CS 3204
Operating Systems
Lecture 14
Godmar Back

Announcements
• Project 1 is due Feb 27, 11:59pm
  – Should have finished alarm clock by now
  – Basic priority by this weekend
    • priority-change, -preempt, -fifo, -sema, -condvar
  – Priority donation & advanced scheduler will likely take
    more time than alarm clock & priority scheduling
• Use forum & office hours
• Check website for reading assignments:
  Stallings Chapter 6 & some of 9

Monitors in C
• POSIX Threads & Pintos
  – are Mesa-style, so must always use “while()”
  – See also book Chapter 5 on discussion of Hoare vs
    Mesa-style
• No compiler support, must do everything manually
  – must declare locks & condition vars
  – must call lock_acquire/lock_release when
    entering/leaving the monitor
  – must use cond_wait/cond_signal to wait for/signal
    condition
• Upside: more flexibility

Monitors in Java
• synchronized block
  means
    – enter monitor
    – execute block
    – leave monitor
  – wait()/notify() use
    condition variable associated with receiver
    – Every object in Java can
      function as a condition var

Per Brinch Hansen’s Criticism
• See Java’s Insecure Parallelism [Brinch Hansen 1999]
• Says Java abused concept of monitors because
  Java does not require all accesses to shared
  variables to be within monitors
• Why did designers of Java not follow his lead?
  – Performance: compiler can’t easily decide if object is
    local or not - conservatively, would have to make all
    public methods synchronized – pay at least cost of
    atomic instruction on entering every time

Readers/Writer w/ Monitor

Q.: does this implementation prevent starvation?
Optimistic Concurrency Control

• Alternative to locks: instead of serializing access, detect when bad interleaving occurred, retry if so

```c
void increment_counter(int *counter) {
    do {
        int oldvalue = *counter;
        int newvalue = oldvalue + 1;
        [BEGIN ATOMIC COMPARE-AND-SWAP INSTRUCTION]
        if (*counter == oldvalue) { *counter = newvalue; success = true; }
        else { success = false; }
        [END CAS]
    } while (!success);
}
```

Optimistic Concurrency Control (2)

• Other names:
  – lock-free synchronization
  – wait-free synchronization
  – non-blocking synchronization

• x86 supports this via cmpxchg instruction

• Advantages:
  – Less overhead for uncontended locks (faster, and need no storage for lock queue)
  – Synchronizes with IRQ handler
  – Easier to clean up when killing a thread

• Disadvantages
  – Can requires lots of retries (more inefficient that even a hot lock since no thread might make progress)

Deadlock

• A situation in which two or more threads or processes are blocked and cannot proceed

  “blocked” either on a resource request that can’t be granted, or waiting for an event that won’t occur

  Possible causes: resource-related or communication-related

  Cannot easily back out

Deadlock Canonical Example (1)
Canonical Example (2)

```cpp
class account {
  pthread_mutex_t lock;
  int amount;  const char *name;
  public:
  account(int amount, const char *name) :
    amount(amount), name(name)  { pthread_mutex_init(&this->lock, NULL); }
  void transferTo(account *that, int amount) {
    pthread_mutex_lock(&this->lock);
    pthread_mutex_lock(&that->lock);
    cout << "Transfering $" << amount << " from " <<
         this->name << " to " << that->name << endl;
    this->amount -= amount;
    that->amount += amount;
    pthread_mutex_unlock(&that->lock);
    pthread_mutex_unlock(&this->lock);
  }
};
```

```cpp
account acc1(10000, "acc1");
account acc2(10000, "acc2");
// Thread 1:
for (int i = 0; i < 10000; i++)
  acc2.transferTo(&acc1, 20);

// Thread 2:
for (int i = 0; i < 10000; i++)
  acc1.transferTo(&acc2, 20);
```

Q.: How to fix?