Threads

Rob Miller

Fall 2007

© Robert Miller 2007
Review

Last lecture
- shared-memory & message-passing paradigms
- race condition (correctness depends on relative timing of events)
- deadlock (concurrent modules get stuck waiting for each other)
- processes & streams

Today
- threads & queues
- thread safety
- design patterns
Processes (review)

- A **process** is an instance of a running program that is isolated from other processes on the same machine (particularly for resources like memory)
  - Simulates a **fresh computer** to run the program, with fresh memory
- Processes have no shared memory (without special effort) but do have a shared filesystem which can be used for shared-memory paradigm
- Automatically ready for message passing (standard input & output streams)

![Diagram of process relationship]

```
% ls | grep java

ls
grep
```

© Robert Miller 2007
Threads

A **thread** is a locus of control inside a process (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
  
  ```
  T1 extends Thread 
  R1 implements Runnable 
  ```

Thread lifecycle

- Starting arguments can be given to the constructor
  
  ```
  new T1(arg1, ...) 
  new Thread(new R1(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

Contrast with processes

- Defined by a Java program (or an executable in any other language)
- Starting arguments given on the command line, “program arg1 arg2...”
- Spawned by calling Runtime.getRuntime().exec()
- Starts its life by calling main() method (if program is Java)
- Process dies when main() returns or throws an uncaught exception
Message Passing with Threads

Use a synchronized queue for message-passing between threads

- interface java.util.concurrent.BlockingQueue is such a queue

ArrayBlockingQueue is a fixed-size queue implementation that uses an array

LinkedBlockingQueue is arbitrary-size queue (no FULL state) that uses a linked list

© Robert Miller 2007
Quoter Using Threads

- **QuoteApp**
- **AsyncQuoter**
- **QuoteDisplayer**
- **RTFGenerator**

**main thread**

**QuoterThread**

`take() : "open, ask, chg"`

`put("open, ask, chg")`

**Network**

**Yahoo server process**

classes in white are unchanged from QuoteGenerator

© Robert Miller 2007
**Time Slicing**

**How can I have many concurrent processes and 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.

A thread may be paused and resumed at any time.
Processes vs. Threads

Threads are “lightweight”, processes “heavyweight”
- Process uses more operating system resources
- Slower to switch to a different process in timeslicing

Processes can be different programs in other languages
- Threads run the same program

Threads are easier to spawn
- Starting a Java process with “java classname” works only in simple cases
- Jars, classpath cause complications

Processes are easier to distribute to other computers
- Need to use network sockets instead of pipes, but they’re all byte streams

© Robert Miller 2007
Processes vs. Threads

Shared memory paradigm
- Processes share no memory by default
- Threads can share any Java objects

Message passing paradigm
- Processes pass messages on byte streams
- Threads can easily exchange Java objects as messages, not just strings of bytes

But thread programming is riskier
- Objects are so easy for threads to share, that it’s possible to share objects that aren’t designed for concurrent access
- Let’s see an example
Example: Comparing Mail Folders

Are there any important messages in my Spam folder?

- Spam filters aren’t 100% accurate
- But I don’t want to look at my 30,000 spam messages by hand
- Heuristic check: do my spam mail and my good mail have any senders in common?

Intersect the senders of two mail folders

- Specifically, \( \text{senders(Spam)} \cap \text{senders(Archive)} \)

© Robert Miller 2007
General Approach

**Email class**
- gotoFolder(name)
- getMessageCount
- getMessages(index,count)

**Main**
- INIT
- OPENED
- IN FOLDER
- CLOSED

**ICommonSenders**
- examineFolderA(name)
- printSendersInCommon
  - INIT
  - KNOW A
  - KNOW A & B
  - DONE

**javax.mail.Store**

**Network**
- IMAP server

**ICommonSenders interface**

© Robert Miller 2007
Single-threaded Solution

**SenderCollector class**
- **vars**
  - email: Email
  - senders: set of addresses
- **init**
  - email is IN FOLDER
- **op**
  - examineMessages(index, count)
  - **pre**
    - email is IN FOLDER
  - **post**
    - email' is IN FOLDER
    - senders' = senders U
      - \{ m.addr | m in email.getMessages(index, count) \}

**email.sync.CommonSenders class**
- examineA
  - KNOW A
  - **INIT**
- examineB
  - KNOW B
- examineA
  - KNOW A & B
- printSenders
  - InCommon
- DONE

© Robert Miller 2007
Multithreaded Approach

Single-threaded solution examines both folders completely before printing any addresses they have in common

➢ Let’s have one thread for each folder instead
➢ As we accumulate senders from each folder, print each common sender that we find immediately
Multithreaded Try #1

Oops: two SenderThreads can’t share the same Email

- In fact we don’t want to share Email at all – we need to replicate it, so that each thread has its own private IMAP server connection and Email state
- Think carefully about what objects need to be replicated for each thread and what objects need to be shared.
Multithreaded Try #2

email.racy2. CommonSenders

email.racy1. SenderThread

gotoFolder

gotSomeSenders, doneFolder

gotoFolder

gotMessages

Email

SenderCollector

gotMessages, examineMessages

new, examineMessages

new

new

is this correct? can these actions happen only in the WAITING state?
Multithreaded Try #2

racy2.CommonSenders has a race condition, because its operations are called by multiple different threads

- printSendersInCommon() is called by the main thread, continuously trying to intersect the two sender sets
- gotSomeSenders() is called by the sender threads, adding senders to the sender sets
- The sender sets are shared data structures, unprotected by a lock
- In this case, Java throws a ConcurrentModificationException – but we’re not always so lucky
  - We might get quiet corruption of the data structure (e.g. set elements thrown away or duplicated)
  - We might even crash Java completely

- A state machine shouldn’t be shared by multiple threads without synchronization: e.g. a message queue or a lock

© Robert Miller 2007
Using a queue for message passing to CommonSenders

- Specifically a BlockingQueue<Set<String>>
- gotSomeSenders(folderName, senders) is represented on the queue by
  senders U {folderName}
- doneFolder() is encoded on the queue as {}
**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 method attribute synchronized

- 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 CommonSenders, which is why we protected it with the queue

© Robert Miller 2007
Lists, Sets, and Maps can be made thread-safe by a wrapper function

- \( t = \text{Collections.synchronizedSet(s)} \) 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
  sendersFromA = Collections.synchronizedSet(new HashSet<String>());
  ```

This doesn’t fix the race condition!

- The printSendersInCommon() code uses both sets to compute the intersection, so it needs to lock them both for the whole loop:

  ```java
  for (String addr: sendersFromA) {
      if (sendersFromB.contains(addr)) ...
  }
  ```
More Thread-Safe Classes

Objects that never change state are usually* thread-safe

Objects that never change state are called **immutable**
- 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

Immutability will be the primary theme of the next sequence of lectures

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

Thread
- Locus of control in a process

Thread-safe
- Objects that can be shared by multiple threads

Design advice
- Decide which objects need to be replicated, and which need to be shared
- If shared objects aren’t thread-safe, protect them by message queues or locks