Multi-Threading II

Andrew Hu

Administrivia

- Attendance: http://tinyurl.com/uwigem/18sp/attendance
- Please fill out the subteam choice survey
  - Opportunity to apply to be a subteam lead
- Sign up for outreach!
  - See Ishira’s post on #general

Agenda

- Review threads & methods of communicating between them
- Parallelism vs Concurrency
- Methods of concurrency
- Race conditions
- Java-isms
  - Synchronized
  - Runnable
  - Thread Pools

Review: Threads

- When you run a program, it is called a process
- Each process has at least one thread
- Each thread has its own “program counter” AKA what part of the code it is executing now
- Each thread can create more “child” threads

Review: Spawning Threads

```
public class Parent{
static void main() {
  // Prepare the child thread
  Child childThread = new Child(42);
  childThread.start();

  // Do your own thing
  computePrimes();

  System.out.println(“Parent is done”);
}
}
```

```
public class Child extends Thread{

private int message;
public Child(int data){
  this.message = data;
}

public void run(){
  // Do something with the data
  System.out.println(“Data is: “+message);
}
}
```

```
Data is: 42
Parent is done
```

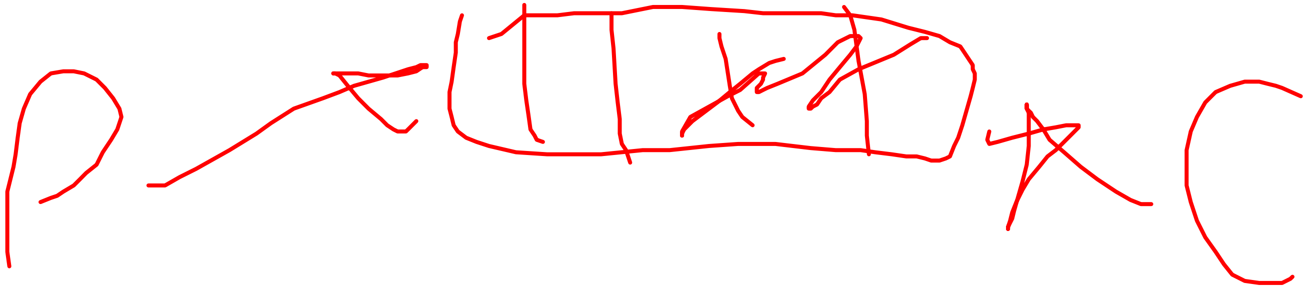
OR

```
Parent is done
```

Spawn Threads with Shared Data

```
public class Parent{
static void main() {
  // Prepare the child thread
  int[] arr = {5, 3, 4, 1, 2};
  System.out.println(arr);
  Child childThread = new Child(arr);
  childThread.start();

  // Do your own thing
  computePrimes();

  System.out.println("Array (hopefully)
                      sorted: "+arr);
}
}
```

```
public class Child extends Thread{

private int[] arr;
public Child(int data){
  this.arr = data;
}

public void run(){
  // Do something with the data
  Collections.sort(arr);
}
}
```



Spawn Threads with Shared Data

```
public class Parent{
static void main() {
   // Prepare the child thread
   int[] arr = {5, 3, 4, 1, 2};
   System.out.println(arr);
   Child childThread = new Child(arr);
   childThread.start();

   // Do your own thing
   computePrimes();

   System.out.println(“Array (hopefully)
                              sorted: “+arr);
}
}
```

```
public class Child extends Thread{

private int[] arr;
public Child(int data){
   this.arr = data;
}

public void run(){
   // Do something with the data
   Collections.sort(arr);
}
}
```

```
5, 3, 4, 1, 2
Array (hopefully) sorted: 1, 2, 3, 4, 5
```

OR

```
5, 3, 4, 1, 2
Array (hopefully) sorted: 5, 3, 4, 1, 2
```

Anatomy of Spawning Threads

```
public class Parent{
static void main() {
    Create the Thread object w/ data

    Call start() on the Thread

    Do something else in the main, while the
    child thread does work
}
}
```

```
public class Child extends Thread{

private int message;
public Child(int data){
  this.message = data;
}

public void run(){
    Process the data passed in
}
}
```

What happens if step 3 depends on step 2 being done?

Sorting Array: Fixed

```
public class Parent{
static void main() {
   // Prepare the child thread
   int[] arr = {5, 3, 4, 1, 2};
   System.out.println(arr);
   Child childThread = new Child(42);
   childThread.start();

   // Do your own thing
   computePrimes();
   arr.wait();

   System.out.println(“Array definitely
                              sorted: “+arr);
}
}
```

```
public class Child extends Thread{

private int[] arr;
public Child(int data){
   this.arr = data;
}

public void run(){
   // Do something with the data
   Collections.sort(arr);
   arr.notify();
}
}
```

```
5, 3, 4, 1, 2
Array definitely sorted: 1, 2, 3, 4, 5
```

Anatomy of Message Passing

```
public class Parent{
static void main() {
```

Create the Thread object w/ shared data

Call start() on the Thread

Do something productive while waiting

Wait for the child thread’s message

```
  System.out.println(“Array definitely
                              sorted: “+arr);
}
}
```

```
public class Child extends Thread{

private int[] arr;
public Child(int data){
  this.arr = data;
}

public void run(){
```

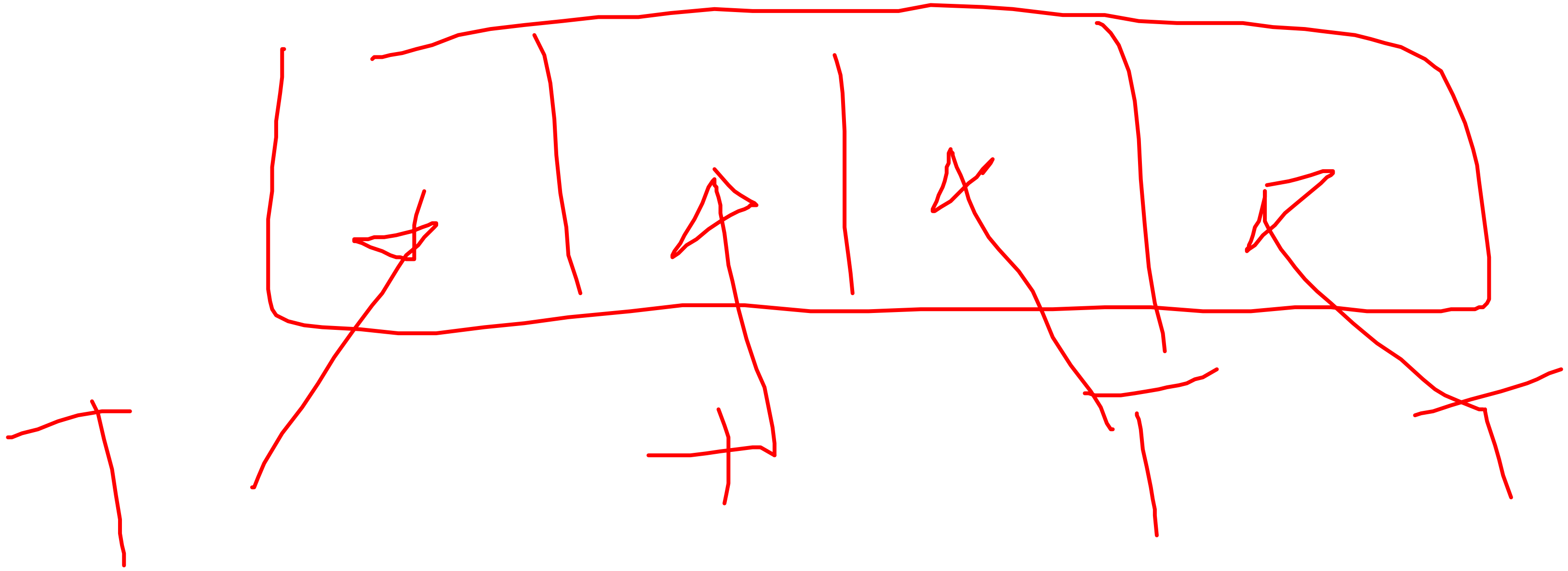
Process the data passed in

Notify the parent thread

```
}
}
```

Parallelism vs Concurrency

- Two different applications of the same concepts
- Using multithreading to be at multiple places in the code at once



Parallelism

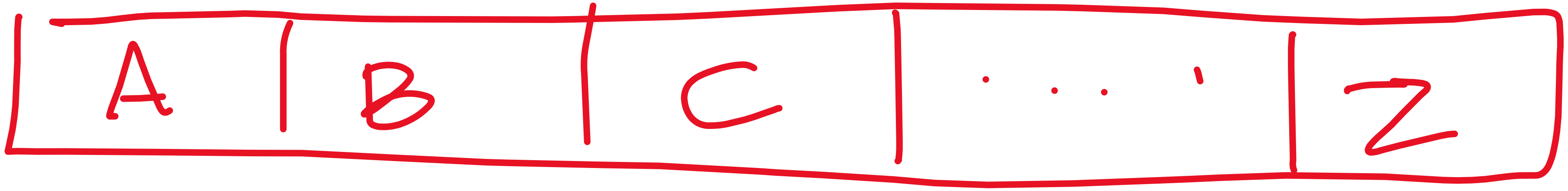
- Using multithreading to split up one tasks amongst many threads
- Example: Summing an array

Concurrency

- Using multithreading to handle different tasks at the same time
- Possibly having multiple threads access the same data (!)
- Example: Web server

Web Server Example

- Suppose it takes 1 minute to service each person’s request
- When people make a request, they are put at the back of the line
- How long to serve request A? request B? request Z?



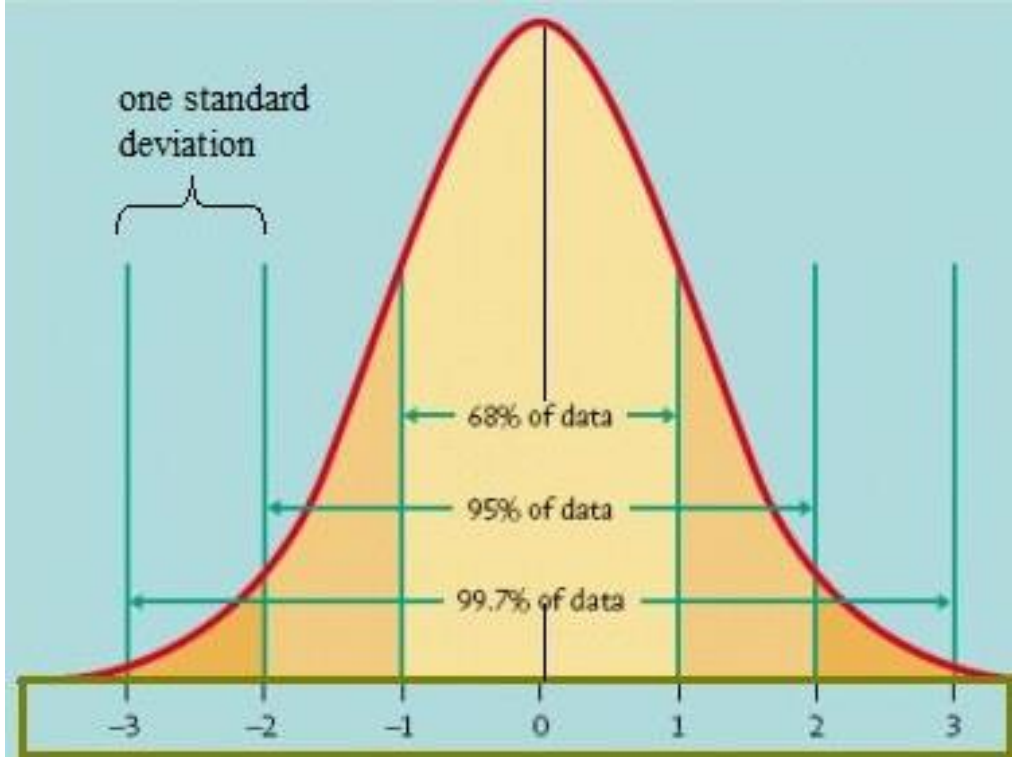
A: 1, Z: 26 can take 1-26 min

How Good Is This?

- Response takes 1-26 minutes
- The mean tells us how long we wait on average
- Standard deviation tells us how much the actual time will vary from the average

$$\text{(mean)} \quad \mu = 13.5$$
$$\text{(std dev)} \; \sigma = 7.5$$

A	B	C	. . . ,	Z



How We Got It

$$\boxed{E[X] = \frac{26+1}{2} = \boxed{13.5}}$$

$$E[X^2] = \sum_{i=1}^{26} i^2 p_X(i)$$

$$= \frac{1}{\cancel{26}} \frac{\cancel{26}(27)53}{6}$$

$$= 238.5$$

$$Var(X) = E[X^2] - \mu^2$$

$$Var(X) = 238.5 - (13.5)^2$$

$$= 56.25 = \sigma^2$$

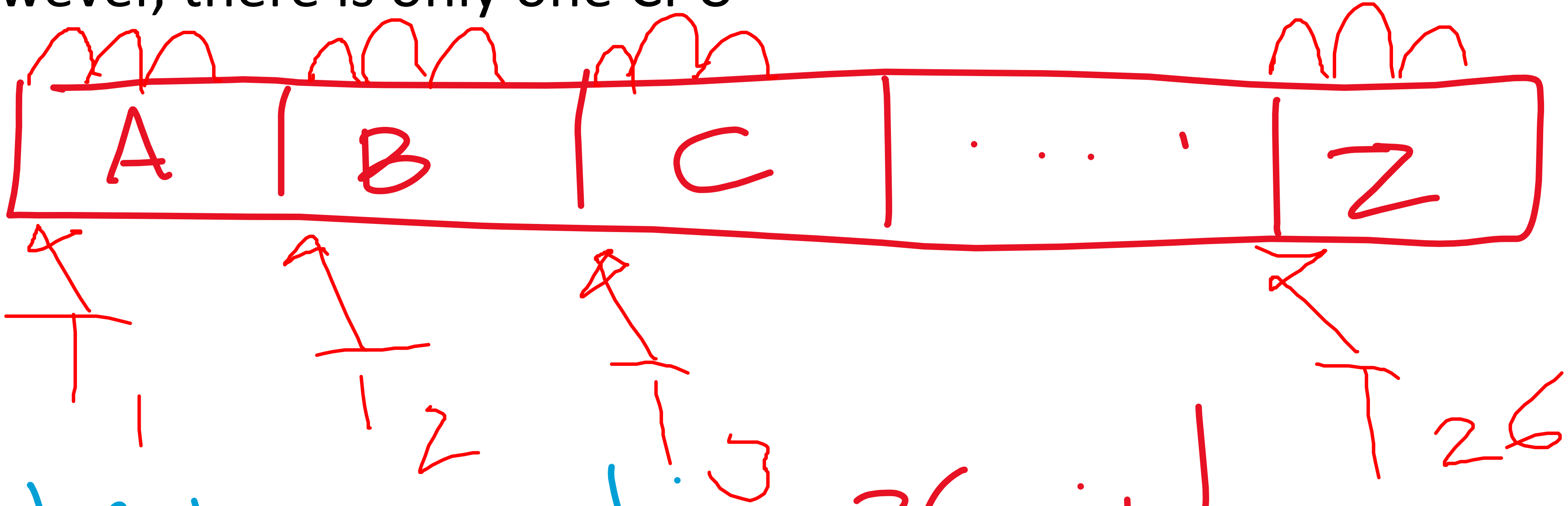
$$\sigma = \sqrt{56.25}$$

$$\boxed{\sigma = 7.5}$$

Web Server with Concurrency

- Now, say that we create a new thread for each request

- However, there is only one CPU



Time per request: 26 min!

Analysis

- $\mu = 26, \sigma = 0$ (w concurrency)
- $\mu = 13.5, \sigma = 7.5$ (w/o concurrency)
- This method has a higher mean, but lower standard deviation

Speed Frustrations

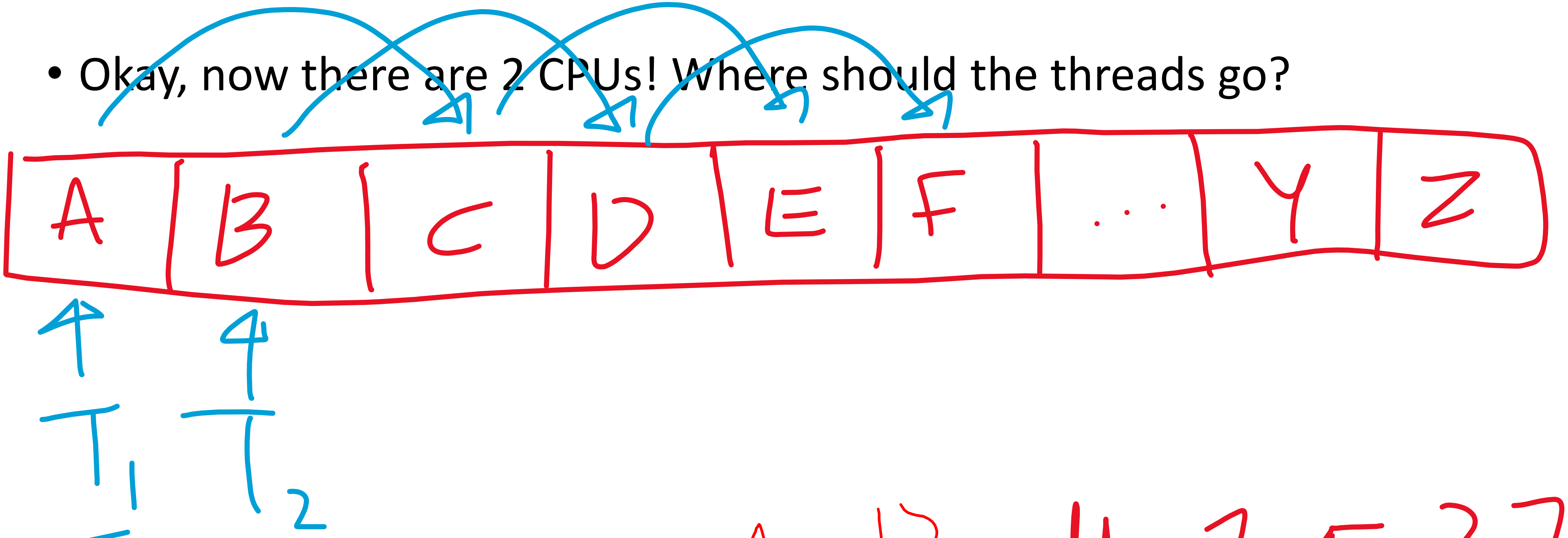
- It looks like all concurrency has done is made the program slow for everyone!
- Concurrency is *not* useful in every situation
- We've just seen there are some situations, where concurrency might make the situation worse!

When Concurrency Can Help

- Two major situations
  - Utilizing multiple cores
  - Blocking calls
- Another form of concurrency
  - Distributed systems (shallow overview)

Web Server with Multi-Core Concurrency

- Okay, now there are 2 CPUs! Where should the threads go?



Time per request: 1-13 $\mu=7$ $\sigma=3.7$

Analysis

- (1 core, par) $\mu = 26, \sigma = 0$

Best guarantee

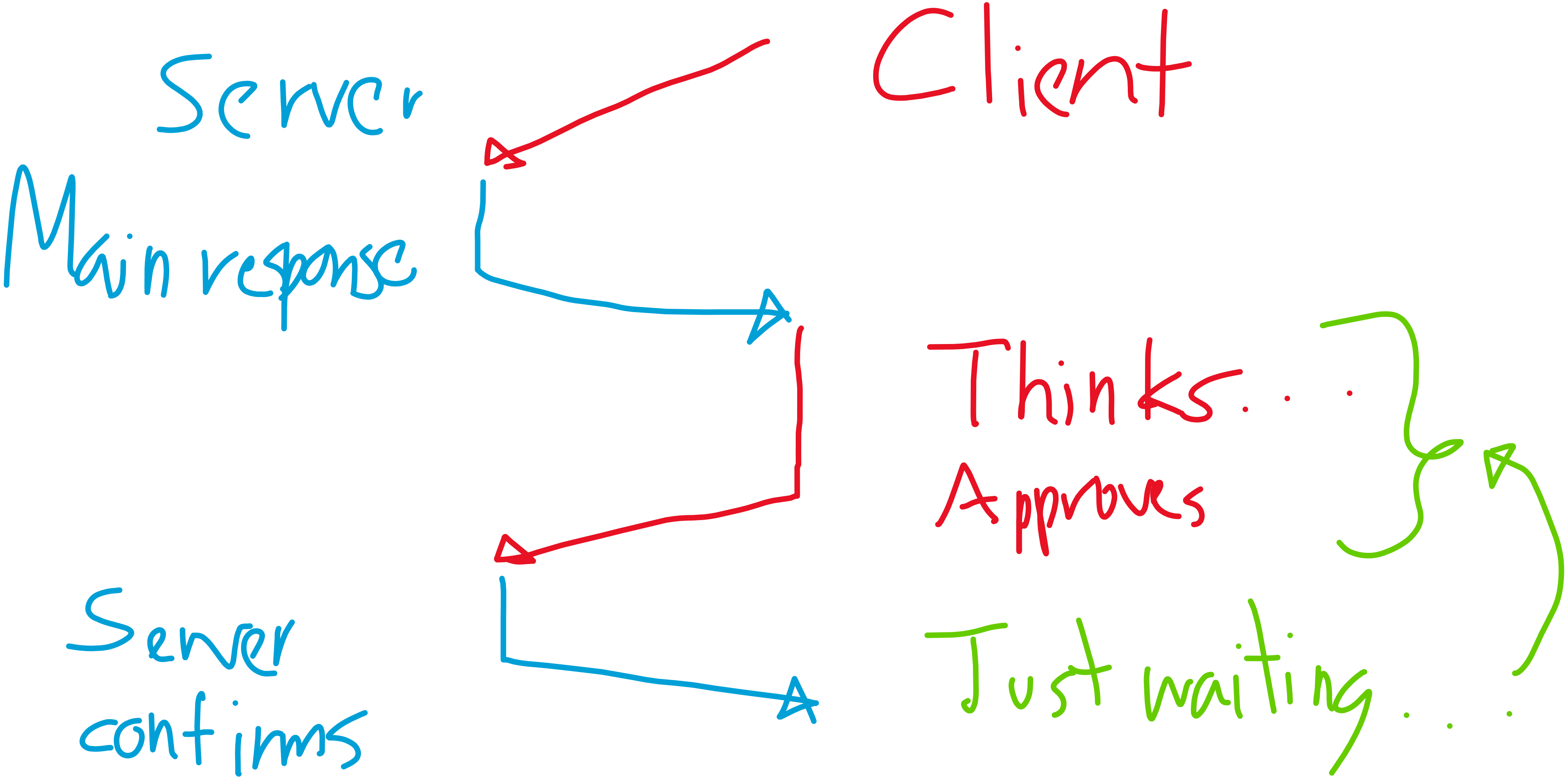
- (1 core, seq) $\mu = 13.5, \sigma = 7.5$
- (2 cores, par) $\mu = 7, \sigma = \sqrt{14} \approx 3.74$

Best avg. time

Blocking Function Calls

- If a function call is blocking, it will wait until something happens
- This event could be
  - Response from a web server
  - Get input from the keyboard (Scanner)
- If we are a web server, we may be both sending and receiving data to complete a request

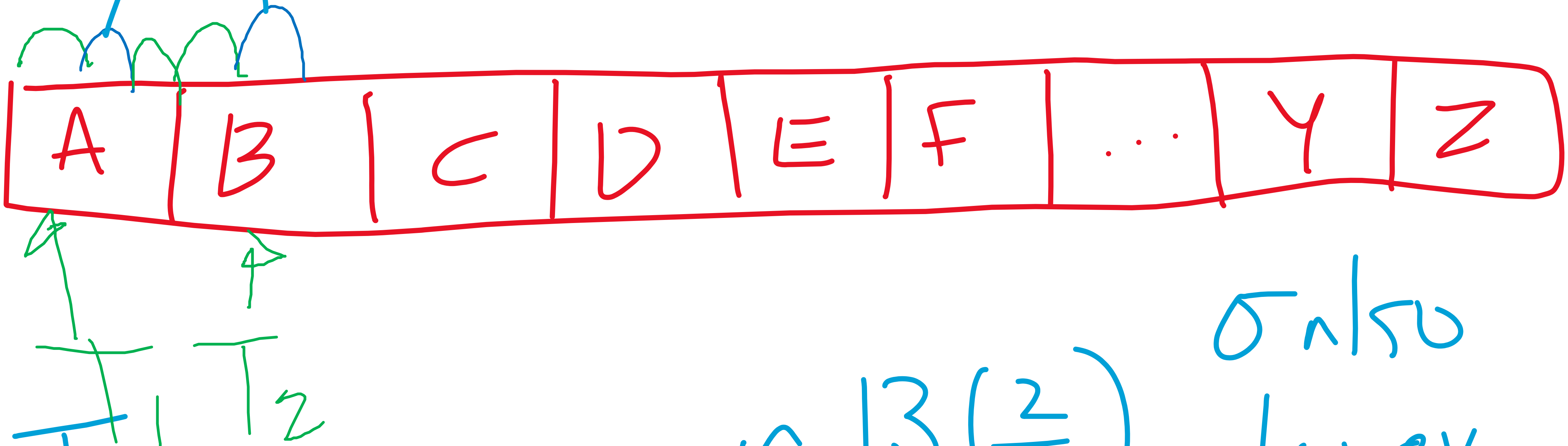
Web Server I/O



Web Server with Non-Blocking I/O

IO

- Now, we can ignore a thread while it is waiting on a response



Time per request: $\sim 13\left(\frac{2}{3}\right)$

σ also lower

Keeping Multiple Threads Alive

- It turns out creating threads is expensive
- Java has something called Thread Pools
- A bunch of threads are created at startup
- You can give new tasks to the thread pool and it will manage the thread overhead

Test

- Test

Hello, there!

```
int main() {
        // Code goes here
}
```