Review: Monitor with Condition Variables

• Lock: the lock provides mutual exclusion to shared data
  – Always acquire before accessing shared data structure
  – Always release after finishing with shared data
  – Lock initially free
• Condition Variable: a queue of threads waiting for something inside a critical section
  – Key idea: make it possible to go to sleep inside critical section by atomically releasing lock at time we go to sleep
  – Contrast to semaphores: Can’t wait inside critical section

Review: Condition Variables

• How do we change the RemoveFromQueue() routine to wait until something is on the queue?
  – Could do this by keeping a count of the number of things on the queue (with semaphores), but error prone
• 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

• Operations:
  – Wait(&lock): Atomically release lock and go to sleep. Re-acquire lock later, before returning.
  – Signal(): Wake up one waiter, if any
  – Broadcast(): Wake up all waiters
• Rule: Must hold lock when doing condition variable ops!
  – In Birrell paper, he says can perform signal() outside of lock – IGNORE HIM (this is only an optimization)

Review: Mesa vs. Hoare Monitors

• Need to be careful about precise definition of signal and wait. Consider a piece of our dequeue code:
  ```java
  while (queue.isEmpty()) {
    dataready.wait(&lock); // If nothing, sleep
  }
  item = queue.dequeue(); // Get next item
  ```
  – Why didn’t we do this?
  ```java
  if (queue.isEmpty()) {
    dataready.wait(&lock); // If nothing, sleep
  }
  item = queue.dequeue(); // Get next item
  ```
• Answer: depends on the type of scheduling
  – Hoare-style (most textbooks):
    » Signaler gives lock, CPU to waiter; waiter runs immediately
    » Waiter gives up lock, processor back to signaler when it exits critical section or if it waits again
  – Mesa-style (most real operating systems):
    » Signaler keeps lock and processor
    » Waiter placed on ready queue with no special priority
    » Practically, need to check condition again after wait
Extended Example: Readers/Writers Problem

• Motivation: Consider a shared database
  – Two classes of users:
    » Readers – never modify database
    » Writers – read and modify database
  – Is using a single lock on the whole database sufficient?
    » Like to have many readers at the same time
    » Only one writer at a time

Basic Readers/Writers Solution

• Correctness Constraints:
  – Readers can access database when no writers
  – Writers can access database when no readers or writers
  – 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

Code for a Reader

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();
}

Why Release the Lock here?

Code for a Writer

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();

Why Give priority to writers?

Why broadcast() here instead of signal()?
Simulation of Readers/Writers solution

- Consider the following sequence of operators:
  - R1, R2, W1, R3

- On entry, each reader checks the following:
  ```java
  while ((AW + WW) > 0) {
    // Is it safe to read?
    WR++;
    // No. Writers exist
    okToRead.wait(&lock);
    // Sleep on cond var
    // No longer waiting
    WR--;
  }
  AR++;
  // Now we are active!
  ```

- First, R1 comes along:
  - AR = 1, WR = 0, AW = 0, WW = 0

- Next, R2 comes along:
  - AR = 2, WR = 0, AW = 0, WW = 0

- Now, readers make take a while to access database
  - Situation: Locks released
  - Only AR is non-zero

- When writer wakes up, get:
  - AR = 0, WR = 1, AW = 1, WW = 0

- Then, when writer finishes:
  ```java
  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
  }
  ```

  - Writer wakes up reader, so get:
    - AR = 1, WR = 0, AW = 0, WW = 0

- When reader completes, we are finished

Simulation(2)

- Next, W1 comes along:
  ```java
  while (((AW + AR) > 0) {
    // Is it safe to write?
    WW++;
    // No. Active users exist
    okToWrite.wait(&lock); // Sleep on cond var
    // No longer waiting
    WW--;
  }
  AW++;
  ```

  - Can't start because of readers, so go to sleep:
    - AR = 2, WR = 0, AW = 0, WW = 1

- Finally, R3 comes along:
  - AR = 2, WR = 1, AW = 0, WW = 1

- Now, say that R2 finishes before R1:
  - AR = 1, WR = 1, AW = 0, WW = 1

- Finally, last of first two readers (R1) finishes and wakes up writer:
  ```java
  if (AR == 0 && WW > 0)
  // No other active readers
  okToWrite.signal(); // Wake up one writer
  ```

Simulation(3)

- Can readers starve? Consider Reader() entry code:
  ```java
  while ((AW + WW) > 0) {
    // Is it safe to read?
    WR++;
    // No. Writers exist
    okToRead.wait(&lock);
    // Sleep on cond var
    // No longer waiting
    WR--;
  }
  AR++;
  // Now we are active!
  ```

- What if we erase the condition check in Reader exit?
  ```java
  AR--; // No longer active
  if (AR == 0 && WW > 0)
  // No other active readers
  okToWrite.signal(); // Wake up one writer
  ```

- Further, what if we turn the signal() into broadcast()?
  ```java
  AR--; // No longer active
  okToWrite.broadcast(); // Wake up one writer
  ```

- Finally, what if we use only one condition variable (call it “okToContinue”) instead of two separate ones?
  - Both readers and writers sleep on this variable
  - Must use broadcast() instead of signal()
Administrivia

- Midterm coming up soon
  - Wednesday 3/9 6-7:30PM
  - 10 EVANS (Seats: 237); 155 DWINELLE (Seats: 481)
    » We will assign you to a room
  - Closed book, no calculators, one double-side page of handwritten notes

- No class that day, extra office hours

- Topics will include the material through lecture 12 (Wed 3/2)
  - Includes lectures, project 1, homeworks, readings, textbook

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(); }

-Does this work better?
  
  Wait(Lock lock) {
    lock.Release();
    semaphore.P();
    lock.Acquire();
  }
  Signal() { semaphore.V(); }

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?
Monitor Conclusion

- 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
  ```

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
  - Catch exceptions, release lock, and re-throw exception:
    ```
    void Rtn() {
      lock.acquire();
      try {
        ...
        DoFoo();
        ...
        catch (...) { // catch exception
          lock.release(); // release lock
          throw; // re-throw the exception
        }
        lock.release();
      }
    }
    void DoFoo() {
      ...
      if (exception) throw errException;
    }
    ```
    - Can deallocate/free lock regardless of exit method
Java Language Support for Synchronization

• Java has explicit support for threads and thread synchronization

• Bank Account example:
  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:
  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:
    synchronized (object) {
      ...
      DoFoo();
      ...
    }
    void DoFoo() {
      throw errException;
    }

Recall: Better Implementation of Locks by Disabling Interrupts

• Key idea: maintain a lock variable and impose mutual exclusion only during operations on that variable

int mylock = FREE;
Acquire(&mylock) — Wait until lock is free, then grab
Release(&mylock) — Unlock, waking up anyone waiting

Acquire(int *lock) {
  disable interrupts;
  if (*lock == BUSY) {
    put thread on wait queue;
    Go to sleep();
  } else {
    // Enable interrupts?
    *lock = FREE;
  }
}

Release(int *lock) {
  disable interrupts;
  if (anyone on wait queue) {
    take thread off wait queue
    Place on ready queue;
  } else {
    *lock = FREE;
  }
}

• Really only works in kernel – why?
In-Kernel Lock: Simulation

value = 0;
Acquire() {
  disable interrupts;
  if (value == 1) {
    put thread on wait-queue;
    go to sleep();
  } else {
    value = 1;
    lock.Release();
    enable interrupts;
  }
}
Release() {
  disable interrupts;
  if anyone on wait queue {
    take thread off wait-queue
    Place on ready queue;
  } else {
    value = 0;
    lock.Release();
    enable interrupts;
  }
}

lock.Acquire();
... critical section; ...
lock.Release();

Value: 0  waiters  owner  READY

Running
Thread A

Value: 0  waiters  owner  READY

Running
Thread B

2/22/16  Joseph CS162 ©UCB Spring 2016  Lec 9.25

In-Kernel Lock: Simulation

Value: 1  waiters  owner  READY

Running
Thread A

Value: 1  waiters  owner  READY

Running
Thread B

2/22/16  Joseph CS162 ©UCB Spring 2016  Lec 9.26

Value: 1  waiters  owner  READY

Running
Thread A

Value: 1  waiters  owner  READY

Running
Thread B

2/22/16  Joseph CS162 ©UCB Spring 2016  Lec 9.27

Value: 1  waiters  owner  READY

Running
Thread A

Value: 1  waiters  owner  READY

Running
Thread B

2/22/16  Joseph CS162 ©UCB Spring 2016  Lec 9.28
Discussion

- Notice that Scheduling here involves deciding who to take off the wait queue
  - Could do by priority, etc.

- Same type of code works for in-kernel condition variables
  - The Wait queue becomes unique for each condition variable
  - Once again, transition to and from queues occurs with interrupts disabled
Synchronization Summary

• **Semaphores**: Like integers with restricted interface
  - Two operations:
    » P(): Wait if zero; decrement when becomes non-zero
    » V(): Increment and wake a sleeping task (if exists)
  - Can initialize value to any non-negative value
  - Use separate semaphore for each constraint

• **Monitors**: A lock plus zero or more condition variables
  - Always acquire lock before accessing shared data
  - Use condition variables to wait inside critical section
  » Three Operations: Wait(), Signal(), Broadcast()
Scheduling Policy Goals/Criteria

- Minimize Response Time
  - Minimize elapsed time to do an operation (or job)
  - Response time is what the user sees:
    » Time to echo a keystroke in editor
    » Time to compile a program
    » Real-time Tasks: Must meet deadlines imposed by World

- Maximize Throughput
  - Maximize operations (or jobs) per second
  - Throughput related to response time, but not identical:
    » Minimizing response time will lead to more context switching than if you only maximized throughput
    » Two parts to maximizing throughput
      » Minimize overhead (for example, context-switching)
      » Efficient use of resources (CPU, disk, memory, etc)

- Fairness
  - Share CPU among users in some equitable way
  - Fairness is not minimizing average response time:
    » Better average response time by making system less fair

First-Come, First-Served (FCFS) Scheduling

- First-Come, First-Served (FCFS)
  - Also “First In, First Out” (FIFO) or “Run until done”
    » In early systems, FCFS meant one program scheduled until done (including I/O)
    » Now, means keep CPU until thread blocks

Example:

<table>
<thead>
<tr>
<th>Process</th>
<th>Burst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>P₁</td>
<td>24</td>
</tr>
<tr>
<td>P₂</td>
<td>3</td>
</tr>
<tr>
<td>P₃</td>
<td>3</td>
</tr>
</tbody>
</table>

Suppose processes arrive in the order: P₁, P₂, P₃
The Gantt Chart for the schedule is:

- Waiting time for P₁ = 0; P₂ = 24; P₃ = 27
- Average waiting time: (0 + 24 + 27)/3 = 17
- Average Completion time: (24 + 27 + 30)/3 = 27

Convoy effect: short process behind long process

FCFS Scheduling (Cont.)

- Example continued:
  - Suppose that processes arrive in order: P₂, P₃, P₁
Now, the Gantt chart for the schedule is:

<table>
<thead>
<tr>
<th>P₂</th>
<th>P₃</th>
<th>P₁</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>30</td>
</tr>
</tbody>
</table>

- Waiting time for P₁ = 6; P₂ = 0; P₃ = 3
- Average waiting time: (6 + 0 + 3)/3 = 3
- Average Completion time: (3 + 6 + 30)/3 = 13

In second case:
  - average waiting time is much better (before it was 17)
  - Average completion time is better (before it was 27)

FIFO Pros and Cons:
  - Simple (+)
  - Short jobs get stuck behind long ones (-)
    » Safeway: Getting milk, always stuck behind cart full of small items. Upside: get to read about space aliens

Semaphores: Like integers with restricted interface
  - Two operations:
    » P(): Wait if zero; decrement when becomes non-zero
    » V(): Increment and wake a sleeping task (if exists)
  - Can initialize value to any non-negative value
  - Use separate semaphore for each constraint

Monitors: A lock plus one or more condition variables
  - Always acquire lock before accessing shared data
  - Use condition variables to wait inside critical section
  - Three Operations: Wait(), Signal(), and Broadcast()

Scheduling: selecting a waiting process from the ready queue and allocating the CPU to it

FCFS Scheduling:
  - Run threads to completion in order of submission
  - Pros: Simple
  - Cons: Short jobs get stuck behind long ones

Round-Robin Scheduling:
  - Give each thread a small amount of CPU time when it executes; cycle between all ready threads
  - Pros: Better for short jobs
  - Cons: Poor when jobs are same length