The Importance of Safety and More Synchronization

CS439: Principles of Computer Systems
February 15, 2017
Last Time

• Readers/Writers
• How to Program Multi-threaded Code
• 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 lock ordering
Today’s Agenda

• The Importance of Safety
• Advanced Synchronization
• Review
  – Atomicity
  – How we get it
  – Tradeoffs and Problems
• Pemberley
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?
A House of Cards?

• Locks and condition variables are a great way to regulate access to a *single* shared object...
• ... but multi-threaded programs often touch *multiple* shared objects
• How can we atomically modify multiple shared objects to maintain
  – Safety: prevent applications from seeing inconsistent states
  – Liveness: avoid deadlock

?
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:

```plaintext
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:

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

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
```
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.
• 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.
Synchronization Review
Concurrency Quiz

If two threads execute this program concurrently, how many different final values of the global variable X are there?

Initially, X == 0.

Thread 1

```java
void increment() {
    int tmp = X;
    tmp = tmp + 1;
    X = tmp;
}
```

Thread 2

```java
void increment() {
    int tmp = X;
    tmp = tmp + 1;
    X = tmp;
}
```

A. 0       B. 1       C. 2       D. More than 2
Schedules/Interleavings

- Model of concurrent execution
- Interleave statements from each thread into a single thread
- If any interleaving yields incorrect results, some synchronization is needed

If \( X == 0 \) initially, \( X == 1 \) at the end. WRONG result!
int flag1=0, flag2=0;

int main(){
    tid id=thread_create(p1, NULL);
    p2(); thread_join(id);
}

void p1(void * ignored){
    flag1=1;
    if(!flag2){
        critical_section_1();
    }
}

void p2(void * ignored){
    flag2=1;
    if(!flag1){
        if(!flag1){
            critical_section_2();
        }
    }
}

Can both critical sections execute during a single execution of the code?

A. Yes
B. No
Atomicity

• Required to reason about multi-threaded code without considering all interleavings
• Requires mutual exclusion
• Locks provide that solution
• Looked at lock implementation
  – Requires waiting
  – Requires hardware support
• Use software abstractions
  – Semaphores
  – Monitors (lock+condition variables)
Tradeoff and Problems: Difficult to Get Right

- Ensure safety
- Ensure liveness
- No race conditions
- No starvation
- No priority inversion
- No deadlock
In Addition... the Cost of Parallelization

\[
\text{for}(k = 0; k < n; k++)
\]

\[
a[k] = b[k]*c[k] + d[k]*e[k];
\]

How would you parallelize this?
How many threads?
The Six Commandments

• Thou shalt always do things the same way
• Thou shalt always synchronize with locks and condition variables
• Thou shalt always acquire the lock at the beginning of a function and release it at the end
• Thou shalt always hold lock when operating on a condition variable
• Thou shalt always wait in a while loop
• (Almost) Never sleep()
Why Thread Coding Standards?

• History has tested this approach
• If you follow these commandments, you will find it easier to write correct code.
• In this class, you must use them or lose points.
• We highly recommend that you continue to do so after this class
But...

- After this class, if you can come up with something better, please use it!

- BUT...
  - Lots of really smart people have thought really hard about this already, so a day or two of thought is unlikely to change the best practice
  - The consequences of getting code wrong can be atrocious
  - People who are confident about their abilities tend to perform *worse*. If you think you are a Threading and Concurrency Ninja and truly understand, then you may wish to re-evaluate...
    - Dunning-Kruger effect
In this class...

- Six commandments
- Coarse-grained locking
- Order your locks
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
Announcements

• Discussion sections this week! Problem Set 4 is posted.
• Project 1 group registration is due today
• 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 Tuesday, contact me again)
  – Show up ON TIME
• Project 1 due NEXT Friday 11:59p