Logged in: ange

Questions about the reading?
Search & ask on Piazza (https://piazza.com/class#spring2017/6031)

Reading 19: Concurrency

Software in 6.031

Safe from bugs	Easy to understand	Ready for change
Correct today and correct in the unknown future.	Communicating clearly with future programmers, including future you.	Designed to accommodate change without rewriting.

Objectives

- The message passing and shared memory models of concurrency
- Concurrent processes and threads, and time slicing
- The danger of race conditions

Concurrency

Concurrency means multiple computations are happening at the same time. Concurrency is everywhere in modern programming, whether we like it or not:

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

In fact, concurrency is essential in modern programming:

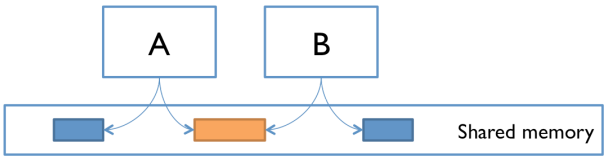
- Web sites must handle multiple simultaneous users.
- Mobile apps need to do some of their processing on servers ("in the cloud").
- Graphical user interfaces almost always require background work that does not interrupt the user. For example, Eclipse compiles your Java code while you're still editing it.

Being able to program with concurrency will still be important in the future. Processor clock speeds are no longer increasing. Instead, we're getting more cores with each new generation of chips. So in the future, in order to get a computation to run faster, we'll have to split up a computation into concurrent pieces.

Two models for concurrent programming

There are two common models for concurrent programming: *shared memory* and *message passing*.

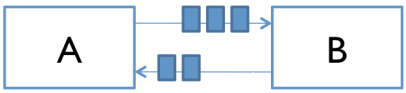
Shared memory. In the shared memory model of concurrency, concurrent modules interact by reading and writing shared objects in memory.



Examples of the shared-memory model:

- A and B might be two processors (or processor cores) in the same computer, sharing the same physical memory.
- A and B might be two programs running on the same computer, sharing a common filesystem with files they can read and write.
- A and B might be two threads in the same Java program (we'll explain what a thread is below), sharing the same Java objects.

Message passing. In the message-passing model, concurrent modules interact by sending messages to each other through a communication channel. Modules send off messages, and incoming messages to each module are queued up for handling. Examples include:



- A and B might be 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 and asks for a web page, and B sends the web page data back to A.
- A and B might be an instant messaging 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` typed into a command prompt.

Processes, threads, time-slicing

The message-passing and shared-memory models are about how concurrent modules communicate. The concurrent modules themselves come in two different kinds: processes and threads.

Process. A process is an instance of a running program that is *isolated* from other processes on the same machine. In particular, it has its own private section of the machine's memory.

The process abstraction is a *virtual computer*. It makes the program feel like it has the entire machine to itself – like a fresh computer has been created, with fresh memory, just to run that program.

Just like computers connected across a network, processes normally share no memory between them. A process can't access another process's memory or objects at all. Sharing memory between processes is *possible* on most operating systems, but it needs special effort. By contrast, a new process is automatically ready for message passing, because it is created with standard input & output streams, which are the `System.out` and `System.in` streams you've used in Java.

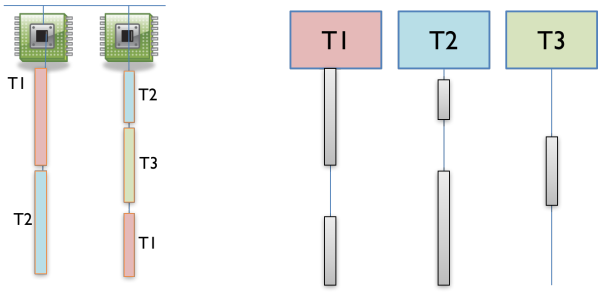
Whenever you start a Java program – indeed, whenever you start any program on your computer – it starts a fresh process to contain the running program.

Thread. A thread is a locus of control inside a running program. Think of it as a place in the program that is being run, plus the stack of method calls that led to that place (so the thread can go back up the stack when it reaches `return` statements).

Just as a process represents a virtual computer, the thread abstraction represents a *virtual processor*. Making a new thread simulates making a fresh processor inside the virtual computer represented by the process. This new virtual processor runs the same program and shares the same memory as other threads in the process.

Threads are automatically ready for shared memory, because threads share all the memory in the process. It takes special effort to get "thread-local" memory that's private to a single thread. It's also necessary to set up message-passing explicitly, by creating and using queue data structures. We'll talk about how to do that in a future reading.

Whenever you run a Java program, the program starts with one thread, which calls `main()` as its first step. This thread is referred to as the *main thread*.



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**, which means that the processor switches between threads. The figure on the right shows how three threads T1, T2, and T3 might be time-sliced on a machine that has only two actual processors. In the figure, time proceeds downward, so at first one processor is running thread T1 and the other is running thread T2, and then the second processor switches to run thread T3. Thread T2 simply pauses, until its next time slice on the same processor or another processor.

On most systems, time slicing happens unpredictably and nondeterministically, meaning that a thread may be paused or resumed at any time.

Starting a thread in Java

You can start new threads by making an instance of `Thread` and telling it to `start()`. You provide code for the new thread to run by creating a class implementing `Runnable`. The first thing the new thread will do is call the `run()` method in this class. For example:

```
// ... in the main method:
new Thread(new HelloRunnable()).start();

// elsewhere in the code
public class HelloRunnable implements Runnable {
    public void run() {
        System.out.println("Hello from a thread!");
    }
}
```

But a very common idiom is starting a thread with an anonymous `Runnable`, which eliminates the need to name the `HelloRunnable` class at all:

```
new Thread(new Runnable() {
    public void run() {
        System.out.println("Hello from a thread!");
    }
}).start();
```

The next two sections discuss the idea of anonymous classes, because they're widely used in Java beyond just threads.

Anonymous classes

Usually when we implement an interface, we do so by declaring a class. For example, given the interface `Comparator` in the Java API:

```
/** A comparison function that imposes a total ordering on some objects.
 *  ... */
public interface Comparator<T> {
    /** Compares its two arguments for order.
     *  ...
     *  @return a negative integer, zero, or a positive integer if the first
     *          argument is less than, equal to, or greater than the second */
    public int compare(T o1, T o2);
}
```

We might declare:

```
/** Orders Strings by length (shorter first) and then lexicographically. */
public class StringLengthComparator implements Comparator<String> {
    @Override public int compare(String s1, String s2) {
        if (s1.length() == s2.length()) {
            return s1.compareTo(s2);
        }
        return s1.length() - s2.length();
    }
}
```

One purpose of `Comparator` is for sorting. A `SortedSet` keeps its items in a total order.

Without a `Comparator`, the `SortedSet` implementation uses the `compareTo` method provided by the objects in the set:

```
SortedSet<String> strings = new TreeSet<>();
strings.addAll(Arrays.asList("yolanda", "zach", "alice", "bob"));
// strings is { "alice", "bob", "yolanda", "zach" }
```

With a `Comparator`:

```
// uses StringLengthComparator declared above
Comparator<String> compareByLength = new StringLengthComparator();
SortedSet<String> strings = new TreeSet<>(compareByLength);
strings.addAll(Arrays.asList("yolanda", "zach", "alice", "bob"));
// strings is { "bob", "zach", "alice", "yolanda" }
```

If we only intend to use this comparator in this one place, we already know how to eliminate the variable:

```
// uses StringLengthComparator declared above
SortedSet<String> strings = new TreeSet<>(new StringLengthComparator());
strings.addAll(Arrays.asList("yolanda", "zach", "alice", "bob"));
// strings is { "bob", "zach", "alice", "yolanda" }
```

An **anonymous class** declares an unnamed class that implements an interface and immediately creates the one and only instance of that class. Compare to the code above:

```
// no StringLengthComparator class!
SortedSet<String> strings = new TreeSet<>(new Comparator<String>() {
    @Override public int compare(String s1, String s2) {
        if (s1.length() == s2.length()) {
            return s1.compareTo(s2);
        }
        return s1.length() - s2.length();
    }
});
strings.addAll(Arrays.asList("yolanda", "zach", "alice", "bob"));
// strings is { "bob", "zach", "alice", "yolanda" }
```

Advantages of anonymous class over named class:

- If we're only using the comparator in this one piece of code, we've reduced its scope by using an anonymous class. With a named class, any other code could start using and depending on `StringLengthComparator`.
- A reader no longer has to search elsewhere for the details of the comparator, everything is right here.

Disadvantages:

- If we need the same comparator more than once, we might be tempted to copy-and-paste the anonymous class. The named class is DRY.
- If the implementation of the anonymous class is long, it interrupts the surrounding code, making it harder to understand. The named class is separated out as a modular piece.

So anonymous classes are good for short one-off implementations of a method.

READING EXERCISES

Comparator

Using an anonymous `Runnable` to start a thread

The `Runnables` we use to create new threads often meet these criteria perfectly.

Here's the example we used above:

```
new Thread(new Runnable() {
    public void run() {
        System.out.println("Hello from a thread!");
    }
}).start();
```

Rather than (1) declare a class that implements `Runnable` where the `run` method calls `System.out.println`, (2) create an instance of that class, and (3) pass that instance to the `Thread` constructor, we do all three steps in one go with an anonymous `Runnable`.

If you're feeling clever, you can go one step further with Java's lambda expressions:

```
new Thread(() -> System.out.println("Hello from a thread!")).start();
```

Whether that's more or less *easy to understand* is up for debate. `Runnable` and `run` never appear at all, so you certainly have to do more research to understand this construction the first time you come across it.

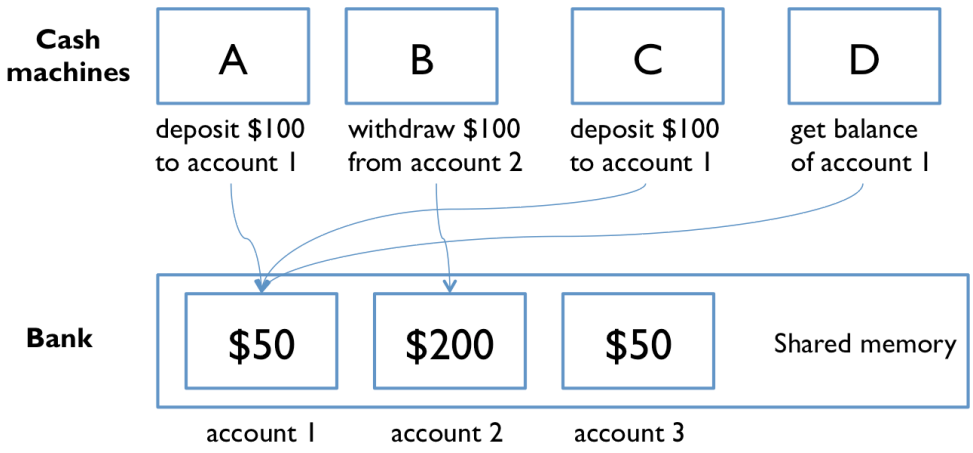
READING EXERCISES

- Processes and threads 1
- Processes and threads 2
- Processes and threads 3

Shared memory example

Let's look at an example of a shared memory system. The point of this example is to show that concurrent programming is hard, because it can have subtle bugs.

Imagine that a bank has cash machines that use a shared memory model, so all the cash machines can read and write the same account objects in memory.



To illustrate what can go wrong, let's simplify the bank down to a single account, with a dollar balance stored in the `balance` variable, and two operations `deposit` and `withdraw` that simply add or remove a dollar:

```
// suppose 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;
}
```

Customers use the cash machines to do transactions like this:

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

Each transaction is just a one dollar deposit followed by a one-dollar withdrawal, so it should leave the balance in the account unchanged. Throughout the day, each cash machine in our network is processing a sequence of deposit/withdraw transactions:

```
// 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
    }
}
```

So at the end of the day, regardless of how many cash machines were running, or how many transactions we processed, we should expect the account balance to still be 0.

But if we run this code, we discover frequently that the balance at the end of the day is *not* 0. 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?

Interleaving

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 breaks down into low-level processor instructions:

get balance (balance=0)
add 1
write back the result (balance=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	B

A get balance (balance=0)	
A add 1	
A write back the result (balance=1)	
	B get balance (balance=1)
	B add 1
	B write back the result (balance=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	**B**
A get balance (balance=0)	
	B get balance (balance=0)
A add 1	
	B add 1
A write back the result (balance=1)	
	B write back the result (balance=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.

Race condition

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 and B. When this happens, we say "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 bank-account 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 modern 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.

Read: **Thread Interference** (just 1 page)

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. Note that it uses a loop that continuously checks for a concurrent condition; this is called busy waiting and it is not a good pattern. In this case, the code is also broken:

```
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();
    }
    if (answer == 0) throw new RuntimeException("answer wasn't ready!");
}
```

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;
}
```

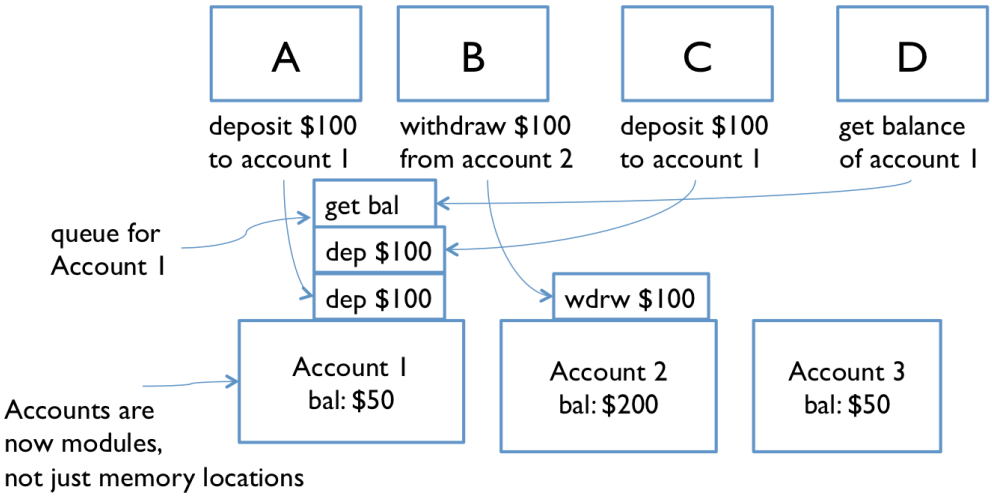
READING EXERCISES

- Interleaving 1
- Interleaving 2
- Race conditions 1
- Race conditions 2

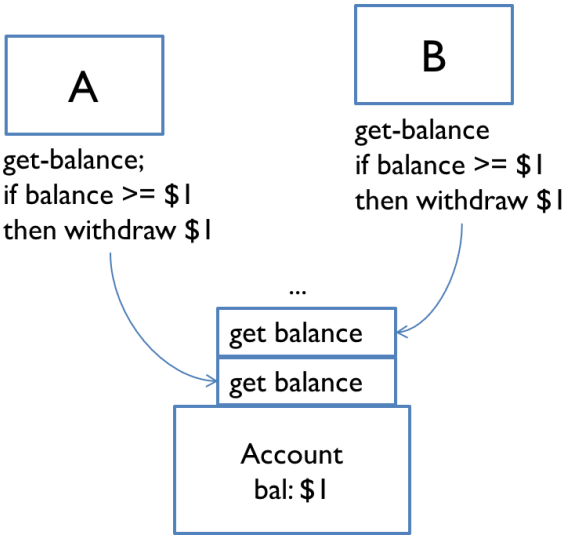
Message passing example

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

Now not only are the cash machine modules, but the accounts are modules, too. Modules interact by sending messages to each other. Incoming requests are placed in a queue to be handled one at a time. The sender doesn't stop working while waiting for an answer to its request. It handles more requests from its own queue. The reply to its request eventually comes back as another message.



Unfortunately, message passing doesn't eliminate the possibility of race conditions. Suppose each account supports `get-balance` and `withdraw` operations, with corresponding messages. Two users, at cash machines A and B, are both trying to withdraw a dollar from the same account. They check the balance first to make sure they never withdraw more than the account holds, because overdrafts trigger big bank penalties:




```
get-balance
if balance >= 1 then withdraw 1
```

The problem is again interleaving, but this time interleaving of the *messages* sent to the bank account, rather than the *instructions* executed by A and B. If the account starts with a dollar in it, then what interleaving of messages will fool A and B into thinking they can both withdraw a dollar, thereby overdrawing the account?

One lesson here is that you need to carefully choose the operations of a message-passing model. `withdraw-if-sufficient-funds` would be a better operation than just `withdraw`.

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 race conditions 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.

Concurrency bugs exhibit very poor reproducibility. It's hard to make them happen the same way twice. Interleaving of instructions or messages depends on the 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. Each time you run a program containing a race condition, you may get different behavior.

These kinds of bugs are *heisenbugs*, which are nondeterministic and 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`! The reason is that printing and debugging are so much slower than other operations, often 100-1000x slower, that they dramatically change the timing of operations, and the interleaving. So inserting a simple print statement into the `cashMachine()`:

```
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 disappear!
    }
}
```

…and suddenly the balance is always 0, as desired, and the bug appears to disappear. But it's only masked, not truly fixed. A change in timing somewhere else in the program may suddenly make the bug come back.

Concurrency is hard to get right. Part of the point of this reading is to scare you a bit. Over the next several readings, we'll see principled ways to design concurrent programs so that they are safer from these kinds of bugs.

READING EXERCISES

Testing concurrency

Summary

- Concurrency: multiple computations running simultaneously
- Shared-memory & message-passing paradigms
- 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 the relative timing of events

These ideas connect to our three key properties of good software mostly in bad ways. Concurrency is necessary but it causes serious problems for correctness. We'll work on fixing those problems in the next few readings.

- **Safe from bugs.** Concurrency bugs are some of the hardest bugs to find and fix, and require careful design to avoid.
- **Easy to understand.** Predicting how concurrent code might interleave with other concurrent code is very hard for programmers to do. It's best to design your code in such a way that programmers don't have to think about interleaving at all.
- **Ready for change.** Not particularly relevant here.

MIT EECS