Concurrency & Parallelism

So far, our programs have been sequential: they do one thing after another, one thing at a time.

Let’s start writing programs that do more than one thing at a time.

Concurrent Work

Concurrency in Multiple Machines

- Datacenters and clusters are everywhere:
  - Industrial: Google, Microsoft, Amazon, Apple, Facebook…
  - Scientific: Several big clusters just in Gates Hall.

Multicore Processors

Every desktop, laptop, tablet, and smartphone you can buy has multiple processors.
Concurrency & Parallelism

Parallelism is about using additional computational resources to produce an answer faster.

Concurrency is about controlling access by multiple threads to shared resources.

A thread or thread of execution is a sequential stream of computational work.

Java: What is a Thread?

- A separate “execution” that runs within a single program and can perform a computational task independently and concurrently with other threads
- Many applications do their work in just a single thread: the one that called main() at startup
- But there may still be extra threads:
  - Garbage collection runs in a “background” thread
  - GUIs have a separate thread that listens for events and “dispatches” calls to methods to process them
- Today: learn to create new threads of our own in Java

Thread

- A thread is an object that “independently computes”
  - Needs to be created, like any object
  - Then “started” --causes some method to be called. It runs side by side with other threads in the same program; they see the same global data
  - The actual executions could occur on different CPU cores, but don’t have to
  - We can also simulate threads by multiplexing a smaller number of cores over a larger number of threads

Java class Thread

- threads are instances of class Thread
  - Can create many, but they do consume space & time
  - The Java Virtual Machine creates the thread that executes your main method.
- Threads have a priority
  - Higher priority threads are executed preferentially
  - By default, newly created threads have initial priority equal to the thread that created it (but priority can be changed)

Threads in Java

```java
public static void main() {
    ...
}
```

Main Thread

```java
private void run() {
    ...
}
```

MainThread

```
public static void main() {
    ...
}
```

MainThread

```
private void run() {
    ...
}
```

MainThread
1. Make a new class that implements Runnable.
2. Put your code for the thread in the run() method. Don’t call the run method directly!
3. Construct a new Thread with the Runnable:
   ```java
   SomeRunnable r = new SomeRunnable(...);
   Thread t = new Thread(r);
   ```
4. Call the Thread’s start() method.

Sequential version:

```
main
starting
check1
prime
done!
check2

main
```

Parallel version:

```
main
starting
done!
check1
prime
check2

check2
```

Creating a new Thread (Method 1)

```java
class PrimeThread extends Thread {
    long a, b;
    PrimeThread(long a, long b) {
        this.a = a;
        this.b = b;
    }
    @Override
    public void run() {
        //compute primes between a and b
        ...
    }
    p = new PrimeThread(143, 195);
    p.start();
}
```
Creating a new Thread (Method 2)

```java
class PrimeRun implements Runnable {
    long a, b;
    PrimeRun(long a, long b) {
        this.a = a;
        this.b = b;
    }
    public void run() {
        //compute primes between a and b
    }
}
PrimeRun p = new PrimeRun(143, 195);
new Thread(p).start();
```

Example

```java
public class ThreadTest extends Thread {
    int M = 1000;
    int R = 600;
    public static void main(String[] args) {
        new ThreadTest().start();
        for (int h = 0; true; h = h+1) {
            sleep(M);
            System.out.format("%s %d
", Thread.currentThread(), h);
        }
    }
    @Override public void run() {
        for (int k = 0; true; k = k+1) {
            sleep(R);
            System.out.format("%s %d
", Thread.currentThread(), k);
        }
    }
}
```

Terminating Threads is Tricky

- The safe way: return from the run() method.
- Use a flag field to tell the thread when to exit.
- Avoid old and dangerous APIs: stop(), interrupt(), suspend(), destroy()...
  - These can leave the thread in a "broken" state.

Background (daemon) Threads

- In many applications we have a notion of "foreground" and "background" (daemon) threads.
  - Foreground threads are doing visible work, like interacting with the user or updating the display.
  - Background threads do things like maintaining data structures (rebalancing trees, garbage collection, etc.) A daemon can continue even when the thread that created it stops.
- On your computer, the same notion of background workers explains why so many things are always running in the task manager.
Background (daemon) Threads

- daemon: an evil spirit
- daemon. Fernando Corbato, 1963, first to use term. Inspired by Maxwell’s daemon, an imaginary agent in physics and thermodynamics that helped to sort molecules.
- from the Greek δαίμων. Unix System Administration Handbook, page 403: “Daemons have no particular bias toward good or evil but rather serve to help define a person’s character or personality. The ancient Greeks’ concept of a “personal daemon” was similar to the modern concept of a “guardian angel”—eudaemonia is the state of being helped or protected by a kindly spirit. As a rule, UNIX systems seem to be infested with both daemons and demons.

Producer & Consumer

```
producer
q = queue.remove();
System.out.println(q);
```

```
consumer
```

Producer & Consumers

```
producer
generate a prime;
queue.add(p);
q = queue.remove();
System.out.println(q);
```

```
consumer
```

Timing is Everything

```
Thread 1
if (!q.isEmpty()) {
    long p = q.remove();
}
```

```
Thread 2
if (!q.isEmpty()) {
    long p = q.remove();
}
```

A Fortunate Interleaving

```
Thread 1
queue length: 1
if (!q.isEmpty()) {
    long p = q.remove();
}
Condition is true!
Remove an element.
queue length: 0
Condition is false.
Do nothing.
```

```
Thread 2
```

Another Fortunate Interleaving

```
Thread 1
queue length: 1
if (!q.isEmpty()) {
    long p = q.remove();
}
```

```
Thread 2
queue length: 0
```

```
```

### An Unfortunate Interleaving

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>queue length: 1</td>
<td>queue length: 1</td>
</tr>
<tr>
<td>Condition is true.</td>
<td>Condition is still true.</td>
</tr>
</tbody>
</table>

```java
if (!q.isEmpty()) {
    queue length: 1
    if (!q.isEmpty()) { Condition is still true. }
    Remove an element. long p = q.remove();
    queue length: 0
    long p = q.remove();
    NoSuchElementException!
}
```

### Beginning to think about avoiding race conditions

You know that race conditions can create problems:

Basic idea of race condition: Two different threads access the same variable in a way that destroys correctness.

- Process t1

  ```java
  x = x + 1;
  ```

- Process t2

  ```java
  x = x + 1;
  ```

But `x = x+1;` is not an "atomic action": it takes several steps.

Two threads may want to use the same stack, or Hash table, or linked list, or ... at the same time.

### Synchronization

- Java has one primary tool for preventing race conditions. You must use it by carefully and explicitly – it isn’t automatic.
- Called a synchronization barrier
- Think of it as a kind of lock
- Even if several threads try to acquire the lock at once, only one can succeed at a time, while others wait
- When it releases the lock, another thread can acquire it
- Can’t predict the order in which contending threads get the lock but it should be “fair” if priorities are the same

### Synchronized Blocks

**a.k.a. locks or mutual exclusion**

- At most one thread can be in a synchronized (obj) block for the same obj at any given time.

```java
synchronized (obj) { ... }
```

### Solution: use with synchronization

```java
private Stack<String> stack = new Stack<String>();
public void doSomething() {
    synchronized (stack) {
        if (stack.isEmpty()) return;
        String s = stack.pop();
    }
    //do something with s...
}
```

- Put critical operations in a synchronized block
- Can’t be interrupted by other synchronized blocks on the same object
- Can run concurrently with non-synchronized code
- Or code synchronized on a different object
Example: a lucky scenario

private Stack<String> stack = new Stack<String>();

public void doSomething() {
    if (stack.isEmpty()) return;
    String s = stack.pop();
    // do something with s...
}

Suppose threads A and B want to call doSomething(), and there is one element on the stack

1. thread A tests stack.isEmpty() false
2. thread A pops ⇒ stack is now empty
3. thread B tests stack.isEmpty() ⇒ true
4. thread B just returns – nothing to do

What Can i Be at the End?

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initially, i = 0</td>
<td>Initially, i = 0</td>
</tr>
</tbody>
</table>

i += 1;

Finally, i = 2 or 1!

A Pretty Good Rule

Whenever you read or write variables that multiple threads might access, always wrap the code in a synchronized block.

(Following this rule will not magically make your code correct, and it is not always strictly necessary to write correct code. But it is usually a good idea.)
Race Conditions

When the result of running two (or more) threads depends on the relative timing of the executions.

- Can cause extremely subtle bugs!
- Bugs that seem to disappear when you look for them!

Race conditions

- Typical race condition: two processes wanting to change a stack at the same time. Or make conflicting changes to a database at the same time.
- Race conditions are bad news
  - Race conditions can cause many kinds of bugs, not just the example we see here!
  - Common cause for “blue screens”: null pointer exceptions, damaged data structures
  - Concurrency makes proving programs correct much harder!

Deadlock

Use synchronized blocks to avoid race conditions.

But locks are shared resources that can create their own problems. Like other resources: files, network sockets, etc.

If thread A holds a resource that thread B needs to continue, and thread B holds a different resource that thread A needs to continue, you have deadlock.

Dining philosopher problem

Five philosophers sitting at a table.

Each repeatedly does this:
1. think
2. eat (2 forks)

What do they eat?
spaghetti.

Need TWO forks to eat spaghetti!

Simple solution to deadlock:
Number the forks. Pick up smaller one first
1. think
2. eat (2 forks)

eat is then:
pick up left fork
pick up right fork
eat spaghetti
put down left fork
put down right fork

At one point, they all pick up their left forks

DEADLOCK!