Review: Definition of Monitor

- Semaphores are confusing because dual purpose:
  - Both mutual exclusion and scheduling constraints
  - Cleaner idea: Use **locks** for mutual exclusion and
    **condition variables** for scheduling constraints
- **Monitor**: a lock and zero or more condition variables
  for managing concurrent access to shared data
  - Use of Monitors is a programming paradigm
- **Lock**: provides mutual exclusion to shared data:
  - Always acquire before accessing shared data structure
  - Always release after finishing with shared data
- **Condition Variable**: a queue of threads waiting for
  something **inside** a critical section
  - Key idea: allow sleeping inside critical section by
    atomically releasing lock at time we go to sleep
  - Contrast to semaphores: Can't wait inside critical
    section

Review: Programming with Monitors

- Monitors represent the logic of the program
  - Wait if necessary
  - Signal when change something so any waiting threads
    can proceed
- Basic structure of monitor-based program:
  
  ```java
  lock
  while (need to wait) {
    condvar.wait();
  }
  unlock
  do something so no need to wait
  lock
  condvar.signal();
  unlock
  ```
  Check and/or update state variables
  Wait if necessary
  Check and/or update state variables

Goals for Today

- Java Support for Monitors
- Tips for Programming in a Project Team
- Discussion of Deadlocks
  - Conditions for its occurrence
  - Solutions for breaking and avoiding deadlock

Note: Some slides and/or pictures in the following are
adapted from slides ©2005 Silberschatz, Galvin, and Gagne.
Many slides generated from my lecture notes by Kubiatowicz.
Java Language Support for Synchronization

- Java has explicit support for threads and thread synchronization
- Bank Account example:
  ```java
  class Account {
    private int balance;
    // object constructor
    public Account (int initialBalance) {
      balance = initialBalance;
    }
    public synchronized int getBalance() {
      return balance;
    }
    public synchronized void deposit(int amount) {
      balance += amount;
    }
  }
  ```
- Every object has an associated lock which gets automatically acquired and released on entry and exit from a synchronized method.

Java Language Support for Synchronization (con't)

- Java also has synchronized statements:
  ```java
  synchronized (object) {
    ...
  }
  ```
  - Since every Java object has an associated lock, this type of statement acquires and releases the object's lock on entry and exit of the body
  - Works properly even with exceptions:
    ```java
    synchronized (object) {
      DoFoo();
      ...
    }
    ```
    ```java
    void DoFoo() {
      throw errException;
    }
    ```

Java Language Support for Synchronization (con't 2)

- In addition to a lock, every object has a single condition variable associated with it
  - How to wait inside a synchronization method or block:
    ```java
    void wait(long timeout); // Wait for timeout
    void wait(long timeout, int nanoseconds); //variant
    void wait();
    ```
  - How to signal in a synchronized method or block:
    ```java
    void notify(); // wakes up oldest waiter
    void notifyAll(); // like broadcast, wakes everyone
    ```
  - Condition variables can wait for a bounded length of time. This is useful for handling exception cases:
    ```java
    t1 = time.now();
    while (!ATMRequest()) {
      wait(CHECKPERIOD);
      t2 = time.new();
      if (t2 - t1 > LONG_TIME) checkMachine();
    }
    ```
- Not all Java VMs equivalent!
  ```java
  » Different scheduling policies, not necessarily preemptive!
  ```

Tips for Programming in a Project Team

- Big projects require more than one person (or long, long, long time)
  - Big OS: thousands of person-years!
- It's very hard to make software project teams work correctly
  - Doesn't seem to be as true of big construction projects
    ```java
    » Empire state building finished in one year: staging iron production thousands of miles away
    » Or the Hoover dam: built towns to hold workers
    ```
- Is it OK to miss deadlines?
  ```java
  » We make it free (slip days)
  » Reality: they're very expensive as time-to-market is one of the most important things!
  ```
  "You just have to get your synchronization right!"
Big Projects

- What is a big project?
  - Time/work estimation is hard
  - Programmers are eternal optimists
    (it will only take two days)
  - This is why we bug you about starting the project early
  - Had a grad student who used to say he just needed “10 minutes” to fix something. Two hours later...
- Can a project be efficiently partitioned?
  - Partitionable task decreases in time as you add people
    - But, if you require communication:
      - Time reaches a minimum bound
      - With complex interactions, time increases!
  - Mythical person-month problem:
    - You estimate how long a project will take
    - Starts to fall behind, so you add more people
    - Project takes even more time!

Techniques for Partitioning Tasks

- Functional
  - Person A implements threads, Person B implements semaphores, Person C implements locks...
  - Problem: Lots of communication across APIs
    - If B changes the API, A may need to make changes
    - Story: Large airline company spent $200 million on a new scheduling and booking system. Two teams “working together.” After two years, went to merge software. Failed! Interfaces had changed (documented, but no one noticed). Result: would cost another $200 million to fix.
- Task
  - Person A designs, Person B writes code, Person C tests
    - May be difficult to find right balance, but can focus on each person’s strengths (Theory vs systems hacker)
    - Since Debugging is hard, Microsoft has two testers for each programmer
  - Most CS162 project teams are functional, but people have had success with task-based divisions

Communication

- More people mean more communication
  - Changes have to be propagated to more people
  - Think about person writing code for most fundamental component of system: everyone depends on them!
- Miscommunication is common
  - “Index starts at 0? I thought you said 1!”
- Who makes decisions?
  - Individual decisions are fast but trouble
  - Group decisions take time
  - Centralized decisions require a big picture view (someone who can be the “system architect”)
- Often designating someone as the system architect can be a good thing
  - Better not be clueless
  - Better have good people skills
  - Better let other people do work

Coordination

- More people ⇒ no one can make all meetings!
  - They miss decisions and associated discussion
  - Example from earlier class: one person missed meetings and did something group had rejected
  - Why do we limit groups to 5 people?
    - You would never be able to schedule meetings
  - Why do we require 4 people minimum?
    - You need to experience groups to get ready for real world
- People have different work styles
  - Some people work in the morning, some at night
  - How do you decide when to meet or work together?
- What about project slippage?
  - It will happen, guaranteed!
  - Ex: phase 4, everyone busy but not talking. One person way behind. No one knew until very end - too late!
  - Hard to add people to existing group
    - Members have already figured out how to work together
How to Make it Work?

- People are human. Get over it.
  - People will make mistakes, miss meetings, miss deadlines, etc. You need to live with it and adapt
  - It is better to anticipate problems than clean up afterwards.
- Document, document, document
  - Why Document?
    » Expose decisions and communicate to others
    » Easier to spot mistakes early
    » Easier to estimate progress
  - What to document?
    » Everything (but don’t overwhelm people or no one will read)
    » Standardize!
      » One programming format: variable naming conventions, tab indents, etc.
      » Comments (Requires, effects, modifies) – javadoc?

Suggested Documents for You to Maintain

- Project objectives: goals, constraints, and priorities
- Specifications: the manual plus performance specs
  - This should be the first document generated and the last one finished
- Meeting notes
  - Document all decisions
  - You can often cut & paste for the design documents
- Schedule: What is your anticipated timing?
  - This document is critical!
- Organizational Chart
  - Who is responsible for what task?

Use Software Tools

- Source revision control software (CVS)
  - Easy to go back and see history
  - Figure out where and why a bug got introduced
  - Communicates changes to everyone (use CVS’s features)
- Use automated testing tools
  - Write scripts for non-interactive software
  - Use “expect” for interactive software
  - Microsoft rebuilds the Longhorn/Vista kernel every night with the day’s changes. Everyone is running/testing the latest software
- Use E-mail and instant messaging consistently to leave a history trail

Test Continuously

- Integration tests all the time, not at 11pm on due date!
  - Write dummy stubs with simple functionality
    » Let’s people test continuously, but more work
  - Schedule periodic integration tests
    » Get everyone in the same room, check out code, build, and test.
    » Don’t wait until it is too late!
- Testing types:
  - Unit tests: check each module in isolation (use JUnit?)
  - Daemons: subject code to exceptional cases
  - Random testing: Subject code to random timing changes
- Test early, test later, test again
  - Tendency is to test once and forget: what if something changes in some other part of the code?
**Administrivia**

- Midterm I coming up in two weeks:
  - Wednesday, 10/11, 4:00-7:00pm, Here (10 Evans)
  - Should be 2 hour exam with extra time
  - Closed book, one page of hand-written notes (both sides)
  - Topics: Everything up to that Monday, 10/9
- No class on day of Midterm (obviously)
  - I will post extra office hours for people who have questions about the material (or life, whatever)

**Resources**

- Resources – passive entities needed by threads to do their work
  - CPU time, disk space, memory
- Two types of resources:
  - Preemptable – can take it away
    » CPU, Embedded security chip
  - Non-preemptable – must leave it with the thread
    » Disk space, plotter, chunk of virtual address space
    » Mutual exclusion – the right to enter a critical section
- Resources may require exclusive access or may be sharable
  - Read-only files are typically sharable
  - Printers are not sharable during time of printing
- One of the major tasks of an operating system is to manage resources

**Starvation vs Deadlock**

- Starvation vs. Deadlock
  - Starvation: thread waits indefinitely
    » Example, low-priority thread waiting for resources constantly in use by high-priority threads
  - Deadlock: circular waiting for resources
    » Thread A owns Res 1 and is waiting for Res 2
    » Thread B owns Res 2 and is waiting for Res 1
- Deadlock ⇒ Starvation but not vice versa
  » Starvation can end (but doesn’t have to)
  » Deadlock can’t end without external intervention

**Conditions for Deadlock**

- Deadlock not always deterministic – Example 2 mutexes:
  
<table>
<thead>
<tr>
<th>Thread A</th>
<th>Thread B</th>
</tr>
</thead>
<tbody>
<tr>
<td>x.P();</td>
<td>y.P();</td>
</tr>
<tr>
<td>y.P();</td>
<td>x.P();</td>
</tr>
<tr>
<td>y.V();</td>
<td>x.V();</td>
</tr>
<tr>
<td>x.V();</td>
<td>y.V();</td>
</tr>
</tbody>
</table>

  - Deadlock won’t always happen with this code
    » Have to have exactly the right timing (“wrong” timing?)
    » So you release a piece of software, and you tested it, and there it is, controlling a nuclear power plant…
- Deadlocks occur with multiple resources
  - Means you can’t decompose the problem
  - Can’t solve deadlock for each resource independently
- Example: System with 2 disk drives and two threads
  - Each thread needs 2 disk drives to function
  - Each thread gets one disk and waits for another one
**Bridge Crossing Example**

- Each segment of road can be viewed as a resource
  - Car must own the segment under them
  - Must acquire segment that they are moving into
- For bridge: must acquire both halves
  - Traffic only in one direction at a time
  - Problem occurs when two cars in opposite directions on bridge: each acquires one segment and needs next
- If a deadlock occurs, it can be resolved if one car backs up (preempt resources and rollback)
  - Several cars may have to be backed up
- Starvation is possible
  - East-going traffic really fast ⇒ no one goes west

**Train Example (Wormhole-Routed Network)**

- Circular dependency (Deadlock!)
  - Each train wants to turn right
  - Blocked by other trains
  - Similar problem to multiprocessor networks
- Fix? Imagine grid extends in all four directions
  - Force ordering of channels (tracks)
    » Protocol: Always go east-west first, then north-south
  - Called "dimension ordering" (X then Y)

**Dining Lawyers Problem**

- Five chopsticks/Five lawyers (really cheap restaurant)
  - Free-for all: Lawyer will grab any one they can
  - Need two chopsticks to eat
- What if all grab at same time?
  - Deadlock!
- How to fix deadlock?
  - Make one of them give up a chopstick (Hah!)
  - Eventually everyone will get chance to eat
- How to prevent deadlock?
  - Never let lawyer take last chopstick if no hungry lawyer has two chopsticks afterwards

**Four requirements for Deadlock**

- Mutual exclusion
  - Only one thread at a time can use a resource.
- Hold and wait
  - Thread holding at least one resource is waiting to acquire additional resources held by other threads
- No preemption
  - Resources are released only voluntarily by the thread holding the resource, after thread is finished with it
- Circular wait
  - There exists a set \( \{ T_1, ..., T_n \} \) of waiting threads
    » \( T_1 \) is waiting for a resource that is held by \( T_2 \)
    » \( T_2 \) is waiting for a resource that is held by \( T_3 \)
    » ... 
    » \( T_n \) is waiting for a resource that is held by \( T_1 \)
Resource-Allocation Graph

- **System Model**
  - A set of Threads $T_1, T_2, \ldots, T_n$
  - Resource types $R_1, R_2, \ldots, R_m$
    - CPU cycles, memory space, I/O devices
  - Each resource type $R_i$ has $W_i$ instances.
  - Each thread utilizes a resource as follows:
    » Request() / Use() / Release()

- **Resource-Allocation Graph**
  - $V$ is partitioned into two types:
    » $T = \{T_1, T_2, \ldots, T_n\}$, the set of threads in the system.
    » $R = \{R_1, R_2, \ldots, R_m\}$, the set of resource types in system
  - request edge – directed edge $T_i \rightarrow R_j$
  - assignment edge – directed edge $R_j \rightarrow T_i$

Resource Allocation Graph Examples

- **Recall:**
  - request edge – directed edge $T_i \rightarrow R_j$
  - assignment edge – directed edge $R_j \rightarrow T_i$

Methods for Handling Deadlocks

- Allow system to enter deadlock and then recover
  - Requires deadlock detection algorithm
  - Some technique for forcibly preempting resources and/or terminating tasks
- Ensure that system will never enter a deadlock
  - Need to monitor all lock acquisitions
  - Selectively deny those that might lead to deadlock
- Ignore the problem and pretend that deadlocks never occur in the system
  - Used by most operating systems, including UNIX

Deadlock Detection Algorithm

- Only one of each type of resource $\Rightarrow$ look for loops
- More General Deadlock Detection Algorithm
  - Let $[X]$ represent an m-ary vector of non-negative integers (quantities of resources of each type):
    - $[\text{FreeResources}]$: Current free resources each type
    - $[\text{Request}_X]$: Current requests from thread $X$
    - $[\text{Alloc}_X]$: Current resources held by thread $X$
  - See if tasks can eventually terminate on their own
  - $[\text{Avail}] = [\text{FreeResources}]$
  - Add all nodes to UNFINISHED
  - do {
      done = true
      Foreach node in UNFINISHED {
        if $([\text{Request}_{\text{node}}] \leq [\text{Avail}])$
        remove node from UNFINISHED
        $[\text{Avail}] = [\text{Avail}] + [\text{Alloc}_{\text{node}}]$
        done = false
      }
    } until(done)
  - Nodes left in UNFINISHED $\Rightarrow$ deadlocked
What to do when detect deadlock?

- Terminate thread, force it to give up resources
  - In Bridge example, Godzilla picks up a car, hurls it into the river. Deadlock solved!
  - Shoot a dining lawyer
  - But, not always possible – killing a thread holding a mutex leaves world inconsistent

- Preempt resources without killing off thread
  - Take away resources from thread temporarily
  - Doesn’t always fit with semantics of computation

- Roll back actions of deadlocked threads
  - Hit the rewind button on TiVo, pretend last few minutes never happened
  - For bridge example, make one car roll backwards (may require others behind him)
  - Common technique in databases (transactions)
  - Of course, if you restart in exactly the same way, may reenter deadlock once again

- Many operating systems use other options

Summary

- Suggestions for dealing with Project Partners
  - Start Early, Meet Often
  - Develop Good Organizational Plan, Document Everything, Use the right tools, Develop Comprehensive Testing Plan
  - (Oh, and add 2 years to every deadline!)

- Starvation vs. Deadlock
  - Starvation: thread waits indefinitely
  - Deadlock: circular waiting for resources

- Four conditions for deadlocks
  - Mutual exclusion
    - Only one thread at a time can use a resource
  - Hold and wait
    - Thread holding at least one resource is waiting to acquire additional resources held by other threads
  - No preemption
    - Resources are released only voluntarily by the threads
  - Circular wait
    - $\exists \{T_1, \ldots, T_n\}$ of threads with a cyclic waiting pattern

Summary (2)

- Techniques for addressing Deadlock
  - Allow system to enter deadlock and then recover
  - Ensure that system will never enter a deadlock
  - Ignore the problem and pretend that deadlocks never occur in the system

- Deadlock detection
  - Attempts to assess whether waiting graph can ever make progress

- Next Time: Deadlock prevention
  - Assess, for each allocation, whether it has the potential to lead to deadlock
  - Banker’s algorithm gives one way to assess this