Recap: Synchronization

- Disabling IRQs – use to protect against concurrent access by IRQ handler
- Locks – use to protect against concurrent access by other threads
  - Locking strategies
  - Implementation of locks on uniprocessor
    - Requires disable_preemption
    - Involves state change of thread if contended
- Today: more examples, multiprocessor locks, semaphores

Announcements

- Project 1 is due Feb 27, 11:59pm
  - Should have finished alarm clock probably by next Wednesday
- Monday Office Hours 3-4pm

Locks in Java/C#

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

```
synchronized 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

Multiprocessor Locks

- Can’t stop threads running on other processors
  - too expensive (interprocessor irq)
  - also would violate protection (locking = unprivileged op, stopping = privileged op)
- Instead: use atomic instructions provided by hardware
  - All variations of ‘read-and-modify’ theme
  - test-and-set, atomic-swap, compare-and-exchange, fetch-and-add
- Locks are built on top of these
Atomic Swap

// In C, an atomic swap instruction would 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 *)
{
    int lockstate = LOCKED;
    while (lockstate == LOCKED)
    {
        atomic_swap(&lockstate, &state);
        continue;
    }
}

lock_release(struct lock *)
{
    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, 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)

Spinlocks (Faster)

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

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

- Only try “expensive” atomic_swap instruction if you’ve seen lock unlocked

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
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