Multithreading; Fork/Join Parallelism

CSE 332 Spring 2021

Instructor: Hannah C. Tang

Teaching Assistants:

Aayushi Modi	Khushi Chaudhari	Patrick Murphy
Aashna Sheth	Kris Wong	Richard Jiang
Frederick Huyan	Logan Milandin	Winston Jodjana
Hamsa Shankar	Nachiket Karmarkar	

gradescope

gradescope.com/courses/256241

- Consider the problem of summing an array of integers:

```
void sum(int[] arr) {
  int ans = 0;
  for (int i = 0; i < arr.length; i++)
       ans += arr[i];
  }
  return ans;
}
```

- You have been *so entranced* by the divide-and-conquer technique that you've decided to rewrite `sum()` using recursion
  - Hint: MergeSort's pseudocode:

```
void mergeSort(int[] arr, int start, int end) {
  if (start == end || start+1 == end) return;

  int mid = (end - start)/2 + start;
  mergeSort(arr, start, mid);
  mergeSort(arr, mid, end);

  merge(arr, start, mid, end);
}
```

Announcements

- P2 CP2 due tomorrow *night*
- Parallelism “mini projects” released soon, due Tue May 18
  - You can use the late days to overlap with quiz 3 … but we don’t advise it

Lecture Outline

- **Shared Memory with Threads**

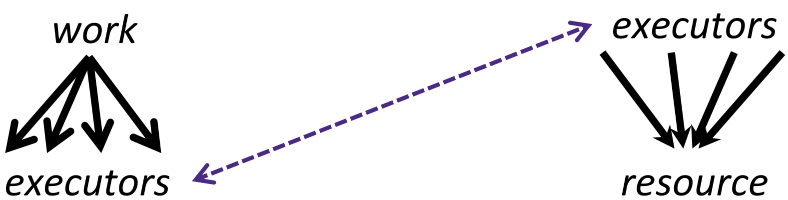
- Concurrency Frameworks in Java
  - Introducing java.lang.Thread
  - Writing good parallel code
  - Improving java.lang.Thread
    - Asymptotically
    - Constants
  - ForkJoin Library

Sequential vs Parallel vs. Concurrent

- **Sequential:** A cook (an executor) making dinner

Parallelism: Use extra executors to solve a problem faster

Concurrency: Manage access to shared resources



- **Parallelism**: *"Extra executors gets the job done faster!"*
  - *Multiple cooks*: One cook in charge of the gravy (and its onions), another in charge of the stuffing (and its onions)
- **Concurrency**: *"We need to manage a shared resource"*
  - *Multiple cooks*: One cook per dish, but only one cutting board

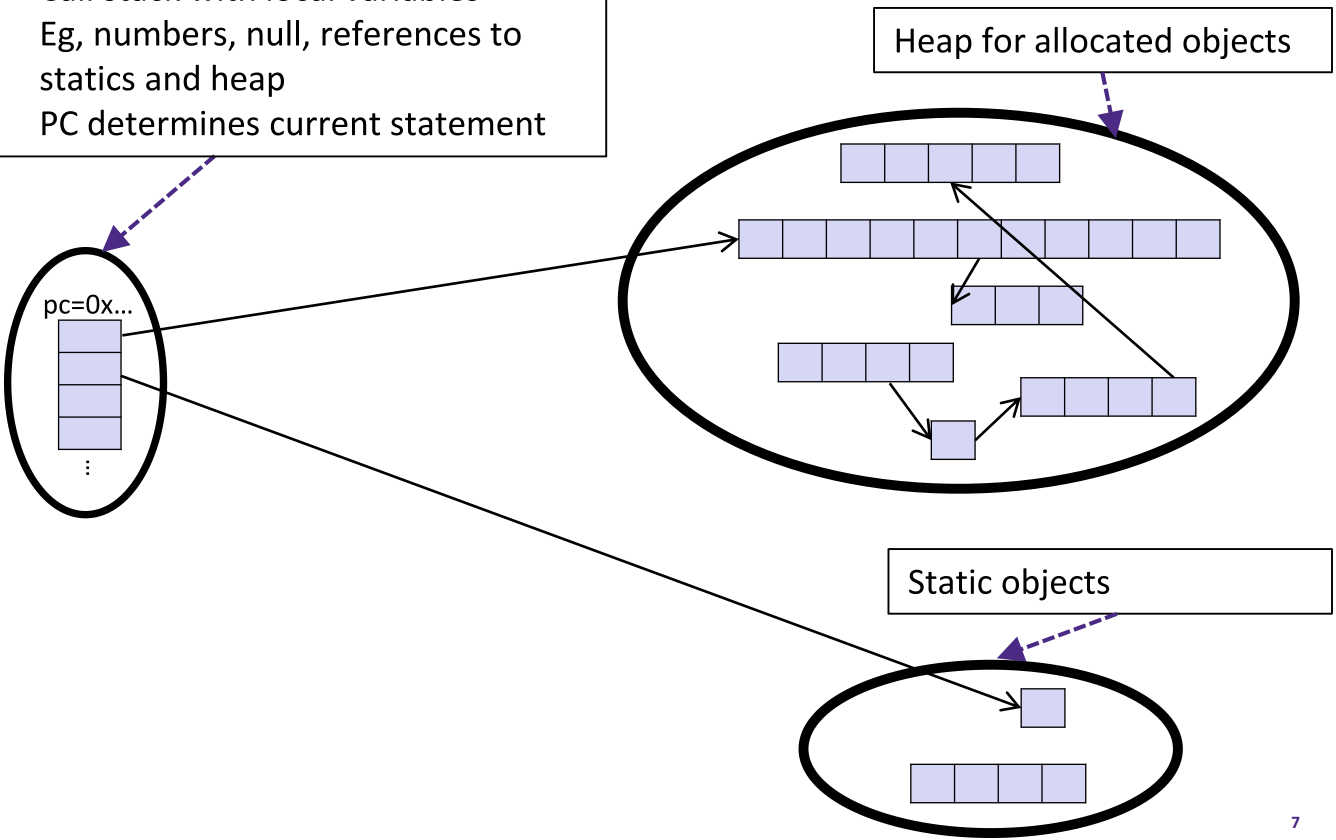
Sequential: One Call Stack and One PC (1 of 2)

- We will assume **shared memory** with **explicit threads**

- *Sequential*: A running program has
  - One ***program counter*** (“PC”): currently executing statement
  - One ***call stack***, with each stack frame holding its local variables
  - ***Objects in the heap*** created by memory allocation (i.e., `new`)
  - ***Static fields*** that are “global” to the entire program

Sequential: One Call Stack and One PC (2 of 2)

- Call stack with local variables
- Eg, numbers, null, references to statics and heap
- PC determines current statement



Our Model: Shared Memory with Threads

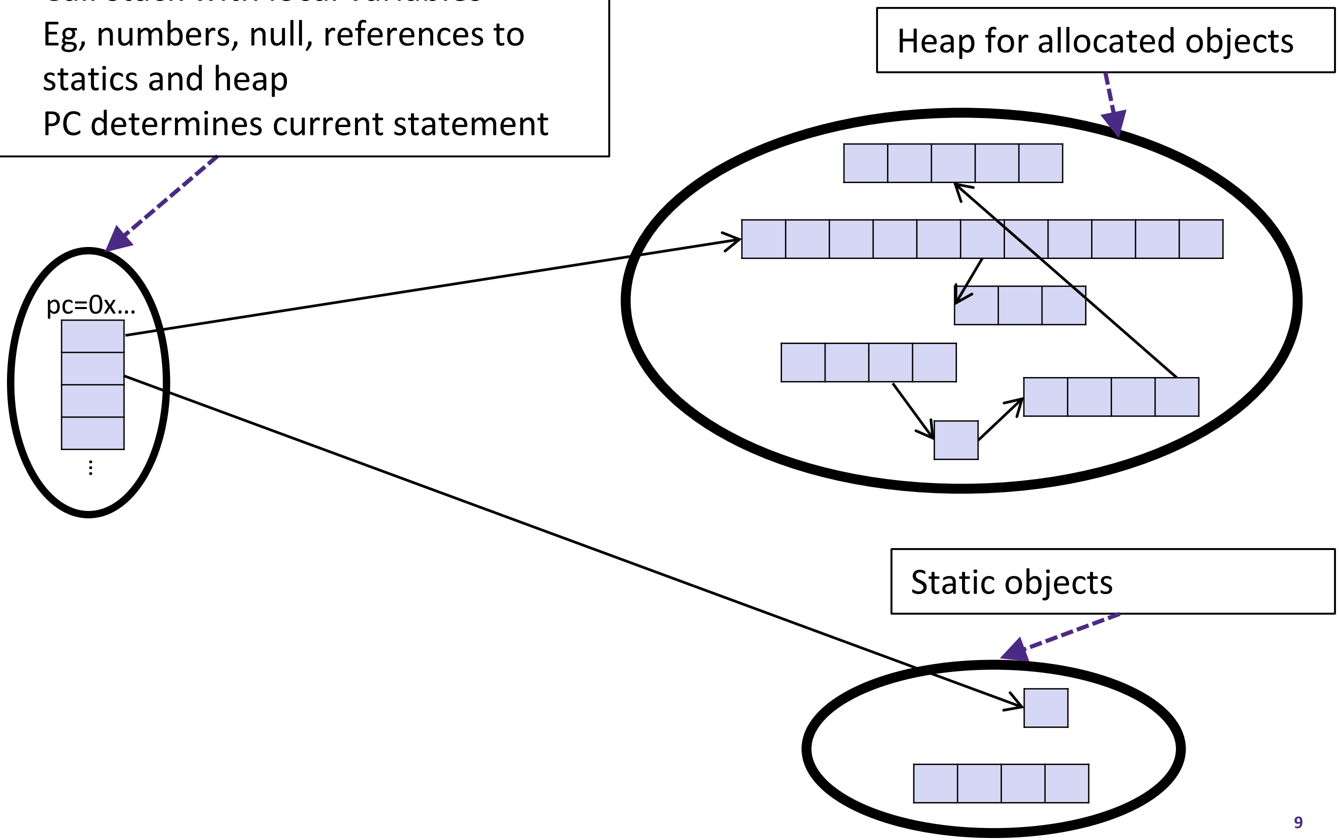
- We will assume **shared memory** with **explicit threads**

- *Sequential*: A running program has
  - One ***program counter*** ("PC"): currently executing statement
  - One ***call stack***, with each stack frame holding its local variables
  - ***Objects in the heap*** created by memory allocation (i.e., new)
  - ***Static fields*** that are "global" to the entire program

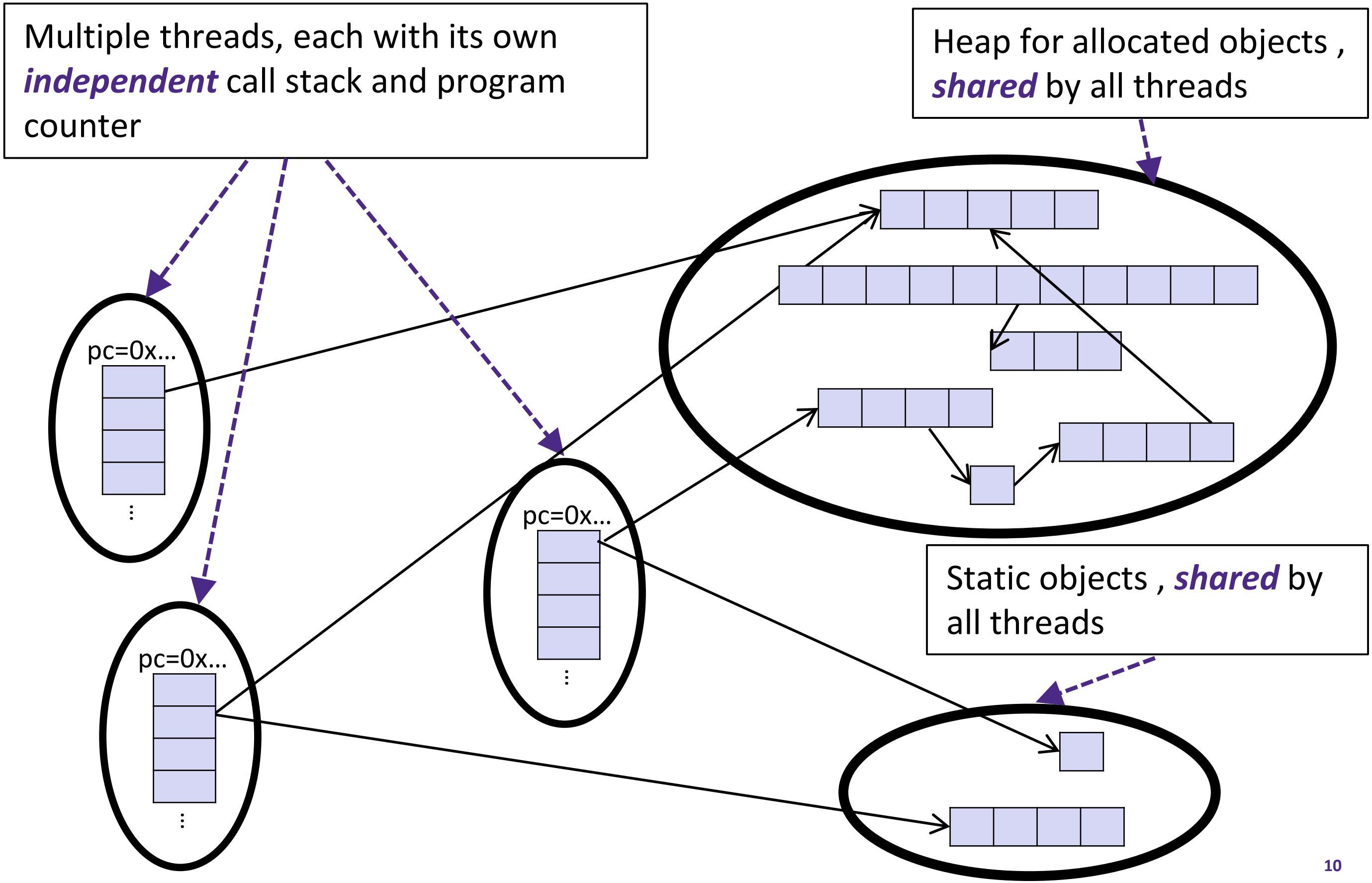
- *Shared Memory with Threads*: A running program has
  - A set of ***threads***, each with its own *program counter* and *call stack*
    - But each thread cannot access to another thread's local variables
  - Threads implicitly share *static fields* and the *heap* (ie, objects)
    - Communication via writing values to some shared location

Sequential: One Call Stack and One PC

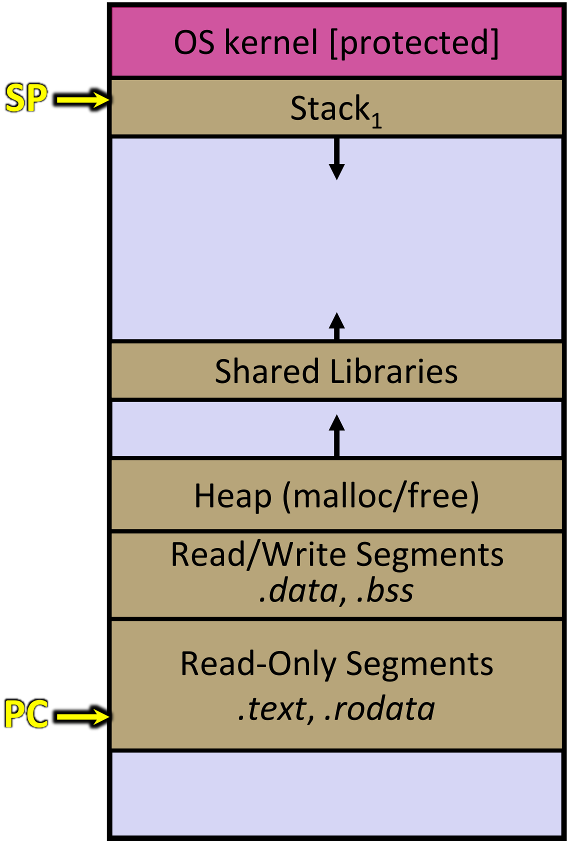
- Call stack with local variables
- Eg, numbers, null, references to statics and heap
- PC determines current statement

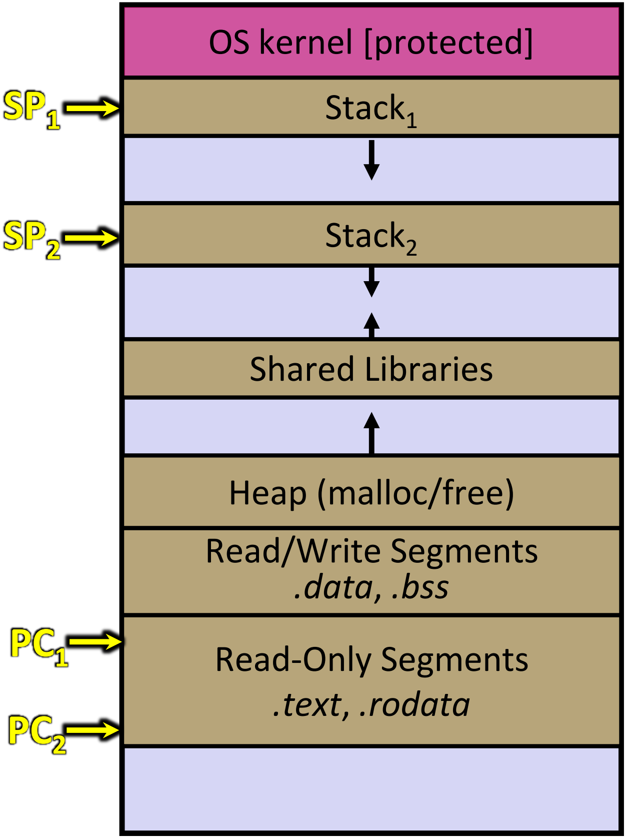


Shared Memory with Threads



Shared Memory with Threads *(if you've taken 351)*





Other Parallelism and Concurrency Models

- We focus on shared memory, but other models exist and have their own advantages
  - **Message-passing**: Each thread has its own collection of objects. Communication happens via explicit messages
    - E.g.: cooks work in separate kitchens and mail around ingredients
  - **Dataflow**: Programmers write programs in terms of a DAG. A node executes after all of its predecessors in the graph
    - E.g.: cooks wait to be handed results of previous steps
  - **Data parallelism**: Primitives for things like "apply this function to every element of an array in parallel"
    - E.g.: cooks wait in their own kitchen for instructions and ingredients

Our Requirements

- To write a shared-memory parallel program, we need new primitives from our *programming language* or a *library*
  - Ways to create and ***execute multiple things at once***
    - i.e. the parallel threads themselves!
  - Ways for threads to ***share memory*** or retain sole ownership
    - Often: just have threads contain references to the same objects
    - How will we pass thread-specific arguments to it? Does the thread have its own "private" (i.e., local) memory?
  - Ways for threads to ***coordinate*** (a.k.a. synchronize)
    - For now, all we need is a way for one thread to wait for another to finish
    - (we'll study other primitives when we get to concurrency)

Lecture Outline

- Shared Memory with Threads
- Concurrency Frameworks in Java
  - **Introducing java.lang.Thread**
  - Writing good parallel code
  - Improving java.lang.Thread
    - Asymptotically
    - Constants
  - ForkJoin Library

Introducing java.lang.Thread

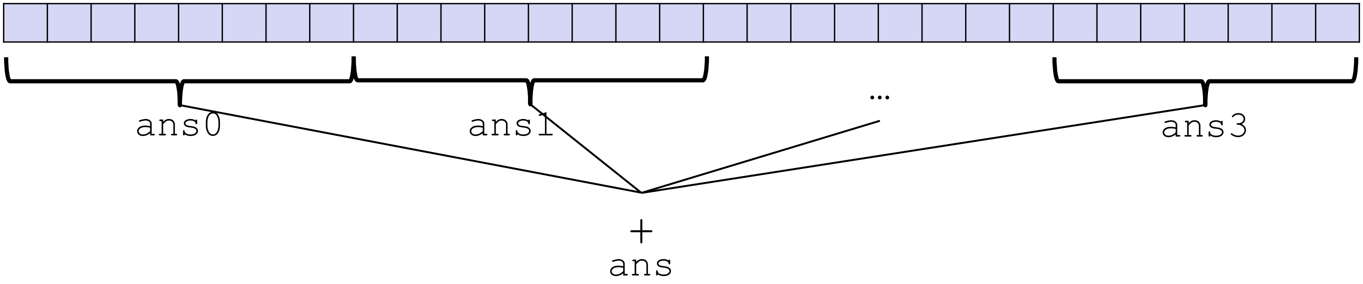
- First, we'll learn basic multithreading with java.lang.Thread
  - Then we'll discuss a different library (used in p3): ForkJoin

- To get a new thread to <u>start</u> executing something:
  1. Define a subclass `C` of `java.lang.Thread`, and override its `run()` method
  2. Create an instance of class `C`
  3. Call that object's `start()` method
     - `start()` creates a new thread and executes `run()` as its "main"

- What if we called C's `run()` method instead?
  - Normal method call executed in the current thread

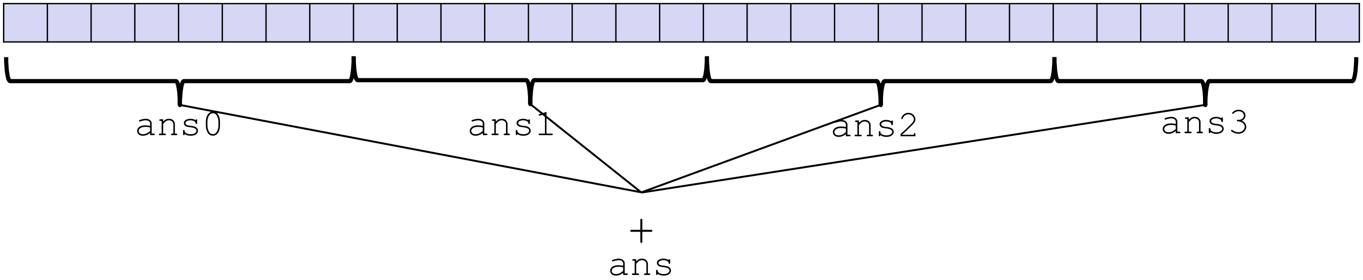
Our Running Example: Summing a Large Array

- *Example*: Sum all the elements of a very large array
- *Idea*: Have n threads simultaneously sum a portion of the array
  a) Create n ***thread objects***, each given a portion of the work
  b) Call `start()` on each object to actually ***execute*** it in parallel
  c) ***Wait*** for each thread to finish
  d) Combine their answers (via addition) to obtain the ***final result***



Attempt #1: Summing a Large Array

- (Warning: this is an inferior first approach)
- Have 4 threads simultaneously sum a portion of the array
  a) Create 4 ***thread objects***, each given a 1/4 of the work
  b) Call `start()` on each object to actually ***execute*** it in parallel
  c) ***Wait*** for each thread to finish
  d) Combine their answers (via addition) to obtain the ***final result***



Attempt #1: Code (1 of 3)

Step 1

```
class SumThread extends java.lang.Thread {
  // We pass arguments to the SumThread instance via
  // member fields that are initialized in the constructor
  int lo;          // input; start index
  int hi;          // input; end index, exclusive
  int[] arr;       // input; the (shared) array

  int ans = 0;     // output; the final sum

  SumThread(int[] a, int l, int h) { lo=l; hi=h; arr=a; }


  @Override
  public void run() { // must have this exact signature
    int i;
    for (i=lo; i < hi; i++)
      ans += arr[i];
  }
}
```

Step 1

- Because we override a no-arguments/no-result `run`, we use member fields to communicate across threads

Attempt #1: Code (2 of 3)

```
class SumThread extends java.lang.Thread {
  int lo, int hi, int[] arr; // input: arguments
  int ans = 0;                // output: result
  SumThread(int[] a, int l, int h) { … }
  public void run(){ … }      // override: implement "main"
}
```

```
int sum(int[] arr){              // can be a static method
  int len = arr.length;
  int ans = 0;
  SumThread[] ts = new SumThread[4];
  for (int i=0; i < 4; i++) { // do parallel computations
    ts[i] = new SumThread(arr,i*len/4,(i+1)*len/4);
}
  for (int i=0; i < 4; i++) { // combine partial results
    ans += ts[i].ans;
  }
  return ans;
}
```

Step 2

Attempt #1: Code (3 of 3)

```
class SumThread extends java.lang.Thread {
  int lo, int hi, int[] arr; // input: arguments
  int ans = 0;               // output: result
  SumThread(int[] a, int l, int h) { … }
  public void run(){ … }     // override: implement "main"
}
```

```
int sum(int[] arr){                // can be a static method
  int len = arr.length;
  int ans = 0;
  SumThread[] ts = new SumThread[4];
  for (int i=0; i < 4; i++) { // do parallel computations
    ts[i] = new SumThread(arr,i*len/4,(i+1)*len/4);
    ts[i].start();                 // call start(), not run!!!
  }
  for (int i=0; i < 4; i++) { // combine partial results
    ans += ts[i].ans;
  }
  return ans;
}
```

Step 3

Introducing java.lang.Thread … part 2

- To get a new thread to start executing something:
  1. Define a subclass `C` of `java.lang.Thread`, and override its `run()` method
  2. Create an instance of class `C`
  3. Call that object's `start()` method
     1. `start()` creates a new thread and executes `run()` as its "main"

- To finish the threads' computation:
  4. ***Wait*** for each thread to finish using `join()`
  5. Optionally: combine their answers to obtain the ***final result***

Attempt #2: Code

Step 1

```
class SumThread extends java.lang.Thread {
  int lo, int hi, int[] arr; // input: arguments
  int ans = 0;               // output: result
  SumThread(int[] a, int l, int h) { … }
  public void run(){ … }     // override: implement "main"
}
```

Step 1

```
int sum(int[] arr){                  // can be a static method
  int len = arr.length;
  int ans = 0;
  SumThread[] ts = new SumThread[4];
  for (int i=0; i < 4; i++) { // do parallel computations
    ts[i] = new SumThread(arr,i*len/4,(i+1)*len/4);
    ts[i].start();                   // call start(), not run!!!
  }
  for (int i=0; i < 4; i++) { // combine partial results
    ts[i].join();                    // wait for thread to finish
    ans += ts[i].ans;
  }
  return ans;
}
```

Step 2 Step 3 Step 4 Step 5

join(): Our "wait" method for Threads

- Framework implements functionality you couldn't on your own
  - E.g.: **`start`**, which creates a new thread
- You "fill in the blanks" for the framework
  - E.g.: we implement **`run()`**, telling Java what to do in the thread
- Something else you can't implement: thread coordination
  - So it also provides the **`join()`** method!
  - **`join()`** blocks the caller until/unless the thread instance is done executing (i.e.: the call to **`run()`** finishes)

Incidentally …

- This code has a compile error because **`join`** may throw **`java.lang.InterruptedException`**
  - In basic parallel code, should be fine to catch-and-exit

```
int sum(int[] arr){                    // can be a static method
  int len = arr.length;
  int ans = 0;
  SumThread[] ts = new SumThread[4];
  for (int i=0; i < 4; i++) { // do parallel computations
    ts[i] = new SumThread(arr,i*len/4,(i+1)*len/4);
    ts[i].start();                     // call start(), not run!!!
  }
  for (int i=0; i < 4; i++) { // combine partial results
    ts[i].join();                      // wait for thread to finish
    ans += ts[i].ans;
  return ans;
}
```

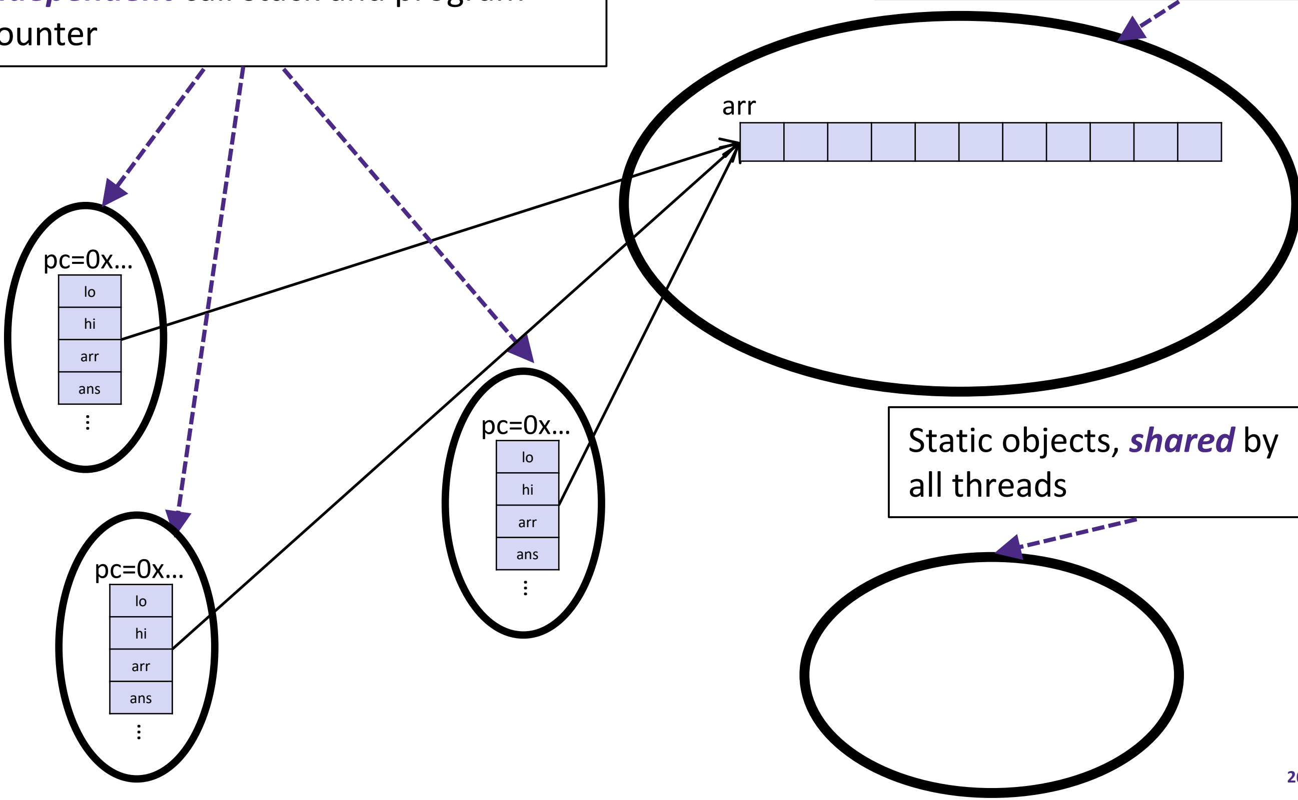
Where is the Shared Memory? Local Memory?

- Our program (implicitly!) shares memory
  - **`lo`** & **`hi`** are inputs: written by "main" thread, read by helper thread
  - **`arr`** reference also an input, but its referred array was shared
  - **`ans`** is an output: written by helper thread, read by "main" thread
- Our program also has thread-local memory
  - Each `SumThread` has a counter it doesn't share with other threads

Summing a Large Array: Shared Memory

Multiple threads, each with its own ***independent*** call stack and program counter

Heap for allocated objects, ***shared*** by all threads



Static objects, ***shared*** by all threads

`join()`ing Forces Against Race Conditions

- Our program (implicitly!) shares memory
  - **`ans`** is an output: written by helper thread, read by "main" thread

- When using shared memory, you must avoid race conditions
  - If "main" thread didn't `join()` before using **`ts[i].ans`**, result is undefined!
  - While studying parallelism (now), we'll stick with **`join`**
  - With concurrency (later), we will learn other ways to synchronize

Lecture Outline

- Shared Memory with Threads

- Concurrency Frameworks in Java
  - Introducing java.lang.Thread
  - **Writing *good* parallel code**
  - Improving java.lang.Thread
    - Asymptotically
    - Constants
  - ForkJoin Library

Issues with Our Earlier Approach (1 of 3)

1. Want code to be portable and efficient across platforms
   - So at the *very very* least, parameterize by the number of threads

```
int sum(int[] arr, int numTs){
  int len = arr.length;
  int chunkLen = arr.length/numTs;
  int ans = 0;
  SumThread[] ts = new SumThread[numTs];
  for(int i=0; i < numTs; i++) {
    ts[i] = new SumThread(arr, i*chunkLen,(i+1)*chunkLen);
    ts[i].start();
  }
  for(int i=0; i < numTs; i++) {
    ts[i].join();
    ans += ts[i].ans;
  }
  return ans;
}
```

Issues with Our Earlier Approach (2 of 3)

2. Want to use only executors "available to you now"
   - Executors used by other programs or threads aren't available!
     - Maybe caller is also using parallelism?
     - Number of available cores changes even while your threads run
   - E.g.: if you have 3 available executors and using 3 threads would take time **X**, then creating 4 threads would take time **1.5X**
     - Example: 12 units of work, 3 executors
       - Dividing work into 3 chunks will take 4 units of time
       - Dividing work into 4 chunks will take 3*2 units of time

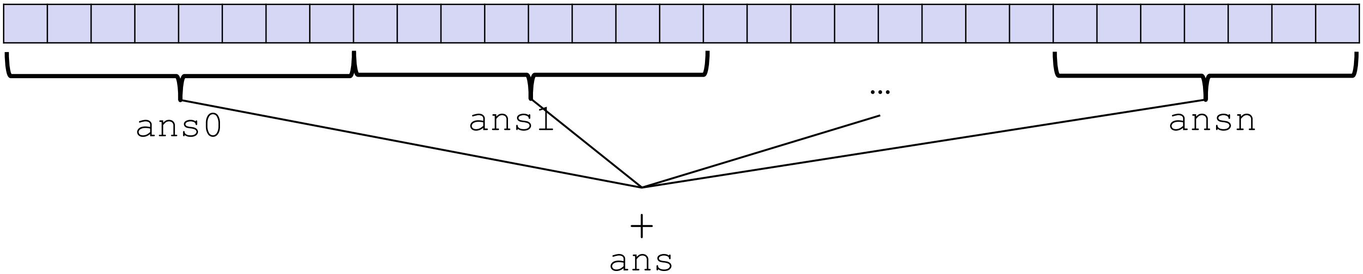
```
// numThreads == numExecutors is bad
// if some are needed for other things
int sum(int[] arr, int numTs){
   …
}
```

Issues with Our Earlier Approach (3 of 3)

3. In general, subproblems take different amounts of time
   - Sometimes drastically different!
   - If we create 100 threads but one chunk takes much much longer, we won't get a ~100x speedup
     - This is called a ***load imbalance***
   - E.g.: apply `f()` to array elements, but `f()` is slower for some elts
     - `f()` checks if the element is prime?

A Better Approach: Smaller Chunks

- The solution: *cut up our problem into many small chunks*
  - We want far more chunks than the number of executors!
  - … but this will require changing our algorithm



1. *Portable?* Yes! (Substantially) more chunks than executors
2. *Adapts to Available executors?* Yes! Hand out chunks as you go
3. *Load Balanced?* Yes(ish)! Variation is smaller if chunks are small

Lecture Outline

- Shared Memory with Threads

- Concurrency Frameworks in Java
  - Introducing java.lang.Thread
  - Writing good parallel code
  - Improving java.lang.Thread
    - **Asymptotically**
    - Constants
  - ForkJoin Library

A Better Approach: Abandoning java.lang.Thread

- For this specific problem (and for p3), the constants for Java's built-in thread framework are not great
- Plus, there's complexity in Java's Thread framework that confuse rather than illuminate

Naïve Thread Creation/Joining (1 of 2)

- Suppose we create 1 thread to process 1000-element chunks

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

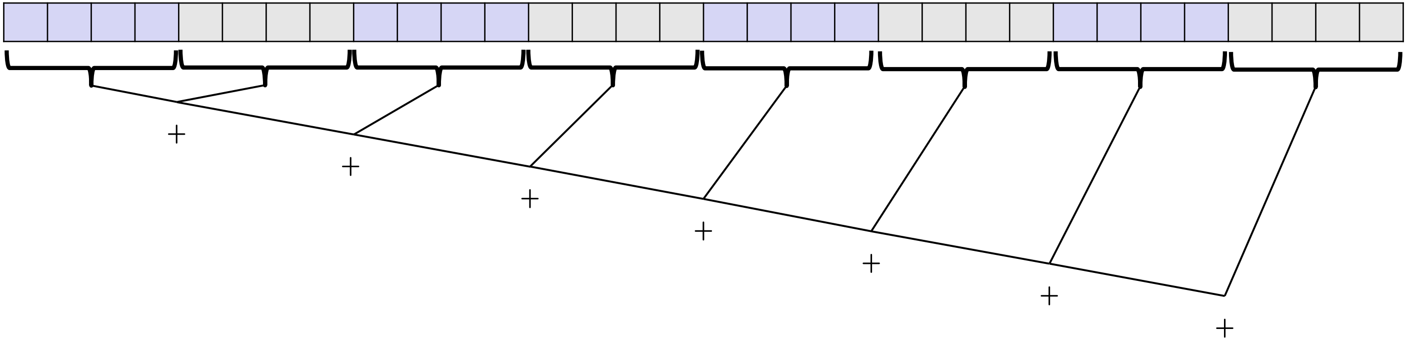
- "Combine results" step has `arr.length/1000` additions
  - Θ(N) to combine!
  - Previously, we had only 4 pieces (Θ(1) to combine)

- Will a Θ(N) algorithm to create threads/combine results be a bottleneck?

Naïve Thread Creation/Joining (2 of 2)

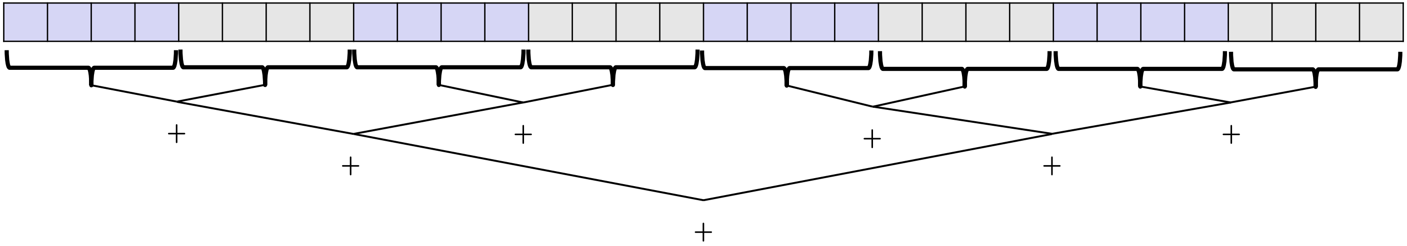
- Yes! The combining has now become a bottleneck

- The calls to `run()` can execute in parallel, but combining intermediate results is still sequential!



Smarter Thread Creation/Joining: Divide and Conquer!

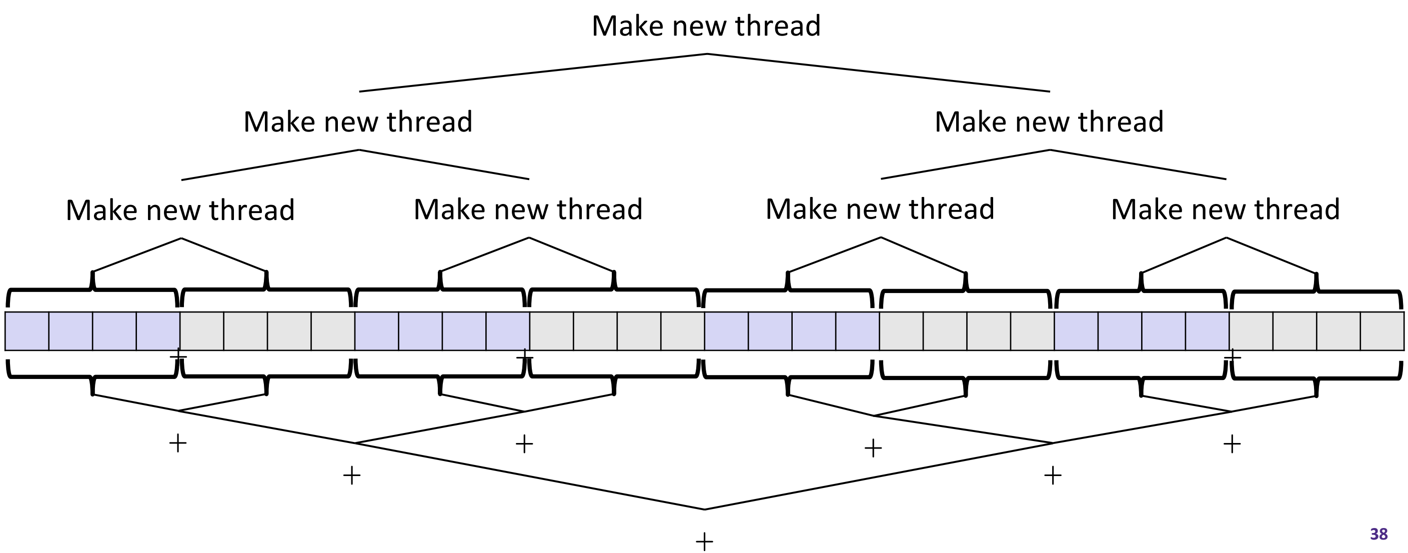
- Divide and Conquer:
  - "Grows" the number of threads to fit the problem
  - Uses parallelism for the recursive calls *and combining*



- This style of parallel programming is called "fork/join"

Smarter Thread Creation/Joining with Fork/Join

- Fork/Join Phases:
  1. Divide the problem
     - Start with full problem at root
     - Make two new threads, halving the problem, until size is at cutoff
  2. Combine answers as we return from recursion



Fork/Join-style Parallelism (1 of 3)

```
class SumThread extends java.lang.Thread {
  // … member fields and constructors elided …
  public void run() {  // override: implement "main"
    if(hi - lo < SEQUENTIAL_CUTOFF) {
      // Just do the calculation in this thread
      for (int i=lo; i < hi; i++)
        ans += arr[i];
    }
    else {
      // Create two new threads to calculate the left and right sums
      SumThread left = new SumThread(arr, lo, (hi+lo)/2);
      SumThread right= new SumThread(arr, (hi+lo)/2, hi);
      left.start();
      right.start();

      // Combine their results
      left.join();  // don't move this up a line (why?)
      right.join();
      ans = left.ans + right.ans;
    }
  }
}
```

Fork/Join-style Parallelism (2 of 3)

```
class SumThread extends java.lang.Thread {
  int lo, int hi, int[] arr; // input: arguments
  int ans = 0;               // output: result
  SumThread(int[] a, int l, int h) { … }
  public void run(){ … }     // override: implement "main"
}
```

```
int sum(int[] arr) {
  SumThread t = new SumThread(arr, 0, arr.length);// just 1 obj since
  t.run();                                        // we don't need
  return t.ans;                                   // parallelism to
}                                                 // start recursion
```

- The computation and the result-combining are both in parallel
  - Using recursive divide-and-conquer makes this natural
  - Easier to write *and* more efficient asymptotically!

Fork/Join-style Parallelism (3 of 3)

- What's up with the sequential cutoff?
  - QuickSort and MergeSort switch to InsertionSort because "the constants are better"
  - Similarly, Fork/Join-style parallelism switches to sequential execution because "the constants are better"

  - In sorting, we said that the recursive call was "expensive"; in parallelism, it's the thread creation/destruction
    - In both cases, it's the setup/teardown overhead!

Fork/Join-style Parallelism Really Works!

- Key idea is parallelizing thread-creation and result-combining
  - If enough executors, runtime is **height of the tree:** $O(\texttt{log}\ n)$
    - Optimal, and exponentially faster than sequential $O(n)$
  - Relies on operations being associative (like +)
- We'll write all our parallel algorithms in this style
  - But using a special library engineered for this style