Lecture 32: Even 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
- Solved with synchronized blocks
  - or synchronized methods

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.

Maze Program

- When user clicks "solve maze" button, spawns Thread to solve maze.
- Event thread responsible for painting screen and responding to GUI components
  - If response takes more than a few milliseconds, spawn a separate thread to do the work!

Example: Maze-Solver

- Start button ⇒ StartListener object
- Clear button ⇒ ClearAndChooseListener
- Maze choice ⇒ ClearAndChooseListener
  - Stops maze from running! How?
- 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?

- When start with run() instead of start(), 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() should have 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.

- 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;
    }
  }
}
```

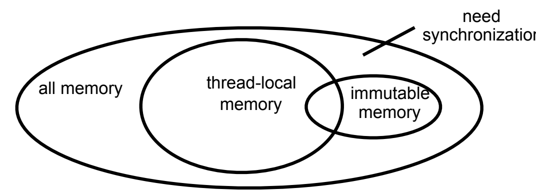
Beware of Accessing Changing Data

- 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



Conventional Wisdom

Thread-Local

- 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
  - 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 (will | did) 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*

Dealing with the Rest

- Guideline: No data races
  - Never allow two threads to read/write or write/write the same location at the same time
- Necessary: In Java or C, a program with a data race is almost always wrong

Worse Than You Think!

```
class C {
  private int x = 0;
  private int y = 0;

  void f() {
    x = 1;
    y = 1;
  }
  void g() {
    int a = y;
    int b = x;
    assert(b >= a);
  }
}
```

- Assertion always true w/ single threaded.
- Looks always true for multithreaded.
  - OK if f not called at all
  - OK after f completes
  - Looks OK if in middle of f
- But have race condition

Memory Reordering

- For performance reasons, compiler and hardware reorder memory operations.
- But, but, ...
  - Compiler/hardware will never perform a memory reordering that affects the result of a single-threaded program
  - The compiler/hardware will never perform a memory reordering that affects the result of a data-race-free multi-threaded program
- So: If no interleaving of your program has a data race, then need not worry: result will be equivalent to some interleaving

A Second Fix

- If label field *volatile,* accesses don't count as data races
- Implementation forces memory consistency
  - though slower!
- Should have used this in CS 51 w/shared variables.
- Really for experts -- better to use locks.

Lock Granularity

- Coarse-grained: Fewer locks, i.e., more objects per lock
  - Example: One lock for entire data structure (e.g., array)
  - Example: One lock for all bank accounts
- Fine-grained: More locks, i.e., fewer objects per lock
  - Example: One lock per data element (e.g., array index)
  - Example: One lock per bank account
- "Coarse-grained vs. fine-grained" is really a continuum.

Trade-Offs

- Coarse-grained advantages
  - Simpler to implement
  - Faster/easier to implement operations that access multiple locations (because all guarded by the same lock)
  - Much easier: ops that modify data-structure shape
- Fine-grained advantages
  - More simultaneous access (performance when coarse-grained would lead to unnecessary blocking)
- Guideline:
  - Start with coarse-grained (simpler) and move to fine-grained (performance) only if contention on the coarser locks becomes an issue. Alas, often leads to bugs.

Critical-section granularity

- A second, orthogonal granularity issue is critical-section size
  - How much work to do while holding lock(s)
- If critical sections run for too long:
  - Performance loss because other threads are blocked
- If critical sections are too short:
  - Bugs because you broke up something where other threads should not be able to see intermediate state
- Guideline: Don't do expensive computations or I/O in critical sections, but also don't introduce race conditions