Concurrency in Managed Runtime

Programming Models for Emerging Platforms

First Model: POSIX Threads

- With threads, programmers must:
  - 1. Manage physical unit of execution (thread)
  - 2. Manage logical unit of execution (code)
  - 3. Synchronize these physical and logical units of execution
  - 4. Handle life cycle dependency *between* threads
  - 5. Handle memory dependency *between* threads
  - 6. Deal with challenging side effect of concurrency: race conditions and deadlock

Second Model: Cilk

- With Cilk, programmers must:
  - 1. Manage logical unit of execution (code, work)
  - 2. Handle life cycle dependency *between* work
  - 3. Handle memory dependency *between* work
  - 4. Deal with challenging side effect of concurrency: race conditions and deadlock

Second Model: Cilk

- With Cilk, programmers must:
  - 1. Manage logical unit of execution (code, work)
  - 2. Handle life cycle dependency *between* work
  - 3. Handle memory dependency *between* work
  - 4. Deal with challenging side effect of concurrency: race conditions and deadlock

Biased toward the concept of **work**, i.e, relatively independent tasks that need processing

Java™

Why Java

- 1. Showed GC was a viable concept (managed runtime)
- 2. Very well designed object oriented “core”.
- 3. Hotspot VM is a mature and well-built example of JIT compiler
- 4. Android applications (emerging platform) use Java.
- 5. Still one of the most widely used languages.

Why Java

- 1. Showed GC was a viable concept (managed runtime)
- 2. Very well designed object oriented “core”.
- 3. Hotspot VM is a mature and well-built example of JIT compiler
- 4. Android applications (emerging platform) use Java.
- 5. Still one of the most widely used languages.

Why Java

- 1. Showed GC was a viable concept (managed runtime)
- 2. Very well designed object oriented “core”.
- 3. Hotspot VM is a mature and well-built example of JIT compiler
- 4. Android applications (emerging platform) use Java.
- 5. Still one of the most widely used languages.

Why Java

- 1. Showed GC was a viable concept (managed runtime)
- 2. Very well designed object oriented “core”.
- 3. Hotspot VM is a mature and well-built example of JIT compiler
- 4. Android applications (emerging platform) use Java.
- 5. Still one of the most widely used languages.

Why Java

- 1. Showed GC was a viable concept (managed runtime)
- 2. Very well designed object oriented “core”.
- 3. Hotspot VM is a mature and well-built example of JIT compiler
- 4. Android applications (emerging platform) use Java.
- 5. Still one of the most widely used languages.

Why Java

- 1. Showed GC was a viable concept (managed runtime)
- 2. Very well designed object oriented “core”.
- 3. Hotspot VM is a mature and well-built example of JIT compiler
- 4. Android applications (emerging platform) use Java.
- 5. Still one of the most widely used languages.

Why Java

- 1. Showed GC was a viable concept (managed runtime)
- 2. Very well designed object oriented “core”.
- 3. Hotspot VM is a mature and well-built example of JIT compiler
- 4. Android applications (emerg
  Java.
- 5. Still one of the most widely used languages.

How to handle concurrency in managed runtime?

Why Java

- 1. Showed GC was a viable concept (managed runtime)
- 2. Very well designed object oriented “core”.
- 3. Hotspot VM is a mature and well-built example of JIT compiler
- 4. Android applications (emerg Java.
- 5. Still one of the most widely used languages.

How to handle *consistently* concurrency in managed runtime?

Third Model: Java Threads

- Java threads are re-bottled form of pthreads
  - Physical unit of execution (Thread)
  - Logical unit of execution (Code)
  - Synchronize Thread and Code
- Some unique differences
  - All Threads are peers (no “main” thread, no join)
  - Thread bootstrapping associated with objects
  - Synchronized blocks

```java
public class FactorialThread extends Thread {
  public int fact;
  public FactorialThread(int fact) {
    this.fact = fact;
  }
  public void run() {
    int f = 1;
    for (int i = this.fact; I > 1; i--) {
      f *= i;
    }
    System.out.format("fact(%d) = %d", this.fact, f);
  };
  public static void main(String[] args) {
    FactorialThread f1 = new FactorialThread(10);
    f1.start();

    FactorialThread f2 = new FactorialThread(20);
    f2.start();
  }
}
```

Must implement **run**

start kicks of thread **run** method

```
public class Factorial implements Runnable {
  public int fact;
  public Factorial(int fact) {
    this.fact = fact;
  }
  public void run() {
    int f = 1;
    for (int i = this.fact; I > 1; i--) {
      f *= i;
    }
    System.out.format(“fact(%d) = %d”, this.fact, f);
  };
  public static void main(String[] args) {
    Factorial f1 = new Factorial(10);
    Thread t = new Thread(f1);
     t.start();
  }
}
```

Must implement **run**

thread “runs” the Runnable obj

```
public class Factorial implements Runnable {
  public int fact;
  public Factorial(int fact) {
    this.fact = fact;
  }
  public void run() {
    int f = 1;
    for (int i = this.fact; I > 1; i--) {
      f *= i;
    }
    System.out.format("fact(%d) = %d", this.fact, f);
  };
  public static void main(String[] args) {
    Factorial f1 = new Factorial(10);
    f1.run();
  }
}
```

Does **NOT** start new thread.
Simply runs in current thread.

```java
public class Factorial { {
  private int fact = 1;
  private int curr = 1;
  public void next() {
    curr++;
    fact *= curr;
  };
  public int getFact() {
    return fact;
  }
  public int getCurr() {
    return curr;
  }
}

public class Compute extends Thread {
  private Factorial f;
  public Compute(Factorial f) {
    this.f = f;
  }
  public void run() {
    for (int j = 0; j < 10; j++) {
      System.out.format("fact(%d) = %d",
       f.getCurr(), f.getFact());
      f.next();
    }
  }
}
```

Objects can be shared among threads

```java
public class Factorial { {
  private int fact = 1;
  private int curr = 1;
  public void next() {
    curr++;
    fact *= curr;
  };
  public int getFact() {
    return fact;
  }
  public int getCurr() {
    return curr;
  }
}

public class Compute extends Thread {
  private Factorial f;
  public Compute(Factorial f) {
    this.f = f;
  }
  public void run() {
    for (int j = 0; j < 10; j++) {
      System.out.format("fact(%d) = %d",
       f.getCurr(), f.getFact());
      f.next();
    }
  }
}
```

Objects can be shared among threads

Do we have race conditions?

```
synchronized ( obj ) {
  // Code
}
```

Only one thread can be in a synchronized block at a time

obj can be shared among multiple **synchronized** blocks

Automatically released at the end of block (prevents accidental deadlock)

```
synchronized void foo() {
  // Code
}
```

```
synchronized ( obj ) {
  // Code
}
```

```
void foo () {
  synchronized (this) {
    // Code
  }
}
```

Only one thread can be in a synchronized block at a time

obj can be shared among multiple **synchronized** blocks

Automatically released at the end of block (prevents accidental deadlock)

Java Locks

- Example (Factorial.java)

Does **synchronized** solve our all of our concurrency / synchronization primitive issues?

Does **synchronized** solve our all of our concurrency / synchronization primitive issues?

What about waiting on conditions?

Java Conditions

- Example (Queue.java)

Do you think Java **properly** encapsulates concurrency in a language defined by object oriented principles?

Do you think Java **consistently** encapsulates concurrency in a language defined by object oriented principles?

Fourth Model: Java Forkjoin

- Java implementation of a work-stealing scheduler
- Instead of **spawn** and **sync**, you have **fork** and **join**
- Work executes on a ForkJoinPool.
- Tasks extend **RecursiveAction** or **RecursiveTask<T>**, and implement **compute**

```java
public class Fib extends RecursiveTask<Integer> {
  int f;

  public Fib(int f) {
    this.f = f;
  }

  @Override
  protected Integer compute() {
    if (f < 2) return f;
    Fib f1 = new Fib(f-1);
    Fib f2 = new Fib(f-2);

    f1.fork();
    f2.fork();

    return f1.join() + f2.join();
  }

  public static void main(String[] args) {
    ForkJoinPool pool = new ForkJoinPool();
    Fib job = new Fib(10);

    int r = pool.invoke(job);
    System.out.format("fib(10) = %d\n", r);
  }
}
```

Type of task return value

fork/join

Kick off execution on pool

```
public class Fib extends RecursiveTask<Integer> {
  int f;

  public Fib(int f) {
    this.f = f;
  }

  @Override
  protected Integer compute() {
    if (f < 2) return f;
    Fib f1 = new Fib(f-1);
    Fib f2 = new Fib(f-2);

    invokeAll(f1,f2);
    int r = 0;
    try {
      r = f1.get() + f2.get();
    } catch (Exception e) { }

    return r;
  }

  public static void main(String[] args) {
    ForkJoinPool pool = new ForkJoinPool();
    Fib job = new Fib(10);

    int r = pool.invoke(job);
    System.out.format("fib(10) = %d\n", r);
  }
}
```

Shorthand for fork, then join

Grab result with get (can throw InterruptedException)

ForkJoin

- Example (Vector.java)

For Reference / Self Study

- Java Classes
  - Thread
  - Runnable
- Java Sync
  - synchronized blocks
- Compiling and linking
  - Compile: javac -d . File.java
  - Run: java -cp . File
  - Be aware of package naming
- ForkJoin Classes
  - ForkJoinPool
  - RecursiveAction,
  - RecursiveTask<T>
- ForkJoin Methods
  - fork
  - join
  - invokeAll
  - compute