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:
  ```c
  lock
  while (need to wait) {
    condvar.wait();
  }
  unlock
  do something so no need to wait
  lock
  condvar.signal();
  unlock
  ```

Review: Basic Readers/Writers Solution
- Correctness Constraints:
  - Readers can access database when no writers
  - Writers can access database when no readers
  - Only one thread manipulates state variables at a time
- Basic structure of a solution:
  - Reader()
    Wait until no writers
    Access database
    Check out - wake up a waiting writer
  - Writer()
    Wait until no active readers or writers
    Access database
    Check out - wake up waiting readers or writers
  - State variables (Protected by a lock called "lock"):
    » int AR: Number of active readers; initially = 0
    » int WR: Number of waiting readers; initially = 0
    » int AW: Number of active writers; initially = 0
    » int WW: Number of waiting writers; initially = 0
    » Condition okToRead = NIL
    » Condition okToWrite = NIL

Review: Code for a Reader
```c
Reader() {
  // First check self into system
  lock.Acquire();
  while ((AW + WW) > 0) {
    // Is it safe to read?
    WR++;  // No Writers exist
    okToRead.wait(&lock);  // Sleep on cond var
    WR--;  // No longer waiting
    AR++;  // Now we are active!
    lock.release();
  }
  // Perform actual read-only access
  // now, check out of system
  lock.Acquire();
  AR--;  // No longer active
  if (AR == 0 & WW > 0) // No other active readers
    okToWrite.signal();  // Wake up one writer
  lock.Release();
}
```
**Review: Code for a Writer**

```java
Writer() {
    // First check self into system
    lock.Acquire();
    while ((AW + AR) > 0) { // Is it safe to write?
        WW++; // No. Active users exist
        okToWrite.wait(&lock); // Sleep on cond var
        WW--; // No longer waiting
    }
    AW++; // Now we are active!
    lock.release();
    // Perform actual read/write access
    AccessDatabase(ReadWrite);
    // Now, check out of system
    lock.Acquire();
    AW--; // No longer active
    if (WW > 0) { // Give priority to writers
        okToWrite.signal(); // Wake up one writer
    } else if (WR > 0) { // Otherwise, wake reader
        okToRead.broadcast(); // Wake all readers
    }
    lock.Release();
}
```

**Goals for Today**

- Discuss language support for synchronization
- Discussion of Deadlocks
  - Conditions for its occurrence
  - Solutions for breaking and avoiding deadlock

**Construction of Monitors from Semaphores (con’t)**

- Problem with previous try:
  - P and V are commutative – result is the same no matter what order they occur
  - Condition variables are NOT commutative
- Does this fix the problem?

```java
Wait(Lock lock) {
    lock.Release();
    semaphore.P();
    lock.Acquire();
}
Signal() {
    if semaphore queue is not empty
        semaphore.V();
}
```

- Not legal to look at contents of semaphore queue
- There is a race condition - signaler can slip in after lock release and before waiter executes semaphore.P()
- It is actually possible to do this correctly
  - Complex solution for Hoare scheduling in book
  - Can you come up with simpler Mesa-scheduled solution?
C-Language Support for Synchronization

- C language: Pretty straightforward synchronization
  - Just make sure you know all the code paths out of a critical section
  ```
  int Rtn() {
    lock.acquire();
    if (exception) {
      lock.release();
      return errReturnCode;
    }
    lock.release();
    return OK;
  }
  ```
  - Watch out for `setjmp`/`longjmp`!
    » Can cause a non-local jump out of procedure
    » In example, procedure E calls longjmp, popping stack back to procedure B
    » If Procedure C had lock.acquire, problem!

C++ Language Support for Synchronization

- Languages with exceptions like C++
  - Languages that support exceptions are problematic (easy to make a non-local exit without releasing lock)
  - Consider:
    ```
    void Rtn() {
      lock.acquire();
      DoFoo();
      lock.release();
    }
    void DoFoo() {
      if (exception) throw errException;
    }
    ```
  - Notice that an exception in `DoFoo()` will exit without releasing the lock

C++ Language Support for Synchronization (con't)

- Must catch all exceptions in critical sections
  - Must catch exceptions, release lock, then re-throw the exception:
    ```
    void Rtn() {
      lock.acquire();
      try {
        DoFoo();
        if (exception) throw errException;
      }
      catch (…) { // catch exception
        lock.release(); // release lock
        throw; // re-throw the exception
      }
    }
    ```

Java Language Support for Synchronization

- Java has explicit support for threads and thread synchronization
- Bank Account example:
  ```java
  public class Account {
    private int balance;
    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();
      ...
    }
    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.now();
      if (t2 - t1 > LONG_TIME) checkMachine();
    }
    ```
  - Not all Java VMs equivalent!
    ```java
    Different scheduling policies, not necessarily preemptive!
    ```

Administrivia

• Midterm I coming up in 2½ weeks:
  - Wednesday, 3/8, TBA, room TBA
  - 1½ hour exam
  - Closed book, one page of hand-written notes (both sides)
  - Topics: Everything up to that Monday, 3/8

• No class on day of Midterm
  - I will post extra office hours for people who have questions about the material (or life, whatever)

Cautionary Tale: OS/2

• Every major OS since 1985 provides threads
  - Makes it easier to write concurrent programs
• Microsoft OS/2 (circa 1988): initially, a failure
• IBM re-wrote it using threads for everything
  - Window systems, Inter-Process Communication, ...
  - OS/2 let you print while you worked!
  - Could have 100 threads, but most not on run queue
    (waiting for something)
• Each thread needs its own stack, say 9 KB
• Result: System needs an extra 1MB of memory
  - $200 in 1988
• Moral: Threads are cheap, but they’re not free
  - <$0.10 today, but context switching is expensive...
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(); y.P();</td>
<td>x.P(); y.P();</td>
</tr>
<tr>
<td>y.P(); x.P();</td>
<td>y.V(); x.V();</td>
</tr>
<tr>
<td>x.V(); y.V();</td>
<td>x.V(); 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 \( T_3 \)
    - ...  
    - \( T_1 \) 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 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

- T1 \rightarrow R1
- T2 \rightarrow R2
- T3 \rightarrow R3
- T4 \rightarrow R4

Simple Resource Allocation Graph

Allocation Graph With Deadlock

Allocation Graph With Cycle, but No Deadlock

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
  \(done = true\)
  do {
    Foreach node in UNFINISHED {
      if \((\text{[Request]}_\text{node}) <= \text{[Avail]}\) {
        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
Techniques for Preventing Deadlock

• Infinite resources
  - Include enough resources so that no one ever runs out of resources. Doesn’t have to be infinite, just large
  - Give illusion of infinite resources (e.g. virtual memory)
  - Examples:
    » Bay bridge with 12,000 lanes. Never wait!
    » Infinite disk space (not realistic yet?)
• No Sharing of resources (totally independent threads)
  - Not very realistic
  - How the phone company avoids deadlock
    » Call to your Mom in Toledo, works its way through the phone lines, but if blocked get busy signal.
    - Technique used in Ethernet/some multiprocessor nets
    » Everyone speaks at once. On collision, back off and retry
    - Inefficient, since have to keep retrying
    » Consider: driving to San Francisco; when hit traffic jam, suddenly you’re transported back home and told to retry!

Techniques for Preventing Deadlock (con’t)

• Make all threads request everything they’ll need at the beginning.
  - Problem: Predicting future is hard, tend to overestimate resources
  - Example:
    » If need 2 chopsticks, request both at same time
    » Don’t leave home until we know no one is using any intersection between here and where you want to go; only one car on the Bay Bridge at a time
• Force all threads to request resources in a particular order preventing any cyclic use of resources
  - Thus, preventing deadlock
  - Example (x.P, y.P, z.P, …)
    » Make tasks request disk, then memory, then...
    » Keep from deadlock on freeways around SF by requiring everyone to go clockwise

Banker’s Algorithm for Preventing Deadlock

• Toward right idea:
  - State maximum resource needs in advance
  - Allow particular thread to proceed if:
    - (available resources - #requested) ≥ max
    - remaining that might be needed by any thread
  - Banker’s algorithm (less conservative):
    - Allocate resources dynamically
      - Evaluate each request and grant if some ordering of threads is still deadlock free afterward
      - Technique: pretend each request is granted, then run deadlock detection algorithm, substituting [Max - Alloc] for [Request]
      - Grant request if result is deadlock free (conservative!)
  - Keeps system in a “SAFE” state, i.e. there exists a sequence (T₁, T₂, … Tₙ) with T₁ requesting all remaining resources, finishing, then T₂ requesting all remaining resources, etc.
  - Algorithm allows the sum of maximum resource needs of all current threads to be greater than total resources

Banker’s Algorithm Example

• Banker’s algorithm with dining lawyers
  - “Safe” (won’t cause deadlock) if when try to grab chopstick either:
    » Not last chopstick
    » Is last chopstick but someone will have two afterwards
  - What if k-handed lawyers? Don’t allow if:
    » It’s the last one, no one would have k
    » It’s 2nd to last, and no one would have k-1
    » It’s 3rd to last, and no one would have k-2
Summary

- Language support for synchronization:
  - Be careful of exceptions within critical sections
  - Java provides synchronized keyword and one condition-variable per object (with `wait()` and `notify()`)

- 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
    - There exists a set \( T_1, \ldots, T_n \) 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

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