Deadlock and Monitors
Bringing It All Together

• Processes
  – Abstraction for protection
  – Define address space

• Threads
  – Share (and communicate) through global and static data, share the heap, each has its own stack and full use of the registers
  – Race conditions may be a problem!

• CPU Schedulers
  – May pre-empt a process or thread at any time

• Solving race conditions OR Ensuring correctness
  – Safety and liveness
  – Atomic instructions
  – Synchronization: mutual exclusion, counted resources...
  – Locks and semaphores!
Too Much Milk: Lock Solution

Lock lock;

All Threads
lock->Acquire();
if(noMilk)
    buy milk;
lock->Release();
Too Much Milk: Semaphore Solution

Semaphore sema = 1;

All Threads
sema->down();
if(noMilk)
    buy milk;
sema->up();
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

• Famous synchronization problems
  – Producers/Consumers (aka Bounded Buffer)
  – Dining Philosophers
• Another synchronization problem: Deadlock
  – The four necessary and sufficient conditions
• Another synchronization technique: Monitors
Producers/Consumers

- Buffer is a global data structure of fixed size $n$
- Producer produces items and puts them in the buffer
- Consumer consumes items from the buffer
- What happens if the producer produces an item and the buffer is full?
- What happens if the consumer consumes an item and the buffer is empty?
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>
}
Let’s make a small change...

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>
}
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
• Not typically used in new application code
  – Where are they used?
• So... *what do we use in application code?*
Dining Philosophers
Dining Philosophers

N philosophers are sitting at a table with N chopsticks, each needs two chopsticks to eat, and each philosopher alternates between thinking, getting hungry, and eating.
Dining Philosophers: Solution

do forever:
  think
  get hungry
  wait(chopstick[i])
  wait(chopstick[(i+1) % N])
  eat
  signal(chopstick[i])
  signal(chopstick[(i+1) % N])

Does this solution work?

A. Yes
B. No
Deadlock
What is Deadlock?

- *Deadlock* occurs when two or more threads are waiting for an event that can only be generated by these same threads.

- Deadlock is not starvation
  - Starvation can occur without deadlock
    - occurs when a thread waits indefinitely for some resources, but other threads are actually using it
  - But deadlock does imply starvation

- We will be discussing how deadlock can occur in the multi-threaded (or multiprocess) code we write (there are other places deadlock may occur)
Conditions for Deadlock

Deadlock occurs when **all** of the following conditions hold:

1. **Mutual Exclusion**: at least one thread must hold a resource in non-sharable mode

2. **Hold and Wait**: at least one thread holds a resource and is waiting for other resources to become available. A different thread holds the resource.

3. **No Pre-emption**: a thread only releases a resource voluntarily; another thread or the OS cannot force the thread to release the resource.

4. **Circular Wait**: A set of waiting threads \( \{t_1, ..., t_n\} \) where \( t_i \) is waiting on \( t_{i+1} \) (\( i=1 \) to \( n \)) and \( t_n \) is waiting on \( t_1 \)
Deadlock Prevention

Prevent deadlock by insuring that at least one of the necessary conditions doesn’t hold

1. **Mutual Exclusion**: make resources sharable
2. **Hold and Wait**: guarantee a thread cannot hold one resource when it requests another (or must request all at once)
3. **No Pre-emption**: If a thread requests a resource that cannot be immediately allocated to it, then the OS pre-empts all the resources the thread is currently holding. Only when all the resources are available will the OS restart the thread
4. **Circular Wait**: Impose an ordering on the locks and request them in order
Deadlock Prevention: Lock Ordering

• Order all locks
• All code grabs locks in a predefined order

Complications:
  – Maintaining global order is difficult in a large project
  – Global order can force a client to grab a lock earlier than it would like, tying up the lock for longer than necessary
Dining Philosophers: Possible Solutions

• *Don’t require mutual exclusion:* ?
• *Prevent hold-and-wait:* Only let a philosopher pick up chopsticks if both are available
• *Pre-empt resources:* Designate a philosopher as the head philosopher. Allow that philosopher to take a chopstick from a neighbor if that neighbor is not currently eating.
• *Prevent circular wait by having sufficient resources:* Kick out a philosopher
• *Prevent circular wait by ordering resources:*  
  – Odd philosophers pick up right then left  
  – Even philosophers pick up left then right
• *Use a monitor...*
Monitors
Monitors

• Encapsulate shared data
  – Collect related shared data into an object/module
    • Struct or File in C (logical encapsulation)
    • All data is private

• Allow operations on the shared data
  – Define functions for accessing the shared data
    • These are the critical sections

• Provide mutual exclusion
  – Associate a lock (exactly one!) with each object/module
  – Acquire the lock before executing any function

• Allow threads to synchronize in the critical section
  – Guarantees against deadlock
  – Has condition variables for this---more in a minute!
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 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 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.
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
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

- Semaphores do have history
  - `down()` and `up()` are commutative
  - 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

- Condition variables do not have any history
  - Condition variables are not commutative
  - On `signal()` if no one is waiting, the signal is a no-op
  - If thread then calls condition->`wait()`, it waits
## Monitors and Semaphores: Recap

### Both
- Provide mutual exclusion and synchronization
- Have `signal()` and `wait()`
- Support a queue of threads that are waiting to access a critical section (e.g., to buy milk)
- No busy waiting!

### Semaphores
- Semaphores are basically generalized locks
- Binary and counting semaphores
- Used for mutual exclusion and synchronization

### 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
- Condition variables release lock temporarily
Pemberley!
Summary

• Code concurrent programs very carefully to help prevent deadlock over resources managed by the program
  – Deadlock is a situation in which a set of threads/processes cannot proceed because each requires resources held by another member of the set

• Use monitors!
  – 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 4 is posted.
• Project 1 is posted and due 10/3
  – Groups must be registered by Wednesday, 9/26
• Exam 1 is NEXT week!
  – Wednesday, 10/1, JES A121A, 7p-9p