Overview

- Review – what constitutes a thread
- Creating threads – general
- Creating threads – Java
- What happens if synchronization is not used?
- Assignment

What constitutes a thread?

- Instruction pointer
- Stack space (how much?)
- Registers/Local variables
- Auxiliary data
  - Exception handling stack
  - Priority
  - Interruption status
  - Thread-private data
  - …
- Thread control block (TCB)

Creating Threads - general

- What has to be supplied to create a thread?
  - Always and most important: some designation for the code it is to run
  - Stack space
  - Initial state
  - Daemon status
  - Human-readable identifier?
  - Thread group?
  - Initial priority?

Creating Threads - Java

- Built-in Thread class
- Supplying the code (1st way)
  - Subclass Thread and override run() method
    - class MyThread (Thread) {
      void run () {
      …this code *can* be run as the body of a new thread
      }
      }
      MyThread mt = new MyThread()
- At this point mt is a Thread object but no new thread of execution exists

Constructing Threads - Java

- Supplying the code (2nd way)
  - Provide an object of a class that implements Runnable when creating a new Thread object.
    - class MyRunnable implements Runnable {
      void run () {
      …this code *can* be run as the body of a new thread
      }
      }
      MyThread mt = new Thread(new MyRunnable())
- Again, mt is just a Thread object. No new executable thread.
- What has to happen?

Providing Initial State

- Why?
  - Want many threads that are "almost the same"
- How
  - Some approaches allow parameters to the run method, but not Java
  - In Java, pass arguments to Runnable (or Thread subclass) constructor then assign to object data fields
    - MyRunnable(int instanceNum) {
      this.instanceNum = instanceNum;
      }

Creating an executing thread

- **mt.start()**
  - **Thread.start() method constructs new stack and tells JVM to call runnable's run() on the new stack**
- **What happens if you just call mt.run()?**

A Bad Example

- The badness is not related to thread creation but rather to synchronization
- This example is also used in the assignment

Assignment

1. Install java on the machine you'll be using for the class (Eclipse or NetBeans or BlueJ IDEs, jdk or openjdk for command line). Which one did you install?
2. Describe the machine that you are using: number of processor cores, physical memory, 64 or 32 bit architecture.
3. What is the default stack size for threads in the Java environment that you installed? (Dive into the documentation) How can it be changed? Can it be changed per-thread?
4. What does this suggest as the maximum number of threads that will run well on your machine using the default stack size? (Consider the available physical memory and how much is used by each thread)
5. Write a test driver program to run the code of the following Runnable in a new thread. How many stack frames does it succeed in creating before throwing StackOverflow when using the default stack size?
6. Compile and run the Race example code from www.eecs.wsu.edu/~hauser/cs580/handouts/Race.java. Vary the number of iterations (class variable `limit`) until you regularly get an error (or you give up trying – don't give up too soon).
7. Bring your answers and a code listing for your stack overflow test to hand in in class next Monday and we will discuss the results.

Stacksize testing Runnable

- class StacksizeTest implements Runnable {
  int depth = 0;
  public void run() {
  try {
  doOverflow();
  } catch (StackOverflowError e) {
  System.out.println(“Overflow occurred at depth ” + depth + “.”);
  }
  }
  void doOverflow() { depth += 1; doOverflow(); }
  }

Racer

Note: Java “main” program

```
class foo {
    public static void main(String argv[]) {
        … code for main program...
    }
}
```

```
% javac foo.java
```

```
% java foo
```

Next time

- **Thread safety – what are the rules about accessing shared variables in Java? (Chapter 2 in the Java book).**