Concurrency Continued...

Spring 2013
Why use Concurrent Programming?

1. Natural Application Structure
   - The world is not sequential! Easier to program multiple independent and concurrent activities.

2. Increased application responsiveness
   - Not blocking the entire application due to blocking IO

3. Performance from multiprocessors and multicores
   - Parallel execution

4. Distributed systems
   - Single application on multiple machines
   - Client/server type or peer-to-peer systems
I. Natural Application Structure

Many problems are easier to decompose as multiple concurrent tasks

- Easier to specify
- Reduces interdependencies
- Easier to implement
- Easier to debug and test

Examples?

- Multiple ATM machines for a bank
2. Increased application responsiveness

**Gracefully handle long delays**
- Blocking IO is simpler than non-blocking IO
- More responsive user interfaces

**Examples?**
3. Performance form Multiprocessors/Multicores

**Multiprocessors**

**Powerful Workstations to Supercomputers**

- Solving very large problems
  - Weather simulation for hurricane prediction
  - Finite element analysis of an aircraft body
  - Protein folding for drug design

- Fastest computers on the planet:
  - China: Tianhe-1A
    - 14,366 Intel X5670 2.93Ghz 6C,
    - 7,168 NVIDIA GPU, FT-1000 8C
  - OakRidge National Lab: Jaguar - Cray XT5-HE
    - 18,688 Dual Opteron 6-core 2.6 GHz

**Multicores**

**From supercomputers to desktops to laptops – they are everywhere**

- Paradigm shift on the way
  - Before: every 18 months single processor performance doubles

- In the future: more performance → parallelize
  - Burden on the programmer
  - Expect to see a lot more parallel programs
Amdhal’s Law

Any computation can be analyzed in terms of a portion that must be executed sequentially, $T_s$, and a portion that can be executed in parallel, $T_p$. Then for $n$ processors:

- $T(n) = T_s + T_p/n$
- $T(\infty) = T_s$, thus maximum speedup $(T_s + T_p) / T_s$
Implications of Amdahl’s Law

\[ \text{Utilization} = \frac{1}{p + N^*(1 - p)} \]

\[ \text{Speedup} = \frac{1}{1 - p + \frac{p}{N}} \]

© Robert Miller 2009
Issus with Parallelism

Amdhal’s Law

➢ Any computation can be analyzed in terms of a portion that must be executed sequentially, $T_s$, and a portion that can be executed in parallel, $T_p$. Then for $n$ processors:

- $T(n) = T_s + \frac{T_p}{n}$
- $T(\infty) = T_s$, thus maximum speedup $(T_s + T_p) / T_s$

Load Balancing

The work is distributed among processors so that all processors are kept busy all of the time.

Granularity

➢ The size of the parallel regions between synchronizations or the ratio of computation (useful work) to communication (overhead)
4. Distributed Systems

Solving Google scale problems

- web search
- Mail service (gmail, yahoo mail)
- Video distribution (utube)

What else?

- PS3: Multiplayer minesweeper

© Robert Miller 2009
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 ATMs simultaneously

- Shared memory model – each cash machine reads and writes the account balance directly

ATMs

- **A**: deposit $100 to account 1
- **B**: withdraw $100 from account 2
- **C**: deposit $100 to account 1
- **D**: get balance of account 1

Bank

- account 1: $50
- account 2: $200
- account 3: $50

Shared memory
Race Condition

Suppose A and C run at the same time

<table>
<thead>
<tr>
<th>A</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>get balance</td>
<td>get balance</td>
</tr>
<tr>
<td>+ $100</td>
<td>+ $100</td>
</tr>
<tr>
<td>write back total</td>
<td>write back total</td>
</tr>
<tr>
<td>$150</td>
<td>$150</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 nasty bugs -- may be rarely observed, hard to reproduce, hard to debug, but may have very serious effects
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)
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
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

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
</tr>
<tr>
<td>deposit $100 to account 1</td>
<td>withdraw $100 from account 2</td>
<td>deposit $100 to account 1</td>
<td>get balance of account 1</td>
</tr>
<tr>
<td>get bal</td>
<td>dep $100</td>
<td>wdrw $100</td>
<td></td>
</tr>
<tr>
<td>dep $100</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Account 1 bal: $50</td>
<td>Account 2 bal: $200</td>
<td>Account 3 bal: $50</td>
<td></td>
</tr>
</tbody>
</table>
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

- Particularly when using finite queues that can fill up
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
void TransferAction (Account fracc) throws Exception {
    out.println("Destination Account ID > ");
    String id = in.readLine(); System.out.println(id);
    Account toacc = bank.get(id);
    if (toacc == null)
        return;

    out.println("your balance is $" + fracc.getbal());
    out.println("Transfer amount > ");
    int val = Integer.valueOf(in.readLine());
    if(val < 0) {
        out.println("Can’t withdraw from other accounts");
        return;
    }

    int fcurr = fracc.getbal();
    if (fcurr - val < 0) {
        out.println("Insufficient Balance");
        return;
    }

    int tocurr = toacc.getbal();
    fcurr = fcurr - val;
    tocurr = tocurr + val;
    toacc.setbal(tocurr);
    fracc.setbal(fcurr);
    out.println("New balance is $" + fcurr);
}
Money Transfers!

```java
void TransferAction (Account fracc) throws Exception {
    out.println("Destination Account ID > ");
    String id = in.readLine(); System.out.println(id);
    Account toacc = bnk.get(id);
    if (toacc == null)
        return;

    out.println("your balance is $" + fracc.getbal());

    out.println("Transfer amount > ");
    int val = Integer.valueOf(in.readLine());
    if(val < 0) {
        out.println("Can’t withdraw from other accounts");
        return;
    }

    synchronized(fracc) {
        int frcurr = fracc.getbal();
        if (frcurr - val < 0) {
            out.println("Insufficient Balance");
            return;
        }

        synchronized(toacc) {
            int tocurr = toacc.getbal();

            frcurr = frcurr - val;
tocurr = tocurr + val;

            toacc.setbal(tocurr);
        }

        fracc.setbal(frcurr);
        out.println("New balance is $" + frcurr);
    }
```

© Saman Amarasinghe 2008
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 $N \times M$ 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
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 that uses an array representation

- LinkedBlockingQueue is a growable queue (no FULL state) using a linked-list representation

no take transition in EMPTY state, so a thread that tries to take from an empty queue must block (wait) until it can
Case Study: A GUI Application

What happens when the UI displays a large album?
Concurrencity 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
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:

  ```java
  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
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.
Thread safe $\neq$ no race conditions!

Atomicity may require across multiple objects

Now in Seat 1
If Seat 2 is empty
Remove from seat 1
Get seat 2

Traveler A
Check avail
Remove me
Add me

Seat 1
Seat 2
Traveler A
Traveler A

Shared memory
Thread safe != no race conditions!

Atomicity may require across multiple objects

But...

```
Now in Seat 1
If Seat 2 is empty
Remove from seat 1
Get seat 2
```

```
Traveler A
Now in Seat 1
If Seat 2 is empty
Remove me
Add me
```

```
Traveler B
Check avail
Add me
```

Seat 1
Traveler A

Seat 2
Traveler B

Shared memory
Thread safe != no race conditions!

Atomicity may require across multiple objects
Create a single queue

Traveler A
Now in Seat 1
Ask to charge to Seat 2 if possible
Add to 2
If successful Remove from 1

Traveler B
If Seat 2 is empty

Seat 1
Traveler A

Seat 2
Traveler A

Shared memory
Now in Seat 1
If Seat 2 is empty
Get it
If successful
Remove from 2

Thread safe != no race conditions!
Atomicity may require across multiple objects
A better solution (transactional)

Shared memory
Thread safe != no race conditions!

Atomicity may require across multiple objects

Traveler A
Now in Seat 1
If Seat 2 is empty
Check avail
Add me
Remove from 2

Traveler B
If Seat 2 is empty
Add me

Seat 1
Traveler A

Seat 2
Traveler B

Shared memory
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.
Java Swing Is Not Threadsafe

**Threadsafe:** A program portion or routine that can be called from multiple programming threads without unwanted interaction between the threads.

**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)
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 boolean array and complements all the bits
- A state machine corresponding to a music player
Client Server Pattern

Just what it’s name implies
Sockets

a network interface is identified by an IP address
➤ (or a hostname, which translates into an IP address)
➤ examples: 127.0.0.1, localhost; web.mit.edu

an interface has 65536 ports
➤ numbered from 0 to 65535

a server process binds to a port (the listening port)
➤ clients have to know which number it’s binding to.
➤ Some numbers are well-known (port 80 is the standard web server port, port 22 is the SSH port, port 25 is the standard SMTP email server port).
➤ When it’s not a standard port for the kind of server, you just treat it as part of the address
Sockets

the listening port is just used to accept incoming client connections.

➢ Once the connection is accepted, the server creates a new socket for the actual connection, with a fresh port number (unrelated to the listening port number).
➢ Both the client and server sockets have port numbers.
Conclusion

Concurrency and Parallelism are important concepts in Computer Science

Concurrency can simplify programming
- However it can be very hard to understand and debug concurrent programs

Parallelism is critical for high performance
- From Supercomputers in national labs to Multicores and GPUs on your desktop