L10: Concurrency

Today

- Processes & threads
- Time slicing
- Message passing & shared memory
- Race conditions
- Deadlocks

Concurrency

- Concurrency is everywhere, whether we like it or not
  - Multiple computers in a network

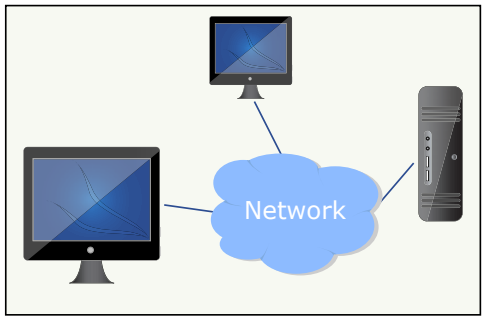
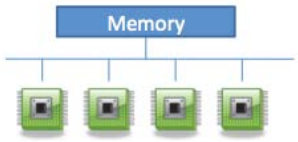


Image by MIT OpenCourseWare.

  - Multiple applications running on one computer
  - Multiple processors in a computer (today, often multiple **processor cores** on a single chip)

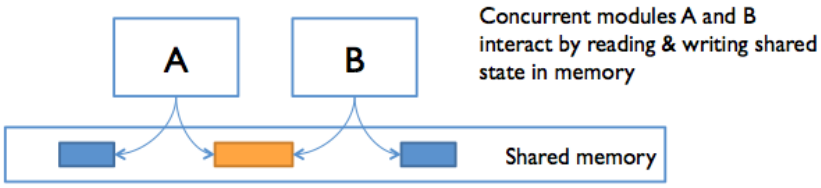


- Concurrency is essential in modern programming
  - Web sites must handle multiple simultaneous users
  - Mobile apps need to do some of their processing on servers ("in the cloud")
  - Graphical user interfaces often require background work (e.g. Eclipse compiling your Java code while you're editing it)
  - Processor clock speeds are no longer increasing – instead we're getting more cores on each new generation of chips. So in the future, we'll have to split up a computation into concurrent pieces in order to get it to run faster.

Two Models for Concurrent Programming

Shared Memory

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



Other ways to think about this:

A and B might be two *threads* in a Java program (we'll explain what a thread is later)

A and B might be two programs running on a computer, sharing a common filesystem

Message Passing

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

A and B might be a web browser and a web server – A opens a connection to B, asks for a web page, and B sends the web page data back to A.

A and B might be an IM client and server.

A and B might be two programs running on the same computer whose input and output have been connected by a pipe (like ls | grep).

Threads & Processes

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)

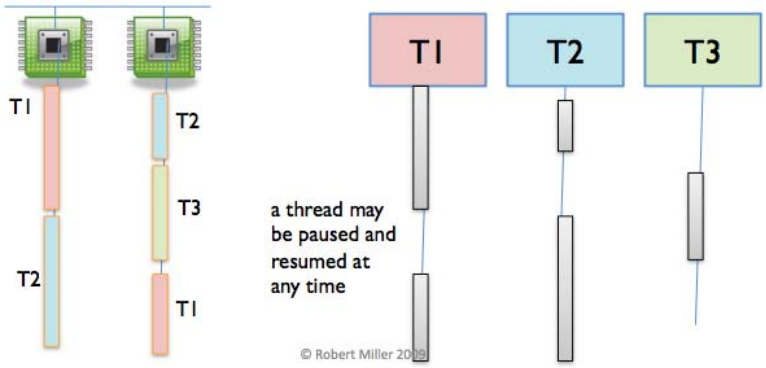
Thread

- 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)

Time-slicing

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

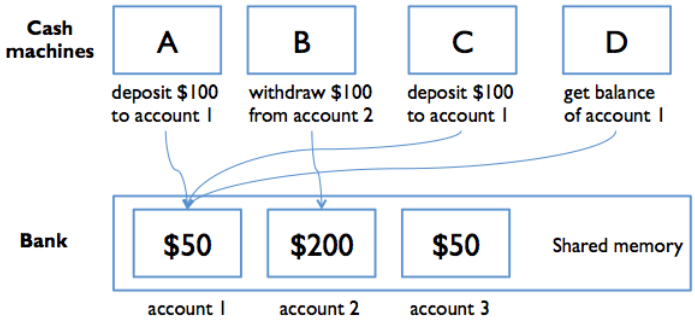
- When there are more threads than processors, concurrency is simulated by **time slicing** (processor switches between threads)
- Time slicing happens unpredictably and nondeterministically



A Shared Memory Example

Four customers using cash machines simultaneously

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




```
// all the cash machines share a single bank account
private static int balance = 0;


private static void deposit() {
    balance = balance + 1;
}
private static void withdraw() {
    balance = balance - 1;
}

// each ATM does a bunch of transactions that
// modify balance, but leave it unchanged afterward
private static void cashMachine() {
    for (int i = 0; i < TRANSACTIONS_PER_MACHINE; ++i) {
```

```
            deposit(); // put a dollar in
            withdraw(); // take it back out
        }
    }
```

Throughout the day, each cash machine in our network is running cashMachine(), processing transactions. In this simple example, every transaction is just a one dollar deposit followed by a one-dollar withdrawal, so it should leave the balance in the account unchanged. So at the end of the day, regardless of how many cash machines were running, or how many transactions we processed, we expect the account balance to still be 0.

But it's not. If more than one cashMachine() call is running at the same time – say, on separate processors in the same computer – then balance may *not* be zero at the end of the day. Why not?

Here's one thing that can happen. Suppose two cash machines, A and B, are both working on a deposit at the same time. Here's how the deposit() step typically break down into low-level processor instructions:

get balance	0
add 1	1
write back the result	1

When A and B are running concurrently, these low-level instructions **interleave** with each other (some might even be simultaneous in some sense, but let's just worry about interleaving for now):

A get balance	0		
A add 1	1		
A write back the result	1		
		B get balance	1
		B add 1	2
		B write back the result	2

This interleaving is fine – we end up with balance 2, so both A and B successfully put in a dollar. But what if the interleaving looked like this:

A get balance	0		
		B get balance	0
A add 1	1		
		B add 1	1
A write back the result	1		
		B write back the result	1

The balance is now 1 – A's dollar was lost! A and B both read the balance at the same time, computed separate final balances, and then raced to store back the new balance – which failed to take the other's deposit into account.

This is an example of a race condition

- A **race condition** means that the correctness of the program (the satisfaction of postconditions and invariants) depends on the relative timing of events in concurrent computations

- "A is in a race with B"

➢ Some interleavings of events may be OK (in the sense that they are consistent with what a single, nonconcurrent process would produce), but other interleavings produce wrong answers – violating postconditions or invariants

Tweaking the Code Won't Help

All these versions of the code exhibit the same race condition:

```
// version 1
private static void deposit() {
    balance = balance + 1;
}
private static void withdraw() {
    balance = balance - 1;
}

// version 2
private static void deposit() {
    balance += 1;
}
private static void withdraw() {
    balance -= 1;
}

// version 3
private static void deposit() {
    ++balance;
}
private static void withdraw() {
    --balance;
}
```

You can't tell just from looking at Java code how the processor is going to execute it. You can't tell what the indivisible operations – the *atomic* operations – will be. It isn't atomic just because it's one line of Java. It doesn't touch balance only once just because the balance identifier occurs only once in the line. The Java compiler, and in fact the processor itself, makes no commitments about what low-level operations it will generate from your code. In fact, a typical Java compiler produces exactly the same code for all three of these versions!

The key lesson is that **you can't tell by looking at an expression whether it will be safe from race conditions.**

Reordering

It's even worse than that, in fact. The race condition on the bank account balance can be explained in terms of different interleavings of sequential operations on different processors. But in fact, when you're using multiple variables and multiple processors, you can't even count on changes to those variables appearing in the same order.

Here's an example:

```
private boolean ready = false;
private int answer = 0;

// computeAnswer runs in one thread
```

```
private void computeAnswer() {
    answer = 42;
    ready = true;
}

// useAnswer runs in a different thread
private void useAnswer() {
    while (!ready) {
        Thread.yield();
    }
    System.out.println(answer);
}
```

We have two methods that are being run in different threads. computeAnswer does a long calculation, finally coming up with the answer 42, which it puts in the answer variable. Then it sets the ready variable to true, in order to signal to the method running in the other thread, useAnswer, that the answer is ready for it to use. Looking at the code, answer is set *before* ready is set, so once useAnswer sees ready as true, then it seems reasonable that it can assume that the answer will be 42, right? Not so.

The problem is that modern compilers and processors do a lot of things to make the code fast. One of those things is making temporary copies of variables like *answer* and *ready* in faster storage (registers or caches on a processor), and working with them temporarily before eventually storing them back to their official location in memory. The storeback may occur in a *different order* than the variables were manipulated in your code. Here's what might be going on under the covers (but expressed in Java syntax to make it clear). The processor is effectively creating two temporary variables, tmpr and tmpa, to manipulate the fields ready and answer:

```
private void computeAnswer() {
    boolean tmpr = ready;
    int tmpa = answer;

    tmpa = 42;
    tmpr = true;

    ready = tmpr;
                        // <== what happens if useAnswer() interleaves here?
                        //     ready is set, but answer isn't
    answer = tmpa;
}
```

Synchronization

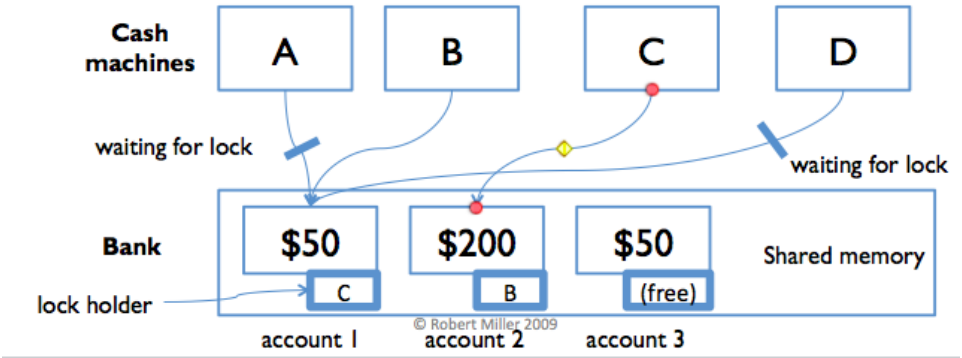
The correctness of a concurrent program should not depend on accidents of timing

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

To avoid race conditions, concurrent modules that share memory 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 B acquires the lock first; then A must wait to read and write the balance until B finishes and releases the lock

- Ensures that A and B are synchronized, but another cash machine C would be able to run independently on a different account (with a different lock)



- Acquiring or releasing a lock also tells the compiler and processor that you're using shared memory concurrently, so that registers and caches will be flushed out to the shared storage (which solves the reordering problem)

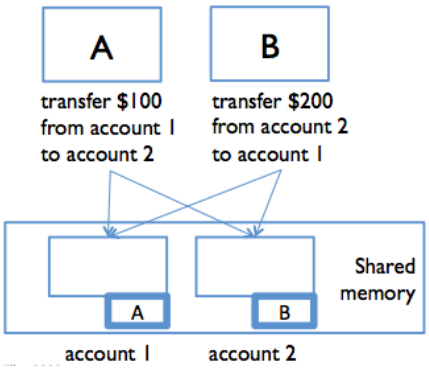
Deadlock

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

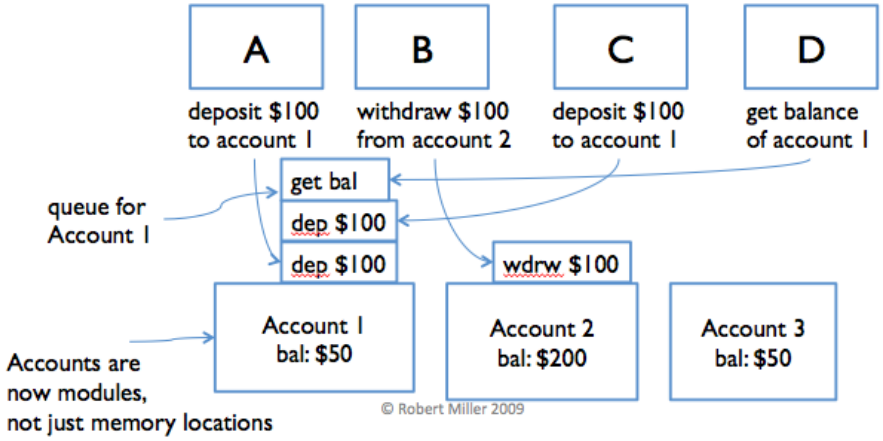


Message Passing Example

Now let's look at the message-passing approach to our bank account example.

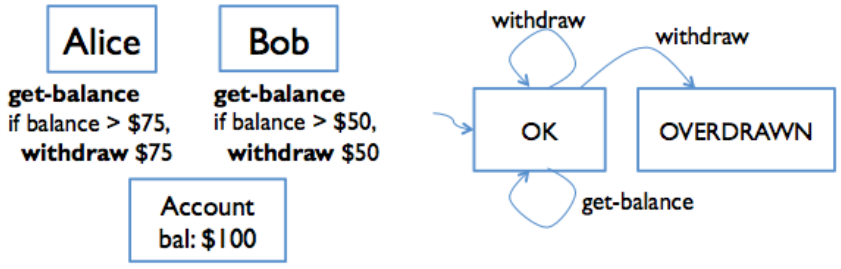
Modules interact by sending messages to each other

- 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



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 and Debug

If we haven't persuaded you that concurrency is tricky, here's the worst of it. It's very hard to discover these kinds of concurrency bugs (race conditions and deadlocks) using testing. And even once a test has found a bug, it may be very hard to localize it to the part of the program causing it.

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

heisenbugs

- a "heisenbug" is nondeterministic, hard to reproduce (as opposed to a "bohrbug", which shows up repeatedly whenever you look at it – almost all bugs in sequential programming are bohrbugs)
- a heisenbug may even disappear when you try to look at it with println or debugger!

```
private static void cashMachine() {
    for (int i = 0; i < TRANSACTIONS_PER_MACHINE; ++i) {
        deposit(); // put a dollar in
```

```
                    withdraw(); // take it back out
                    System.out.println(balance);  // makes the bug
unreproducible!
                }
            }
```

one approach

- build a lightweight event log (circular buffer)
- log events during execution of program as it runs at speed
- when you detect the error, stop program and examine logs

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

Processes & threads

- Process is like a virtual computer; thread is like a virtual processor

Race conditions

- When correctness of result (postconditions and invariants) depends on relative timing of events

Deadlock

- When concurrent modules get stuck waiting for each other

6.005 Elements of Software Construction

Fall 2011