Monitors

CS439: Principles of Computer Systems
February 8, 2017
Threads and the Scheduler
(or, Why Multi-threaded Programming is Hard)

Given two threads, A and B, how might their executions be scheduled?
Concurrence is great...

```c
int a=1, b=2;
main() {
    createThread(fn1, 4);
    createThread(fn2, 5);
    thread_join(all);
}

fn1(int arg1){
    if(a) b++;
}

fn2(int arg1){
    a=arg1;
}
```

What are the values of a and b after execution?

A. a=1, b=2
B. a=1, b=3
C. a=5, b=2
D. a=5, b=3
... but can be problematic

```c
int a=1, b=2;
main() {
    createThread(fn1, 4);
    createThread(fn2, 5);
    thread_join(all);
}

fn1(int arg1){
    if(a) b++;
}

fn2(int arg1){
    a=0;
}
```

What are the values of a and b after execution?

A. a=0, b=2
B. a=0, b=3
C. a=1, b=2
D. a=1, b=3
Last Time

• Terminology
  – Safety and liveness
  – Atomic Instructions, Synchronization, Mutual Exclusion, Critical Sections

• Synchronization in Software
  – Abstractions built on top of hardware support
  – Locks
  – Semaphores
Semaphore Recap

• Two types: binary and counted
• Used for locking and synchronization
• Two *atomic* operations:
  – down(): wait for semaphore to be available, decrement value
  – up(): signal waiting threads semaphore is available, increment value
• Need separate semaphores for locking and synchronizing
Implementing Down() and Up()

```cpp
int value = val; //initial value depends on the problem and
                //indicates number of resources available

Semaphore::Down()
{
    if(value == 0)
    {
        add t to wait queue;
        t->block()
    }
    value = value - 1;
}

Semaphore::Up()
{
    value = value + 1;
    if(t on wait queue)
    {
        remove t from wait queue;
        wakeup(t);
    }
}
```
When to Use Semaphores

- **Mutual Exclusion**
  - Use to protect the critical section (see Too Much Milk Example)

- **Control Access to a Pool of Resources**
  - Counted semaphore

- **General Synchronization**
  - Use to enforce general scheduling constraints where the threads must wait for some circumstance
  - Value is typically 0 to start
Today’s Agenda

• Producers/Consumers problem
• More Synchronization in Software
  – Monitors
    • Locks and Condition Variables
• Readers/Writers problem
Semaphore Example: Producers/Consumers

Semaphore mutex = 1 //access to buffer
Semaphore empty = N //count of empty slots
Semaphore full = 0 //count of full slots
int buffer[N]

BoundedBuffer::Producer(){
   <produce item>
   empty->Down() //get empty spot
   mutex->Down() //get access to buffer

   <add item to buffer>
   mutex->Up() //release buffer
   full->Up() //another item in buffer
}

BoundedBuffer::Consumer(){
   full->Down() //get item
   mutex->Down() //get access to buffer

   <remove item from buffer>
   mutex->Up() //release buffer
   empty->Up() //another empty slot
   <use item>
}
Semaphore Summary

• Semaphores can be used for three purposes:
  – to ensure mutually exclusive execution of a critical section (like locks)
  – to control access to a shared pool of resources (using a counting semaphore)
  – to cause one thread to wait for a specific event

• AND
  – No busy wait

• So... They’re perfect! Right?
Um, No.
(Problems with Semaphores)

• Huge step up from what we had, but...
• Essentially shared global variables
• Too many purposes
  – Waiting for a condition is independent of mutual exclusion
• No control or guarantee of proper usage
• Difficult to read (and develop) code
• Often studied for history
  – Not typically used in new application code
  – (Where are they used?)
• So...
What NOT to do

Semaphore mutex = 1  //access to buffer
Semaphore empty = N  //count of empty slots
Semaphore full = 0  //count of full slots
int buffer[N]

BoundedBuffer::Producer(){
   <produce item>
   empty->Down()  //get empty spot
   mutex->Down()  //get access to buffer

   <add item to buffer>
   mutex->Up()  //release buffer
   full->Up()  //another item in buffer
}

BoundedBuffer::Consumer(){
   mutex->Down()  //get access to buffer
   full->Down()  //get item

   <remove item from buffer>
   mutex->Up()  //release buffer
   empty->Up()  //another empty slot
   <use item>
}
Monitors
Introducing Monitors

• Monitors guarantee mutual exclusion
  – only one thread may execute a given monitor method at a time
• First introduced as a programming language construct (Mesa, Java)
• Now also define a programming convention and can be used in any language (C, C++, ...)
• Object-oriented style:
  – Collect related shared data into an object/module
    • Associate a lock with each one
    • All data is private
  – Define methods for accessing the shared data
    • These are critical sections
Monitors, Formally

A monitor defines a *lock* and zero or more *condition variables* for managing concurrent access to shared data.

- uses the *lock* to ensure that only a single thread is active in the monitor at any point
- the *lock* also provides mutual exclusion for shared data
- *Condition variables* enable threads to block waiting for an event inside of critical sections
  - release the lock at the same time the thread is put to sleep
Monitor Design

- Goal is to encapsulate shared data
  - Class in C++ or Java
  - Struct or File in C (logical encapsulation)

- Goal is to provide for mutual exclusion
  - Has a lock (exactly one!)

- Goal is to allow for synchronization
  - Has condition variables

- Goal is to allow operations on the shared data
  - Has functions or methods
Monitor Functions: Implementation

• Acquire the lock at the start of every function (first thing!)
  – Operate on the shared data
  – Temporarily release the lock if they can’t complete due to missing resource (use condition variable for this)
  – Reacquire the lock when they can continue (again, condition variable!)
  – Operate on the shared data

• Release the lock at the end
Semaphore Example: Producers/Consumers (Recall)

Semaphore mutex = 1  //access to buffer
Semaphore empty = N  //count of empty slots
Semaphore full = 0    //count of full slots
int buffer[N]

BoundedBuffer::Producer(){
    <produce item>
    empty->down() //get empty spot
    mutex->down() //get access to buffer

    <add item to buffer>
    mutex->up() //release buffer
    full->up() //another item in buffer
}

BoundedBuffer::Consumer(){
    full->down()  //get item
    mutex->down() //get access to buffer

    <remove item from buffer>
    mutex->up() //release buffer
    empty->up() //another empty slot
    <use item>
}
Semaphore mutex = 1  //access to buffer
Semaphore empty = N  //count of empty slots
Semaphore full = 0    //count of full slots
int buffer[N]

BoundedBuffer::Producer(){
    <produce item>
    empty->down() //get empty spot
    mutex->down() //get access to buffer

    <add item to buffer>

    mutex->up() //release buffer
    full->up() //another item in buffer
}
Condition Variables

• Enable threads to wait efficiently for changes to shared state protected by a lock
• Each one is a queue of waiting threads (no state!)
• Enable the thread to block inside a critical section by *atomically* releasing the Lock at the same time the thread is blocked

Rule: A thread *must* hold the lock when doing condition variable operations
Condition Variable Operations

1. Wait(Lock lock)
   – atomic (release lock, move thread to waiting queue, suspend thread)
   – when the thread wakes up it re-acquires the lock (before returning from wait)
   – thread will always block
2. Signal(Lock lock)
   – wake up waiting thread, if one exists. Otherwise, it’s a no-op
3. Broadcast(Lock lock)
   – wake up all waiting threads, if any exist. Otherwise, it’s a no-op
Monitor Operations

**Lock->Acquire()** //acquires lock, when returns, thread has lock

**Lock->Release()** //releases lock

**CondVar::Wait(lock){**
  *move thread to wait queue and suspend thread
  *release lock
  *on signal, wake up, re-acquire lock
  *return

**CondVar::Signal(lock){**
  *wake up a thread waiting on condVar
  *return

**CondVar::Broadcast(lock){**
  *wake up ALL threads waiting on condVar
  *return

*The pseudocode on this slide assumes Mesa/Hansen semantics (more in a minute)*
Resource Variables

• Conditions variables (unlike semaphores) keep no state
• Each condition variable should have a resource variable that tracks the state of that resource
  – You must maintain this variable
• Check the resource variable before calling wait on the associated condition variable to ensure the resource really isn’t available
• Once the resource is available, claim it (subtract the amount you are using!)
• Before signaling that you are finished with a resource, indicate the resource is available by increasing the resource variable
Signal() Semantics

Which thread executes once signal() is called?

– If there are no waiting threads, the signaler continues and the signal is effectively lost

– If there is a waiting thread (or two):
  • There are at least two ready threads: the one that called signal() and the one that was (or will be) awakened
  • Exactly one of the threads can execute or we will have more than one thread active in the monitor (violates mutual exclusion!)
  • So which thread gets to execute?
Whose turn is it?

Mesa/Hansen Style
- The thread that signals keeps the lock (and thus the processor)
- The waiting thread waits for the lock
  - Signal is only a hint that the condition may be true; shared state may have changed!
  - Adding signals affects performance, but never safety
- Implemented in Java and most real operating systems

Hoare Style
- The thread that signals gives up the lock and the waiting thread gets the lock
  - Signaling is atomic with the resumption of the waiting thread
  - Shared state cannot change before waiting thread is resumed
- When the thread that was waiting and is now executing exits or waits again, it releases the lock back to the signaling thread
- Implemented in most textbooks (not yours!)
More on Turns

• With Mesa-style, waiting thread may need to wait again after it is wakened (Why?), so while is important
• With Hoare-style, we can change while to if because a waiting thread runs immediately

```java
public void remove()
{
    lock->Acquire()
    while (queue_size <= 0)
    {
        full->wait(lock);
    }
    queue_size--;
    remove item;
    lock->Release();
}
```

Regardless, if you assume there is NO atomicity between signal() and the return from wait(), your code will always work.
Monitor Example: Items in a Queue

Lock lock;
Condition fullCV;
int queue_size;

void Add(item){
    lock->acquire()
    put item on queue;
    queue_size++;
    fullCV->signal(lock)
    lock->release()
}

void remove(){
    lock->acquire()
    while(queue_size <=0)
        fullCV->wait(lock);
    queue_size--;
    remove item;
    lock->release();
}
Semaphore Example: Producers/Consumers (Recall)

Semaphore mutex = 1 //access to buffer
Semaphore empty = N //count of empty slots
Semaphore full = 0 //count of full slots
int buffer[N]

BoundedBuffer::Producer()
{
    <produce item>
    empty->down() //get empty spot
    mutex->down() //get access to buffer

    <add item to buffer>
    mutex->up() //release buffer
    full->up() //another item in buffer
}

BoundedBuffer::Consumer()
{
    full->down() //get item
    mutex->down() //get access to buffer

    <remove item from buffer>
    mutex->up() //release buffer
    empty->up() //another empty slot
    <use item>
}
Semaphore Example: Producers/Consumers (Recall)

Semaphore mutex = 1  //access to buffer
Semaphore empty = N  //count of empty slots
Semaphore full = 0    //count of full slots
int buffer[N]

BoundedBuffer::Consumer(){
  full->down()   //get item
  mutex->down()  //get access to buffer

  <remove item from buffer>

  mutex->up()    //release buffer
  empty->up()    //another empty slot
  <use item>
}
Monitor Example:
Producers/Consumers

class BBMonitor{
    public: <methods>
    private: item buffer[N];
        Lock lock;
        Condition fullCV, emptyCV;
    int empty=N;
    int full=0;
}

BoundedBuffer::Producer(){
    lock->Acquire()
    while(empty == 0)
        emptyCV->Wait(lock) //get empty spot
        empty--;
    <add item to buffer>
    full += 1
    fullCV->Signal(lock) //another item in buffer
    lock->Release()
}

BoundedBuffer::Consumer(){
    lock->Acquire()
    while(full == 0)
        fullCV->Wait(lock) //get item
        full--
    <remove item from buffer>
    empty++; 
    emptyCV->Signal(lock) //another empty slot
    lock->Release()
}
Every monitor function should begin with what command?

A. Wait()
B. Signal()
C. Lock->Acquire()
D. Lock->Release()
E. Broadcast()
When using monitors, you should:

A. Call wait() to determine resource availability AND block if the resource is unavailable
B. Call wait() in an if statement that checks resource availability
C. Call wait() in a while loop that checks resource availability
Comparing Monitors and Semaphores

• Condition variables do not have any history
  – on signal() if no one is waiting, the signal is a no-op
  – if thread then calls condition->wait(), it waits.

• Semaphores do have history
  – on up() if no one is waiting, the value of the semaphore is incremented
  – if a thread then calls semaphore->down(), the value is decremented and the thread continues
So... signal() and down()

• In semaphores, down() and up() are commutative
  – result is the same regardless of the order of execution

• Condition variables are not commutative
  – so they must be in a critical section to access state variables and do their job

You *can* implement Monitors with Semaphores.
# Monitors and Semaphores: Recap

<table>
<thead>
<tr>
<th>Both</th>
<th>Semaphores</th>
<th>Monitors</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Provide mutual exclusion and synchronization</td>
<td>• Semaphores are basically generalized locks</td>
<td>• Consist of a lock and one or more condition variables</td>
</tr>
<tr>
<td>• Have signal() and wait()</td>
<td>• Binary and counting semaphores</td>
<td>• Encapsulate shared data</td>
</tr>
<tr>
<td>• Support a queue of threads that are waiting to access a critical section (e.g., to buy milk)</td>
<td>• Used for mutual exclusion and synchronization</td>
<td>• Use locks for mutual exclusion</td>
</tr>
<tr>
<td>• No busy waiting!</td>
<td></td>
<td>• Use condition variables for synchronization</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Wrap operations on shared data with a lock</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Condition variables release lock temporarily</td>
</tr>
</tbody>
</table>

Both provide mutual exclusion and synchronization and have signal() and wait(). Semaphores support a queue of threads waiting to access a critical section (e.g., to buy milk) and no busy waiting! Monitors consist of a lock and one or more condition variables, encapsulate shared data, use locks for mutual exclusion, use condition variables for synchronization, wrap operations on shared data with a lock, and condition variables release lock temporarily.
A Different Type of Problem

• We’ve looked at problems where we protect shared data by only allowing one thread in the critical section at a time

• Is this always appropriate? When might we want to let more threads access shared data at once?
Readers/Writers Problem

• Data is shared among several threads
  – Some only read
  – Some only write

• To get correct results, we allow *multiple readers* at a time, but only *one writer* at a time

• How can we control access to the object to permit this protocol?
Correctness Criteria

• Each read or write of the shared data must happen within a critical section

• Guarantee mutual exclusion for writers

• Allow multiple readers to execute in the critical section at once

• Allow one writer (and no readers) to execute in the critical section at once
Readers and Writers: Monitor Solution

• What methods do we need?
• How many locks?
• How many condition variables?
• What should we name them?
• Any other variables?

• Assume we’re going to say <read> and <write> for accesses to the shared data.
Readers and Writers: Monitor Solution

Variables:

read()

write()

Is our solution fair?

A. Yes
B. No, favors readers
C. No, favors writers
Understanding Our Solution

It works, but it favors readers over writers

– Any reader blocks all writers
– All readers must finish before a writer can start
– Last reader will wake any writer, but a writer wakes all readers and writers
– If a writer exits and a reader goes next, then all readers that are waiting will get through
Readers and Writers: Monitor Solution

Variables:

```c
read()

write()

read()
```

```c
}

}
```
Alternative Semantics

- It may be that you would like a writer to enter its critical section as soon as possible.
- How could we implement that?
Signal vs. Broadcast

• It is always safe to use broadcast() instead of signal()
  – only performance is affected

• signal() is preferable when
  – at most one waiting thread can make progress
  – any thread waiting on the condition variable can make progress

• broadcast() is preferable when
  – multiple waiting threads may be able to make progress
  – the same condition variable is used for multiple predicates
    • some waiting threads can make progress, others can’t
Summary

• A monitor wraps operations with a lock
• Condition variables release lock temporarily
• Monitors do not keep state---a call to wait() will always wait
• Monitors can be implemented by following the monitor rules for acquiring and releasing locks
Announcements

- Discussion sections are Friday! Problem Set 3 is posted.
- Project 0 is due Friday at 11:59p
- Project 1 is posted and due 2/24
  - Groups must be registered by 2/15
- Exam 1 is in TWO weeks
  - Wednesday, 2/22! 7p! WEL 2.224!