Last Time

• Bounded Buffer
• Software abstractions for Mutual Exclusion
  – Monitors
    • Condition Variables (Zero or more)
    • Locks (Exactly One!)
Today’s Agenda

• Readers and Writers
• How to Program Multi-threaded code
• Dining Philosophers
• Deadlocks
• The Importance of Safety (Therac-25)
Readers and Writers
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:

read()

write()

}
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?
Writing Multi-threaded (User-Level) Code
Designing Multithreaded Programs

• Building a shared object class (or pseudo-class) involves familiar steps:
  – Decompose the problem into objects
  – For each object:
    • define a clear interface
    • implement methods that manipulate the state appropriately

• The new steps are straightforward:
  – Add a lock
  – Add code to acquire and release the lock
  – Identify synchronization points and add condition variables
  – Add loops to check resource status and wait using condition variables
  – Add signal() and broadcast() calls
Managing Locks

• Add a lock as a member variable for each object in the class to enforce mutual exclusion on the object’s shared state

• Acquire a lock at the start of each public method

• Release the lock at the end of each public method
  – You will be tempted to acquire/release a lock midway through a method.
  – RESIST!
Identifying Condition Variables

• Ask yourself: when can this function wait?
• Map each opportunity for waiting to a condition variable
  – full and empty in producers/consumers
• You may be able to manage with fewer condition variables
  – such as somethingChanged
  – but you must call broadcast() and not signal() (why?)
Waiting Using Condition Variables

• Every call to Condition::Wait() should be enclosed in a loop

• Loop tests the appropriate resource

• When Condition::Wait() is called, **all invariants must hold**
  – Remember, you are releasing the lock!
The Six Commandments

• Thou shalt always do things the same way
  – Habit allows you to focus on the core problem
  – Easier to review, maintain, and debug your code

• Thou shalt always synchronize with locks and condition variables
  – Condition variables and locks make code clearer
The Six Commandments

• Thou shalt always acquire the lock at the beginning of a function and release it at the end
  – Put a chunk of code that requires a lock into its own function

• Thou shalt always hold lock when operating on a condition variable
  – Condition variables are useless without shared state
  – Shared state should only be accessed using a lock
The Six Commandments

• Thou shalt always wait in a while loop
  – while() works every time if() does
  – Makes signals hints
  – Protects against spurious wake ups

• (Almost) Never sleep()
  – Use sleep() only if an action should occur at a specific real time
  – Never use sleep() to wait for an event
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)
Deadlock Example

Producer{
lock->down()
empty->down()

<do stuff>

full->up()
lock->up()
}

Consumer{
full->down()
lock->down()

<do stuff>

empty->up()
lock->up()
}
Conditions for Deadlock

Deadlock can happen **if 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 resources 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 \( \{t_1, \ldots, t_n\} \) 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 (or semaphores)
• 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
Therac-25
or The Importance of Safety
What is the Therac-25?

• Linear accelerator
• Used to treat patients ...
<table>
<thead>
<tr>
<th>Modes of Operation</th>
<th>Beam Energy</th>
<th>Beam Current</th>
<th>Beam Modifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>For electron therapy</td>
<td>5-25 MeV</td>
<td>low</td>
<td>magnets</td>
</tr>
<tr>
<td>For X-ray therapy, photo mode</td>
<td>25 MeV</td>
<td>high (100x)</td>
<td>flattener</td>
</tr>
<tr>
<td>For field light mode</td>
<td>0</td>
<td>0</td>
<td>none</td>
</tr>
</tbody>
</table>
What Went Wrong?

• Two (major) software problems
• Tons of bad software design/human failures that might have prevented this:
  – False alarms
  – Errors reported by number only and there was no documentation!
  – No clearinghouse for mistakes and company hid failures from other users
  – No end-to-end consistency checks
  – No quality control
  – Don’t trust software---hardware should have prevented this, too
What about more recent disasters?

• We don’t know for sure

• Possibly software lost treatment plan and defaulted to “all leaves open”

• Software should have sensible defaults!
Lessons

• Complex systems fail for complex reasons
• Be tolerant of inputs
• Be strict on outputs

• Assume buggy software and protect against it!
Summary

• Please Think!

• Safety first!
  – Coarse-grained locking is the easiest to get right, so do that
  – Don’t worry about performance at first
  – In fact, don’t even worry about liveness at first
Announcements

• Discussion Sections on Friday! Problem Set 4 is posted.

• Exam 1 is next Wednesday at 7p WEL 2.224
  – If you have a conflict, you should have already told me (if you don’t receive instructions by noon next Tuesday, contact me again)
  – Show up ON TIME

• Project 1 due next Friday 11:59p