Lecture 25: Parallelism

CS 62
Fall 2016
Kim Bruce & Peter Mawhorter

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

Sharing is the Key

• Common to have:
  • Different threads access the same resources in an unpredictable order or even at about the same time
    • But program correctness requires that simultaneous access be prevented using synchronization
  • Simultaneous access is rare
    • Makes testing difficult
    • Must be much more disciplined when designing / implementing a concurrent program
    • Will discuss common idioms known to work

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
int nb = b + amount;
if(nb < 0)
  throw new ...
balance = nb;

Thread 2
int nb = b + amount;
if(nb < 0)
  throw new ...
balance = nb;

“Lost withdraw” – unhappy bank
Interleaving is the Problem

- Suppose:
  - Thread T1 calls \texttt{changeBalance(-100)}
  - Thread T2 calls \texttt{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 \texttt{x} and \texttt{y} refer to different accounts, no problem
  - “You cook in your kitchen while I cook in mine”
  - But if \texttt{x} and \texttt{y} alias, possible trouble...

Problems with Account

- Get wrong answers!
- Try to fix by getting balance again, rather than using \texttt{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 \texttt{A} at one time.
- Areas where don't want two threads executing called \textit{critical sections}.
- Programmer needs to decide where, as compiler doesn't know intentions.

Java Solution

- \textit{Re-entrant locks} via \texttt{synchronized} blocks
- Syntax:
  - \texttt{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 \texttt{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 pop0 {
    if(isEmpty())
      throw new StackEmptyException();
    ...
  }

  E peek() {
    E ans = pop0;
    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.
• peek0 has no overall effect on data structure
  • reads rather than writes