WEEK TWELVE

Acknowledgements: Slides created based off material provided by Dr. Michael Raymer and Dr. Travis Doom

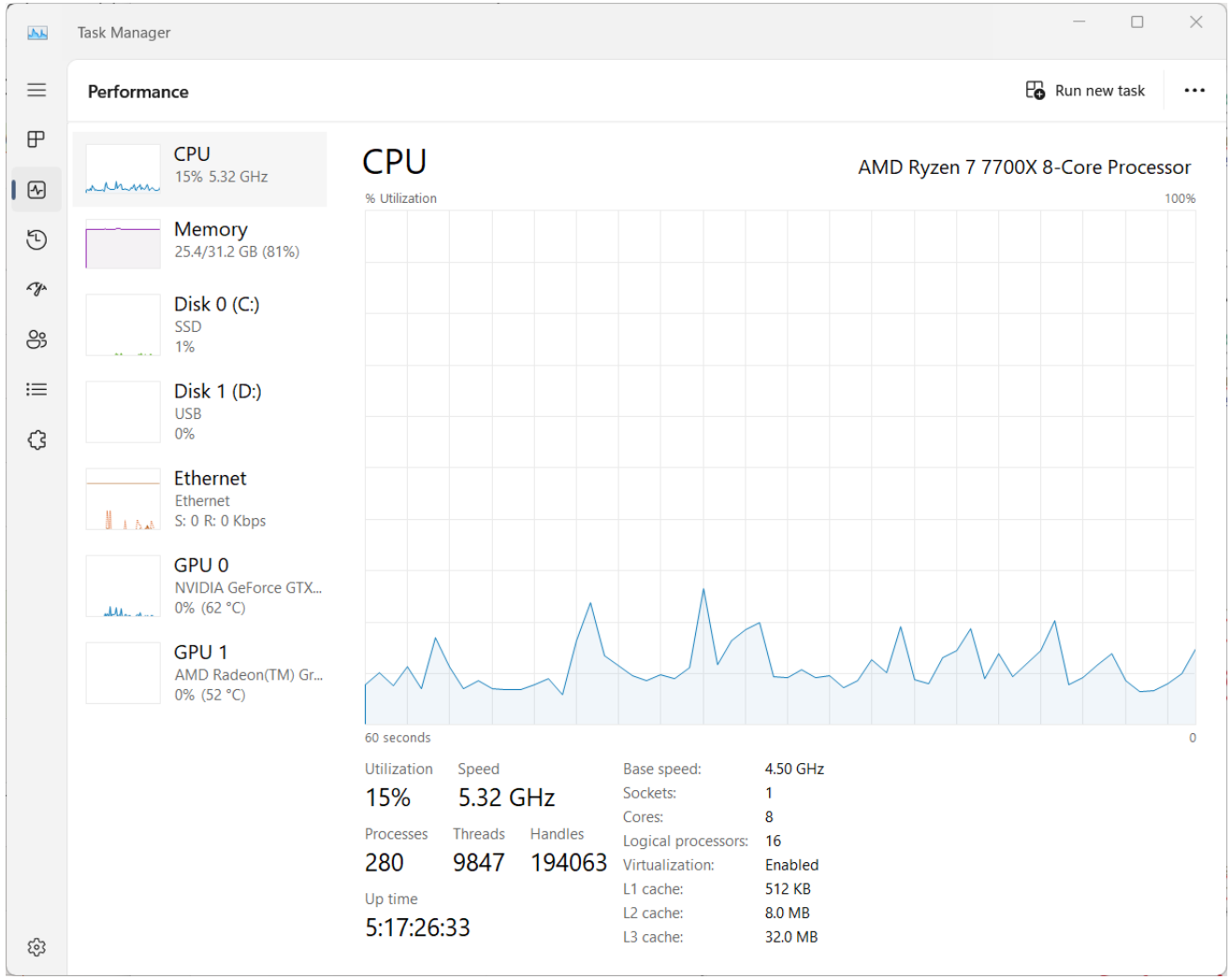
HOW DO COMPUTERS RUN MANY PROGRAMS AT THE SAME TIME?

Multiple cores

Pre-emptive multitasking

EXECUTION CORES

- Task manager shows number of cores on your CPU



PROCESSES

- A process consists of:
  - Memory Space
    - Heap
    - Stack
    - Globals/Statics
    - Code
  - Program State
    - Program counter
    - Execution state: Running, waiting, sleeping, etc.

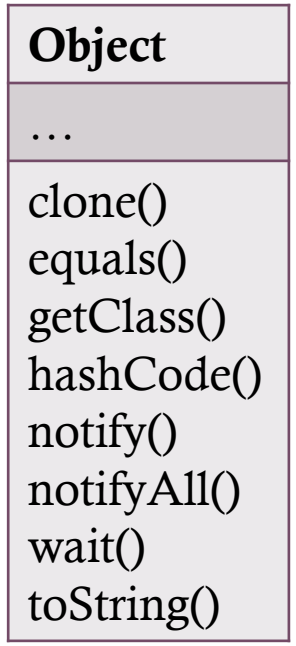
Each program executes as a single process.

THREADS

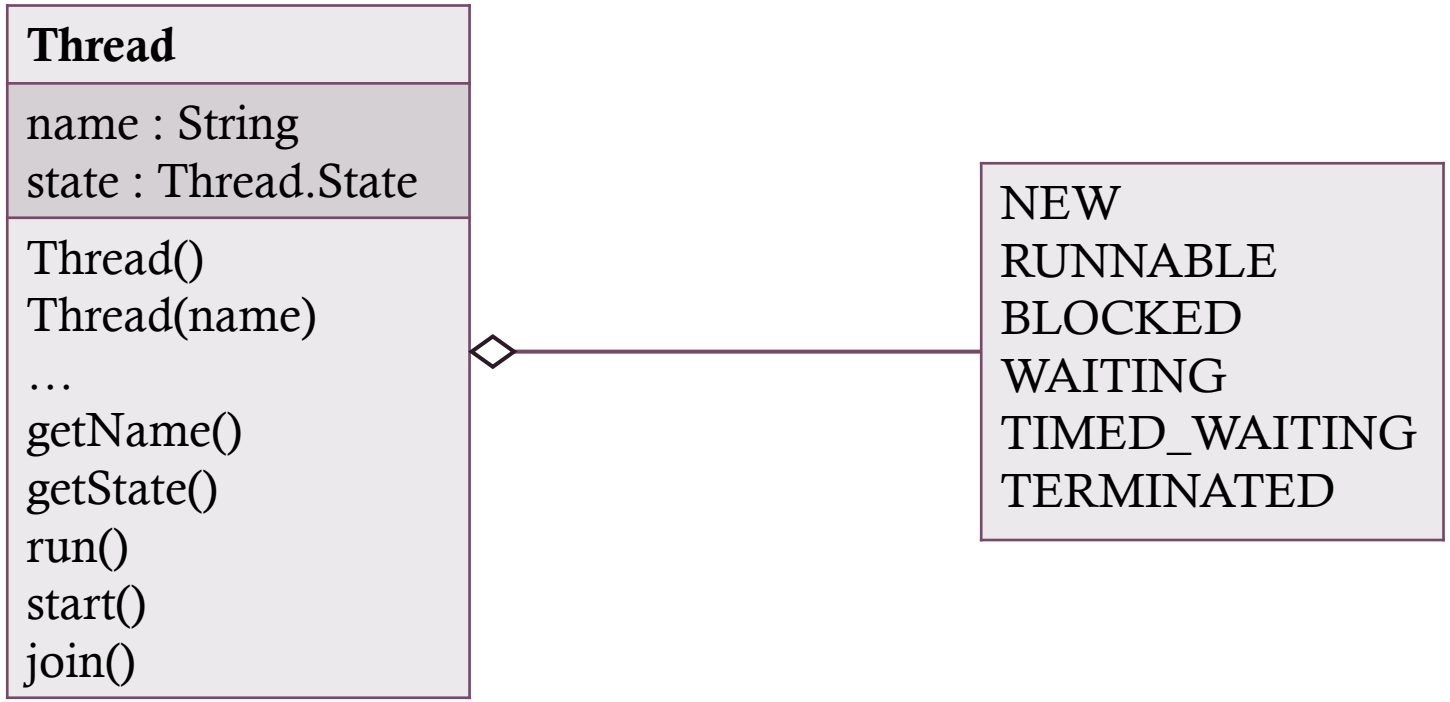
- An execution thread (aka a lightweight process) is a sequence of instructions being executed
- Threads share memory space but have their own stack frames
- A process can have multiple threads
- So far, we've only seen one or two execution threads running at a time

THREADS IN JAVA

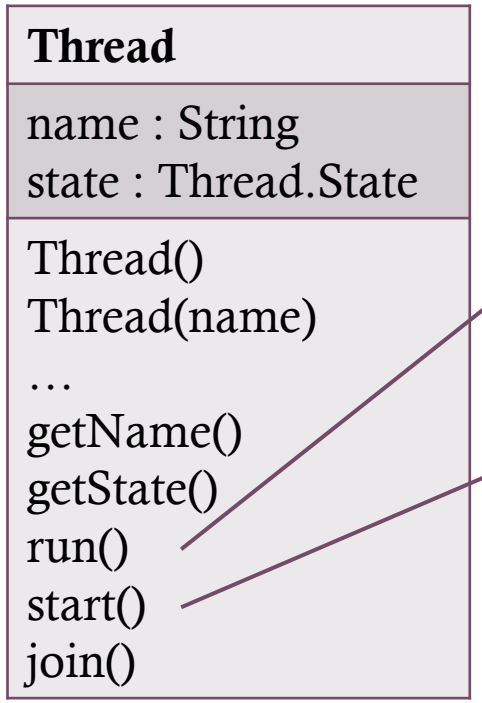
- Concurrency and threads are a core part of the java language (no import required!)



CLASS THREAD



CLASS THREAD



- run():
  - An ordinary method. Your code goes here!
- start():
  - Creates a new execution thread and starts executing run() in it.

RUN() VS START()

- Any code we want executed by a separate thread should go in the run() method
- When it's time to generate the threads, call start()
- start() has code behind the scenes to create a new thread and execute the run() method

- Tips:
  - Do not override start(), only override run()
  - Do not call run(), unless you want everything to run on the same thread

LET'S TRY AN EXAMPLE

THREAD STATES

- **NEW:** No execution thread (start() hasn't been called yet)
- **RUNNABLE:** Execution thread is running or waiting for the OS to run it
- **BLOCKED:** Waiting for a monitor lock
- **WAITING:** Waiting for another thread to wake this thread
- **TIMED_WAITING:** Waiting for a fixed amount of time to wake
- **TERMINATED:** Execution thread is finished

BASIC THREAD STATE PROGRESSION

State:

```
public static void main(String[] args) {
        Thread t1 = new Thread("thread1");
```

NEW


```
        t1.start();
```

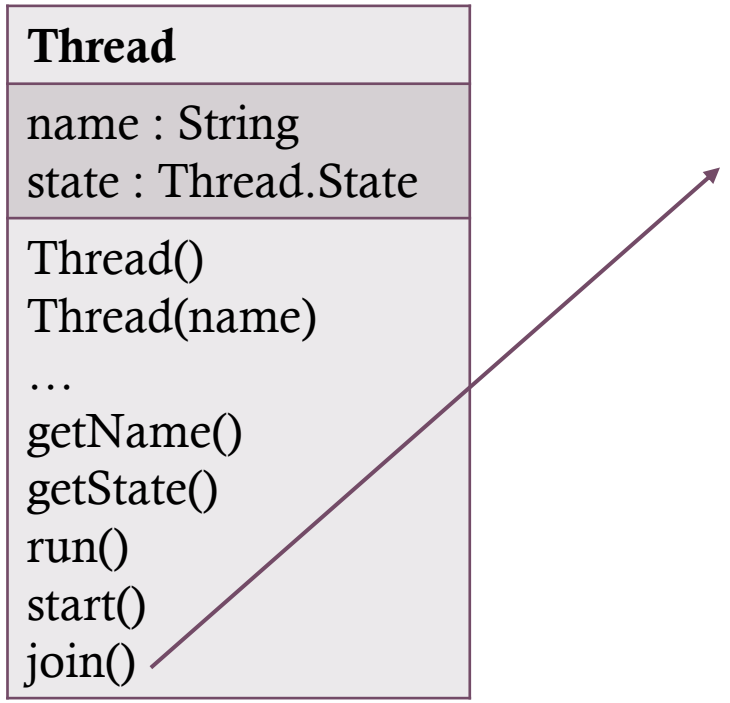
RUNNABLE

```
                    public void run(){
                        for (int i = 0; i < 20; i++){
                            System.out.println(i);
                        }
                    }
```

```
}
```

TERMINATED

JOIN()



- join():
  - Wait for a thread to reach the TERMINATED state, then move on to the next instruction
  - Can be used when worker threads need to finish before the main thread continues

MULTITHREADING EXAMPLES

- When you want things running independently
  - For example, if you want an animation running while a file downloads
  - Games often require multiple threads for smooth play
  - Web server that handles multiple clients simultaneously
- When you have a lot of work to do
  - Divide up into parts and run one thread on each part
  - Protein folding problem

REVIEW: DOES JAVA SUPPORT MULTIPLE INHERITANCE?

Suppose I want to make a Java Swing program like a clock

I probably need to extend JComponent or JPanel

But what if I also want the clock to run in a separate thread?

RUNNABLE INTERFACE

- Requires implementation of one method:
  - `public void run()`
- Allows any class (even a subclass) to have its own execution thread
- Bypasses issues with multiple inheritance

RUNNABLE INTERFACE CONTINUED

- Thread class has a constructor that takes in a Runnable object:
  - `Thread(Runnable target)`
- Assume the Clock class is defined as follows:

```
public class Clock extends JPanel implements Runnable
```

- Then, a Clock object could be passed into the Thread constructor:

```
Clock myClock = new Clock();
Thread t1 = new Thread(myClock);
t1.start();
```

- When start is called, Thread will execute myClock's run() method

UTILIZING RUNNABLE VIA LAMBDA EXPRESSION

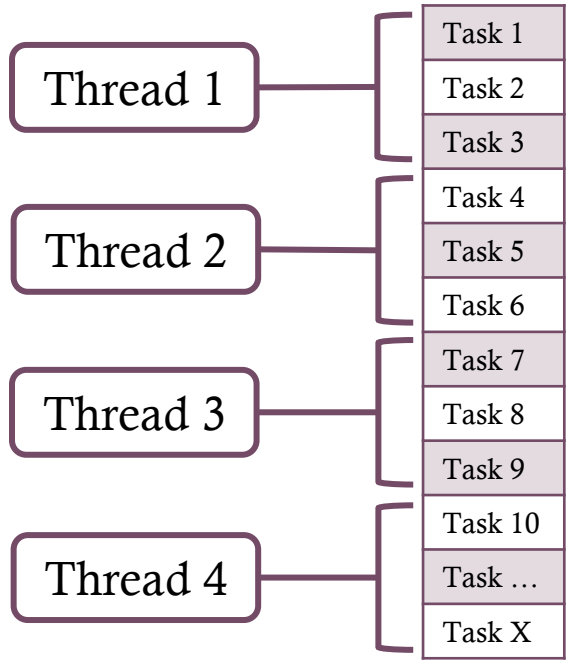
- Can be set up using a lambda expression:

```
Runnable r2 = () -> {
    System.out.println("Runnable with Lambda Expression");
};
new Thread(r2).start();
```

LET’S DO AN EXAMPLE

THREADS WORKING TOGETHER

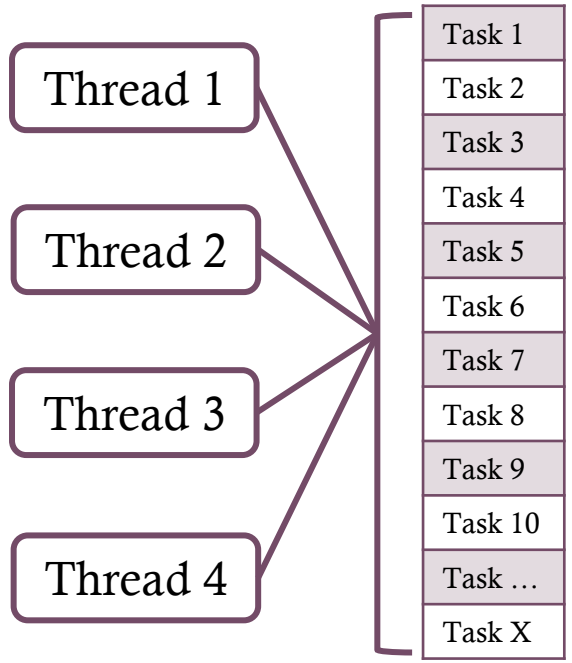
- Threads can easily do separate task
  - Unrelated tasks
  - Smaller pieces of the same task



THREADS WORKING TOGETHER

```
public static int[] work;
public static int nextJob = 0;
```

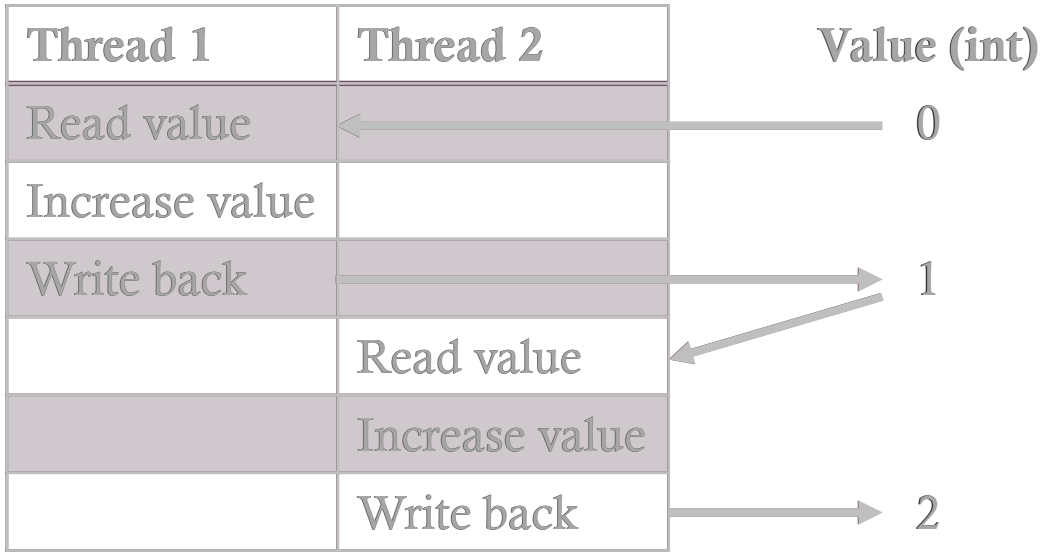
- Threads can easily do separate task
  - Unrelated tasks
  - Smaller pieces of the same task
- What if our pieces of work are not the same size?
  - Could choose to store all tasks in an array (or similar data structure)
  - Then, each thread just needs to know which task is next
  - This could be represented via a variable that indicates the next task
  - Threads would need to update the variable each time they take a task

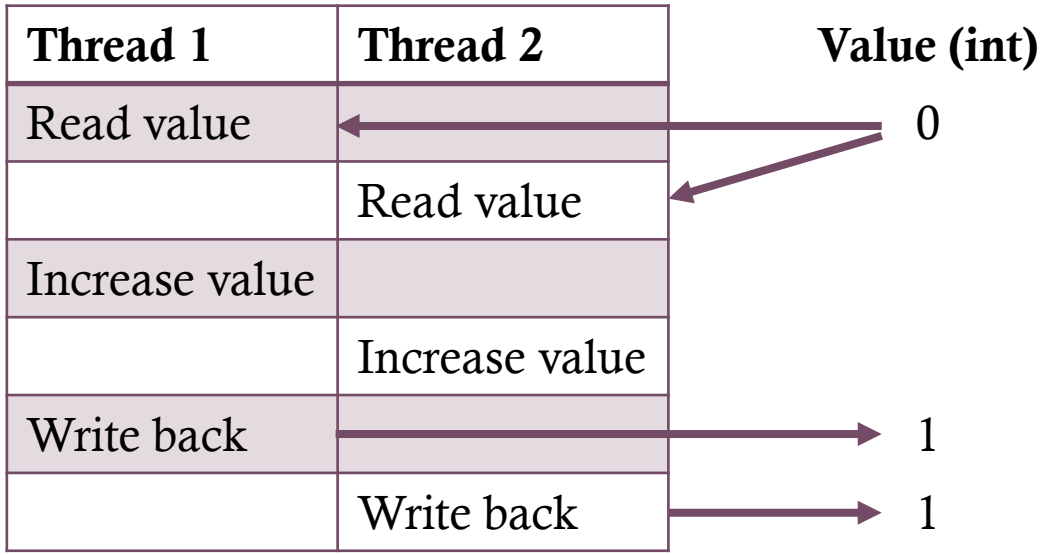


RACE CONDITIONS

- Each time a thread executes a piece of code, many machine code instructions are run
- Sometimes the OS will cause a delay in one of these instructions which could freeze the thread
- Increases the chance of multiple threads accessing the same variable/method simultaneously
- In the best case, this could lead to redundancy
- Could cause worse issues, like skipped tasks or unexpected/wrong output

RACE CONDITION VISUALIZATION





SOLUTIONS

- **Java monitors:** a “lock” or “stoplight” in each object that controls access to code sections
- **Synchronized:** Java keyword that says only one thread may access the code at a time
  - Note: if a thread is frozen while executing synchronized code, all other threads are locked out of that method until the frozen thread wakes up and finishes
- **Volatile:** Java keyword that ensures a variable is not cached and always accessed in a synchronized fashion
  - Can replace synchronized methods
  - Slows down code significantly

SYNCHRONIZED & VOLATILE EXAMPLE

- Assume each worker's run() method will call getNextJob() and doWork()
- Do either of these methods need to be synchronized?
- Do any of the variables need to be volatile?