L16: Message Passing

Today

- Client/server
- Network sockets
- Blocking
- Wire protocols
- Deadlocks

Required reading (from the Java Tutorial)

Make sure that you read and understand these parts of the Java API:

- I/O Streams, up to and including I/O from the Command Line (8 pages)
  [http://docs.oracle.com/javase/tutorial/essential/io/streams.html](http://docs.oracle.com/javase/tutorial/essential/io/streams.html)
- Network Sockets (4 pages)
  [http://docs.oracle.com/javase/tutorial/networking/sockets/index.html](http://docs.oracle.com/javase/tutorial/networking/sockets/index.html)

Review

Shared memory vs. message passing

Race conditions caused by shared memory access

Today: dig deeper into message passing, and see our first example of deadlock

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 looking at or changing this; don’t touch it right now.” Locks have two operations:

- lock.acquire() allows a thread to take ownership of a lock; it waits until the lock is free, and then acquires it
- lock.release() releases ownership of the lock, allowing other threads to get it
In the example below, both A and B are trying to access the same account. Suppose B acquires the lock first; then A must wait to read and write the balance until B finishes and releases the lock. This ensures that A and B are synchronized, but another cash machine C would be able to run independently on a different account (because it has 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 that we saw last lecture).

**Deadlock**

When used properly and carefully, locks can prevent race conditions. But then another problem rears its ugly head. Because use of locks require threads to wait (when a lock is held by another thread), it’s possible to get into a situation two threads are waiting for each other to do something – and hence neither can make progress.

Suppose A and B are making simultaneous transfers between two accounts in our bank:

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 its respective “from” account – A acquires account 1’s lock, and B acquires account 2's lock. 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 in a “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 also have deadlock without using locks – we’ll see an example later in this lecture.
Message Passing Example

Now let’s look at a message-passing approach to our bank account example. Instead of sharing objects in memory, the message-passing approach has concurrent 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 necessarily stop working while waiting for an answer to its request; it can handle more requests from its own queue. The reply eventually comes back as another message.

Here’s an example showing the accounts and cash machines sending messages to each other for deposit, withdraw, and get-balance operations:

![Message Passing Diagram](image)

In many situations, message passing may be the *only* choice, since no shared memory exists. Web applications communicate by message passing.

Unfortunately 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?

![Message Passing Example Diagram](image)

Lesson: you need to carefully choose the **atomic** (indivisible) operations of the state machine – withdraw-if-sufficient-funds would be a better operation.

Message-passing can have deadlocks too – particularly when using finite queues that can fill up and cause a sending module to block.
Client/Server Design Pattern

In today’s lecture (and in the next problem set) we’re going to use a well-established design pattern for message passing called client/server.

This pattern has multiple processes communicating by message passing. There are two kinds of processes: clients and servers. A client initiates the communication by connecting to a server. The client sends requests to the server, and the server sends replies back. Finally the client disconnects.

Many Internet applications work this way: web browsers are clients for web servers, an email program like Thunderbird or Outlook is a client for a mail server, etc.

On the Internet, client and server processes are often running on different machines connected by the network, but it doesn’t have to be that way – the server can be a process running on the same machine as the client.

Network Sockets

A network interface is identified by an IP address (or a hostname, which translates into an IP address; so there may be many synonyms).

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 the server is 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. You may have seen URLs like http://128.2.39.10:9000. In this URL, 9000 refers to the port number to connect to on the computer at IP address 128.2.39.10.

A socket represents one end of the connection between client and server. Client sockets also have port numbers, usually chosen at random from available non-well-known numbers.

- A listening socket is used by a server process to wait for connections from remote clients. In Java, use java.net.ServerSocket and its accept method.
- An established socket can send and receive messages to and from the process on the other end of the connection. It is identified by both the local IP address and port number plus the remote address and port, which allows a server to differentiate between concurrent connections from different IPs (or from the same IP from different remote ports). In Java, clients use java.net.Socket. Servers get a Socket from ServerSocket.accept.

Buffers

Data is sent over a network in chunks. Rarely just byte-sized chunks (though they may be). The sending side typically writes a big chunk (maybe a whole string like “Hello, world!”), or maybe 20 megabytes worth of video data all at once). The network chops that chunk up into packets, which are routed separately over the network. And the receiver reassembles the packets together to a stream of bytes.

The result is a bursty kind of data transmission – the data may be there when you want to read it, or you may have to wait for it.

When data arrives, it is put into a buffer, which is simply an array in memory that is holding it until you read it.
## Blocking

**Blocking** means a thread waits (doing nothing) until an event occurs. It’s usually used to refer to a method call: when a method call **blocks**, it delays returning to its caller until the event occurs.

Socket streams exhibit blocking behavior:

- When an incoming socket’s buffer is empty, **read()** blocks.
- When the destination socket’s buffer is full, **write()** blocks.

Blocking is very convenient from a programmer’s point of view, because the programmer can write code as if the **read()** call will always succeed, no matter what the timing of data arrival. The operating system takes care of the details of delaying your thread until **read()** can succeed.

Blocking happens throughout concurrent programming, not just in I/O. Concurrent modules don’t work in lockstep, like sequential programs do, so they typically have to wait for each other to catch up.

We’ll see, though, that all this waiting causes the second major kind of bug in concurrent programming: **deadlocks**.

## Wire Protocols

Now that we’ve got our client and our server and they’re connected up with sockets, what do they pass back and forth over those sockets?

A **protocol** is a set of messages that can be exchanged by two communicating parties. A **wire protocol**, in particular, is a set of messages represented as byte sequences, like “hello world” and “bye”.

Most Internet applications use simple ASCII-based wire protocols. You can even use a Telnet program to check them out. For example, you can speak HTTP, the language of the world wide web:

```
telnet www.eecs.mit.edu 80
GET /
```

The **GET** command gets a web page; the / is the path of the page you want on the **www.eecs.mit.edu** server. So this command effectively fetches the page at **http://www.eecs.mit.edu:80/**

Internet protocols are defined by RFC specifications (RFC stands for “request for comment”). RFC 1945 defined HTTP version 1.0, and was superseded by HTTP 1.1 in RFC 2616:

```
telnet web.mit.edu 80
GET / HTTP/1.1
Host: web.mit.edu
```

That request must end with a blank line (why?), and you may need to quit telnet manually using the escape character after receiving a response (why?).

## Designing a Wire Protocol

Similar to defining operations for an abstract data type: small, coherent, adequate

The equivalent of representation independence is platform-independence

Ready for change – e.g., version number that client and server can announce to each other:

```
GET / HTTP/1.0
```
Deadlock

When buffers fill up, message passing systems can experience deadlock.

Deadlock: two concurrent modules are both blocked waiting for each other to do something. Since they’re blocked, neither will be able to make it happen, and neither will break the deadlock.

In general, in a system of multiple concurrent modules communicating with each other, we can imagine drawing a graph in which the nodes are the modules and there’s an edge from A to B if A is blocked waiting for B to do something. The system is deadlocked if at some point in time, there’s a cycle in this graph. The simplest case is the two-node deadlock, A -> B and B -> A, but more complex systems can have larger deadlocks.

Deadlocked systems appear to simply hang. They’re not done, there’s still work to be done, they just can’t make any progress.

One solution to deadlock is to design the system so that there is no possibility of a cycle – so that if it’s possible for A to wait for B, then it’s never possible for B to wait for A.

Another approach to deadlock is timeouts – if a module has blocked for too long (maybe 100 milliseconds? maybe 10 seconds? it depends on the application and how long you need to wait), then you stop blocking and throw an exception. Then the problem becomes: what do you do when that exception gets thrown?

Message Passing with Threads

We’ve talked about message passing in the process context (e.g. a client and a server communicating through network sockets). We can also use message passing between threads, and this is often preferable to a shared memory design with locks.

Use a synchronized queue for message passing between threads. This queue serves the same function as the buffered network communication channel in client/server message passing.

Producer threads put requests on the queue, and consumer threads (often just one, but possibly more) take requests off the queue:

The interface java.util.concurrent.BlockingQueue is such a queue. Here’s a state machine spec for it:

Note that there’s no take transition in the EMPTY state, so a consumer thread that tries to take from an empty queue will block (wait) until something is in the queue (SOME or FULL).

There is also no put in the FULL state. Producer threads will block until there is space in the queue.
BlockingQueue has two implementations in Java:

- **ArrayBlockingQueue** is a fixed-size queue that uses an array representation.
- **LinkedBlockingQueue** is a growable queue using a linked-list representation. If a maximum capacity is not specified, the queue will effectively never reach the FULL state, and putting an item on the queue will never block.

Unlike the streams of bytes sent and received by sockets, these synchronized queues – like the other collections classes in Java – contain objects of an arbitrary type. Instead of designing a wire protocol, we must choose or design a type for messages in the queue.