The Importance of Safety and Advanced Synchronization

CS439: Principles of Computer Systems
September 27, 2017
Last Time

• Bounded Buffer
• Dining Philosophers
• Deadlock
  – When a set of threads cannot progress because each requires a resource held by another member of the set
• Prevent deadlock through resource ordering
• Monitors
  – Condition Variables (Zero or more)
  – Locks (Exactly One!)
Today’s Agenda

• Comparing Monitors and Semaphores
• The Importance of Safety (Therac-25)
• Advanced Synchronization
• How to Program Multi-threaded Code
• Pemberley
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)
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

<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>
</tbody>
</table>

- Condition variables release lock temporarily
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
Therac-25
or The Importance of Safety
What is the Therac-25?

• Linear accelerator
• Used to treat patients ...
## Modes of Operation

<table>
<thead>
<tr>
<th></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!
Advanced Synchronization
So Far: One Big Lock

• Advantage: Simple
  – Relatively easy to get correct
  – This is a big advantage!

• Disadvantages:
  – Often not great for performance
    • Eliminates advantages due to multi-threading for that part of your code
    • Eliminates advantages due to multicore in that part of your code

Any other problems?
Multi-Object Synchronization

• Transfer $100 from account A to account B
  A->subtract(100)
  B->add(100)

  Individual operations are atomic. Sequence is not.

• How should we ensure atomicity?
  – One lock for each account?
  – One lock for all accounts?
  – All accounts at one bank?
  – All accounts everywhere?
Fine-Grained Locking

• Better for performance
  – This will matter more in the kernel than in an application (the kernel affects every application!)

• Complex
  – May need to acquire multiple locks to accomplish a task (a lock for each account?)
  – Incorrect code becomes more likely
  – Deadlock
Conservative Two-Phase Locking

- **A thread must:**
  - Acquire all locks it will need
    - If all locks cannot be acquired, release any already acquired and begin again
  - Make necessary changes, commit, and release locks

- Thus B cannot see any of A’s changes until A commits and releases the lock
- Provides serializability
- Prevents deadlock
Transactions

• *Transactions* group actions together so that they are:
  – *atomic*: they all happen or they all don’t
  – *serializable*: transactions appear to happen one after the other
  – *durable*: once it happens, it sticks

• Critical sections give us atomicity and serializability, but not durability
Achieving Durability

To get durability, we need to be able to:

• *Commit*: indicate when a transaction is finished
• *Roll back*: recover from an *aborted* transaction
  – If we have a failure in the middle of a transaction, we need to be able to undo what we have done so far
• In other words, we do a set of operations tentatively.
  – If we get to the commit stage, we are okay.
  – If not, roll back operations as if the transaction never happened.
Implementing Transactions

- Key idea: Turn multiple disk updates into a single disk write!
  
  ```
  begin transaction
  x = 300
  y = 512
  Commit
  ```

- Keep `write-ahead` (or `redo`) log on disk of all changes in the transaction

- The log records everything the OS does (or tries!) to do

- Once the OS writes both changes on the log, the transaction is committed

- Then `write-behind` changes to the disk, logging all writes

- If the crash comes after a commit, the log is replayed
Imagine two threads executing this code:

```sql
write begin transaction
lock x, y
   x = x + 3
   y = y + 5
write x, y to log
unlock x, y
write Commit
```

Does this code work?

A. Yes
B. No
In the transaction log...

Assuming all goes well and initial values of x and y are 0:

```plaintext
Begin transaction
x=3
y=5
Commit
Begin transaction
x=6
y=10
Commit
```
Implementing Multi-Threaded Transactions

write begin transaction
lock x, y
    x = x + 3
    y = y + 5
write x, y to log
unlock x, y
write Commit

Given two threads A & B that execute that code in the following sequence:

1. A gets the lock, reads an modifies x and y, writes to the log, and unlocks
2. The B grabs the lock before A commits
3. B reads A’s modifications, then modifies x and y, writes to the log, unlocks, and commits
4. Then the system crashes before A commits
In the transaction log...

Assuming our crash scenario and initial values of $x$ and $y$ are 0:

Begin transaction
$x=3$
$y=5$
Begin transaction
$x=6$
$y=10$
Commit
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
Pemberley!
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
• Sometimes more fine-grained synchronization techniques are required for efficiency, but you must be extra careful
• Sometimes serializability is necessary; in those cases, use two-phase locking
• Sometimes durability is necessary; in those cases, use transactions
Announcements

• Discussion sections this week! Problem Set 4 is posted.
• Project 1 group registration is due today
• Exam 1 is MONDAY at 7p BEL 328
  – If you have a conflict, you should have already told me (if you don’t receive instructions by noon Sunday, contact me again)
  – Show up ON TIME
• My Monday office hours will be canceled
• Project 1 due NEXT Friday 11:59p