Multithreading

Working in Parallel

Objectives

- Topics
  - Threading - concepts and terminology
  - Using Java’s threading tools
- Goals: after this lecture, you will be able to
  - contrast multiprocessing, multiprogramming, and multithreading
  - use Thread and Runnable to create new threads

Review

What is a socket?

a) The integer number of a server's connection

b) A class uses to access the network

c) A class used only for UDP network connections

d) The place you plug into a network

Review

What server method blocks while waiting for a client connection?

a) ServerSocket's constructor

b) getInputStream( )

c) None, all its methods return immediately

d) accept( )

Threads, Cores, Programs

- Modern computers typically have multiple cores in one cpu
- A different program can be running in each core at the same time
  - Historically: old computers had one cpu with one core
- "Normal" programs don't share data, so running them in parallel is (kind of) not a problem
- A threaded program can do ***parallel*** computation: it can run different parts of its code in different cores at the same time
  - Why? <u>**Do to the job faster**</u> by doing parts concurrently

Concurrency and Parallel Programming

- More definitions:
  - ***Concurrency***: OS runs more than one program or thread by context switching
    - Does not require multiple cpu’s or cores
  - ***Parallel programming***: two or more programs or threads running on two or more cpu’s or cores
  - ***Multiprocessing***: more than one cpu or core
  - ***Multiprogramming***: using concurrency
  - ***Multithreading***: multiprogramming restricted to threads
- We’re only focusing on threads

Process and Thread

- Some definitions:
  - ***Process***: a running program
    - Keeps track of overall program state: data, memory, and the program counter (points to the next line of code to be run)
  - ***Thread***: within a process, running the program's code
    - Shares the overall program state, but has its own program counter and local state
- The main( ) program, when executed, is a process, and has one thread
  - So "normal" programs that don't use threads still have one thread, main

Threads

- A thread is sometimes called a "unit of control" for running code
  - A main program, and anything else it calls, in a regular program, is one thread
  - Any other threads have to be created manually – it's not automatic
- Threads can ***share memory***, or not, depending on your design
  - It depends on the problem: if you can break up a program into multiple independent (non-sharing) tasks, great!
  - If not, then you may have issues sharing memory among threads – this will be discussed later

Threads, cont.

- Why should you use threads?
  - concurrent execution, or more likely, parallel execution
  - either to ***speed up*** some calculation or …
  - for ***separation of concerns*** or …
  - to make keep some part of the program responsive (e.g. UI) when some other part is taking time (e.g. network connection, computation)
- Why shouldn’t you use threads?
  - Harder to debug
  - May cause data consistency problems
  - Speed up may not be what you expected

Threading Examples

- Example: loading a web page
  - A page is composed of HTML (formatted text), images, tables, etc.
  - Text downloads pretty fast, but the images, not so much
  - Threaded web browsers use threads to download the different parts
  - These are independent of each other if the parts are stored in different places
  - So threading can speed things up by parallelizing the downloads
  - … and the faster parts can be displayed while the slower parts are in progress

Threading Examples, cont.

- Another example: analyze some Covid-19 data
  - load some covid data from Pennsylvania counties from the web
  - sort it by county name
  - create a bar chart from the sorted data
- These are clearly dependent: can't sort before you have the data, can't create the chart without sorted data
- So parallelization using threads won't help, speed-wise

Threads, cont.

- One more example: sort a giant patient data set
  - divide the set up into "manageable" pieces
  - sort each part in parallel – one set per thread
  - when all the sets are sorted, merge them together into one giant sorted list

Java Thread Creation

- Two basic ways to create threads manually:
  - Create a class that implement the `Runnable` interface, override `run();` create an object, pass it to a `Thread` object, call `start()`

must implement run()

```
public class SomeWork implements Runnable {
   @Override
   public void run( ) { … do some stuff …}
}

// main:
Thread t = new Thread( new SomeWork() );
t.start();
```

wrapped in a Thread object

Java Thread Creation, cont.

- Another way:
  - Create a class that `extends` the `Thread` class, override `run();` create an object, call `start()`

must override run()

```
public class MyThread extends Thread {
   @Override
   public void run( ) { … do some stuff …}
}

// main:
MyThread t = new MyThread( );
t.start();
```

this *is* a Thread object

Java Thread Creation, cont.

- Think of `run( )` as the main of the thread
- Main then news-up a `Thread` object with parameter a new object of that class type, then calls `thread.start()`
- You write the `run()` method but never call it – `start()` does the overhead work behind the scenes, then calls your `run()` method – which it knows is there because of `Runnable` or `extends Thread`

Java Thread Issues

- How many threads can you start? As many* as you need
- What should they do? That's the hard part
  - Writing safe and effective parallel programs is an art
- Why might they not be safe? If they use shared data, they might not share it correctly via bad programming
- Why might they not be effective? Overhead can steal the gains you think you'll get from parallelization

* *up to some practical limit*

Example

- You could just call `Arrays.sort(array)` on the whole array
- Sorter( ) sorts one array
- Parallel sort to try for speedup
  - Divide an array into two halves
  - Two threads, one sorts each half
  - Use Sorter twice

Runnable interface

```
public class Sorter implements Runnable {
   private int[] array;

   // Breaking encapsulation, I know, I know
   public Sorter(int[] array) {  this.array = array;   }

   @Override
   public void run() {  Arrays.sort(array);  }
}
```

Example

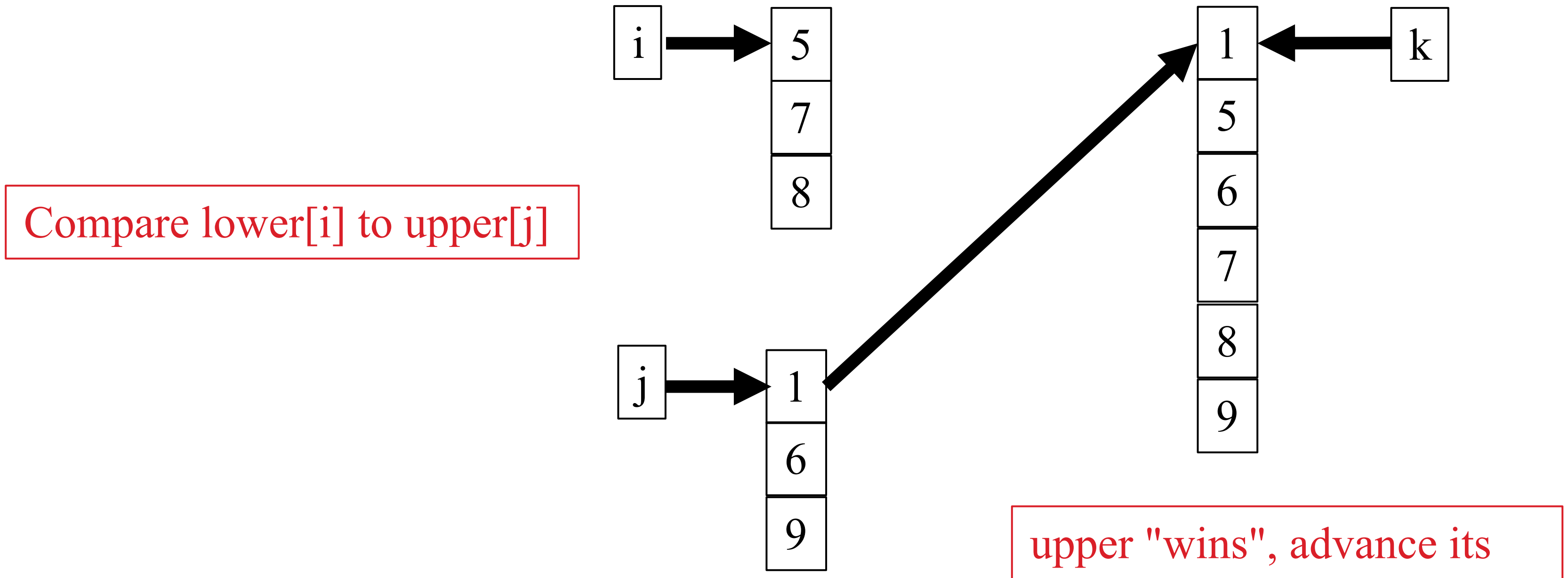
- Main program
  - Create copies of upper, lower halves of the array
  - Create sorter1 with new Sorter(lower)
  - Create sorter2 with new Sorter(upper)
  - Wait for each thread to finish (`join` throws an exception)
  - Merge the sorted halves together
- Just an example – better parallel sorts exist than this

```
int[] lower = Arrays.copyOfRange(list, 0, list.length/2);
int[] upper = Arrays.copyOfRange(list, (list.length/2), list.length);

// Create two threads and start them
Thread sorter1 = new Thread( new Sorter(lower) );
Thread sorter2 = new Thread( new Sorter(upper) );
sorter1.start();
sorter2.start();
try {
  sorter1.join();        wait for sorter1 to finish
} catch (InterruptedException e) {
  e.printStackTrace();
}
try {
  sorter2.join();        wait for sorter2 to finish
} catch (InterruptedException e) {
  e.printStackTrace():
}
list = merge(lower, upper);  // Back in the main thread
```

Merge Operation

- Given two sorted arrays, "shuffle" them together



Performance Comparison

- Built-in sort (Arrays.sort()) versus my (sad) parallel sort
- Built-in sort versus built-in parallel sort (Arrays.parallelSort)

More on the example

- Copying the data is inefficient
  - could have manipulated the array based on indexes
  - but sometimes, this is necessary: now each thread is operating on independent data
- So it's not really a fair comparison

Diminishing Returns

- Why only two threads? Why not a zillion?
  - Overhead (thread management, data management) will eventually slow things down
  - ***Amdahl's Law***:
    - every program has *sequential part* + *parallelizable part*
    - sequential part: initialization, cleanup – e.g. split array into two parts
    - only the second part can be speeded up: if the split is 10% sequential, 90% parallel, the fastest this will run is (10% of the sequential time) + (90% parallel time/zillion) -> 10% sequential
    - ***limit = 1/(1-p)***, so here p = 0.9, limit = 1/(1-0.9) = 1/(0.1) = 10 times speedup, even with a zillion threads

Executors

- Another way to start threads is to use a ***thread pool*** – that is, a set of threads that are then assigned a task
- Example: web server. To be performant, use a thread to handle each new incoming request
- Two styles:
  - ***create a new thread for each new task***, and reuse the threads after they complete
  - ***create a set of threads initially***, assign new tasks to threads, reuse as above, but if there are more tasks than threads, the tasks are queued.

Executors

- First style in Java:

```
ExecutorService exec = Executors.newCachedThreadPool();
```

- Second style in Java (using 5 threads, for example):

```
ExecutorService exec = Executors.newFixedThreadPool(5);
```

- Then in both cases:

```
exec.execute( new taskThatImplementsRunnable() );
exec.shutdown(); // no new threads are started
```

Executor, Callable<T>, and Future

- Another feature is tasks that return a value
- Use a task that implements `Callable<T>`
  - For returning a result from the task of type T
  - Override `<T> call()` instead of `run()`
  - Create a `Future<T>` to store the results
  - Run using the `ExecutorService submit()` method to run the thread aysnchronously *within* the Future add( ) method
  - Call the Future get( ) method to retrieve the result – if the thread hasn’t finished its work, you’ll wait here until it does
  - Between add(… submit()) and get( ), there might be some other code, run in parallel with the thread

Example

```
public class SomeWork implements Callable<String>
{
   @Override
   public String call( ) {
      … do some stuff …
     return someString; // for example
  }
}
```

```
ExecutorService exec = Executors.newCachedThreadPool();
Future<String> result;

result.add( exec.submit( new SomeWork( ) );

… possibly some code here …

try {
   System.out.println("Result = " + result.get() );
} catch(InterruptedException e) { System.out.println(e);
} catch(ExecutionException e) { System.out.println(e);
} finally {
   exec.shutdown();
}
```

asynchronous operation; runs SomeWork.call( ), stores answer in result

synchronous call – wait here for the result