Announcements

- Maps due next Friday (10 days)
- Concurrency lab this week.
- Providence-based paid coding internships
  - students.brown.edu/bearsatwork
  - Year-round, Apply now!
- One-hour study: learn a tool, do two programming tasks in Java
  - Contact Qi Xin — qx5@cs.brown.edu.
  - Earn $20, and I’ll let you use it as a lab makeup.
Threads

John Jannotti

course/cs0320/www/docs/lectures/

Apr 3, 2018
• A “computer” is CPU, registers, memory (ignore I/O for now).
• Special register: Program Counter (PC)
• A program executes instruction at PC, continues.
• Memory logically divided into heap and stack (and code).
  ▶ The stack holds local variables, state required for function calls.
  ▶ The heap contains explicitly allocated memory. (new)
• Another special register (SP) indexes the stack.
  ▶ During nested function calls, the stack “grows”.
  ▶ Unfun fact: It grows down on x86
Computers are expensive

- Often, a program is “waiting” (for disk, network) anyway.
- So it makes sense to share the hardware.
- Provide the illusion of multiple CPUs, sets of registers, memory.
  - Each program gets its own virtual address (memory) space.
  - Hardware support allows OS kernel to run “occasionally”.
  - Kernel swaps registers, address space periodically.
- Now we have *processes*.
- And the *illusion* of simultaneous execution.
- Processes have no direct control over when they are switched.
Address spaces need not be totally independent

- Memory might be shared
  - Kernel just uses (some of) the same mappings for two processes.
  - Read-only shared memory works well for library code.
  - Writable shared memory
    - Might be used for communications between processes.
    - Lets processes interact quickly (skip the filesystem).
    - But typical processes don’t share much (or any) writable memory

- What happens if two “processes” share all memory?
  - Both at the start, and as they grow.
  - These “processes” are *threads*.
  - They still need independent register state! (recall PC and SP)
How might you use concurrency to speed up Stars?

- Building the KD-Tree
- Looking up neighbors
How might you use concurrency to speed up Stars?

- Building the KD-Tree
  - At each split, the build is independent.
  - But you are building a single shared tree.
  - How should you decide whether to create more threads?

- Looking up neighbors
How might you use concurrency to speed up Stars?

- Building the KD-Tree
  - At each split, the build is independent.
  - But you are building a single shared tree.
  - How should you decide whether to create more threads?

- Looking up neighbors
  - At each split, the radius search is independent.
  - Be very careful about collecting results.
  - The overhead may not be worth it. Profile!
How might you use concurrency to speed up Stars?

- Building the KD-Tree
  - At each split, the build is independent.
  - But you are building a single shared tree.
  - How should you decide whether to create more threads?

- Looking up neighbors
  - At each split, the radius search is independent.
  - Be very careful about collecting results.
  - The overhead may not be worth it. Profile!
  - Top-K search is much harder.
The browser is doing many different things
- Downloading from multiple URLs
- Updating layouts
- Handling GUI interactions (clicks, scrolls)
- Evaluating Javascript

Even when one of these things is slow, UI should not “hang”
- Threads can make the design of such a program simpler.
- They’re not just about parallelism for speed.

For a long time, many browsers made do without threads.
The browser is doing many different things
- Downloading from multiple URLs
- Updating layouts
- Handling GUI interactions (clicks, scrolls)
- Evaluating Javascript

Even when one of these things is slow, UI should not “hang”
- Threads can make the design of such a program simpler.
- They’re not just about parallelism for speed.

For a long time, many browsers made do without threads.
- Asynchronous interfaces offer an alternative.
- Javascript works this way. Callbacks proliferate.
Web Servers (Apache)

- What does a web server do?
  - Receive, parse, process a request.
  - Emit the response.
  - Each client is handled almost independently.

- Simplest design
  - Process each client in a separate thread.
  - Start a thread for each request.
  - This doesn’t work very well. Why?
What does a web server do?
- Receive, parse, process a request.
- Emit the response.
- Each client is handled almost independently.

Simplest design
- Process each client in a separate thread.
- Start a thread for each request.
- This doesn’t work very well. Why?

Pool of worker threads
- Queue of requests
- Workers take a request off the queue, process, repeat.
Computers Today

- Multiple *processors* in a single computer.
  - Most servers, some workstations
- Multiple *cores* on a single processor.
  - Hard (impossible?) to buy a desktop that isn’t multi-core.
  - Even phones are dual/quad core and more.
  - $40$ Raspberry PI is quad-core!
  - More and more cores are coming.
- Several ways to take advantage of this.
  - Processes
  - Threads
  - Virtual machines
Why Not Threads

- Programming is more complex
  - Considerably
  - Really
- Different conceptual model
  - Need to think differently about the program.
  - Yet it seems so similar.
- New problems are introduced
  - Programs become nondeterministic and thus, unpredictable.
  - Much harder to debug and test.
  - Problems can be very subtle and temporal. Hard to replicate.
Threads in Java

- Implement the Runnable interface.
  - Just one method: `run()`
  - You are trying to think of a nice separable unit of work.
  - You might use Java 8's lambda expression

- Create a `java.lang.Thread`
  - Supply the Runnable object.
  - The work in `run()` begins when you `start()` the Thread.
  - You *could* inherit from Thread, and implement `public void run()`
  - But don't. (I'll explain more in the next lecture.)
public class PrimeFinder extends Thread {
    public int n = 0;

    public PrimeFinder(n) { this.n = n; }

    @Override
    public void run() {
        while (!Prime.test(n)) n++;
        // what do I do with n?
    }
}

Thread t = new PrimeFinder(64);
t.start();
// Use t.n?
int n = 64;
Thread t = new Thread(() -> {
    while (!Prime.test(n)) n++;
});
t.start();
/* Use n? */
Implementation notes

- You can, and should, name your threads. `t.setName()`
- Execution begins (and ends) in the `run()` method.
- Your program won’t exit while any threads are still running.
  - Unless they are *daemon* threads.
- Java Threads are resource intensive (relative to other objects)
  - Require an OS thread.
  - Require a stack (4-8k).
  - Think in terms of “10s of threads”, not thousands.
Thread Problems

- Race Conditions
- Coordination
- Deadlock
- Overhead
- Livelock, Priority Inversion, Convoying...
Race Conditions

- Two threads accessing/modifying the same thing.
- Consider an event counter

```java
public class Counter {
    private int count = 0;
    public int nextCount() {
        count = count + 1;
        return count;
    }
}
```
Race Conditions

- What does the counter do if called from many threads?
  - Does it always go up?
  - Does it ever go down?

- The counter may be wrong.

- Complex data structures can be corrupted
  - Lists, maps, queues, trees, ... 
  - When private data is “wrong” your invariants may not hold.
  - Unsynchronized HashMap? ⇒ Infinite loop!
Avoiding Race Conditions

```java
public int nextCount() {
    return ++count;
}
```
Avoiding Race Conditions

```java
public int nextCount() {
    return ++count;
}
```

- NO!

Critical regions (or sections) are a simple, common way:
- Portions of code in which only one thread executes at a time.
- The JVM decides which gets to run.

`nextCount()` would be a good choice.
Avoiding Race Conditions

```
public int nextCount() {
    return ++count;
}
```

- **NO!**
- Need to explicitly control access to shared state
  - In a consistent and fair(?) manner
  - Do not assume an operation is atomic because it is “short”.
- Critical regions (or sections) are a simple, common way
  - Portions of code in which only one thread executes at a time.
  - The JVM decides which gets to run.
  - `nextCount()` would be a good choice.
Critical Regions

- Mutex (MUTual EXclusion) or Lock
  - Operations: lock and unlock
  - Only one thread can hold a lock.
  - Others wait if they try to acquire it.
- “Built-in” to the Java language at the syntax level.
- Yet you should prefer higher-level constructs when available.
Java’s “synchronized” regions

- Every object can act as a lock/mutex
  - Every object is a different lock.
  - Only one thread can hold the lock.
  - Acquires lock on entry, else waits.

```java
1 Map<String, Double> bank = new HashMap<>();
2 . . .
3 synchronized (bank) {
4   double balance = bank.get("John");
5   balance += 231.23;
6   bank.put("John", balance);
7 }
```

Every other piece of code must be written to obey the same “locking discipline”!
A mental model for synchronized blocks

```java
synchronized (bank) {
    double balance = bank.get("John");
    balance += 231.23;
    bank.put("John", balance);
}

becomes (CAPITALS == built in “magic”)

LOCK lock = bank.GETLOCK();
try {
    lock.OBTAIN()
    double balance = bank.get("John");
    balance += 231.23;
    bank.put("John", balance);
} finally {
    lock.RELEASE();
}
```
Java’s synchronized methods

synchronized keyword in front of method.

```java
synchronized void deposit(String name, double amount) {
    double balance = table.get(name);
    balance += amount;
    table.put(name, balance);
}
```

Equivalent to synchronized (this) for body.

```java
void deposit(String name, double amount) {
    synchronized (this) {
        double balance = table.get(name);
        balance += amount;
        table.put(name, balance);
    }
}
```

What about a static synchronized method?
Examples

- **List**
  - add() / remove() in synchronized regions
  - Can you safely do unsynchronized lookups in ArrayList?
  - Iteration?
  - What is the difference between Vector & ArrayList?
    - Prefer Collections.synchronizedList(alist);

- **Maps, Sets, and other collections**
  - Unsynchronized by default.
  - Map sm = Collections.synchronizedMap(m);
  - Avoids corruption, but { get(); ...; put(); } still isn’t atomic!
  - Consider java.util.concurrent.Concurrent(Hash)Map