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:
  ```
  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 writer
  - 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

```cpp
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
    AccessDatabase(ReadOnly);
    // 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    AR--; // No longer active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

Writer() {
    // First check self into system
    lock.Acquire();
    while ((AW + AR) > 0) { // Is it safe to write?
        AW++; // 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    AW--; // No longer active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

Can we construct Monitors from Semaphores?

• Locking aspect is easy: Just use a mutex
• Can we implement condition variables this way?
  Wait() { semaphore.P(); }
  Signal() { semaphore.V(); }
  - Doesn’t work: Wait() may sleep with lock held
• Does this work better?
  Wait(Lock lock) {
    lock.Release();
    semaphore.P();
    lock.Acquire();
  }
  Signal() { semaphore.V(); }
  - No: Condition vars have no history, semaphores have history:
    » What if thread signals and no one is waiting? NO-OP
    » What if thread later waits? Thread Waits
    » What if thread V’s and none is waiting? Increment
    » What if thread later does P? Decrement and continue

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?
  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

```c
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:

```c++
void Rtn() {
    lock.acquire();
    ...
    DoFoo();
    ...
    lock.release();
    }
}
```

- 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:

```c++
void Rtn() {
    lock.acquire();
    try {
        ...
        DoFoo();
        ...
    } catch (...) { // catch exception
        lock.release(); // release lock
        throw; // re-throw the exception
    }
    lock.release();
    }
}
```

- 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

- 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();
    ...
} 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:
    » void wait(long timeout); // Wait for timeout
    » void wait(long timeout, int nanoseconds); // variant
    » void wait();
  - How to signal in a synchronization method or block:
    » 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!
    » Different scheduling policies, not necessarily preemptive!

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

Administrivia

• Midterm I coming up in two weeks:
  - Wednesday, 10/12, 5:30 – 8:30, Here
  - Should be 2 hour exam with extra time
  - Closed book, one page of hand-written notes (both sides)
  - Topics: Everything up to that Monday, 10/10
• No class on day of Midterm
  - I will post extra office hours for people who have questions about the material (or life, whatever)
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 doesn't have to be deterministic.
  - Consider mutexes 'x' and 'y':
    - Thread A: x.P(); y.P();
    - Thread B: y.P(); x.P();
  - 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 has managed to get one disk and is waiting 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 \( \{ T_1, \ldots, T_n \} \) 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 \( T_3 \)
    » \( \ldots \)
    » \( T_n \) is waiting for a resource that is held by \( T_1 \)

Symbols

- 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

- Recall:
  - request edge – directed edge \( T_1 \rightarrow R_1 \)
  - assignment edge – directed edge \( R_1 \rightarrow T_1 \)
Methods for Handling Deadlocks

- Allow system to enter deadlock and then recover
  - Requires deadlock detection algorithm
  - Some technique for selectively 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 ⇒ 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
    - \([\text{Avail}] = [\text{FreeResources}]\)
    - \(\text{Add all nodes to UNFINISHED}\)
    - \(\text{done} = \text{true}\)
    - \(\text{do} \{\)
      - \(\text{Foreach node in UNFINISHED} \{
        \text{if} ([\text{Request}_\text{node}] \leq [\text{Avail}]) \{
          \text{remove node from UNFINISHED}
          \text{[Avail]} = [\text{Avail}] + [\text{Alloc}_\text{node}]
          \text{done} = \text{false}
        \}
      \}\) \(\text{until(done)}\)
    - \(\text{Nodes left in UNFINISHED} ⇒ \text{deadlocked}\)

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 will 12,000 lanes. Never wait!
    - Infinite disk space (not realistic yet?)
- No Sharing of resources (totally independent threads)
  - Not very realistic
- Don’t allow waiting
  - 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. If collision, back off and try again
  - Inefficient, since have to keep retrying
    - Consider: trying to drive to San Francisco; when hit traffic jam, suddenly you were transported back home and told to try again!
Techniques for Preventing Deadlock (con’t)

• Make all threads request everything they’ll need at the beginning.
  - Problem: Predicting future is hard, tend to over-estimate 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 Prevents any cyclic use of resources
  - Thus preventing deadlock
  - Example
    » 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-resource] - [Alloc-resource] for [Request-resource]
      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 {T1, ..., Tn} 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 every 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