Deadlock and Advanced Synchronization

CS439: Principles of Computer Systems
September 23, 2015
Last Time

• Software abstractions for Mutual Exclusion
  – Monitors
    • Condition Variables (Zero or more)
    • Locks (Exactly One!)

• Readers and Writers
Today’s Agenda

• How to Program Multi-threaded code
• Dining Philosophers
• Deadlocks
  – What causes them
• Advanced Synchronization
• Pemberley!
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 less 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
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 \( \{t_1, ..., t_n\} \) 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 resources and request them in order
Deadlock Prevention: Resource Ordering

• Order all locks (or semaphores or resources)
• 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 a resource for longer than necessary
Dining Philosophers: Possible Solutions

- *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
- *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.
- *Don’t require mutual exclusion:* ?
Advanced Synchronization
A House of Cards?

- Locks and condition variables are a great way to regulate access to a *single* shared object...
- ... but general multi-threaded programs 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
  
  \[ \begin{align*}
  \text{A} & \rightarrow \text{subtract}(100) \\
  \text{B} & \rightarrow \text{add}(100)
  \end{align*} \]

- How should we ensure atomicity?
  - One lock for each account?
  - One lock for all accounts?
  - All accounts at one bank?
  - All accounts everywhere?

Individual operations are atomic. Sequence is not.
One Big Lock

• Simple
  – Relatively easy to get correct
• Often not great for performance
  – No advantage from multi-threading for that part of your code
  – No advantage of multicore in that part of your code
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
Two-Phase Locking

Basic two-phase locking requires that the thread:
1. Acquire all locks it will need
2. 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
  - May cause deadlock

Strict: All unlocks happen at the commit
Conservative: If all locks cannot be acquired, release any already acquired and begin again (No 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
iClicker Question

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:

\[
\begin{align*}
\text{Begin transaction} \\
x &= 3 \\
y &= 5 \\
\text{Commit} \\
\text{Begin transaction} \\
x &= 6 \\
y &= 10 \\
\text{Commit}
\end{align*}
\]
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
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
• 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

• Homework 4 due Thursday at 9:45a
• Project 1 is posted is due 10/2
• Exam 1 is NEXT week! (Wednesday, 9/30! 7p!)
  – If you have a conflict, you should have already notified me via email.