CS 3204
Operating Systems

Lecture 9

Godmar Back

Virginia Tech

Announcements

- Project 1 due Monday Sep 25, 11:59pm
  - Only 6 days left
  - Should be busy debugging priority donation & the advanced scheduler
- Project 0 scores are forthcoming
- Reading assignments:
  - Read carefully Chapter 6.1-6.4

Concurrency & Synchronization

Virginia Tech

Recap: Critical Section Problem

- Defined Critical Section Problem:
  - mutual exclusion, progress, bounded waiting
- Approaches to CS on uniprocessor:
  - Disabling IRQs – use this to protect against concurrent access by IRQ handler
  - Locks – use to protect against concurrent access by other threads
    - Involves state change of thread if contended
    - Requires a "disable_preemption" primitive (such as disable irq)
- Using Locks:
  - correctness first – see "Rules for Easy Locking", then performance
- Today:
  - More on using locks
  - Locks in Java/C#
  - Multiprocessor implementations of locks

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 should not protect too many
  - The lock protects the variable, not a region of code or a function! All accesses to variable must be protected by *same* lock no matter where they occur.
  - Encapsulation helps: e.g., use one lock for all fields in an object
- If manipulating multiple variables, acquire locks protecting each variable
  - always in same order (doesn't matter which)
  - release in opposite order
  - don't mix acquires & release (two-phase locking)

Rul

Aside: LockSet algorithm.
To determine race conditions in a program, use following idea:
For each variable *v*, set **lock set** (*v*) := ALL.
Whenever a variable is accessed by a thread, compute:

lock set (v) := **lock set** (v) ∩
{ set of locks that is held by the thread at access }

- Every shared
  - Acquire loc
  - Release wh
  - One lock m
  protect too m
  - The lock protects the variable, not a region of c unction! All accesses to variable must be protected by *same* lock no matter where they occur.
  - Encapsulation helps: e.g., use one lock for all fields in an object
- If manipulating multiple variables, acquire locks protecting each variable
  - always in same order (doesn't matter which)
  - release in opposite order
  - don't mix acquires & release (two-phase locking)

Coarse- vs fine-grained Locking

- Coarse-grained
  - "a single lock protects a set of resources"
- Unnecessary serialization: other threads must wait to access any resource even while unrelated ones are accessed
  - This means processors on which those threads could run will idle, leading to decrease in throughput
- Fine-grained
  - "each resource is protected by its own lock"
- More potential for mistakes & deadlocks
  - Requires locking order
- Potential for unnecessary context switches
  - Can limit scalability for highly contended locks
- May not provide benefit on uniprocessors

Locks in Java/C#

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

Locks "this"

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

Multiprocessor Locks

- Problem: stopping threads running on other processors is
  - too expensive (requires interprocessor irq)
  - also would violate protection (locking = unprivileged op, stopping = privileged op) so couldn't do it in unprivileged code
- Instead: use atomic instructions provided by hardware
  - These are all variations of a "read-and-modify" theme
  - test-and-set, atomic-swap, compare-and-exchange, fetch-and-add
- Locks are then built on top of these

Atomic Swap

```
// In C, an atomic swap instruction would look like this
void
atomic_swap(int *memory1, int *memory2)
{
        [ disable interrupts in CPU;
          lock memory bus for other processors ]
        int tmp = *memory1;
        *memory1 = *memory2;
        *memory2 = tmp;
         [ unlock memory bus; reenable interrupts ]
}
```


Spinlocks

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

```
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

- 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 & Disabling Preemption

- 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, we usually combine spinlocks with disabling preemption on the same processor
  - 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)

Spinlocks (Optimized)

```
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;
}
```

- Idea: only try "expensive" atomic_swap instruction if you've seen lock unlocked

Locks: More Practical Issues

- How expensive are locks by themselves – aside from the performance impact of the serialization they can cause?
- Two considerations:
  - Cost to acquire uncontended lock
    - UP Kernel: disable/enable irq + memory access
    - In most 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
  - When implementing locks, optimizing uncontended case is important
  - "Hot locks" can sack performance easily

Locks: Ownership & Recursion

- Locks typically (not always) have explicit 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