

Lecture 31: More Concurrency

CS 62
Fall 2017
Kim Bruce & Alexandra Papoutsaki

*Some slides based on those from Dan Grossman,
U. of Washington*

Concurrent Programming

- Concurrency: Allowing simultaneous or interleaved access to shared resources from multiple clients
- Requires coordination, particularly synchronization to avoid incorrect simultaneous access: make somebody block
 - join is not what we want
 - block until another thread is “done using what we need” not “completely done executing”

Canonical Example

- Several ATM's accessing same account.
 - See ATM2

Bad Interleavings

Interleaved **changeBalance(-100)** calls on the same account
– Assume initial **balance** 150

	Thread 1	Thread 2
	<code>int nb = b + amount;</code>	<code>int nb = b + amount;</code>
		<code>if (nb < 0)</code>
		<code> throw new ...;</code>
		<code> balance = nb;</code>
Time ↓	<code>if (nb < 0)</code>	
	<code> throw new ...;</code>	
	<code> balance = nb;</code>	

“Lost withdraw” –
unhappy bank

Interleaving is the Problem

- Suppose:
 - Thread T1 calls `changeBalance(-100)`
 - Thread T2 calls `changeBalance(-100)`
- If second call starts before first finishes, we say the calls **interleave**
 - Could happen even with one processor since a thread can be pre-empted at any point for time-slicing
- If x and y refer to different accounts, no problem
 - “You cook in your kitchen while I cook in mine”
 - But if x and y alias, possible trouble...

Problems with Account

- Get wrong answers!
- Try to fix by getting balance again, rather than using `newBalance`.
 - Still can have interleaving, though less likely
 - Can go negative w/ wrong interleaving!

Solve with Mutual Exclusion

- At most one thread withdraws from account A at one time.
- Areas where don't want two threads executing called *critical sections*.
- Programmer needs to decide where, as compiler doesn't know intentions.

Java Solution

- *Re-entrant locks* via synchronized blocks
- Syntax:
 - **synchronized (expression) {statements}**
- Evaluates expression to an object and tries to grab it as a lock
 - If no other process is holding it, grabs it and executes statements. Releasing when finishes statements.
 - If another process is holding it, waits until it is released.
- Net result: Only one thread at a time can execute a synchronized block w/same lock

Correct Code

```
public class Account {
    private Object myLock = new Object();
    ...
    // return balance
    public int getBalance() {
        synchronized(myLock){ return balance; }
    }

    // update balance by adding amount
    public void changeBalance(int amount) {
        synchronized(myLock) {
            int newBalance = balance + amount;
            display.setText("" + newBalance);
            balance = newBalance;
        }
    }
}
```

Better Code

```
public class Account {
    ...
    // return balance
    public int getBalance() {
        synchronized(this){ return balance; }
    }

    // update balance by adding amount
    public void changeBalance(int amount) {
        synchronized(this) {
            int newBalance = balance + amount;
            display.setText("" + newBalance);
            balance = newBalance;
        }
    }
}
```

Best Code

```
public class Account {
    ...
    // return balance
    synchronized public int getBalance() {
        return balance;
    }

    // update balance by adding amount
    synchronized public void changeBalance(int amount) {
        int newBalance = balance + amount;
        display.setText("" + newBalance);
        balance = newBalance;
    }
}
```

Reentrant Locks

- If thread holds lock when executing code, then further method calls within block don't need to reacquire same lock.
 - E.g., Methods *m* and *n* are both synchronized with same lock (e.g., with *this*), and execution of *m* results in calling *n*. Then once thread has the lock executing *m*, no delay in calling *n*.

Responsiveness

Maze Program

- Uses stack to solve a maze.
- When user clicks “solve maze” button, spawns Thread to solve maze.
- What happens if send “run” instead of “start”?

Non-Event-Driven Programming

- Program in control.
- Program can ask for input at any point, with program control depending on input.
- But user can't interrupt program
 - Only give input when program ready

Event-Driven Programming

- Control inverted.
 - User takes action, program responds
- GUI components (buttons, mouse, etc.) have “listeners” associated with them that are to be notified when component generates an event.
- Listeners then take action to respond to event.