What is concurrency

Concurrency is not just about multicores
- it is about having many things taking place at once

What is a concurrent program?
- It's a program where multiple streams of instructions are executing at once

You don’t need a multi-core to have a concurrent program

© Robert Miller 2009
Concurrenty and Parallelism

Concurrenty does not require multiple cores (although those help)

Interleaved Concurrency
- Logically simultaneous processing
- Interleaved execution on a single processor

Parallelism
- Physically simultaneous processing
- Requires a multiprocessors or a multicore system
Many ways to achieve concurrency

**Multiprogramming**
- Threads multiplex their executions on a single processor.

**Multiprocessing**
- Threads multiplex their executions on a multiprocessor or a multicore system

**Distributed Processing**
- Processes multiplex their executions on several different machines
Why use Concurrent Programming?

Natural Application Structure

- The world is not sequential!
  - Web sites must handle multiple simultaneous users
  - Graphical user interfaces often require background work (e.g. Eclipse compiling your Java code while you’re editing it)
- 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
Processes Vs. Threads

**Process**

Tries to make the program feel like it has the whole machine to itself; like a fresh computer has been created, with fresh memory

It is an instance of a running program that is isolated from other processes on the same machine

**Thread**

Simulates making a fresh processor inside the computer, running the same program and sharing the same memory as other threads in process

It is a locus of control inside a running program (i.e. position in the code + stack, representing the current point in the computation)
Concurrency in Java

Java has a predefined class `java.lang.Thread` which provides the mechanism by which threads are created

```java
public class MyThread extends Thread {
    public void run() {
    } // method body
}
```

However to avoid all threads having to be subtypes of `Thread`, Java also provides a standard interface

```java
public interface Runnable {
    public void run();
}
```

Hence, any class which wishes to express concurrent execution must implement this interface and provide the `run` method

Threads do not begin their execution until the `start` method in the `Thread` class is called
**Safety: Example**

**“The too much milk problem”**

<table>
<thead>
<tr>
<th>time</th>
<th>You</th>
<th>Your Roommate</th>
</tr>
</thead>
<tbody>
<tr>
<td>3:00</td>
<td>Arrive home</td>
<td>Arrive home</td>
</tr>
<tr>
<td>3:05</td>
<td>Look in fridge, no milk</td>
<td>Look in fridge, no milk</td>
</tr>
<tr>
<td>3:10</td>
<td>Leave for grocery</td>
<td>Leave for grocery</td>
</tr>
<tr>
<td>3:15</td>
<td>Arrive at grocery</td>
<td></td>
</tr>
<tr>
<td>3:20</td>
<td></td>
<td>Buy Milk</td>
</tr>
<tr>
<td>3:25</td>
<td>Buy milk</td>
<td></td>
</tr>
<tr>
<td>3:35</td>
<td>Arrive home, put milk in fridge</td>
<td>Arrive home, put up milk</td>
</tr>
<tr>
<td>3:45</td>
<td></td>
<td>Buy Milk</td>
</tr>
<tr>
<td>3:50</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3:50</td>
<td></td>
<td>Oh no!</td>
</tr>
</tbody>
</table>

**Model of need to synchronize activities**

Courtesy of Emery Berger @ UMASS
The solution

Went to get Milk

© Robert Miller 2009
Synchronization

All the interleavings of the threads are **NOT** acceptable correct programs.

Java provides **synchronization** mechanism to restrict the interleavings

**Synchronization serves two purposes:**

- **Ensure safety** for shared updates
  - Avoid **race conditions**
- **Coordinate** actions of threads
  - Parallel computation
  - Event notification
Multiple threads access shared resource simultaneously

Safe only if:

- All accesses have no effect on resource,
  - e.g., reading a variable,

- or

- All accesses idempotent
  - E.g., \( y = \text{sign}(a), a = a*2; \)

- or

- Only one access at a time:
  
  mutual exclusion
Mutual Exclusion

Prevent more than one thread from accessing **critical section** at a given time

- Once a thread is in the critical section, no other thread can enter that critical section until the first thread has left the critical section.
- No interleavings of threads within the critical section
- **Serializes** access to section

```java
synchronized int getbal() {
    return balance;
}

synchronized void post(int v) {
    balance = balance + v;
}
```
Atomicity

Synchronized methods execute the body as an atomic unit
May need to execute a code region as the atomic unit
Block Synchronization is a mechanism where a region of code can be labeled as synchronized
The *synchronized* keyword takes as a parameter an object whose lock the system needs to obtain before it can continue
Example:

```java
synchronized (acc) {
    if (acc.getbal() + val > 0)
        acc.post(val);
    else
        throw new Exception();
    out.print("your balance is "+acc.getbal());
}
```
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
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
Avoiding Deadlock

Cycle in locking graph = deadlock

Standard solution:
canonical order for locks

- Acquire in increasing order
- Release in decreasing order

Ensures deadlock-freedom, but not always easy to do
public class Account {
    String id;
    String password;
    int balance;
    static int count;

    Account(String id, String password, String balance) {
        this.id = id;
        this.password = password;
        this.balance = balance;
    }

    public boolean transfer(Account from, Account to, int val) {
        synchronized(from) {
            synchronized(to) {
                if (from.getbal() > val)
                    from.post(-val);
                else
                    throw new Exception();
                to.post(val);
            }
        }
    }

    ...
public class Account {
    String id;
    String password;
    int balance;
    static int count;
    public int rank;

    Account(String id,
            String password,
            String balance) {
        this.id = id;
        this.password = password;
        this.balance = balance;
        rank = count++;
    }

    public boolean transfer(Account from, Account to, int val) {
        Account first = (from.rank > to.rank)?from:to;
        Account second = (from.rank > to.rank)?to:from;
        synchronized(first) {
            synchronized(second) {
                if (from.getbal() > val)
                    from.post(-val);
                else
                    throw new Exception();
            }
            to.post(val);
        }
    }

    ...
Models for Concurrent Programming

Shared Memory

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

![Diagram of Shared Memory](image)

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

![Diagram of Message Passing](image)

A and B interact by sending messages to each other through a communication channel.
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>
</tbody>
</table>

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