Review

• Why deal with distributed systems?
  • Scalability
  • Performance
  • Latency
  • Availability
  • Fault Tolerance
Review

• More machines, more problems
  • Replication solves everything, and makes everything a lot worse
• CAP Theorem
  • Consistency, Availability, Partition Tolerance
CAP Theorem

Set A = 5

"OK"!

Read A

"6"!

Assume replica failed

Set A = 5
Review: HW1

• Building a cache in front of a filesystem
• In Java
• Tricky part:
  • Consistency (multiple processes can access filesystem at once)
Today

• Systems fundamentals and abstractions
• Synchronization + Critical Section Problem
• A ton of Java
• Some tips to get you through HW1
• Resources:
  • http://winterbe.com/posts/2015/04/07/java8-concurrency-tutorial-thread-executor-examples/
  • http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/Lock.html
  • https://developers.google.com/protocol-buffers/
Today
Today

- User
- User
- User
- User

System & Application Programs

Operating System

Hardware

To network
The role of the Operating System

• Mediates access to hardware from many simultaneous processes
• Manages resources
  • CPU, network, memory, storage, etc
• Receives requests from processes to access resources, provides responses
Processes

• What’s a process?
  • Some program that is executing; execution is always sequential

• Represented as:
  • Code
  • Program counter (position in code)
  • Stack
  • Heap

• A program is what we store on disk: can run a program more than once simultaneously: multiple processes

• By default, hard to communicate
Processes

```
public class Sample {
    static int i;

    public static void main(String[] args) {
        int k = 10;
        foo(k);
    }

    public static void foo(int in) {
        bar(in);
    }

    public static void bar(int in) {
        i = in;
        System.out.println("bar");
    }
}
```
Java: Heap vs Stack

- Stack contains local variables and method arguments
- Heap contains everything else
- Stack might contain pointers to the heap

```java
public class Sample {
    static int i;
    static String str = "myString";
    public static void main(String[] args) {
        int k = 10;
        String stackPointer = str;
        foo(k);
    }
    public static void foo(int in) {
        bar(in);
    }
    public static void bar(int in) {
        i = in;
        System.out.println("bar");
    }
}
```
Threads

Program execution: a series of sequential method calls (✩'s)

App Starts

App Ends
Threads

Program execution: a series of sequential method calls (⭐️'s)

App Starts

Multiple threads can run at once --> allows for asynchronous code

App Ends
What do we use threads for?

- Run multiple tasks seemingly at once
  - Update UI
  - Fetch data
  - Respond to network requests
- Process creation: heavyweight, thread creation: lightweight
- Improve responsiveness, scalability
- Concurrency + Parallelism
Concurrency vs Parallelism

4 different threads: T1 T2 T3 T4

Concurrency: (1 processor)

Parallelism: (2 processors)
Threads: Memory View

Single-Threaded Process

Multi-Threaded Process

Each thread has its own stack
Threads: Memory View

Each thread might be executing the same code, but with different local variables (and hence doing different stuff).

Single-Threaded Process

Multi-Threaded Process

Heap data: still shared between threads
Threads: Memory View

Single-Threaded Process

Multi-Threaded Process

Each thread might be executing totally different code, too
How to split up the work

- **Data parallelism** - distribute subsets of same data across multiple cores, perform same operation on each
- **Task parallelism** - distribute multiple threads, each thread performs a different operation
- In either case, there is some need to coordinate and share data between threads
Threads in Java

• In Java, make a new thread by instantiating the class java.lang.Thread
• Pass it an object that implements Runnable
• When you call thread.start(), the run() method of your runnable is called, from a new thread
• join() waits for a thread to finish

```java
Thread t = new Thread(new Runnable() {
    @Override
    public void run() {
        //This code will now run in a new thread
    }
});
t.start();
```
Threads in Java

```java
public static void main(String[] args) throws InterruptedException {
    Thread t = new Thread(new Runnable() {
        @Override
        public void run() {
            //This code will now run in a new thread
            System.out.println("Hello from the thread!");
        }
    });
    t.start();
    System.out.println("Hello from main!");
    t.join();
}
```

What is the output of this code?

- **#1** Hello from the thread!
  Hello from main!

- **#2** Hello from main!
  Hello from the thread!

This is a race condition
Race Conditions

```java
static int i = 0;
public static void increment()
{
    i = i + 1;
}
```

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>increment()</td>
<td>increment()</td>
</tr>
<tr>
<td>read i = 0</td>
<td>read i = 0</td>
</tr>
<tr>
<td>write i = 1</td>
<td>write i = 1</td>
</tr>
</tbody>
</table>
Critical Section Problem

- Each thread/process has some *critical section* of code that:
  - Changes shared variables, files etc
  - When one thread/process is in a critical section, no other may be in the same critical section
- Critical section problem: design a protocol to solve this
- Each process/thread asks for permission to enter critical section, does its work, then exits the section
Critical Section in increment()

static int i = 0;
public static void increment()
{
    enterSection();
    i = i + 1;
    exitSection();
}

Only one thread can read/write i at once

But how to implement enterSection() and exitSection()?
Solution to Critical-Section Problem

• Need to guarantee **mutual exclusion**
  • If one thread/process is executing its critical section, no other can execute in their critical sections

• Need to guarantee **progress**:
  • If no process is executing in its critical section, and some other would like to, then some process must be allowed to continue

• Need to guarantee **bounded waiting**:
  • If some process wants to enter its critical section, it must eventually be granted access
Peterson’s Solution

- Simple algorithm that solves critical section problem
- Requires two variables: int turn, boolean flag[]
- turn indicates which process can enter the critical section
- flag is an array indicating which process is ready to enter its critical section, flag[i] = true means P_i is ready to enter its critical section
Peterson’s Solution

Algorithm for \( P_i \)

```java
while(true) {
    flag[i] = true;
    turn = j;
    while(flag[j] && turn == j); //wait if j is in its critical section
    //critical section
    flag[i] = false; //signal we are done
    //do anything else that is not in critical section
}
```

“Busy waiting”

**Problem: Inefficient** - this thread keeps checking flag[], preventing other things from running on your CPU
Busy Waiting

while(true) {
    flag[i] = true;
    turn = j;
    while(flag[j] && turn == j);  //wait if j is in its critical section
    //critical section
    flag[i] = false;  //signal we are done
    //do anything else that is not in critical section
}
Locks

- Most systems have some hardware support for implementing this, based on **locks**

- This tells the OS and processor that when a thread is waiting for a lock, **not to bother running it** until it can receive the lock

```
Acquire lock | Wait for lock | Acquire lock | Release lock | Acquire lock | Wait for lock | Acquire lock | Release lock | Acquire lock | Release lock | Acquire lock | Release lock | Acquire lock | Release lock | Acquire lock | Release lock | Acquire lock | Release lock | Acquire lock | Release lock | Acquire lock | Release lock
```

T1 finished faster, T2 got to start sooner
Deadlocks & Starvation

- Starvation: one or more threads are blocked from gaining access to a resource, and hence, can’t make progress

- Deadlock: Two or more threads are waiting for the others to do something
  - T1: Has lock 1, needs lock 2
  - T2: Has lock 2, needs lock 1
  - Hence, neither T1 nor T2 can execute
Locking in Java

• Most locks are *reentrant*: if you hold it, and ask for it again, you don’t have to wait (because you already have it)

• Basic primitives:
  • `synchronized{}`
  • `wait`
  • `notify`

• Plus...
  • Lock API... `lock.lock()`, `lock.unlock()`
  • The *preferred* way
Synchronized methods in Java

```java
public synchronized static void increment()
{
    i = i + 1;
}
```

Result: Before entering `increment()`, thread gets a lock on the Class object of `increment()`
wait and notify()

- Two mechanisms to enable coordination between multiple threads using the same monitor (target of synchronized)
  - While holding a monitor on an object, a thread can wait on that monitor, which will temporarily release it, and put that thread to sleep
  - Another thread can then acquire the monitor, and can notify a waiting thread to resume and re-acquire the monitor
wait and notify() example

```java
public class BlockingQueue<T> {

    private Queue<T> queue = new LinkedList<T>();
    private int capacity;

    public BlockingQueue(int capacity) {
        this.capacity = capacity;
    }

    public synchronized void put(T element) throws InterruptedException {
        while (queue.size() == capacity) {
            wait();
        }
        queue.add(element);
        notify(); // notifyAll() for multiple producer/consumer threads
    }

    public synchronized T take() throws InterruptedException {
        while (queue.isEmpty()) {
            wait();
        }
        T item = queue.remove();
        notify(); // notifyAll() for multiple producer/consumer threads
        return item;
    }
}
```

Only one thread can be in put or take of the same queue.
Synchronized methods in Java

```java
public synchronized static void increment()
{
    i = i + 1;
}
```

Result: Before entering `increment()`, thread gets a lock on the Class object of `increment()``

```java
public synchronized static void incrementOther()
{
    j = j + 1;
}
```

Result: Before entering `incrementOther()`, thread gets a lock on the Class object of `incrementOther()``

Problem?
Synchronized blocks in Java

• Can also use *any* object as that monitor

```java
static Object someObject = new Object();
public static void increment()
{
    synchronized(someObject){
        i = i + 1;
    }
}

static Object someOtherObject = new Object();
public static void incrementOther()
{
    synchronized(someOtherObject){
        j = j + 1;
    }
}
```

Now, two different threads could call `increment()` and `incrementOther()` at the same time
Java Lock API

- **Synchronized** gets messy: what happens when you need to synchronize many operations? What if we want more complicated locking?

- **ReentrantLock**: same semantics as synchronized

- **ReadWriteLock**: allows many readers simultaneously, but writes are exclusive

```java
static ReentrantLock lock = new ReentrantLock();
public static void increment()
{
    lock.lock();
    try{
        i = i + 1;
    } finally{
        lock.unlock();
    }
}
```
Java Lock API

```java
static ReadWriteLock lock = new ReentrantReadWriteLock();
static int i;
public static void increment()
{
    lock.writeLock().lock();
    try{
        i = i + 1;
    } finally{
        lock.writeLock().unlock();
    }
}

public static int getI()
{
    lock.readLock().lock();
    try{
        return i;
    } finally{
        lock.readLock().unlock();
    }
}
```
Locking Granularity

• BIG design question in writing concurrent programs: how many locks should you have?

• Example: Distributed filesystem
  • It would be correct to block all clients from reading any file, when one client writes a file
  • However, this would not be performant at all!
  • It would be much better to instead lock on individual files
  • More locks -> more complicated semantics and tricky to avoid deadlocks, races
I/O

• OS manages all access to network, filesystem, memory
• I/O is typically *synchronous*
  • Program requests some I/O, then waits
  • Eventually, the I/O completes, and the program resumes
• Storage hierarchy:
  • Main memory
  • SSD
  • Magnetic (hard disks)
  • Network
• Exposed interfaces: files (local), sockets (remote hosts)
Nonblocking & Asynchronous I/O

• Blocking: Your thread is suspended until I/O is completed (e.g. read(), write() are not instantaneous so you wait)

• Nonblocking: I/O calls return only as much as they can
  • select() to find if there is data then read() or write() it if it’s there

• Asynchronous: Process runs while I/O occurs
  • Can be very tricky to do correctly
Files

- File:
  - Name
  - Size (bytes)
  - Create/Access/Modification Time
  - Contents (binary)

- Directory:
  - Maintains a list of the files (and their metadata) in that directory
File Operations

• Create
• Read, Write
  • OS normally provides this functionality with small buffers, CFS simplifies this for you to read/write an entire file at once
• Delete
• Truncate
  • CFS simplifies this by turning it into a write with no contents
• Open, close file
• Open, close directory
Open file locking

- When a file is opened, it can also be locked on.
- In CFS (the homeworks for this course), opening a file will always acquire a lock in the following ways:
  - Several processes can read a file concurrently.
  - Only one process can write a file at a time; no process can read that file simultaneously.
Directory Structure

• Directories contain information about the files in them
• Directories can be nested
• Operations on directories:
  • Create file
  • List files
  • Delete file
  • Rename file
Filesystems

• Define how files and directory structure is maintained
• Exposes this information to the OS via a standard interface (driver)
• OS can provide user with access to that filesystem when it is **mounted**
• (Example: NFS, AFP, SMB… CFS)
Mounting Filesystems

Filesystem driver is passed path only from its mount point (e.g. it doesn’t care where it is mounted)
Sockets

- Basic abstraction for network communication
- Server socket: listens on an IP Address + network port
- Client socket: connects to some IP address + network port
TCP & UDP

• The two underlying protocols that almost every network are built upon
• TCP: server acknowledges receipt of packet to client
• UDP: no acknowledgement -> less overhead, less reliable

• Note: TCP doesn’t guarantee your message will get there - just that you will probably know if it doesn’t
Sockets in Java

- java.net.Socket: client TCP socket
- java.net.ServerSocket: server TCP socket
  - Server listens for connections, client establishes them
- java.net.DatagramSocket: UDP socket
  - UDP is connectionless, so no difference between client/server
Abstractions

- Using sockets directly is annoying - very low level, likely someone else already figured out a way to solve your problem and their fix is bug free
- High level protocols:
  - FTP, SMTP
  - HTTP (REST, SOAP and other web service stuff)
- For general purpose, lower level:
  - RPC - Remote Procedure Call, and for Java: RMI - Remote Method Invocation
  - Language agnostic: Google Protocol Buffers
Remote Procedure Calls

• Example: Address book
• We’ll store our address book on a server
• But to simplify writing the code, can we pretend that the address book is stored on the client?
• RPC
  • Client program will think it’s directly talking to the server
  • Server program will think it’s directly talking to the client
  • In reality, there’s a ton of glue in between them
Remote Procedure Calls

Address Book
Client Stub
addPerson("Prof Bell","ENGR 4422");

Client

RPC Magic

Server

Address Book
Server Stub
addPerson("Prof Bell","ENGR 4422");

54
RPC Magic

- RPC generates these stubs based on an *interface* that you define
- Many RPC implementations: RMI, CORBA, COM, SOAP
- With Java RMI, we define that interface as literally, an interface, which extends *Remote*
- Java RMI can pass any Java object that is *Serializable*
- In general, deciding how to convert application data into bytes to send over the network is tricky
RPC Challenges

• Communications failures
  • Delayed, lost messages, connection resets

• Machine Failures
  • Server or client fails
  • Was request processed?

• How can we tell if a request was processed?
RPC Challenges

• How do we know that our remote call succeeded?
  • With “call and return” style (as in example) we wait for a response. What if it doesn’t come?
Delivery Semantics

• At least once:
  • Keep sending your call until you hear a response
  • If a failure is transient, then you’ll eventually get your message delivered (at least once)

• At most once:
  • Send it once. Hopefully it works. If it didn’t?
  • Tricky. Send some sequence number with each message. Server must guarantee it can keep track of all of the messages it received. Each duplicate message indicates what it is duplicating

• Exactly once:
  • Impossible

• The magic is not perfect :(
RPC Challenges

• How do we know that our remote call succeeded?

• With “call and return” style (as in example) we wait for a response. What if it doesn’t come?

How do we know if our message got through?
Message Formats

• Q: How can we make our RPC cross platform and cross language?
• A: Write our messages in XML/JSON
• Problem: slow, not very friendly
• Answers:
  • Protocol Buffers (Protobuf): efficient, binary, typed, cross platform/language, versioning support
  • Thrift: very similar to Protobuf, also provides an RPC implementation (protobuf is just message format)
Java RMI

• Synchronous (client method doesn’t return until server completes)
• At most once delivery
• Hence, in the event of a communication failure, an exception is thrown on your client
• Implications:
  • Client code needs to be aware that failures might happen (and exception might be thrown)
  • Client code needs to have some plan to handle when a message fails to get through (application specific)
Java RMI

• Threading model:
  • What happens when there are multiple simultaneous RMI requests to the same server?
• RMI creates a thread pool, a set of threads ready to handle each request
  • Subsequent calls from the same client might or might not use the same thread
  • Subsequent calls from other clients might use the same thread as others
• Implications:
  • Can process multiple requests simultaneously
  • Need to be cognizant of thread safety
Java RMI

```java
public interface AddressBook extends Remote {
    public LinkedList<Person> getAddressBook() throws RemoteException;

    public void addPerson(Person p) throws RemoteException;
}

AddressBook book = new AddressBookServer();
AddressBook stub = (AddressBook) UnicastRemoteObject.exportObject(book, 0);
Registry registry = LocateRegistry.createRegistry(port);
registry.rebind("AddressBook", stub);

Registry registry = LocateRegistry.getRegistry("localhost", 9000);
AddressBook addressBook = (AddressBook) registry.lookup("AddressBook");
```
Lab: RMI + Threading

Also: need to collect your GitHub IDs