Concurrency

Rob Miller
Fall 2009

Concurrency

Multiple computations running at the same time
- Concurrency is everywhere, whether we like it or not
- Concurrency is useful, too
  - Splitting up a computation into concurrent pieces is often faster
  - Many apps must handle multiple simultaneous users (e.g. web sites)
  - Even single-user applications are better with concurrency (e.g. Eclipse compiling your Java code in the background while you're editing it)

Network

Memory

Multiple computers in a network
Multiple processors in a computer (or multiple cores in a single chip)

Models for Concurrent Programming

Shared Memory
- Analogy: two processors in a computer, sharing the same physical memory
  - Concurrent modules A and B interact by reading & writing shared state in memory

Message Passing
- Analogy: two computers in a network, communicating by network connections
  - A and B interact by sending messages to each other through a communication channel

Shared Memory Example

Four customers using cash machines simultaneously
- Shared memory model – each cash machine reads and writes the account balance directly

Cash machines
- A deposits $100 to account 1
- B withdraws $100 from account 2
- C deposits $100 to account 1
- D gets balance of account 1

Bank
- Account 1: $50
- Account 2: $200
- Account 3: $50

© Robert Miller 2009
**Race Condition**

**Suppose A and C run at the same time**

- A get balance $50
- C get balance $50
- add deposit $100
- add deposit $100
- write back total $150
- write back total $150

Neither answer is right!

This is an example of a race condition

- A race condition means that the correctness of the program depends on the relative timing of events in concurrent computations
- A race condition means that the correctness of the program depends on the relative timing of events in concurrent computations
- Some interleavings of events may be OK, e.g.: C
- but other interleavings produce wrong answers

Correctness of a concurrent program should not depend on accidents of timing

- Race conditions are nasty bugs — may be rarely observed, hard to reproduce, hard to debug, but may have very serious effects

**Deadlocks**

**Suppose A and B are making simultaneous transfers**

- A transfer between accounts needs to lock both accounts, so that money can’t disappear from the system
- A and B each acquire the lock on the “from” account
- Now each must wait for the other to give up the lock on the “to” account
- Stalemate! A and B are frozen, and the accounts are locked up.

**Deadly embrace**

- Deadlock occurs when concurrent modules are stuck waiting for each other to do something
- A deadlock may involve more than two modules (e.g., a cycle of transfers among N accounts)
- You can have deadlock without using locks – example later

**Synchronization**

**A and C need to synchronize with each other**

- Locks are a common synchronization mechanism
- Holding a lock means “I’m changing this; don’t touch it right now”
- Suppose C acquires the lock first, then A must wait to read and write the balance until C finishes and releases the lock
- Ensures that A and C are synchronized, but B can run independently on a different account (with a different lock)

Correctness of a concurrent program should not depend on accidents of timing

A and C are in a race with C

Some interleavings of events may be OK, e.g.: but other interleavings produce wrong answers

Correctness of a concurrent program should not depend on accidents of timing

- Race conditions are nasty bugs — may be rarely observed, hard to reproduce, hard to debug, but may have very serious effects

**Lock Granularity**

**Preventing the deadlock**

- One solution is to change the locking granularity – e.g., use one lock on the entire bank, instead of a lock on each account

Choosing lock granularity is hard

- If locking is too coarse, then you lose concurrency (e.g., only one cash machine can run at a time)
- If locking is too fine, then you get race conditions and/or deadlocks
- Easy to get this wrong
**Message Passing Example**

- Modules interact by sending messages to each other
  - Incoming requests are placed in a queue to be handled one at a time
  - Sender doesn’t stop working while waiting for an answer to its request; it handles more requests from its own queue
  - Reply eventually comes back as another message

Accounts are now modules, not just memory locations

| Account 1 | bal: $50 | Account 2 | bal: $200 | Account 3 | bal: $50 |
|-----------|----------|-----------|------------|-----------|

**Concurrency Is Hard to Test**

**Poor coverage**
- Recall our notions of coverage
  - All states, all transitions, or all paths through a state machine
- Given two concurrent state machines (with N states and M states), the combined system has N x M states (and many more transitions and paths)
- As concurrency increases, the state space explodes, and achieving sufficient coverage becomes infeasible

**Poor reproducibility**
- Transitions are nondeterministic, depending on relative timing of events that are strongly influenced by the environment
  - Delays can be caused by other running programs, other network traffic, operating system scheduling decisions, variations in processor clock speed, etc.
- Test driver can’t possibly control all these factors
- So even if state coverage were feasible, the test driver can’t reliably reproduce particular paths through the combined state machine

**Use Message Passing in 6.005**

- We’ll focus on message passing, not shared memory
  - Locking strategy for shared-memory paradigm is hard to get right
  - Message-passing paradigm often aligns directly with the real-world workflow of a problem
  - But message passing is less suited to some problems, e.g. a big shared data structure

**Message Passing Has the Same Risks**

- Message passing doesn’t eliminate race conditions
  - Suppose the account state machine supports get-balance and withdraw operations (with corresponding messages)
  - Can Alice and Bob always stay out of the OVERDRAWN state?

Lesson: need to carefully choose the atomic (indivisible) operations of the state machine – withdraw-if-sufficient-funds would be better

**Message-passing can have deadlocks too**
- Particularly when using finite queues that can fill up
Threads

- A thread is a locus of control (i.e., program counter + stack, representing a position in a running program).
- Simulates a fresh processor running the same program in a different place.
- A process always has at least one thread (the main thread).
- Threads can share any memory in the process, as long as they can get a reference to it.
- Threads must set up message passing explicitly (e.g., by creating queues).

Threads in Java

A thread is represented by java.lang.Thread object.
- To define a thread, either override Thread or implement Runnable.
  
  Thread T1 extends Thread
  
  T1 implements Runnable

Thread lifecycle

- Starting arguments can be given to the constructor.
  
  new T1(arg1, ...) new Thread(new T1(arg1, ...))

- Thread is spawned by calling its start() method.
- New thread starts its life by calling its own run() method.
- Thread dies when run() returns or throws an uncaught exception.

Time Slicing

How can I have many concurrent threads with only one or two processors in my computer?

- When there are more threads than processors, concurrency is simulated by time slicing (processor switches between threads).
- Time slicing happens unpredictably and nondeterministically.

Message Passing with Threads

Use a synchronized queue for message-passing between threads.

- interface java.util.concurrent.BlockingQueue is such a queue.
  
  no take transition in EMPTY state, so a thread that tries to take from an empty queue must block (wait) until it can

- ArrayBlockingQueue is a fixed-size queue that uses an array representation.
- LinkedBlockingQueue is a growable queue (no FULL state) using a linked-list representation.
Case Study: Photo Organizer

What happens when the UI displays a large album?

© Robert Miller 2009

Java Swing Is Not Threadsafe

The view hierarchy is a big meatball of shared state

- And there’s no lock protecting it at all
- It’s OK to access user interface objects from the event-handling thread (i.e., in response to input events)
- But the Swing specification forbids touching – reading or writing – any Component objects from a different thread
  - The truth is that Swing's implementation does have one big lock (Component.getTreeLock()) but only some Swing methods use it (e.g., layout)

© Robert Miller 2009

Concurrency in GUIs

Mouse and keyboard events are accumulated in an event queue

- Event loop reads an input event from the queue and dispatches it to listeners on the view hierarchy
- In Java, the event loop runs on a special event-handling thread, started automatically when a user interface object is created

© Robert Miller 2009

Message Passing Via the Event Queue

The event queue is also a message-passing queue

- To access or update Swing objects from a different thread, you can put a message (represented as a Runnable object) on the event queue
  
  SwingUtilities.invokeLater(new Runnable() {
    public void run() {
      content.add(thumbnail); ...
    }
  });
- The event loop handles one of these pseudo-events by calling run()
Thread Safety

**BlockingQueue is itself a shared state machine**

- But it's OK to use from multiple threads because it has an **internal lock** that prevents race conditions within the state machine itself
  - So state transitions are guaranteed to be **atomic**
  - This is done by the Java `synchronized` keyword

- BlockingQueue is therefore **thread-safe** (able to be called by multiple threads safely without threat to its invariants)
- HashSet is not thread-safe; neither is the Swing view hierarchy

HashSet is not thread-safe; neither is the Swing view hierarchy

- Swing thread
- BlockingQueue
- DisplayThread

More Thread-Safe Classes

**Objects that never change state are usually* thread-safe**

- **Immutable** objects never change state
  - e.g., `java.lang.String` is immutable, so threads can share strings as much as they like without fear of race conditions, and without any need for locks or queues

* Caveat: some apparently immutable objects may have hidden state: e.g. memoizing (caching) method return values.

Other Thread-Safe Classes

**Lists, Sets, and Maps can be made thread-safe by a wrapper function**

- `t = Collections.synchronizedSet(s)` returns a thread-safe version of set `s`, with a lock that prevents more than one thread from entering it at a time, forcing the others to block until the lock is free
- So we could imagine synchronizing all our sets:
  ```java
  thumbnails = Collections.synchronizedSet(new HashSet<Thumbnail>());
  ```
  **This doesn't fix all race conditions!**
  - Doesn't help preserve invariants involving more than one data structure
  - ` thumbnails.add(t); content.add(t);`
    - these operations need to be atomic together, to avoid breaking the rep invariant of PreviewPane
    - (that all thumbnails are children of content)

Thread-safe or Not?

**Which of the following are thread-safe? If not, how could you ensure that they are thread-safe?**

- a `findPrimes()` method that remembers all the primes it's ever found in an `ArrayList`
- a method that times itself, using a static variable to store its start time
- a method that takes a String and replaces all the spaces in it with underscores
- a method that takes an integer array and replaces all zeroes in it with ones
- HTMLGenerator from the quote generator case study
Summary

Concurrency
- Multiple computations running simultaneously

Shared-memory & message-passing paradigms
- Shared memory needs a synchronization mechanism, like locks
- Message passing synchronizes on communication channels, like queues

Pitfalls
- Race when correctness of result depends on relative timing of events
- Deadlock when concurrent modules get stuck waiting for each other

Design advice
- Share only immutable objects between threads
- Use blocking queues and SwingUtilities.invokeLater()