems

©2003-07 Back

Infinite Buffer Problem

Synchronization 10

```
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!)

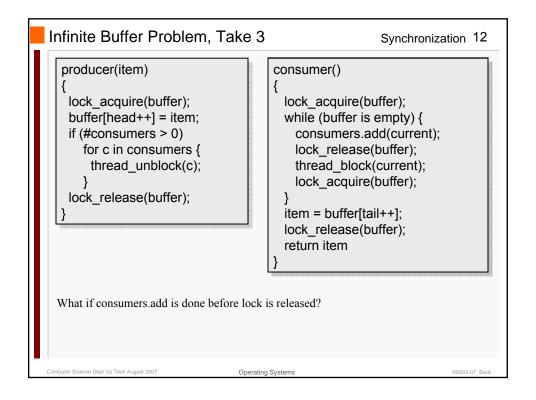
Locks cannot express precedence constraint: A must happen before B.

Computer Science Dept Va Tech August 2007

Operating Systems

©2003-07 Back

```
Infinite Buffer Problem, Take 2
                                                          Synchronization 11
  producer(item)
                                        consumer()
   lock acquire(buffer);
                                          lock acquire(buffer);
   buffer[head++] = item;
                                          while (buffer is empty) {
   if (#consumers > 0)
                                           lock release(buffer);
      for c in consumers {
                                            consumers.add(current);
       thread_unblock(c);
                                            thread block(current);
                                            lock_acquire(buffer);
   lock_release(buffer);
                                          item = buffer[tail++];
                                          lock_release(buffer);
                                          return item
  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)
```



Infinite Buffer Problem, Take 4

Synchronization 13

```
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
}
```

This is correct, but complicated and very easy to get wrong

Want abstraction that does not require direct block/unblock call

Computer Science Dept Va Tech August 2007

Operating Systems

@2003.07 Back

Low-level vs. High-level Synchronization

Synchronization 14

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

- Semaphores
- Monitors

Computer Science Dept Va Tech August 2007

Operating Systems

©2003-07 Back