Lecture 25: Concurrency & Responsiveness

CS 62
Spring 2011
Kim Bruce & Kevin Coogan

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(isEmpty0)
            throw new StackEmptyException();
        ...
    }
    E peek0 {
        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
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

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

```java
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.

```java
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

Conventional Wisdom

- Whenever possible, don’t share resources
  - Easier to have each thread have its own thread-local copy of a resource than to have one with shared updates
  - This is correct only if threads don’t need to communicate through the resource
    - That is, multiple copies are a correct approach
    - Example: Random objects
  - Note: Since each call-stack is thread-local, never need to synchronize on local variables

- In typical concurrent programs, the vast majority of objects should be thread-local: shared-memory should be rare – minimize it.
Immutable

• Whenever possible, don’t update objects
  • Make new objects instead

• One of key tenets of functional programming
  • Hopefully you study this in 52
  • Generally helpful to avoid side-effects
  • Much more helpful in a concurrent setting

• If a location is only read, never written, no synchronization is necessary!
  • Simultaneous reads are not races and not a problem

• Programmers over-use mutation – minimize it.