How to Create a Thread in Java

1. Define class C extends Thread
   • Override public void run()
   • Thread in java.lang

2. Create object of class C

3. Call that thread’s start method
   • Creates new thread and starts executing run method.
   • Direct call of run won’t work, similarly to the issue as paint-repaint.
class ExampleThread extends java.lang.Thread {
    int i;
    ExampleThread(int i) {
        this.i = i;
    }
    public void run() {
        System.out.println("Thread "+i+" says hi");
        System.out.println("Thread "+i+" says bye");
    }
}

class M {
    public static void main(String[] args) {
        for(int i=1; i <= 20; ++i) {
            ExampleThread t = new ExampleThread(i);
            t.start();
        }
    }
}

Pass any arguments for the new thread to constructor
No guarantees in order of output

Thread 1 says hi
Thread 2 says hi
Thread 1 says bye
Thread 2 says bye
Thread 3 says hi
Thread 3 says bye
Thread 4 says hi
Thread 4 says bye
Thread 5 says hi
Thread 5 says bye
Thread 6 says hi
Thread 6 says bye
Thread 7 says hi
Thread 7 says bye
Thread 8 says hi
Thread 8 says bye
Thread 9 says hi

Thread 1 says hi
Thread 2 says hi
Thread 3 says hi
Thread 4 says hi
Thread 2 says bye
Thread 4 says bye
Thread 5 says hi
Thread 1 says bye
Thread 5 says bye
Thread 6 says hi
Thread 6 says bye
Thread 7 says hi
Thread 3 says bye
Thread 7 says bye
Thread 8 says hi
Thread 8 says bye
Thread 9 says hi

...
Another example of parallelism

- Method to calculate sum of elements of an array
  - Sequential algorithm takes $O(n)$
- Use 4 threads, which each sum 1/4 of the array
- Steps:
  - Create 4 thread objects, assigning each their portion of the work
  - Call `start()` on each thread object to actually run it
  - Wait for threads to finish
  - Add together their 4 answers for the final result
First Attempt – wrong!

class SumThread extends Thread{
    int lo, int hi, int[] arr;
    int ans = 0; // for communicating result
    SumThread(int[] a, int l, int h) {
        lo=l; hi=h; arr=a;
    }
    public void run(){
        for(int i=lo; i < hi; i++) ans += arr[i];
    }
}

//some other class
static int sum(int[] arr){
    int len = arr.length;
    int ans = 0;
    SumThread[] ts = new SumThread[4];
    for(int i=0; i < 4; i++){
        ts[i] = new SumThread(arr,i*len/4,(i+1)*len/4);
        ts[i].start(); // use start not run
    }
    for(int i=0; i < 4; i++)
        ans += ts[i].ans;
    return ans;
}
class SumThread extends Thread {
    int lo, int hi, int[] arr
    int ans = 0;
    SumThread(int[] a, int l, int h) { ... }
    public void run(){ ... }
}
//some other class
static int sum(int[] arr){
    int len = arr.length;
    int ans = 0;
    SumThread[] ts = new SumThread[4];
    for(int i=0; i < 4; i++){
        ts[i] = new SumThread(arr,i*len/4,(i+1)*len/4);
        ts[i].start();
    }
    for(int i=0; i < 4; i++){
        ts[i].join(); // wait for helpers to finish!
        ans += ts[i].ans;
    }
    return ans;
}

Needs to be within a try/catch block for java.lang.InterruptedException
Thread class methods

• void start(), which calls void run()
• void join() which blocks until receiver thread is done
  • Style called fork/join parallelism
  • It needs a try-catch around join as it can throw InterruptedException
• Some memory sharing:
  • lo, hi, arr fields written by “main” thread, read by helper thread
  • ans field written by helper thread, read by “main” thread
• Later, we will learn how to protect data (race conditions) using synchronized
Great, right? Actually, no!

• If we time it, it’s slower than sequential!!
• We want our code to be reusable and efficient as core count grows (“forward-portable”).
  • At minimum, make #threads a parameter (e.g., in the sum method)
• Want to effectively use processors available now
  • Not being used by other programs or threads in your program
  • Can change while your threads running
Problem

• Suppose we have a computer with 3 processors and a problem of size $n$
  • If we create 3 threads and all 3 processors are available, overall $t$ time.
  • If we create 4 threads, total time $1.5t$
    • Example: 12 units of work, 3 processors
    • 3 threads will take 4 units of time
    • 4 threads: After first 3 are finished (3 units), run 4th which takes another 3 units
    • Total time ends up $3 + 3 = 6$
    • Runs 50% slower than with 3 threads!
More problems

• Subproblems can take significantly different amounts of time
  • Apply method \( f \) to every array element, but maybe \( f \) is much slower for some data items. e.g., is a large integer prime?
  • If unlucky, all slow operations may be assigned to the same thread
    • Certainly, won’t see \( n \) speedup with \( n \) threads
    • May be much worse, due to load imbalance
Toward a solution

• To avoid having to wait too long for any one thread, instead create lots of threads, far more than #cores (counterintuitive)
• Schedule threads as processors become available.
• If a thread is very slow, many others will get scheduled on other processors while that one runs.
• Will work well if the slow thread is scheduled relatively early
Naive idea

Suppose we create 1 thread to process every 1000 elements

```java
int sum(int[] arr){
    ... int numThreads = arr.length / 1000;
    SumThread[] ts = new SumThread[numThreads];
    ...
}
```

- Combining results will be linear in size of array (1/1000 constant)
- Extreme case: 1 thread per element, like original sequential algorithm
Divide and Conquer

1. Divide problem into pieces recursively:
   • Start with full problem at root – Halve and make new thread until size is at some cutoff

2. Combine answers in pairs as we return from recursion
   • If have $numProc$ processors then total time
     \[ O\left(\frac{n}{numProc} + \log n\right) \]
In practice

- Creating so many threads and synchronizing their communication swamps savings
- Instead, use sequential cutoff about 500-1000
  - Eliminates almost all the recursive thread creation (bottom levels of tree)
  - Exactly like quicksort switching to insertion sort for small subproblems, but more important here
- Don’t create two recursive threads: create one thread and do the other piece of work “yourself”. Cuts #threads in half
- Instead of:
  - `left.start(); right.start();
  left.join(); right.join();`
- do:
  - `left.start(); right.run(); left.join();`
ForkJoin Framework to the rescue

• Java’s threads are too heavyweight
  • Expensive to create and destroy
• ForkJoin Framework addresses the need for divide-and-conquer fork-join parallel programming
• Part of Java 7
Java Threads VS ForkJoin

- Create a ForkJoinPool
  `ForkJoinPool.commonPool().invoke`
- Don’t subclass `Thread` → Subclass `RecursiveTask<V>`
- Don’t override `run` → Do override `compute`
- Do not use an `ans` field → Do return a `V` from `compute`
- Don’t call `start` → Do call `fork`
- Call `join` that returns answer
Getting good results in practice

• Sequential threshold
  • Library documentation recommends doing approximately 100-5000 basic operations in each “piece” of your algorithm

• Library needs to “warm up” – May see slow results before the Java virtual machine reoptimizes the library internals

• Wait until your computer has more processors
  • Seriously, overhead may dominate at 4 processors, but parallel programming is likely to become much more important
class SumArray extends RecursiveTask<Integer> {
    int lo; int hi; int[] arr; // arguments
    SumArray(int[] a, int l, int h) { ... }
    protected Integer compute(){// return answer
        if(hi - lo < SEQUENTIAL_CUTOFF) {
            int ans = 0;
            for(int i=lo; i < hi; i++)
                ans += arr[i];
            return ans;
        }
        else {
            SumArray left = new SumArray(arr,lo,(hi+lo)/2);
            SumArray right= new SumArray(arr,(hi+lo)/2,hi);
            left.fork();
            int rightAns = right.compute();
            int leftAns = left.join();
            return leftAns + rightAns;
        }
    }
}
static final ForkJoinPool fjPool = new ForkJoinPool();
int sum(int[] arr){
    return ForkJoinPool.commonPool().invoke(new SumArray(arr,0,arr.length));
}
Examples

- Maximum or minimum element
- Is there an element satisfying some property (e.g., is there a 47)?
- Left-most element satisfying some property (e.g., first 47)
- Smallest rectangle encompassing a number of points
- Counts; for example, number of strings that start with a vowel
- Are these elements in sorted order?
- Create a Histogram of test results from a much larger array of actual test results
CPU vs GPU

From Mythbusters:

https://www.youtube.com/watch?v=-P28LKWTrzrl&feature=youtu.be

In a bit more detail:

https://www.youtube.com/watch?v=1kypaBjJ-pg