Lecture 25: Concurrency & Responsiveness

CS 62
Spring 2015
Kim Bruce & America Chambers

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

For Lab

- Will be using in-line tools for Java
  - Must do the reading before lab!!!!!
- Learn how to compile and run from command line (e.g., if want to use our parallel servers)
- Learn how to set up and use subversion repository
  - Allows you to recover old copies,
  - helps prevent problems when more than one person working on program.

Assignment

- AI'ish program to play simple chess-like game, Hex-A-Pawn.
- Build game tree
  - Players move from root to leaves (win/lose configs)
- Smart Player:
  - Trim sub-tree corresponding to last move when make a losing move.

Good Habits

- Start now (or, better yet, yesterday)!
- TA's have been instructed not to stay over regular hours. Plan on finishing early!
- Remember, exams have limited overlap with actual programming.
  - Both are very important!!

No class (or quiz) Friday

Finishing ATM Example

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 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.

Event-Driven Programming in Java

- When an event occurs, it is posted to appropriate event queue.
  - Java GUI components share an event queue.
  - Any thread can post to the queue
  - Only the "event thread" can remove event from the queue.
- When event removed from queue, thread executes the appropriate method of listener w/ event as parameter.

Example: Maze-Solver

- Start button ⇒ StartListener object
- Clear button ⇒ ClearAndChooseListener
- Maze choice ⇒ ClearAndChooseListener
- Speed slider ⇒ SpeedListener

Listeners

- Different kinds of GUI items require different kinds of listeners:
  - Button -- ActionListener
  - Mouse -- MouseListener, MouseMotionListener
  - Slider -- ChangeListener
- See GUI cheatsheet on documentation web page

Event Thread

- Removes events from queue
- Executes appropriate methods in listeners
- Also handles repaint events
- Must remain responsive!
  - Code must complete and return quickly
  - If not, then spawn new thread!

Why did Maze Freeze?

- Solver animation was being run by event thread
- Because didn't return until solved, was not available to remove events from queue.
  - Could not respond to GUI controls
  - Could not paint screen

Off to the Races

- A *race* condition occurs when the computation result depends on scheduling (how threads are interleaved). Answer depends on shared state.
- Bugs that exist only due to concurrency
  - No interleaved scheduling with 1 thread
- Typically, problem is some intermediate state that "messes up" a concurrent thread that "sees" that state

Example

```
class Stack<E> {
  ...
  synchronized void push(E val) { ... }
  synchronized E pop() {
      if(isEmpty())
          throw new StackEmptyException();
      ...
  }

  E peek() {
    E ans = pop();
    push(ans);
    return ans;
  }
}
```

Sequentially Fine

- Correct in sequential world
- May need to write this way, if only have access to push, pop, & isEmpty methods.
- peek() has no overall effect on data structure
  - reads rather than writes

Concurrently Flawed

- Way it's implemented creates an inconsistent intermediate state
  - Even though calls to push and pop are synchronized so no data races on the underlying array/list/whatever
  - (A data race is simultaneous (unsynchronized) read/write or write/write of the same memory: more on this soon)
- This intermediate state should not be exposed
  - Leads to several wrong interleavings...

Lose Invariants

- Want: If there is at least one push and no pops, then isEmpty always returns false.
- Fails with two threads if one is doing a peek, other isEmpty, & unlucky.
- Gets worse: Can lose LIFO property
  - Problem do push while doing peek.
- Want: If # pushes > # pops then peek never throws an exception.
  - Can fail if two threads do simultaneous peeks

Solution

- Make peek synchronized (w/same lock)
  - No problem with internal calls to push and pop because locks reentrant
- Just because all changes to state done within synchronized pushes and pops doesn't prevent exposing intermediate state.

A Fix!

- Re-entrant locks allows calls to push and pop if use same lock

From within Stack

```
class Stack<E> {
  …
  synchronized E peek(){
     E ans = pop();
     push(ans);
     return ans;
  }
}
```

From outside Stack

```
class C {
  <E> E myPeek(Stack<E> s){
    synchronized (s) {
      E ans = s.pop();
      s.push(ans);
      return ans;
    }
  }
}
```

Beware of Accessing Changing Data

- Even if unsynchronized methods don't change it.

```
class Stack<E> {
  private E[] array = (E[])new Object[SIZE];
  int index = -1;
  boolean isEmpty() { // unsynchronized: wrong?!
    return index==-1;
  }
  synchronized void push(E val) {
   array[++index] = val;
  }
  synchronized E pop() {
   return array[index--];
  }
  E peek() { // unsynchronized: wrong!
    return array[index];
  }
}
```

Providing Safe Access

- For every memory location (e.g., object field) in your program, you must obey at least one of the following:
  - Thread-local: Don't access the location in > 1 thread
  - Immutable: Don't write to the memory location
  - Synchronized: Use synchronization to control access to the location