# CS 3204
# Operating Systems

Lecture 9

Godmar Back

## Announcements

- Project 1 due on Sep 29, 11:59pm
  - Additional GTA office hours
    - Nick: Wed 4-6pm
    - Peter: TBA
- Midterm coming up Oct 2
- Reading:
  - Read carefully 1.5, 3.1-3.3, 6.1-6.4



## Project 1 Suggested Timeline

- By now, you have:
  - Have read relevant project documentation, set up CVS, built and run your first kernel, designed your data structures for alarm clock
- And should be finishing:
  - Alarm clock by Sep 16
- Pass all basic priority tests by Sep 18
- Priority Inheritance & Advanced Scheduler will take the most time to implement & debug, start them in parallel
  - Should have design for priority inheritance figured out by Sep 23
  - Develop & test fixed-point layer independently by Sep 23
- Due date Sep 29



## 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



## 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

## 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



## - Note -

- Examples on following slides assume a slightly different version of Project 0 (used several semesters ago) where used blocks were also kept on a list, called the "used list."
  - mem_alloc would add a block to used list
  - mem_free would remove block from used list
- In this case, the code needed to protect both the free and used lists
- The following slides discuss correct and incorrect ways of doing so



## 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);
}
```



## 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 */
```

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

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

## 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 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)

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

```
    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
critical section efficiency may be low (particularly for O(1) operations)



## 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#

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

is transformed to

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

- 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

## Subtle Race Condition

```
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

- 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



## 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