Concurrency

Rob Miller
Fall 2007
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)
Models for Concurrent Programming

Shared Memory

- Analogy: two processors in a computer, sharing the same physical memory

![Diagram of concurrent modules A and B interacting through shared memory]

Message Passing

- Analogy: two computers in a network, communicating only by network connections

![Diagram of concurrent modules A and B interacting 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: deposit $256 to account 1
- B: withdraw $65 from account 2
- C: deposit $127 to account 1
- D: get balance of account 1

Bank:
- Account 1: $964
- Account 2: $115
- Account 3: $14

© Robert Miller 2007
Race Condition

Suppose A and C run at the same time

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th></th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>get balance</td>
<td>$964</td>
<td></td>
<td>$964</td>
</tr>
<tr>
<td>add deposit</td>
<td>+ $256</td>
<td></td>
<td>+ $127</td>
</tr>
<tr>
<td>write back total</td>
<td>$1220</td>
<td></td>
<td>$1081</td>
</tr>
</tbody>
</table>

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 is 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 very bad! They may be rarely observed, hard to reproduce, hard to debug, but usually have very serious effects

© Robert Miller 2007
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).

---

**Diagram:**

- **Cash machines:**
  - **A**
  - **B**
  - **C**
  - **D**

- **Bank**:
  - **Account 1:**
    - $964
    - Lock holder: C
  - **Account 2:**
    - $115
    - Lock holder: B
  - **Account 3:**
    - $14
    - (Free)

- Shared memory

© Robert Miller 2007
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.

© Robert Miller 2007
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

**Queue for Account 1**

- **Account 1**
  - bal: $964
- **Account 2**
  - bal: $115
- **Account 3**
  - bal: $14
**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?

Alice

- `get-balance`
  - if balance > $75, withdraw $75

Bob

- `get-balance`
  - if balance > $50, withdraw $50

Account

- bal: $100

- 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 – example later
Concurrenty Is Hard to Test

**Poor coverage**
- Recall our notions of coverage from last lecture: 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 non-deterministic, 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.
We’ll 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
Mechanisms for Message-Passing Concurrency

Between computers in a network: sockets & streams

- A **socket** represents one end of a bidirectional network connection
  - Suppose your laptop opens a connection to `web.mit.edu`
  - Then your laptop has a socket representing its end of the connection, and `web.mit.edu` has a socket on its end

- A **byte stream** is a sequence of bytes (possibly finite, but not necessarily)
  - Each direction of the network connection is a byte stream
  - Byte streams are also used for reading and writing files

```
write("hello")
read() → "hello"
```
Mechanisms for Message-Passing Concurrency

Between processes in a computer

- A **pipe** is a stream between two processes
  - Every process has at least two such streams: standard input & standard output (in Java, System.in and System.out)
  - Unix shell commands construct pipes between concurrent processes:
    ```
    ls | grep java
    ```
- Sockets are sometimes used, too

Between threads in a process

- A **queue** is a data structure that stores objects and returns them in FIFO order
  - In Java: java.util.concurrent.BlockingQueue
  - Graphical user interface toolkits have event queues

© Robert Miller 2007
Processes vs. Threads

Process

- A **process** is an instance of a running program that is isolated from other processes on the same machine (particularly for resources like memory)
- Tries to make the program feel like it has the whole machine to itself – like a **fresh computer** has been created, with fresh memory
- By default, processes have no shared memory (needs special effort)
- Automatically ready for message passing (standard input & output streams)

Thread

- A **thread** is a locus of control inside a running program (i.e. position in the code + stack, representing the current point in the computation)
- Simulates making a **fresh processor** inside the computer, running the same program and sharing the same memory as other threads in process
- Automatically ready for shared memory, because threads share all the memory in the process (needs special effort to get “thread-local” memory that's private to the thread)
- Must set up message passing explicitly (e.g. by creating queues)
Process Example: Quote Generator

One quote lookup at a time was a bottleneck
- The lookups were serialized – one had to finish before next could start

But Yahoo can handle multiple concurrent requests
- Overlapping the work for each lookup will speed up the overall program
Concurrent Quote Generator

Design rules

- Every object belongs to exactly one process (actually Java enforces this)
- Modules in different processes communicate using streams

© Robert Miller 2007
Processes actually have three streams by default

- Standard input, standard output, and **standard error** (System.err in Java)

- **QuoteApp** redirects subprocess’s standard error to its own std error

- This is useful, because it lets you see error messages from the subprocess (like uncaought exceptions) in the Eclipse console view
Deadlock

Not just helpful for debugging

➢ You **must** consume the standard error stream from a subprocess, even if you don’t care what it says
➢ Failing to do this can lead to deadlock
  • Streams have **finite buffers** – when the buffer is full, the writer has to wait (“block”) until a reader starts to empty it
  • So if subprocess fills up its error stream, and main process doesn’t empty it, then the subprocess will eventually block
  • Meanwhile the main process eventually has to wait for replies back from the subprocess – so both processes are waiting for each other
  • **Deadlock!**

Can you deadlock on the input and output streams too?
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 streams or queues

Race conditions
- When correctness of result depends on relative timing of events

Deadlock
- When concurrent modules get stuck waiting for each other

Processes & threads
- Process is like a virtual computer; thread is like a virtual processor

© Robert Miller 2007