Operating Systems Programming

Discussion Section
February 9, 2010
Plan for Today

• Project Stuff

• Mutual Exclusion, Critical Sections, Locks, Semaphores, Spinlocks, Mutexes, Monitors, Condition Variables, Deadlock, Priority Inversion, Priority Donation
Project 1 Timeline

• **Initial design doc:** due Thursday Feb 11th

• **Design review** with me: Monday Feb 15th and Tuesday Feb 16th. I’ll add times on Wed Feb 17th if necessary.

• **Code:** due Monday Feb 22nd

• **Final design doc:** due Tuesday Feb 23rd

• **Peer evaluations:** also Tuesday Feb 23rd
Design Doc Expectations

• < 4000 words long

• Contents:
  – Overview of solution (2-4 lines)
  – Correctness constraints
  – Data structures
  – Design decisions
  – Descriptions of Java classes and methods
  – Pseudocode for key algorithms
  – Test plan and test cases!
The classic example

- Suppose we have to implement a function to withdraw money from a bank account:

```c
int withdraw(account, amount) {
    int balance = get_balance(account);
    balance -= amount;
    put_balance(account, balance);
    return balance;
}
```

- Now suppose that you and your partner share a bank account with a balance of $100.00
  - what happens if you both go to separate ATM machines, and simultaneously withdraw $10.00 from the account?
• Represent the situation by creating a separate thread for each person to do the withdrawals – have both threads run on the same bank mainframe:

```c
int withdraw(account, amount) {
    int balance = get_balance(account);
    balance -= amount;
    put_balance(account, balance);
    return balance;
}

int withdraw(account, amount) {
    int balance = get_balance(account);
    balance -= amount;
    put_balance(account, balance);
    return balance;
}
```
Interleaved schedules

• The problem is that the execution of the two threads can be interleaved, assuming preemptive scheduling:

```c
// Execution sequence as seen by CPU
balance = get_balance(account);
balance -= amount;
context switch
balance = get_balance(account);
balance -= amount;
context switch
put_balance(account, balance);
context switch
put_balance(account, balance);
```

• What’s the account balance after this sequence? — who’s happy, the bank or you?
• How often is this unfortunate sequence likely to occur?
Other Execution Orders

- Which interleavings are ok? Which are not?

```c
int withdraw(account, amount) {
    int balance = get_balance(account);
    balance -= amount;
    put_balance(account, balance);
    return balance;
}
```
int xfer(from, to, amt) {
  int bal = withdraw( from, amt );
  deposit( to, amt );
  return bal;
}

int xfer(from, to, amt) {
  int bal = withdraw( from, amt );
  deposit( to, amt );
  return bal;
}
And This?

```
i++;  

i++;  
```

```
What’s the Problem?

• Perspective #1:
  – Want to maintain an invariant on a shared resource (account) between some set of instructions
  – Need atomicity to guarantee invariant is maintained

• Perspective #2:
  – Two concurrent entities want to access a shared resource (account) without any synchronization
  – Creates a race condition
    • output is non-deterministic, depends on timing
  – Need some way to re-introduce determinism, so we can reason about programs

• Synchronization is necessary for any shared data structure
  – buffers, queues, lists, hash tables, scalars, ...
How To Maintain Invariants?

• We want to use mutual exclusion to synchronize access to shared resources

• Code that uses mutual exclusion to synchronize its execution is called a critical section
  – only one thread at a time can execute in the critical section
  – all other threads are forced to wait on entry
  – when a thread leaves a critical section, another can enter
Mechanisms for building critical sections

• Locks
  – very primitive, minimal semantics; used to build others

• Semaphores
  – basic, easy to get the hang of, hard to program with
  – binary semaphore effectively a lock

• Monitors
  – high level, requires language support, implicit operations
  – easy to program with; Java “synchronized()” as an example
  – includes concept of waiting and signaling (i.e., condition variables)
Locks

• A lock is a object (in memory) that provides the following two operations:
  – acquire(): a thread calls this before entering a critical section
  – release(): a thread calls this after leaving a critical section

• Threads pair up calls to acquire() and release()
  – between acquire() and release(), the thread owns/holds the lock
  – acquire() does not return until the caller owns/holds the lock
    • at most one thread can hold a lock at a time
  – so: what can happen if the calls aren’t paired?

• Two basic flavors of locks
  – spinlock (busy-waiting)
  – blocking (a.k.a. “mutex”)
Using locks

int withdraw(account, amount) {
    acquire(lock);
    balance = get_balance(account);
    balance -= amount;
    put_balance(account, balance);
    release(lock);
    return balance;
}

• What happens when green tries to acquire the lock?
• Why is the “return” outside the critical section?
  – is this ok?
Spinlocks

• **How do we implement locks?** Here’s one attempt:

  ```c
  struct lock {
      int held = 0;
  }
  void acquire(lock) {
      while (lock->held);
      lock->held = 1;
  }
  void release(lock) {
      lock->held = 0;
  }
  ```

  the caller “busy-waits”, or spins, for lock to be released ⇒ hence spinlock

• **Why doesn’t this work?**
  – where is the race condition?
Implementing locks (cont.)

• Problem is that implementation of locks has critical sections, too!
  – the acquire/release must be **atomic**
    • atomic == executes as though it could not be interrupted
    • code that executes “all or nothing”

• Need help from the hardware
  – atomic instructions
    • test-and-set, compare-and-swap, ...
  – disable/reenable interrupts
    • to prevent context switches
Spinlocks redux: Test-and-Set

• CPU provides the following as one atomic instruction:

```c
bool test_and_set(bool *flag) {
    bool old = *flag;
    *flag = True;
    return old;
}
```

• Remember, this is a single uninterruptible instruction...
Problems with spinlocks

• Spinlocks work, but are horribly wasteful!
  – if a thread is spinning on a lock, the thread holding the lock cannot make progress
  – And neither can anyone else! (Why?)

• Only want spinlocks as primitives to build higher-level synchronization constructs
  – Why is this okay?
Problems with disabling interrupts

• Only available to the kernel
  – Can’t allow user-level to disable interrupts!

• Insufficient on a multiprocessor
  – Each processor has its own interrupt mechanism
Monitors and Condition Variables

• A monitor is a programming language construct that supports controlled access to shared data
  – synchronization code is added by the compiler
    • why does this help?

• A monitor encapsulates:
  – shared data structures
  – procedures that operate on the shared data
  – synchronization between concurrent threads that invoke those procedures

• Data can only be accessed from within the monitor, using the provided procedures
  – protects the data from unstructured access
Monitor

waiting queue of threads trying to enter the monitor

at most one thread in monitor at a time
Problem: Bounded Buffer Scenario

- Buffer is empty
- Now what?
Problem: Bounded Buffer Scenario

- Buffer is full
- Now what?
Bounded Buffer Scenario with CV’s

- Buffer is full
- Now what?
Condition Variables

- A place to wait; sometimes called a rendezvous point

- “Required” for monitors
  - So useful they’re often provided even when monitors aren’t available

- Three operations on condition variables
  - `wait()`, `sleep()`
    - release monitor lock, so somebody else can get in
    - wait for somebody else to signal condition
    - thus, condition variables have associated wait queues
  - `signal()`, `notify()`, `wake()`
    - wake up at most one waiting thread
      - “Hoare” monitor: wakeup immediately, signaller steps outside
    - if no waiting threads, signal is lost
      - this is different than semaphores: no history!
  - `broadcast()`, `notifyAll()`, `wakeAll()`
    - wake up all waiting threads
Bounded Buffer using (Hoare) Monitors

```java
monitor BoundedBuffer<T> {
    T[] buffer;
    Condition not_full, not_empty;

    void produce(T t) {
        if (buffer is full, determined maybe by a count)
            not_full.sleep();
        insert t into buffer;
        not_empty.wake();
    }

    T consume() {
        if (buffer is empty, determined maybe by a count)
            not_empty.sleep();
        T t = get resource from buffer;
        not_full.wake();
        return t;
    }
}
```
Bounded Buffer using Locks and CVs

```java
class BoundedBuffer<T> {
    T[] buffer;
    Lock l;
    Condition not_full(l), not_empty(l);

    void produce(T t) {
        l.acquire();
        if (buffer is full, determined maybe by a count)
            not_full.sleep();
        insert t into buffer;
        not_empty.wake();
        l.release();
    }

    T consume() {
        l.acquire();
        if (buffer is empty, determined maybe by a count)
            not_empty.sleep();
        T t = get resource from buffer;
        not_full.wake();
        l.release();
        return t;
    }
}
```
There is a subtle issue with that code...

- Who runs when the `wake()` is done and there is a thread waiting on the condition variable?

- **Hoare monitors:** `wake(c)` means
  - run waiter immediately
  - waker blocks immediately
    - condition guaranteed to hold when waiter runs
    - but, waker must **restore monitor invariants** before signalling!
      - cannot leave a mess for the waiter, who will run immediately!

- **Mesa monitors:** `wake(c)` means
  - waiter is made ready, but the waker continues
    - waiter runs when waker leaves monitor (or waits)
  - waker need not restore invariant until it leaves the monitor
  - **being woken up is only a hint that something has changed**
    - condition may no longer hold
    - must recheck conditional case
Hoare vs. Mesa Monitors

• Hoare monitors:  
  \[
  \text{if (notReady) sleep(c)}
  \]

• Mesa monitors:  
  \[
  \text{while (notReady) sleep(c)}
  \]

• Mesa monitors easier to use
  – more efficient
  – fewer context switches
  – directly supports broadcast

• Hoare monitors leave less to chance
  – when wake up, condition guaranteed to be what you expect
Deadlock

a.acquire();
...
b.acquire();
...
b.release();
a.release();

b.acquire();
...
a.acquire();
...
a.release();
b.release();
Deadlock

Execution sequence as seen by CPU

```
b.acquire();
...
```

```
a.acquire();
...
```

```
b.acquire();
```

```
a.acquire();
```

context switch

context switch
A deadlock exists if there is an *irreducible cycle* in the resource graph (such as the one above).
Deadlock

```
b.acquire();
...
```

```
a.acquire();
...
b.acquire();
a.acquire();
```
Deadlock

• Can a thread get deadlock with itself?
  – Yes, unless locks are **re-entrant**
Priority Inversion

Priority: 2

Priority: 3

Priority: 4

run queue

lock

wait queue

lock.acquire();
...

lock.acquire();
...

...
Because a thread owns lock, and the max priority of the lock wait queue is 4, the effective priority of the thread is 4!

NOTE! A thread may own more than one resource (i.e., have threads waiting on it in more than one wait queue).

```
lock.acquire();
...
lock.acquire();
...
...
...
...
...
```