

Recall CS680's HW on Distance Metrics

- Suppose you are implementing a machine learning (ML) API.
 - You are expected to implement many **distance metrics** for various ML algorithms
 - e.g., Clustering

1

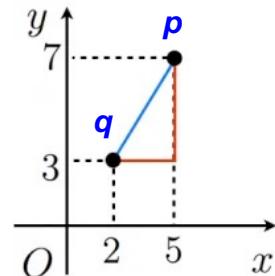
Example Distance Metrics

- Euclidean distance

$$- d_e(p, q) = \sqrt{\sum_{i=0}^n (p_i - q_i)^2}$$

- Manhattan distance

$$- d_m(p, q) = \sum_{i=0}^n |p_i - q_i|$$



- $d_e = \sqrt{(5 - 2)^2 + (7 - 3)^2} = \sqrt{3^2 + 4^2} = 5$
- $d_m = (5 - 2) + (7 - 3) = 7$

2

Using Strategy for Pluggable Distance Metrics

- There exists **no “one-size-fits-all” metric.**
 - Different algorithms and datasets require different metrics.
 - e.g., Standardized Euclidean, Chebyshev, Mahalanobis, Minkowski, Cosine, Jaccard, Canberra, Kulsinski, etc.
 - **You (API designer)** will implement extra metrics in the future, for sure.
 - Some **API users** will want to implement their own (custom) metrics, for sure.
- How to allow distance metrics to be introduced and maintained in a pluggable manner?

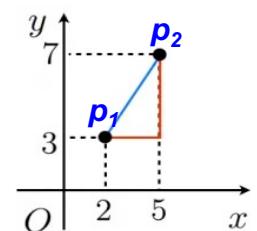
Distance	Returns the Euclidean distance b/w p1 and p2 by default.
+ <code>get(p1: List<Double>, p2: List<Double>): double</code>	
+ <code>matrix(points: List<List<Double>>): List<List<Double>></code>	Returns a distance matrix for given points with the Euclidean metric by default

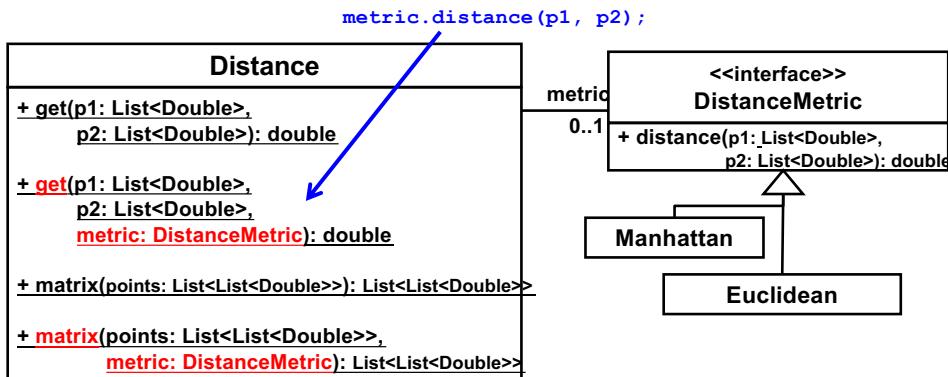
Client of Distance:

```
List<Double> p1, p2;
p1 = Arrays.asList(2.0, 3.0);
p2 = Arrays.asList(5.0, 7.0);
Distance.get(p1, p2);           // returns 5

List<List<Double>> points = new ArrayList<>();
points.add(p1); points.add(p2);
Distance.matrix(points);
// returns [[0,5], [5,0]]
```

3



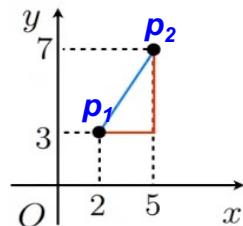


```

Distance.get(p1, p2);           // returns 5
Distance.get(p1, p2,
            new Manhattan()); // returns 7

Distance.matrix(points);        // returns [[0,5],
                                //          [5,0]]
Distance.matrix(points,
                new Manhattan());
                // returns [[0,7],
                //          [7,0]]

```



CS680's HW

- Read `Distance.java` and other source code.
 - DO NOT modify `Distance`, `DistanceMetric` and `Euclidean`.
 - Keep them as they are, and just “use” them.
- Add the `Manhattan` class, so you can...
 - compute a Manhattan distance b/w `p1` and `p2`
 - compute a distance matrix for given `points` with the Manhattan metric.
- [Optional] Implement one extra metric.

6

CS681's HW 4

- Revise the body of `Euclidean.distance()` with Stream API.

```

public double distance(List<Double> p1, List<Double> p2) {
    double sumOfSquared = 0;
    for(int i=0; i < p1.size(); i++) {
        sumOfSquared += Math.pow( p1.get(i)-p2.get(i), 2 );
    }
    return Math.sqrt(sumOfSquared); }

public double distance(List<Double> p1, List<Double> p2) {
    double sumOfSquared =
        IntStream
            .range(0, p1.size())
            .mapToDouble(i -> p1.get(i)-p2.get(i))
            .map(n -> n*n)
            .sum();
    return Math.sqrt(sumOfSquared); }

```

- Revise the body of `Manhattan.distance()` with Stream API.
- [Optional] Revise the body of `distance()` of an extra distance metric class(es).
 - If you worked on that in CS680.
- Revise the body of `Distance.matrix()` with Stream API.
 - Test your code with more than 1,000 points. Have 100+ dimensions for each point.

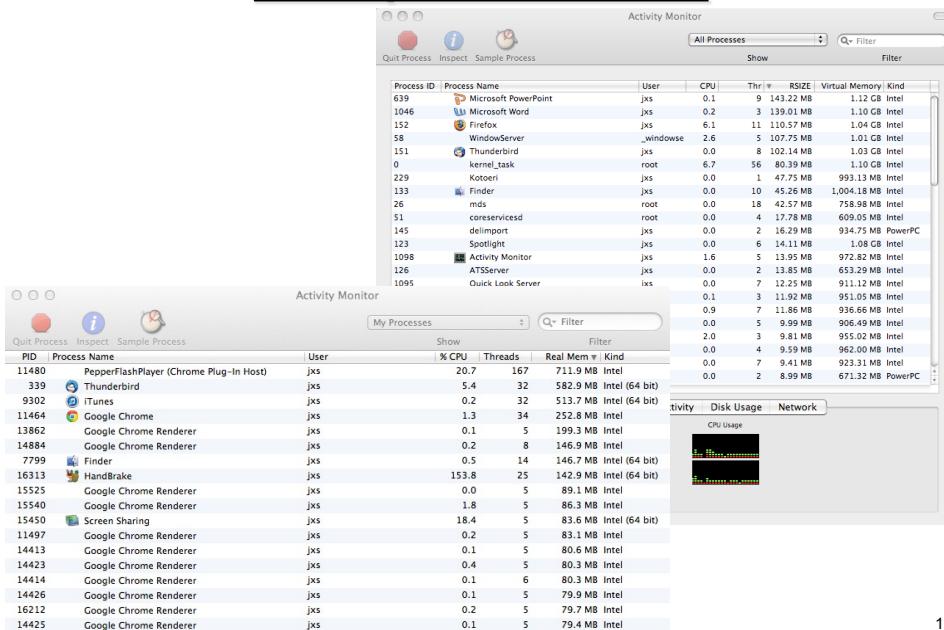
Processes

- One of the most important concepts in all modern operating systems
- A process is a **container** (or execution environment) for a program (software) in execution.
- Any software is executed with one or more processes.
 - Java VM (JVM), MS Word, Excel, PPT, Firefox, iTunes, Google Chrome, Kindle, Java IDE, etc.

9

Concurrency with Threads

Example Processes



11

Multi-Tasking (Time Sharing)

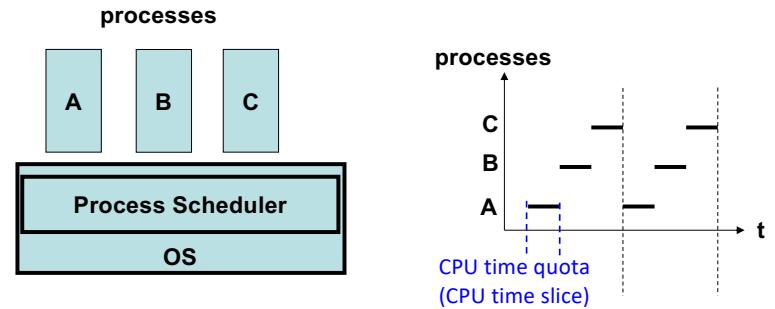
- All modern OSes support **multi-tasking**.
 - Doing **multiple things** with **multiple processes** on a single **CPU** (more precisely, a single CPU core)
 - e.g., Writing a document with an editor while having an online meeting with a conferencing tool.
 - Writing code with an IDE while playing a YouTube video
 - Reading an e-book with an e-book reader while playing music with a music player.
 - Running an activity monitoring (e.g. step counting) app while running other apps
 - e.g., Walking/running on a treadmill while watching a movie

10

12

Process Scheduling

- At any moment, a single CPU core can execute a single program (process).
- **Pseudo concurrency/parallelism:** An **illusion** for human eyes as if multiple processes run at the same time.
 - OS periodically assigns one process to another to the CPU core.
 - every several tens of milliseconds to several hundreds of milliseconds
 - CPU time quota (CPU time slice)
 - A CPU core is multiplexed among processes; processes do NOT run in completely parallel.

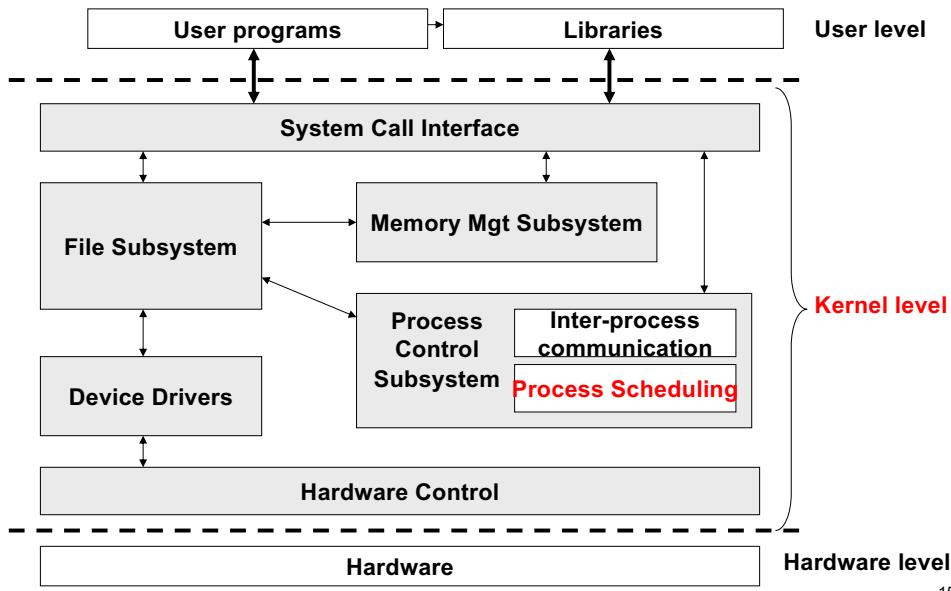


- Processes run in turn by spending the same amount of CPU time at a time.

13

14

An Architectural View of an OS



15

Why Threads?

- **Good old days:** Process-based, **coarse-grained concurrency** was enough
 - e.g., Writing a document with an editor while having an online meeting with a conferencing tool.
- **Now:** **Finer-grained concurrency** is required.
 - Many programs are required to run multiple **in-process tasks** at the same time.
 - Running different tasks at the same time.
 - Running the same or a similar tasks for different users/clients at the same time.
 - Splitting a computationally heavy (big) task into lightweight (smaller) tasks, and running those smaller tasks at the same time.
 - Assign those **in-process tasks** to **threads**.

16

- Many programs are required to run **different in-process tasks** at the same time.

- Document editor
 - Editing a document
 - Performing spelling and grammar check in the background
 - Uploading/downloading (synching) documents to/from a cloud
- Web browser
 - Running multiple tabs (e.g., displaying a web page on a tab while playing a YouTube video on another tab).
- Music player
 - Playing music
 - Downloading music and its metadata (lyrics, etc.)
 - Streaming music to
- Activity monitoring app
 - Running sensors (e.g. accelerometer, gyro, barometric pressure sensor)
 - Calculating step counts, flight/floor climbing, walking/running/swim distance

17

- Many programs are required to run **the same or a similar task** for different users/clients at the same time.

- Web server
 - Does the same sequence of tasks to different clients (remote browsers) at the same time
 - Accepting and parsing an HTTP request
 - Finding a requested file
 - Making a response HTTP message (incl. header, payload, etc.)
 - Returning the requested file's content with the HTTP message

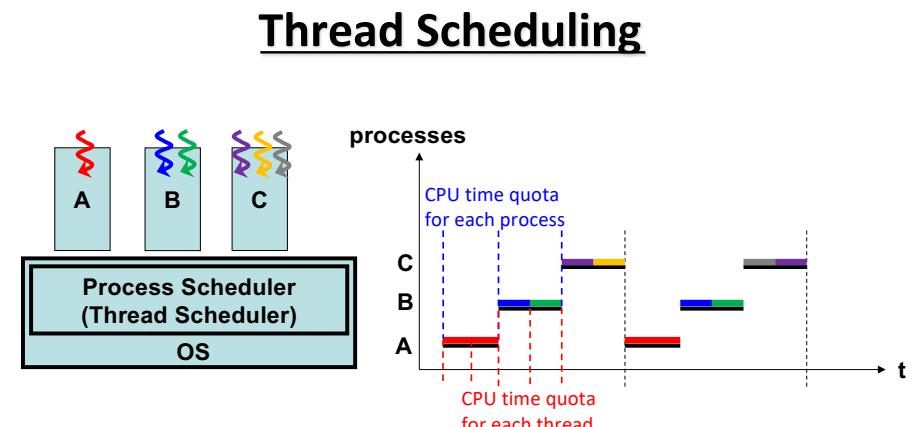
- Each process splits a big task and runs smaller tasks at the same time.
 - e.g., Calculating tax rate for each county in the US

18

- **Fine-grained pseudo parallelism:** An **illusion** for human eyes as if multiple threads run on a process at the same time.

- OS periodically assigns one thread to another to the CPU core.
 - Every tens of milliseconds to approx. 100 milliseconds
 - Usually, shorter CPU time quota for desktop and mobile OSes, and longer CPU time quota for server-side OSes
 - A CPU core is multiplexed among processes and threads; processes/threads do NOT run in a completely parallel manner on a single CPU core.

19



- Threads run in turn by spending the same amount of CPU time at a time.

20

Summary: Why Threads?

• Responsiveness/availability

- If in-process tasks runs sequentially, your app would not be responsive/available for you.
- Threads allow your app to continue running even if a part of it is blocked for I/O or is performing a long operation.

• Efficiency

- Threads are more lightweight than processes.
 - Creating a process is approx. 30 times slower than creating a thread.
 - Process switching is approx. 5 times slower than thread switching.
- In-process tasks can run in parallel on a multi-core CPU

• In mobile computing,

- Mobile devices often use multi-core CPUs with lower clock speed to reduce their power consumption and extend their battery life.
 - Favor increased density of cores, not increased speed on a each core.
 - Kindle Fire (\$50): 4 cores, 1.3 GHz
 - iPhone: 6 cores (2 high-performance and 4 energy-efficient cores)
 - iPad Pro: 8 cores (4 high-performance and 4 energy-efficient cores)

• In cloud computing

- Servers often use multi-core CPUs with lower clock speed to reduce power supply cost and CO2 emission.
 - Enormous energy consumption
 - Google paid \$2M/mo for electricity bills ('07).
 - Significant CO2 emission
 - The IT industry produces more CO2 emission than the aviation industry.

Other Viewpoints to Threads

- CPU clock speed does not increase as it did in the past.

- Physical material barrier to increase clock speed and # of transistors in a single CPU

- Excessive power consumption and heat

- Intuitively, power consumption increases exponentially as clock speed increases.
- Harder to economically justify power supply cost with increased clock speed in many industry sectors.

21

22

- You are expected to increase your app's performance by increasing its concurrency.

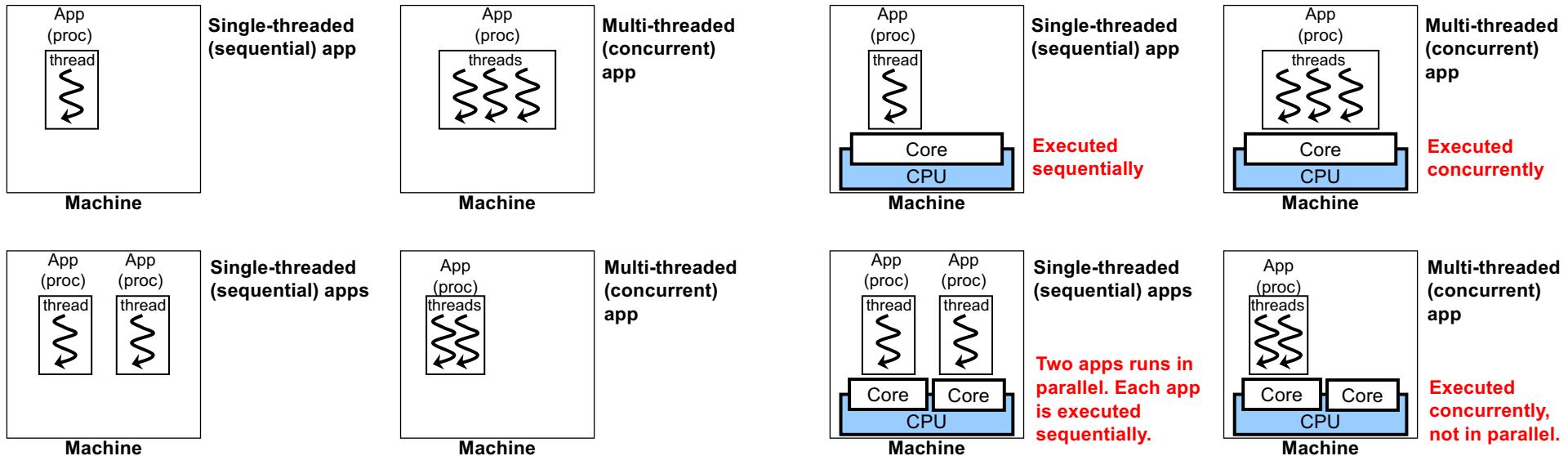
- This will lead you to “green” computing.

- Ultimate goals of multi-threaded applications:
Responsiveness and efficiency improvement

23

24

Terminology: Single- and Multi-threaded



25

26

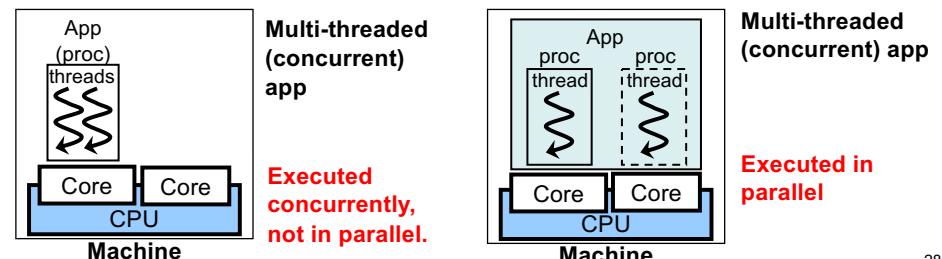
Terminology: Concurrent and Parallel

- Traditionally, the semantics of concurrency and parallelism have been orthogonal (separated clearly).
 - Being “concurrent” means...
 - A program is implemented with multiple threads.
 - Concurrency: a concept (or technique) in program implementation
 - Being “parallel” means...
 - Programs are executed in parallel.
 - Parallelism: a concept (or technique) in program execution
- As a result...
 - Sequential (non-concurrent) programs may be executed in parallel.
 - Concurrent programs may not be executed in parallel.

27

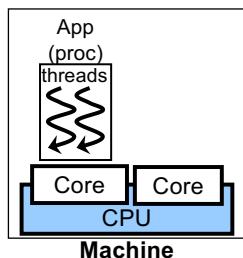
Modern Thread Distribution (Thread Dispatching)

- Many modern OSes automatically distribute in-process threads onto different CPU cores in multi-core CPU environments.
- Concurrent programs often run in parallel (i.e., concurrent programs often means parallel programs)
 - as far as they run on modern OSes in multi-core CPU environments.



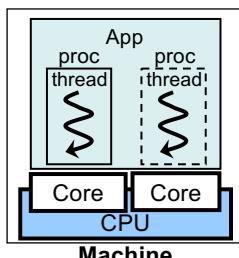
28

Terminology: Concurrent or Parallel?



[1] Multi-threaded (concurrent) app

NOT parallel



[2] Multi-threaded (concurrent) app

Parallel

- The terms “concurrent” [1] and “parallel” [2] are NOT equivalent in principle, but they are becoming equivalent more often in practice.
- 2 terminology preferences exist
 - If you think [2] is a special case of [1], you may prefer the term “concurrent.”
 - If you think [1] is a special case of [2], you may prefer the term “parallel.”

29

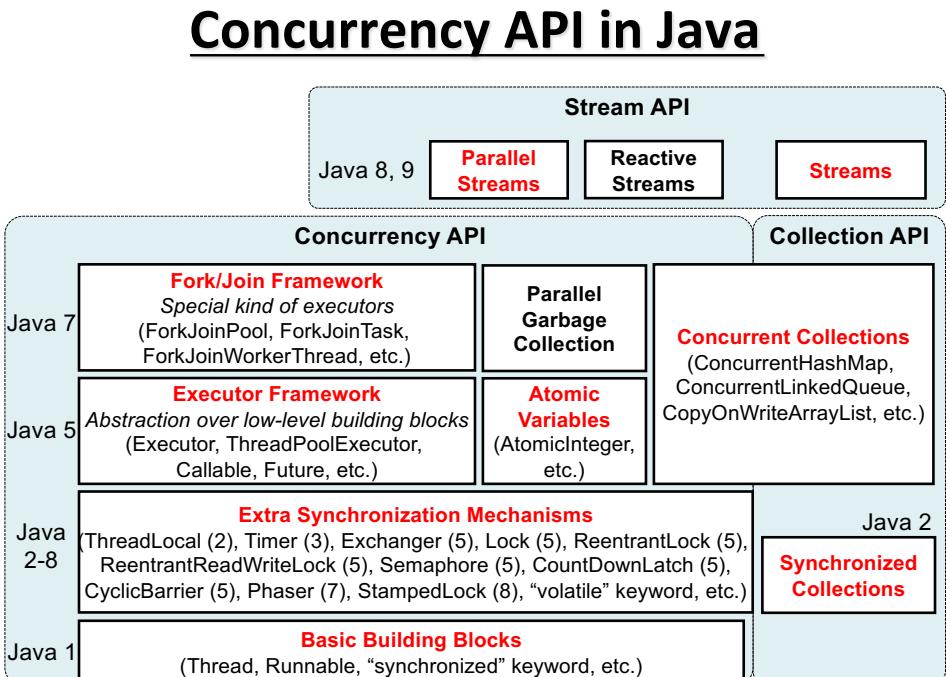
In Java...

- The terms “concurrent” and “parallel” are somewhat mixed up in Java.
 - Java 5 (2004) introduced `java.util.concurrent`, which has been enhanced afterwards until now.
 - “Concurrent” collections such as `concurrentHashMap`
 - The Executor framework
 - An extension/abstraction over low-level threads
 - Java 7 (2011)
 - The Fork/Join framework, an extension to the Executor framework
 - The term “parallel” appeared in its documentation, although it was placed in `java.util.concurrent`.
 - “Parallel” garbage collector, which performs garbage collection with multiple threads.
 - Java 8 (2014)
 - “Parallel” streams, which allows for “parallel” (multi-threaded) operations on streams with lambda expressions

30

Java Threads

31



Java Threads

- Every Java program has at least one *thread of control*.
 - `main()` runs with a thread (of control) on a JVM.
 - When a JVM starts, it implicitly (or automatically) creates the “`main` thread.
 - ```
class HelloWorldApp{
 public static void main(String[] args){
 System.out.println("Hello World!");
 }
}

>java HelloWorldApp
```

- If you need *extra threads* in addition to the main thread, you need to create them *explicitly*.
- 4 things to do:
  - Define a class implementing the `java.lang.Runnable` interface
    - `public abstract void run();`
  - Write a threaded/concurrent task in `run()` in the class
  - Instantiate `java.lang.Thread` and associate a Runnable object with the thread
  - Start (call `start()` on) the instantiated thread.
    - `run()` is automatically called on the thread.

33

34

## An Example Code: Creating a Thread

- GreetingRunnable.java
  - ```
class GreetingRunnable implements Runnable{
    private String greetingMsg;

    public GreetingRunnable(String msg){
        this.greetingMsg = msg;
    }

    public void run(){
        for( int i=0; i<10; i++ ){
            System.out.println( greetingMsg );
        }
    }
}
```
- HelloWorldTest.java
 - ```
main(...){
 GreetingRunnable runnable = new GreetingRunnable("Hello World");
 Thread t = new Thread(runnable);
 t.start();
}
```

## Thread.start()

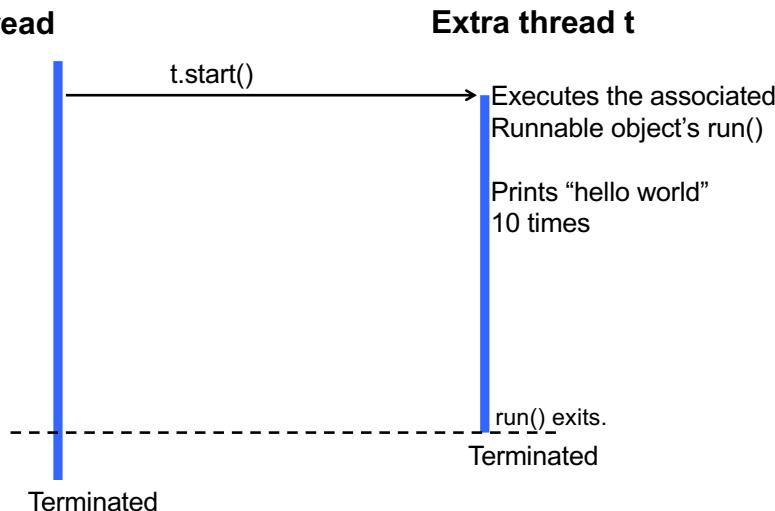
- Creating a `Thread` object does not mean creating a new thread on the local JVM and in turn OS.
  - It is `start()` that actually creates a thread.
- `start()`
  - Allocates memory and initializes a new thread on the a JVM (and its underlying OS).
  - Calls `run()` of a specified `Runnable` object.
    - DO NOT call `run()` directly, but...  
let `start()` call `run()` on behalf of yourself.

35

36

# Program Execution

Main thread



• Output:

- Hello World

37

38

## An Example Code: Creating Threads

- HelloWorldTest2.java and GreetingRunnable.java

```
main(...){
 GreetingRunnable runnable1 = new GreetingRunnable("Hello World");
 GreetingRunnable runnable2 = new GreetingRunnable("Goodbye World");

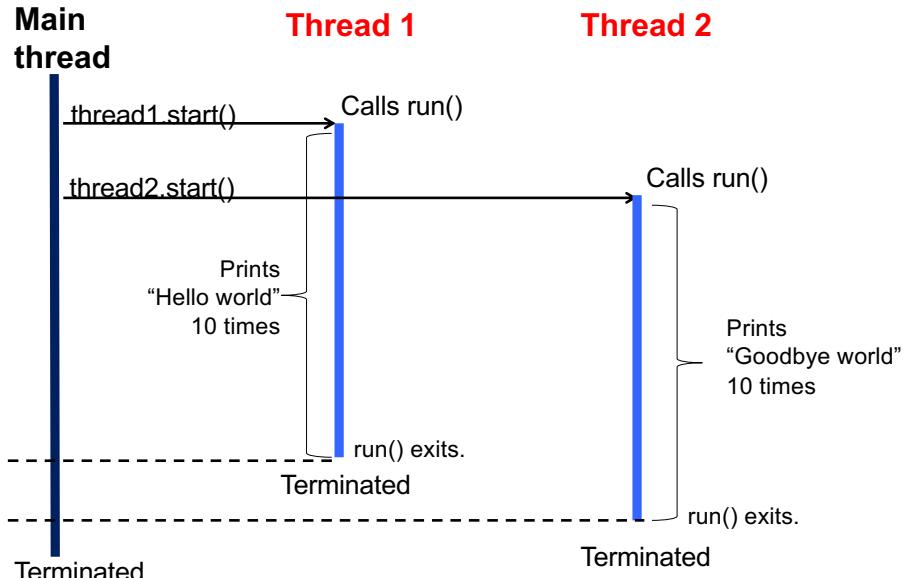
 Thread thread1 = new Thread(runnable1);
 Thread thread2 = new Thread(runnable2);

 thread1.start();
 thread2.start();
}
```

39

## Expected Program Execution

Main  
thread

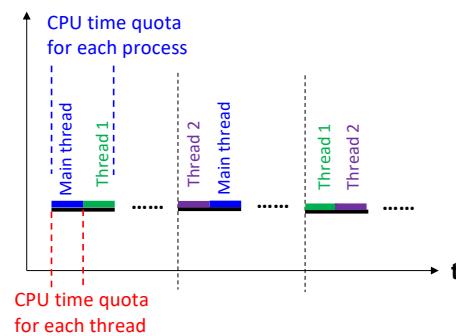


40

# Thread Scheduling



Main thread  
2 extra threads  
(Threads 1 and 2)



- Other processes (than the Java program) are ignored here.
  - 1 per-process CPU time quota is assumed to be equal to 2 per-thread CPU time quotas here.
    - Actual ratio depends on the JVM and OS implementations.
      - May be 1:3, 1:1.5, etc.

41

## A Possible Output

- Output:
    - Goodbye World
    - Hello World
    - Hello World
    - Goodbye World
    - Hello World
    - Goodbye World
    - Goodbye World
    - Hello World
    - Hello World
    - Goodbye World
    - Hello World
    - Hello World
    - Goodbye World
    - Hello World
    - Hello World
    - Goodbye World
  - Two message sets (Hello and Goodbye) are NOT exactly interleaved.

42

# The Order of Thread Execution

- JVM's thread scheduler gives you NO guarantee about the order of thread execution.
  - There are always slight variations in the time to run a concurrent task
    - especially when calling OS system calls (typically I/O related system calls)
  - Expect that the order of thread execution is somewhat random.

## Exercise

- Modify HelloWorldTest.java;
    - Replace the following highlighted lines
      - GreetingRunnable runnable1 = new GreetingRunnable("Hello");  
GreetingRunnable runnable2 = new GreetingRunnable("Bye");  
`Thread thread1 = new Thread(runnable1);`  
`Thread thread2 = new Thread(runnable2);`  
`thread1.start();`  
`thread2.start();`
      - with the following lines:
        - `runnable1.run();`
        - `runnable2.run();`
    - What does the program output?

43

44

# States of a Thread

- Modify HelloWorldTest.java;

- Replace the highlighted lines

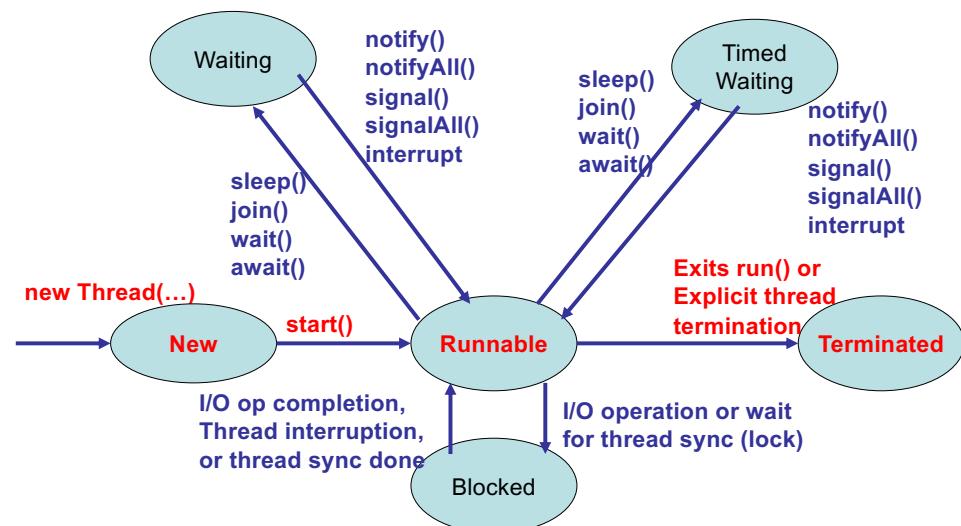
```
• GreetingRunnable runnable1 = new GreetingRunnable("Hello");
GreetingRunnable runnable2 = new GreetingRunnable("Bye");
Thread thread1 = new Thread(runnable1);
Thread thread2 = new Thread(runnable2);
thread1.start();
thread2.start();
```

- with the following lines:

```
• runnable1.run();
• runnable2.run();
```

- Output: “Hello” in 10 lines, followed by “Bye” in 10 lines.
  - The program runs **sequentially**, NOT concurrently.

45



46

- **New**

- A Thread object is created. **start()** has **NOT** been called on the object yet.

- **Runnable**

- **start()** has been called.
  - Java does not precisely distinguish **ready-to-run** and **running**. A running thread is in the Runnable state.

- **Terminated (dead)**

- A thread automatically dies when **run()** returns.

- ```
public class Thread{
    public enum State{
        NEW, RUNNABLE, BLOCKED, WAITING,
        TIMED_WAITING, TERMINATED
    }

    public Thread.State getState()

    public boolean isAlive()
```

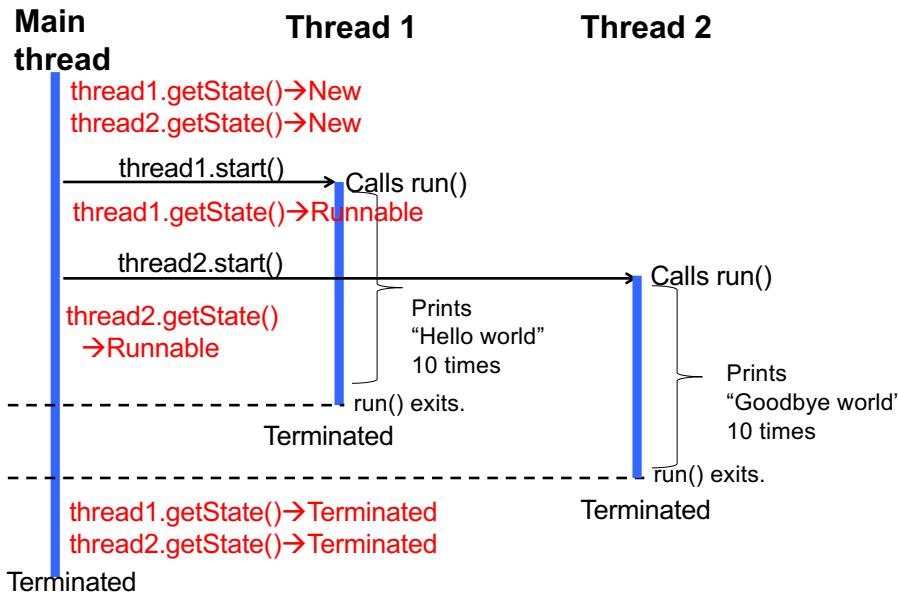
- **Alive**

- in the Runnable, Blocked, Waiting or Timed Waiting state.
 - NOT in New nor Terminated.
 - **isAlive()** can be used to check if a particular thread has been terminated.

47

48

Program Execution w/ HelloWorldTest3



Terminated

49

Important Note

- Do NOT associate a single `Runnable` object with multiple `Thread` objects.

- Do (less error-prone):

```
GreetingRunnable runnable1 = new GreetingRunnable(...);
GreetingRunnable runnable2 = new GreetingRunnable(...);
Thread thread1 = new Thread(runnable1);
Thread thread2 = new Thread(runnable2);
thread1.start();
thread2.start();
```

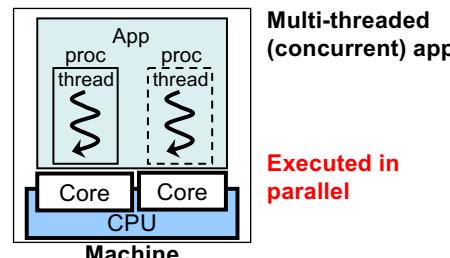
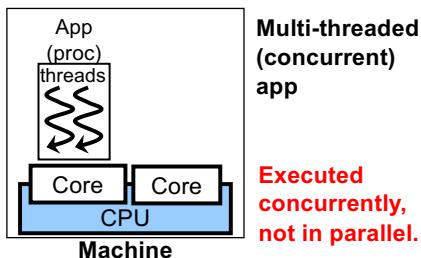
- Don't:

```
GreetingRunnable runnable1 = new GreetingRunnable(...);
Thread thread1 = new Thread(runnable1);
Thread thread2 = new Thread(runnable1);
thread1.start();
thread2.start();
```

50

Modern Thread Distribution (Thread Dispatching)

- Many modern OSes automatically distribute in-process threads onto different CPU cores in multi-core CPU environments.
- Concurrent programs often run in parallel (i.e., concurrent programs often means parallel programs)
 - as far as they run on modern OSes in multi-core CPU environments.

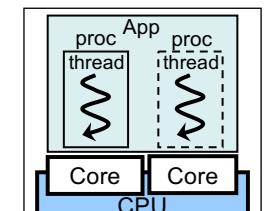
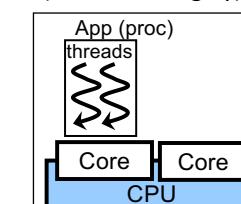
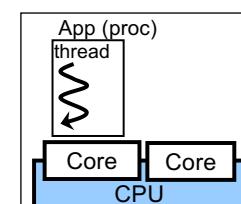


51

Run MCTest.java

- MCTest.java
 - `run()`: calculates 25×25 multiple times.
 - Can vary the number of threads
 - Calculated 25×25 10 billion times with Java 15 on Mac OS X
 - Intel Core i7 2.8 GHz (quad-core) and 16 GB RAM

# of threads	Time (sec)
• 1	2.9
• 2	2.9 (< 2.9 * 2. Two threads run in parallel!)
• 4	3.0 (< 2.9 * 4. Four threads run in parallel!)
• 8	6.1 (= 3.0 * 2, roughly)
• 16	12.3 (= 6.1 * 2, roughly)

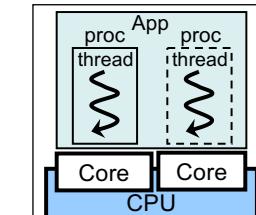
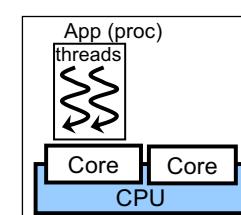
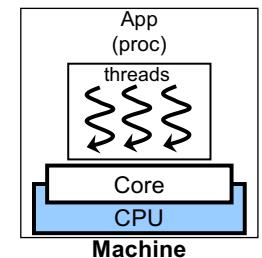


In CS681...

- Run `MCTest.java` with multiple threads on your machine.

- e.g., `java edu.umb.cs681.basics.MCTest 1000000000 4`
 - First param: # of 25*25 multiplication
 - Second param: # of threads

- I will assume there is **ONLY ONE** CPU core in the underlying machine
 - when I explain/analyze the safety of multi-threaded programs.
- If a program is thread-safe in the single-core setting, it is always thread-safe as well in multi-core settings.



53

Runnable as a Functional Interface

- `java.lang.Runnable`: functional interface
 - Abstract method: `run()`

- Thread's constructor
 - Takes a `Runnable` object
 - Can receive a lambda expression (LE)

- Traditional

```
- GreetingRunnable runnable =
    new GreetingRunnable("Hello World");
Thread thread = new Thread(runnable);
thread.start();
```

- LE-based

```
- Thread thread = new Thread( ()->{
    System.out.println("Goodbye World"); } );
thread.start();
```

- It makes less/no sense to use a lambda expression to implement a concurrent task when the task is complex.
 - Lambda expressions are useful/powerful when their code blocks are reasonably short.

Sample Code: PrimeGenerator

- Generates prime numbers in between two input numbers (`from` and `to`)

```

• Class PrimeGenerator {
    protected long from, to;
    protected LinkedList<Long> primes;

    public void generatePrimes(){ ... }
    public LinkedList<Long> getPrimes(){ return primes };
    protected boolean isPrime(long n){ ... };
}

```

- Client code (single-threaded)

```

• PrimeGenerator gen = new PrimeGenerator(1L, 1000000L);
gen.generatePrimes();
gen.getPrimes().forEach( (prime)-> System.out.println(prime) );
//forEach() takes a LE typed with Consumer

```

57

Notes: main() in PrimeGenerator

- PrimeGenerator's `main()` implements 2 types of single-threaded client code to generate primes.

- 1st client code

- Uses `generatePrimes()` to generate primes
 - c.f. previous slide
 - `for(long n = from; n <= to; n++) { if(isPrime(n)){ primes.add(n); } }`

- 2nd client code

- Does NOT use `generatePrimes()`
- Uses Stream API and `isPrime()`
 - `gen.primes = LongStream.rangeClosed(gen.from, gen.to)
 .filter((long n)->gen.isPrime(n))
 ...}`

58

Sample Code: RunnablePrimeGenerator

- A Runnable class that generates prime numbers in between two input numbers (`from` and `to`)

```

• class RunnablePrimeGenerator extends PrimeGenerator
    implements Runnable {
    public RunnablePrimeGenerator(long from, long to){
        super(from, to); }

    public void run(){
        generatePrimes(); }
}

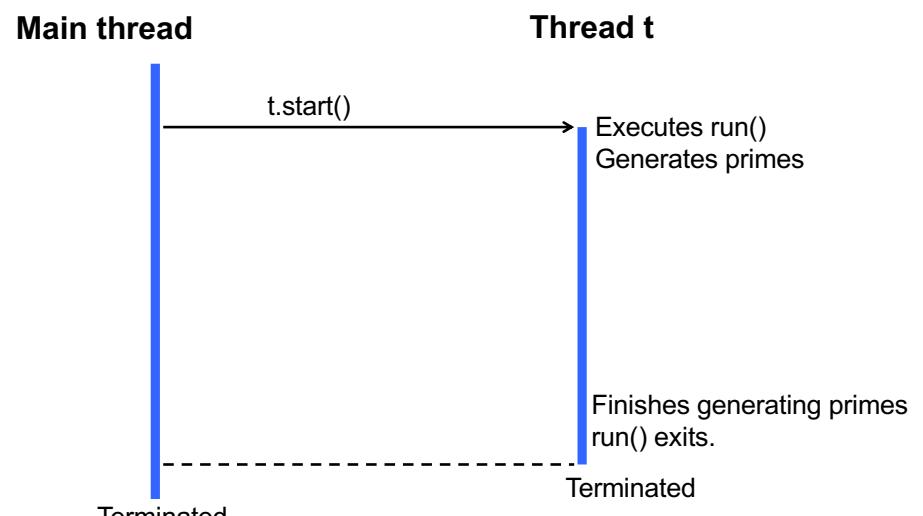
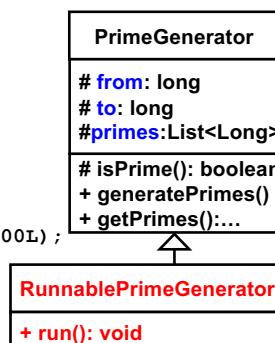
```

- Client code (multi-threaded)

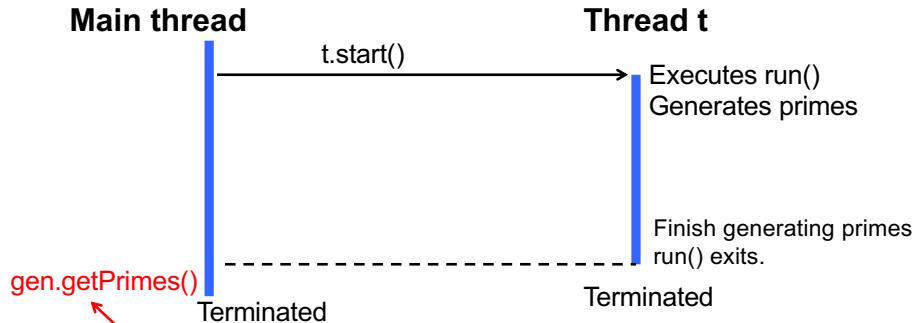
```

• RunnablePrimeGenerator gen =
    new RunnablePrimeGenerator(1L, 2000000L);
Thread t = new Thread(gen);
t.start();

```



60

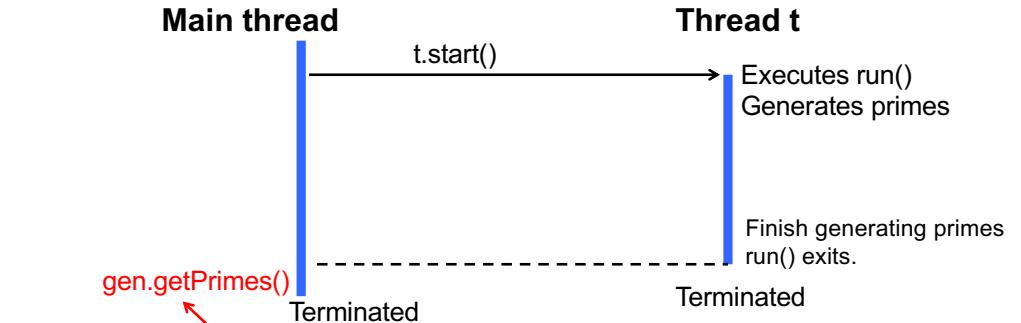


To get generated prime numbers, the main thread needs to call `getPrimes()` on a `RunnablePrimeGenerator`.

The main thread can call `getPrimes()` on the `Runnable` object **even after t dies**.

Thread **t** dies when `run()` exits. However, it NEVER kill/delete the `Runnable` object.

The `Runnable` object is still available **even after t dies**.



To get generated prime numbers, the main thread needs to call `getPrimes()` on a `RunnablePrimeGenerator`.

The main thread can call `getPrimes()` on the `Runnable` object **even after t dies**.

- Client code

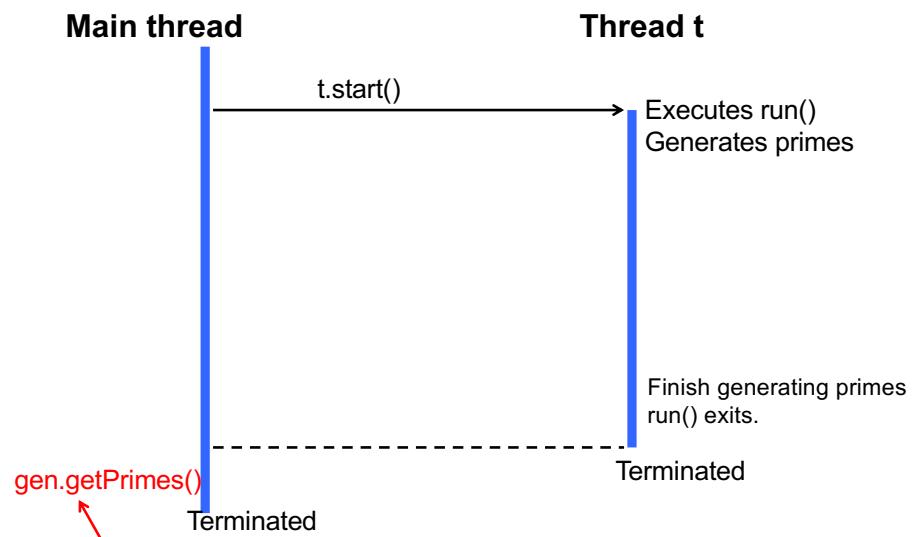
```

• RunnablePrimeGenerator gen = new RunnablePrimeGenerator(...);
  Thread t = new Thread(gen);
  t.start();
  gen.getPrimes();

```

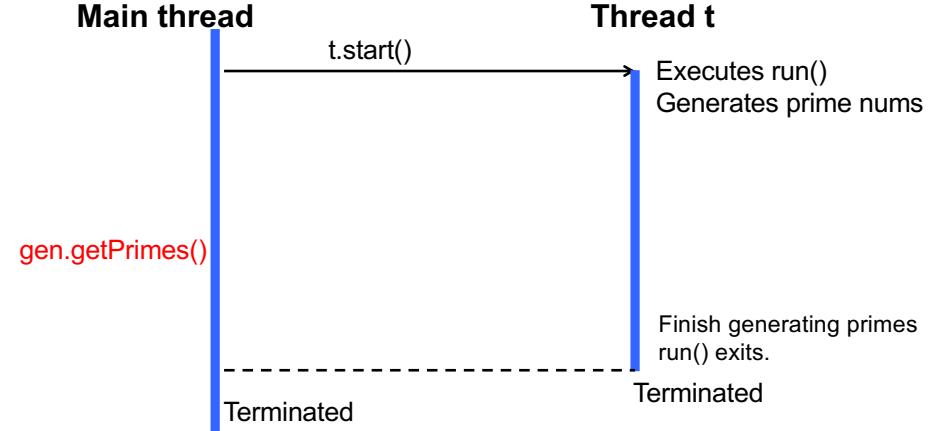
61

62



To get generated prime numbers, the main thread needs to call `getPrimes()` on a `RunnablePrimeGenerator`.

How to guarantee to call `getPrimes()` after `run()` exits?



- Client code

```

• RunnablePrimeGenerator gen = new RunnablePrimeGenerator(...);
  Thread t = new Thread(gen);
  t.start();
  gen.getPrimes();

```

`getPrimes()` may be called before `run()` exits.

How to guarantee to call `getPrimes()` after `run()` exits?

63

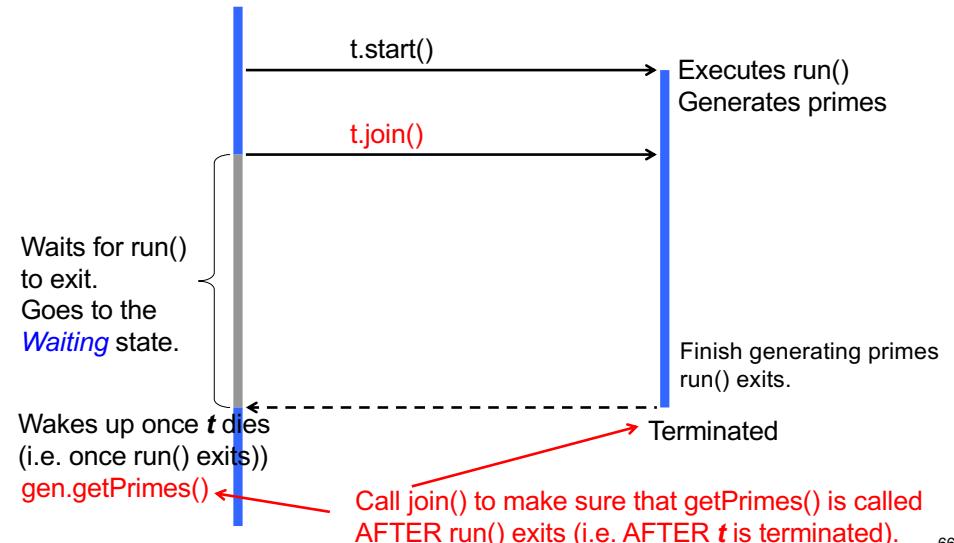
64

Thread.join()

- Revised client code

```
RunnablePrimeGenerator gen = new RunnablePrimeGenerator(...);
Thread t = new Thread(gen);
t.start();
t.join();
gen.getPrimes().forEach(...);
```

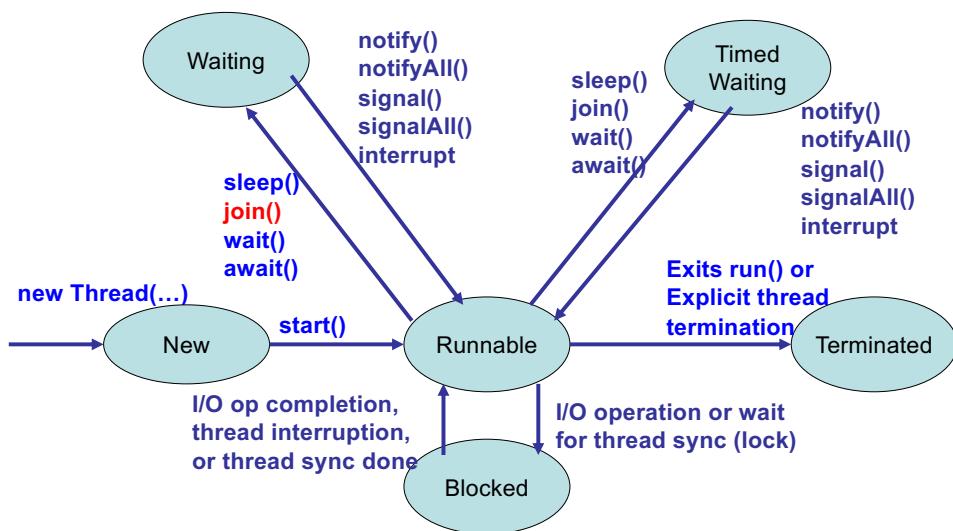
Main thread



65

66

States of a Thread



67

Thread.join()

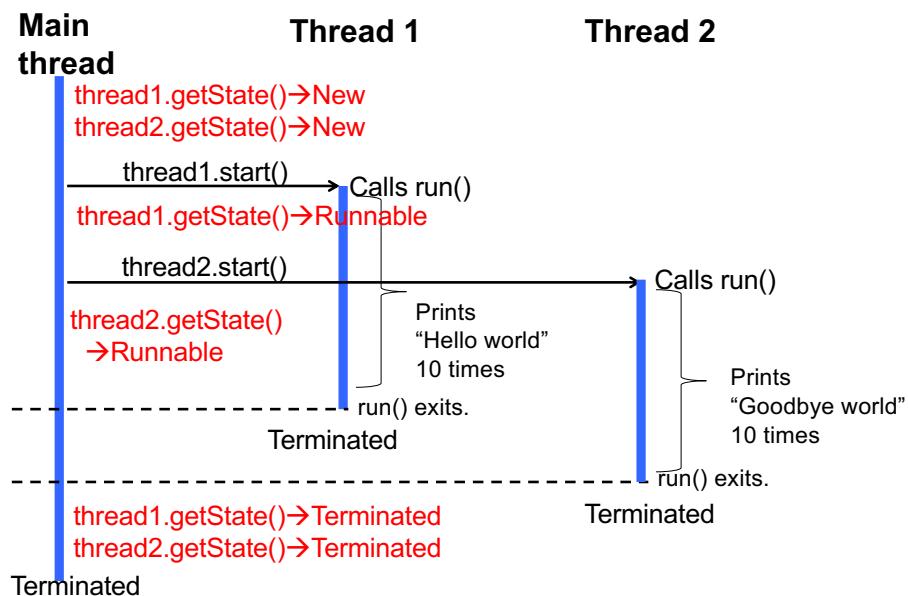
- By default, there is no guarantee about the order of thread execution
- `join()` allows you to **control the order of thread execution** to some extent.

68

Exercise

- See how program behavior changes with and without `join()`.

Note: `join()` is used in `HelloWorldTest3`



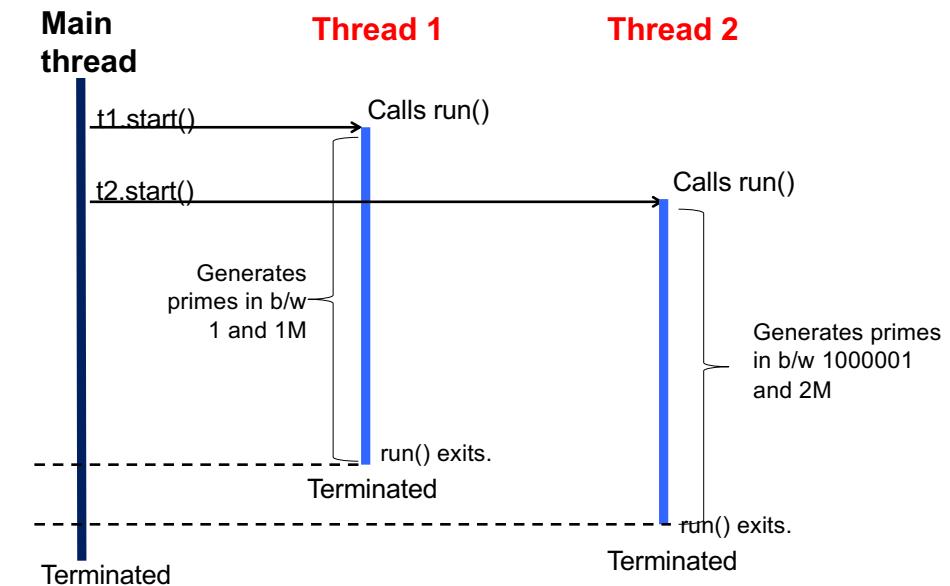
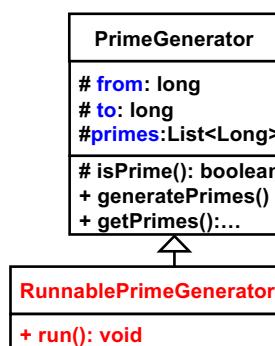
69

70

Sample Code: RunnablePrimeGenerator

- Client code (multi-threaded), which uses **2 threads**

```
RunnablePrimeGenerator g1 = new RunnablePrimeGenerator(  
    1L, 1000000L);  
RunnablePrimeGenerator g2 = new RunnablePrimeGenerator(  
    10000001L, 2000000L);  
  
Thread t1 = new Thread(g1);  
Thread t2 = new Thread(g2);  
t1.start();  
t2.start();  
t1.join();  
t2.join();  
g1.getPrimes().forEach(...);  
g2.getPrimes().forEach(...);
```



72

Exercise

- Run `RunnablePrimeGenerator` to generate primes in b/w 1 and 2,000,000.

- With 1 thread

```
• RunnablePrimeGenerator g = new RunnablePrimeGenerator(  
    1L, 2000000L);  
Thread t = new Thread(g);  
t.start();  
t.join();  
g1.getPrimes().forEach(...);
```

- With 2 threads

```
• RunnablePrimeGenerator g1 = new RunnablePrimeGenerator(  
    1L, 1000000L);  
RunnablePrimeGenerator g2 = new RunnablePrimeGenerator(  
    1000001L, 2000000L);  
Thread t1 = new Thread(g1); // MAKE SURE TO CREATE 2 RUNNABLE  
Thread t2 = new Thread(g2); // OBJECTS  
t1.start(); t2.start();  
t1.join(); t2.join();  
g1.getPrimes().forEach(...);  
g2.getPrimes().forEach(...);
```

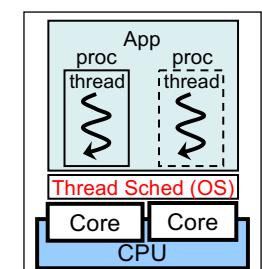
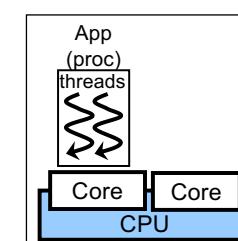
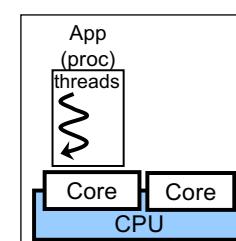
- With more threads

-

- Measure the overhead of generating primes in b/w 1 and 2M to see if automatic thread distribution/dispatching works.

- # of threads Time (sec)

- 1	???
- 2	??? (2 threads run in parallel?)
- 4	??? (4 threads run in parallel?)
- 8	???
- 16	???



73

```
• // Stream to array conversion  
//  
String[] arr = stream.toArray(String[]::new);  
  
// Array to stream conversion  
//  
Stream<String> stream = Arrays.stream(arr);  
  
// Stream to list conversion  
//  
List<String> list = stream.collect(Collectors.toList());  
  
// List to stream conversion  
//  
Stream<String> stream = list.stream();  
  
// List to array conversion  
//  
String[] arr = list.toArray(new String[0]);  
  
// Array to list conversion  
//  
List<String> list = Arrays.asList(arr);
```

79

74