

Intro to Multithreading

CSE 332 Spring 2020

Instructor: Hannah C. Tang

Teaching Assistants:

Annie Mao

Howard Xiao

Lucy Jiang

Chris Choi

Jade Watkins

Nathan Lipiarski

Ethan Knutson

Khushi Chaudhari

Richard Jiang

Announcements

- ❖ 🍰🍰🍰 No exercises this week! 🍰🍰🍰
- ❖ Quiz this week tho: 🙄🗨️🙄
- ❖ And also P2 checkpoint #2
- ❖ Tips for asking for help in Piazza / Office Hours:
 - Please push your code to gitlab *before* asking for help
 - Include your group name in your posts
- ❖ Lecture questions? pollev.com/cse332

Learning Objectives

- ❖ Understand concurrency vs parallelism
- ❖ Understand the shared memory model
- ❖ Become familiar with the “raw” `java.lang.Thread` model

Lecture Outline

- ❖ **Changing Another Major Assumption**
 - Definitions: Parallelism vs Concurrency
- ❖ Shared Memory with Threads
- ❖ Concurrency Frameworks in Java
 - Introducing `java.lang.Thread`
 - Writing *good* parallel code

Sequential Programming: A Major Assumption

- ❖ So far, most / all of your study has assumed:

One thing happened at a time

- ❖ This is **sequential programming**: everything in one sequence
- ❖ Removing this assumption creates major challenges & opportunities
 - *Programming*: How to divide work among **threads of execution** and coordinate (**synchronize**) among them
 - *Algorithms*: How to utilize parallel activity to gain speed
 - More **throughput**: work done per unit time
 - *Data structures*: May need to support **concurrent access**
 - ie, multiple threads operating on data at the same time

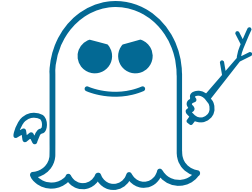
A Simplified View of Computing History

- ❖ Writing *correct* and *efficient* multithreaded code is much more difficult than for sequential code
 - Especially in common languages like Java and C
- ❖ Roughly 1980-2005, computers got exponentially faster
 - Sequentially-written programs doubled in speed every couple years
 - So there was little motivation to write non-sequential code
- ❖ But nobody knows how to continue making computers faster
 - Increasing clock rate generates too much heat
 - Relative cost of memory access is too high
- ❖ But we *can* continue “making wires exponentially smaller” (“**Moore’s ‘Law’**”)
 - Result: multiple processors on the same chip (“**multicore**”)

What to do with Multiple Processors/Cores?

- ❖ Next computer you buy will likely have 4 cores
 - Wait a few years and it will be 8, 16, 32, ...
 - The chip companies have decided to do this (not a “law”)

- ❖ What can you do with these processors?
 - Run multiple, totally different, programs at the same time
 - Already do that? It certainly *appears* that way, thanks to **time-slicing**
 - Run multiple, possibly different, tasks at the same time in one single program
 - Our focus for the next few lectures; it’s more difficult!
 - Requires rethinking everything from asymptotic complexity to how to implement data-structure operations



Lecture Outline

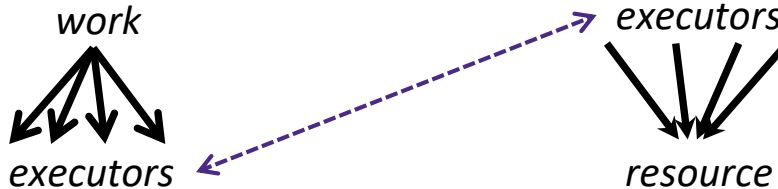
- ❖ Changing Another Major Assumption
 - **Definitions: Parallelism vs Concurrency**
- ❖ Shared Memory with Threads
- ❖ Concurrency Frameworks in Java
 - Introducing `java.lang.Thread`
 - Writing *good* parallel code

Parallelism vs. Concurrency

- ❖ Note: Terms not yet standard but precision here is essential
 - (many programmers confuse these concepts)

Parallelism: Use extra executors to solve a problem faster

Concurrency: Manage access to shared resources



- ❖ There is some connection (confusion!) between them:
 - We commonly use threads for both parallelism and concurrency
 - If parallel computations access shared resources, the concurrency needs to be managed

Parallelism vs Concurrency: An Analogy

- ❖ **Sequential:** A program is like a cook making dinner
 - *One cook:* Makes gravy and stuffing one at a time!
- ❖ **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)
 - Increase throughput via simultaneous execution!
 - Too many cooks means you spend all your time coordinating
- ❖ **Concurrency:** *“We need to manage a shared resource”*
 - *Multiple cooks:* One cook per dish, but only one cutting board
 - Correctness: Don’t want spills or ingredient mixing
 - Efficiency: Who should use the boards and in what order?

efficiency

correctness

Parallelism Example

- ❖ **Parallelism**: Using extra executors to solve a problem faster
- ❖ *Pseudocode* for summing an array:
 - No such 'FORALL' construct, but we'll see something similar
 - Bad style, but with 4 processors may get roughly 4x speedup

```
int sum(int[] arr) {
    int[] res = new int[4];
    int len = arr.length;
    FORALL (i=0; i < 4; i++) { // parallel iterations
        res[i] = sumRange(arr, i*len/4, (i+1)*len/4);
    }
    return res[0] + res[1] + res[2] + res[3];
}

int sumRange(int[] arr, int lo, int hi) {
    int result = 0;
    for(int j=lo; j < hi; j++)
        result += arr[j];
    return result;
}
```

Bad interleaving:

Thread 1
Node n1 = new Node();
head = n1;

Thread 2
Node n2 = new Node();

head = n2;
BOOM! We've lost n1!!!

Concurrency Example

- ❖ **Concurrency:** *Correctly and efficiently* manage access to shared resources, from multiple possibly-simultaneous clients
 - Correctness: prevent bad inter-leavings
 - Efficiency: do not prevent good concurrency
 - One 'solution' to preventing bad inter-leavings is to do it all sequentially
- ❖ *Pseudocode* for a separate chaining hashtable

```
class Hashtable<K,V> {
  ...
  void add(K key, V value) {
    int bucket = hash(key);
    // 1. Prevent other add/finds in table[bucket]
    // 2. Do the insertion
    // 3. Re-enable access to table[bucket]
  }
  V find(K key) {
    // Similar to add(), but can allow concurrent
    // lookups to the same bucket  }
  }
}
```

Lecture Outline

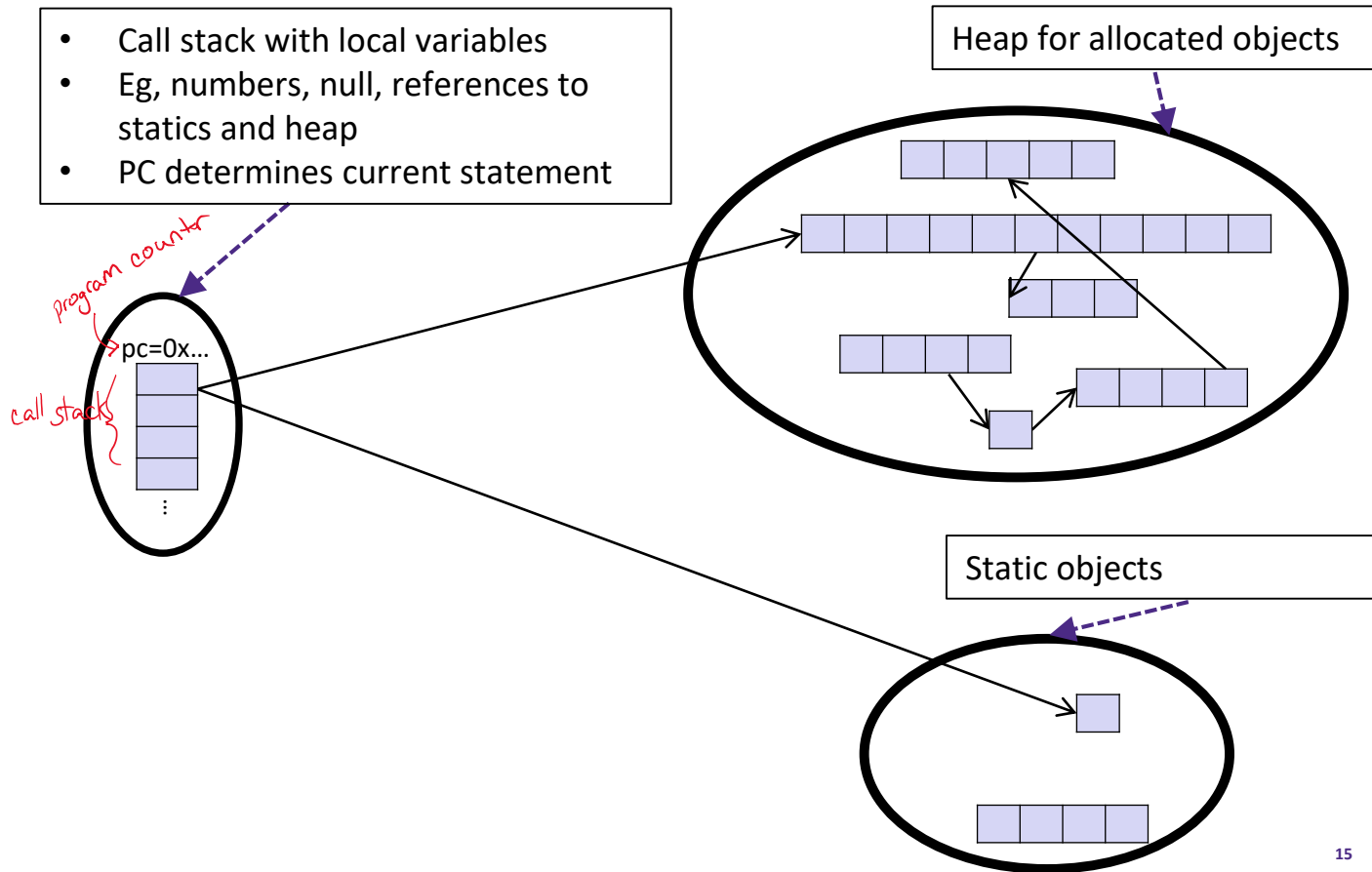
- ❖ Changing Another Major Assumption
 - Definitions: Parallelism vs Concurrency
- ❖ **Shared Memory with Threads**
- ❖ Concurrency Frameworks in Java
 - Introducing `java.lang.Thread`
 - Writing *good* parallel code

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

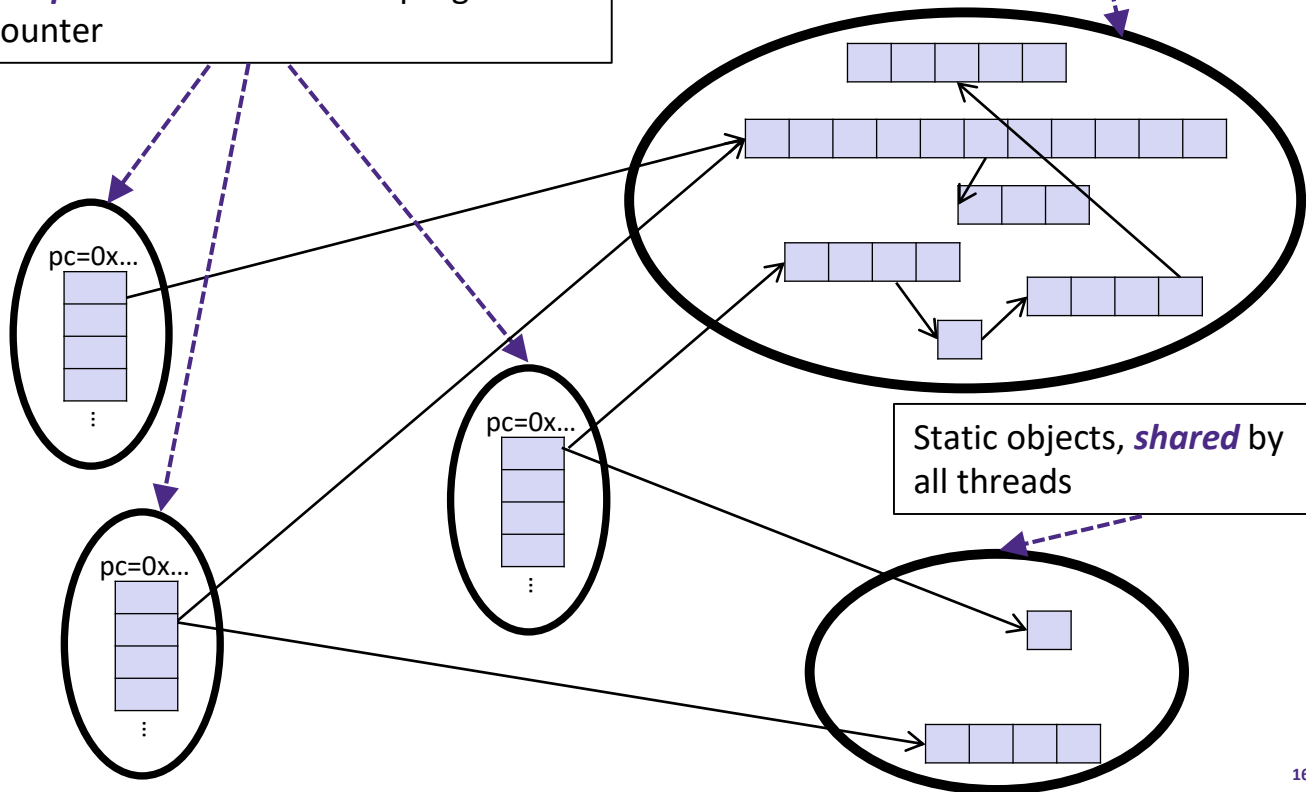


Shared Memory with Threads

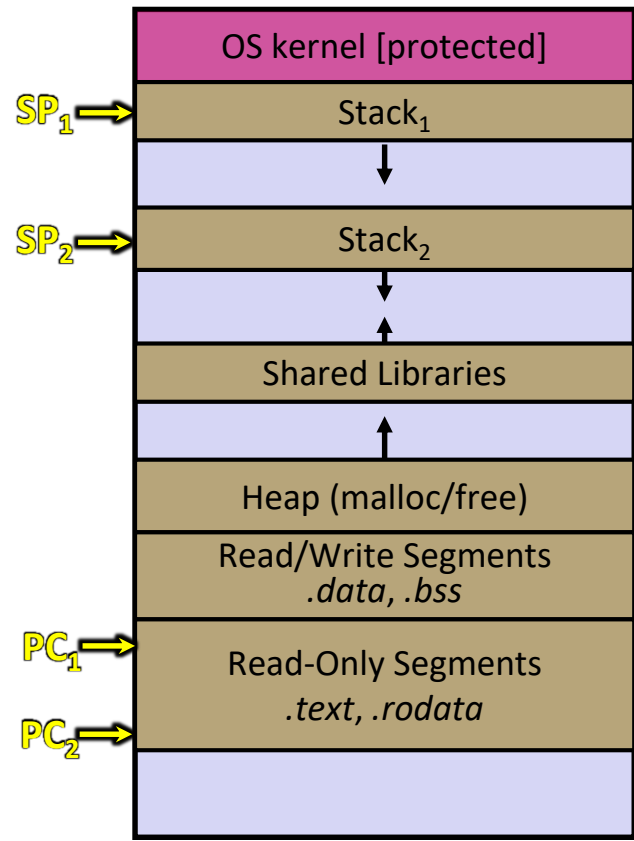
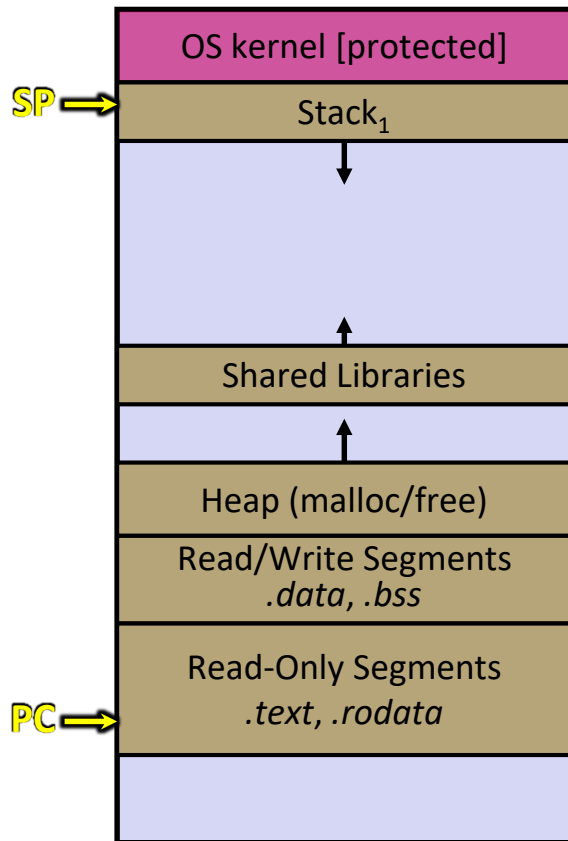
Multiple threads, each with its own *independent* call stack and program counter

Heap for allocated objects, *shared* by all threads

Static objects, *shared* by all threads



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

Lecture Outline

- ❖ Changing Another Major Assumption
 - Definitions: Parallelism vs Concurrency
- ❖ Shared Memory with Threads
- ❖ Concurrency Frameworks in Java
 - **Introducing `java.lang.Thread`**
 - Writing *good* parallel code

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)

Introducing `java.lang.Thread`

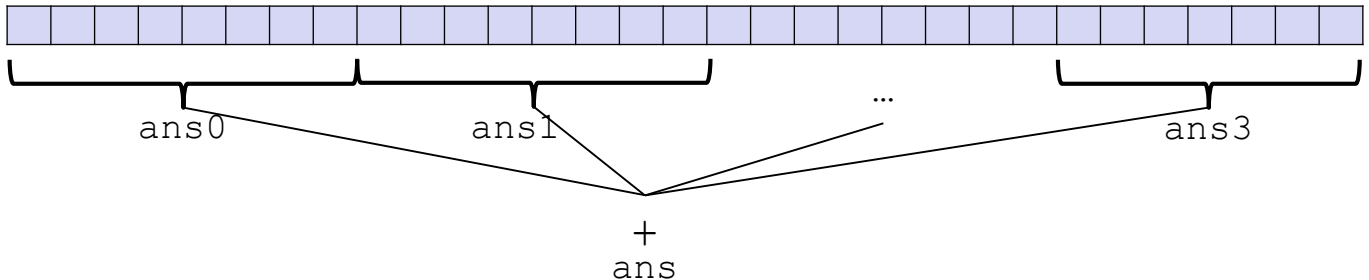
- ❖ First, we'll learn basic multithreading with `java.lang.Thread`
 - Then we'll discuss a different library (used in p3): `ForkJoin`

- ❖ To get a new thread to 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
 - `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

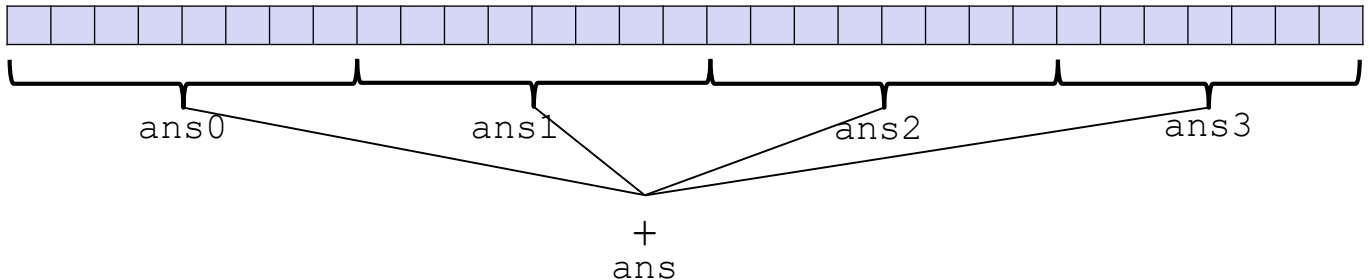
Our Running Example: Summing a Large Array

- ❖ *Example*: Sum all the elements of a very large array
- ❖ *Idea*: Have n threads simultaneously sum a portion of the array
 - Create n **thread objects**, each given a portion of the work
 - Call `start()` on each object to actually **execute** it in parallel
 - **Wait** for each thread to finish using its `join()` method
 - 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
 - Create 4 *thread objects*, each given a 1/4 of the work
 - Call `start()` on each object to actually *execute* it in parallel
 - *Wait* for each thread to finish using its `join()` method
 - Combine their answers (via addition) to obtain the *final result*



Attempt #1: Code (1 of 2)

Step #: subclass Thread & override run()

```
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
        for (int i=lo; i < hi; i++)
            ans += arr[i];
    }
}
```

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

Attempt #1: Code (2 of 2)

STEP 1: instantiate subclass

```
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 2: instantiate subclass

```
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;
}
```

Attempt #2: Code

```
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 3: Call start

```
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;
}
```

Attempt #3: Code

```
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
        ts[i].join();           // wait for thread to finish
        ans += ts[i].ans;
    }
    return ans;
}
```

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)
 - If it didn't, we would have a *race condition* on **ts[i].ans**

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?

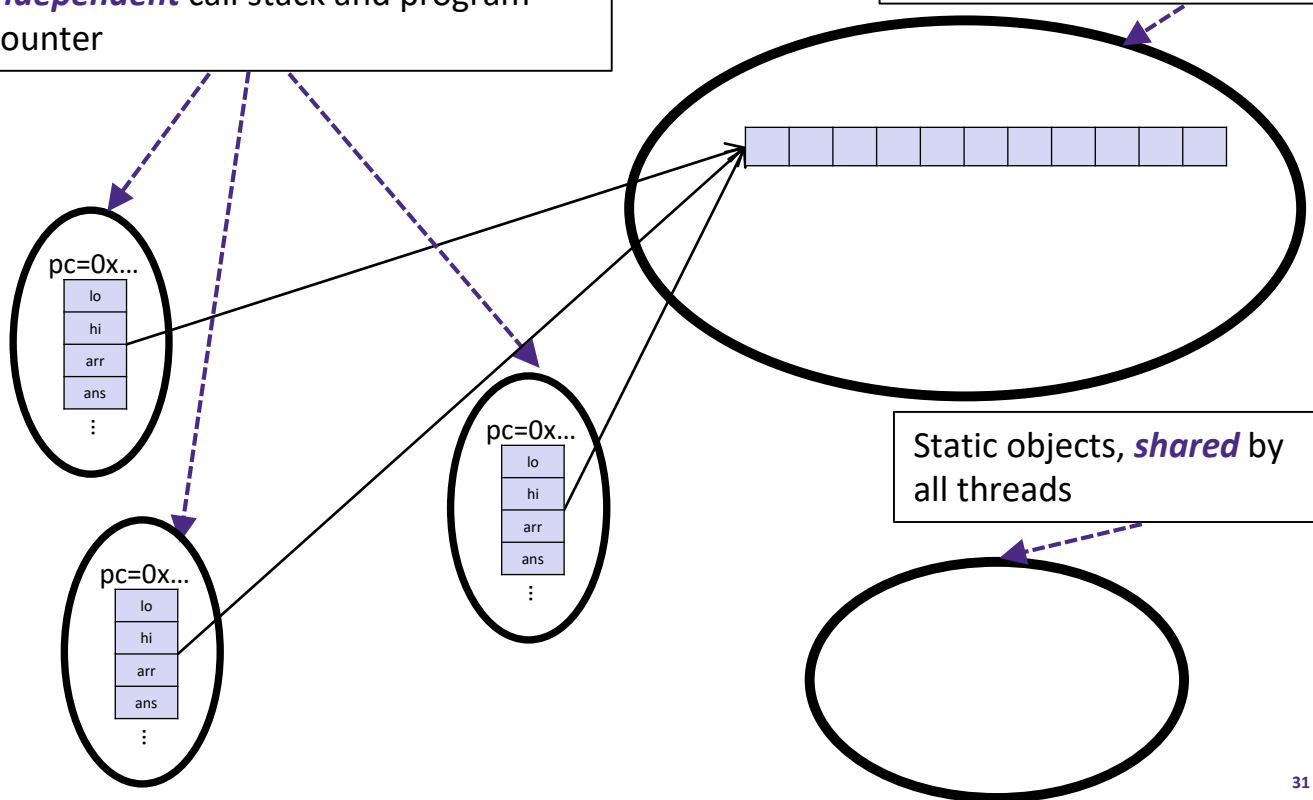
- ❖ Our program (implicitly!) shares memory
 - `lo` & `hi` are inputs: written by “main” thread, read by helper thread
 - `arr` reference also an input, but its referred array was shared
 - `ans` is an output: written by helper thread, read by “main” thread
- ❖ Our program didn't have thread-local memory
 - But you could imagine `SumThread` containing a local variable (maybe a loop counter?) which it doesn't share with other threads
- ❖ When using shared memory, you must avoid race conditions
 - While studying parallelism (now), we'll stick with `join`
 - With concurrency (later), we will learn other ways to synchronize

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



Lecture Outline

- ❖ Changing Another Major Assumption
 - Definitions: Parallelism vs Concurrency
- ❖ Shared Memory with Threads
- ❖ Concurrency Frameworks in Java
 - Introducing `java.lang.Thread`
 - **Writing *good* parallel code**

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 processors “available to you now”
 - Processors 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 processors and using 3 threads would take time **X**, then creating 4 threads would take time **1.5X**
 - Example: 12 units of work, 3 processors
 - Dividing work into 3 chunks will take 4 units of time
 - Dividing work into 4 chunks will take 3*2 units of time

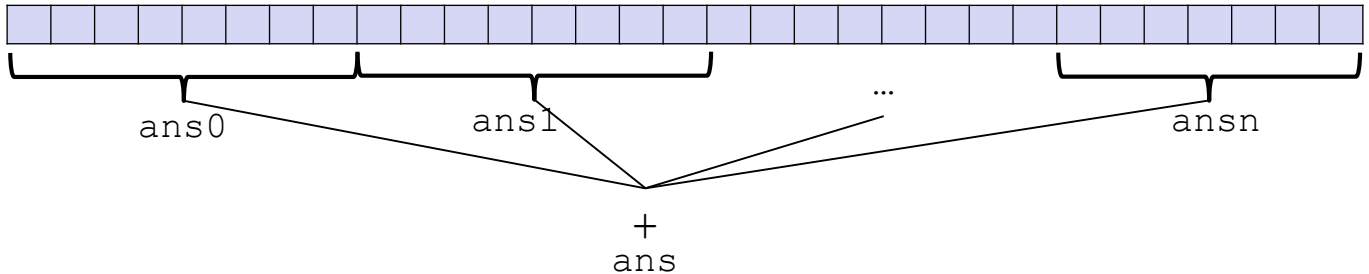
```
// numThreads == numProcessors is bad
// if some are needed for other things
int sum(int[] arr, int numTs){
    ...
}
```

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 processors!
 - ... but this will require changing our algorithm



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

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