CMSC 132: OBJECT-ORIENTED PROGRAMMING II

Threads in Java

Department of Computer Science
University of Maryland, College Park

Multitasking

- **Process** – an instance of a computer program that is currently executing
- **Multitasking** – the ability of a computer to give the illusion that multiple processes are running at the same time

  Example: Listening to music while working on a document and downloading a file

- **How does multitasking work?**

  - The CPU does some work on one task
  - Then quickly switches to the next task
  - This switching is managed by the operating system's **scheduler**

- **As a result:**

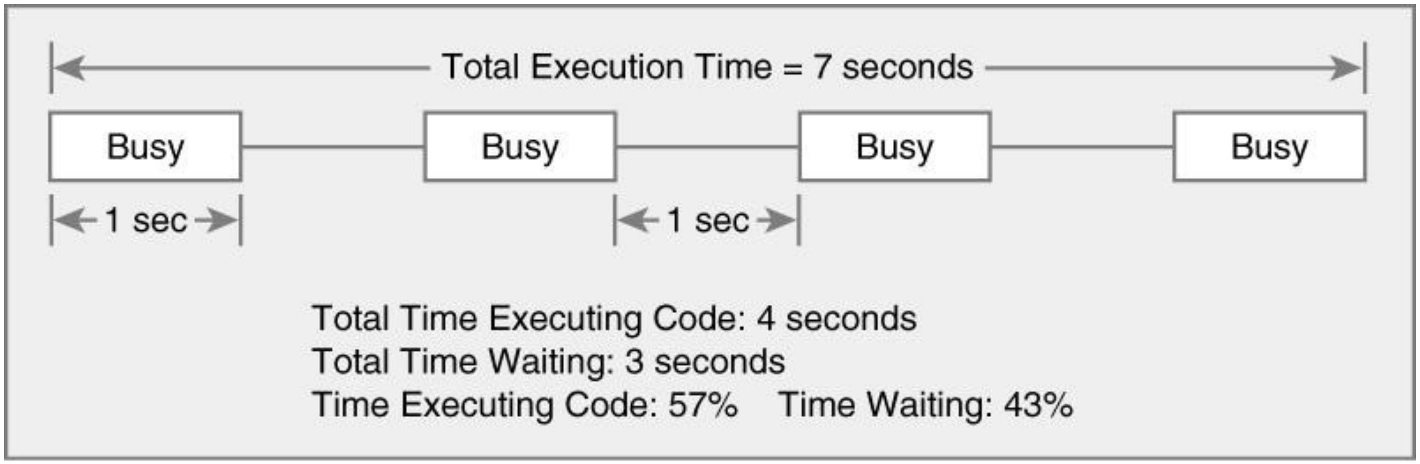
  - The computer *seems* to run tasks **concurrently**, even if there's only one CPU

- **What if the computer has multiple CPUs?**

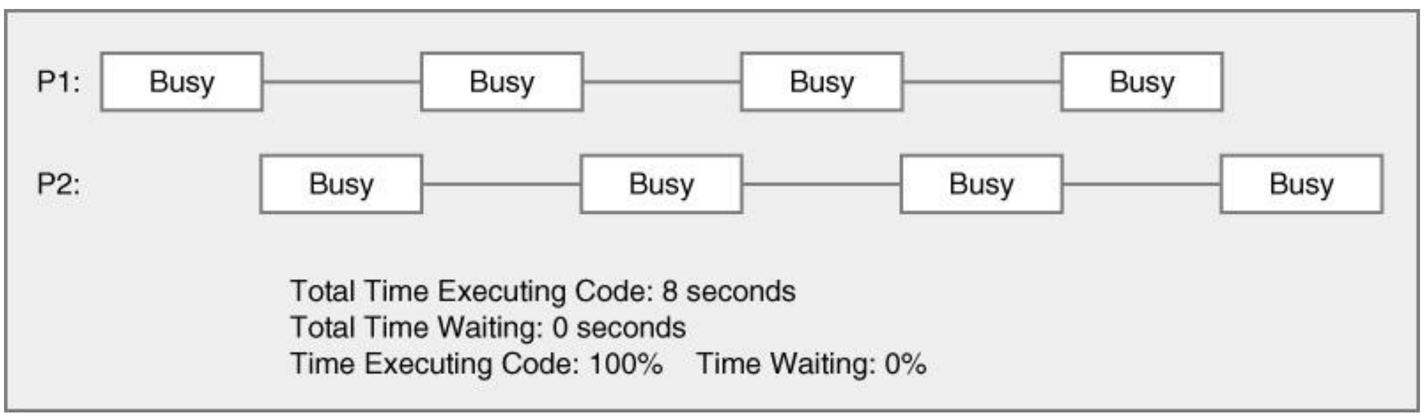
  - Now it can truly run multiple tasks in **parallel**
  - Same concept applies — but now the number of processes is not strictly limited by the number of CPUs

Multitasking Can Aid Performance

- Single task



- Two tasks



Perform Multiple Tasks Using Processes

- **Process –** an executable program loaded into memory
  - Has its own **address space** (independent memory)
  - Contains its own **variables** and **data structures**
  - Each process may run a **different program**
  - Processes communicate via the **operating system**, **files**, or **network**
- A process may contain multiple **threads**
  - Threads share the same memory space within the process
  - Allow a single program to perform **multiple tasks concurrently**
- **Example:**

  An audio/video application may:
  - Download data
  - Decompress video/audio
  - Play the media
  - Respond to user input (e.g., pause/seek)

More about Threads

Thread – a sequential stream of instructions within a program
- Also known as a **“lightweight process”**
- Shares memory with other threads in the same process

• **Each thread has its own execution context:**
- **Program counter** (tracks current instruction)
- **Call stack** (local variables and method calls)
- But shares the **heap** (global variables, objects) with other threads

• **Threads communicate via shared data access**
- **Advantage:** Less overhead than inter-process communication (IPC)
- **Disadvantage:** Increased risk of bugs due to shared mutable state (e.g., race conditions, deadlocks)

• **So far:**
- Our programs have had **one process, one thread** (the **main thread**)

• **Now:**
- We’ll study **multithreaded programming in Java**
  → One process, **multiple threads** working in parallel

Threads Match Real-World Structure

- **Many real-world systems involve multiple, independent tasks**
  - Example: A **web server** handles requests from many users
  - Each request is **separate** but served by the same program
- **Using threads to handle this structure:**
  - Server creates a **new thread** (or uses a thread from a pool)
    - → One thread per incoming request
  - Each thread handles:
    - → Reading the request
    - → Generating a response
    - → Sending back HTML or data
  - Threads **run in parallel**, sharing memory, cache, and network sockets
- **Benefits of this approach:**
  - Matches how clients interact: **simultaneous, independent requests**
  - **Improves responsiveness**: no request has to wait for others to finish
  - **Simplifies logic**: programmer can focus on one request at a time

Threads Improve Performance on Modern Hardware

- **Threads enable better hardware utilization:**
  - When one thread is **waiting** (e.g., for disk or network), others can run
  - **Multiple CPU cores** = true **parallel execution**
  - Threads share resources (like memory), reducing overhead
- **Benefits include:**
  - Higher **throughput**: more tasks done in less time
  - Shorter **response time** for users
  - Efficient handling of **mixed workloads** (CPU + I/O)
- **Summary:**
  - → Threads help model the problem **naturally**
  - → And take full advantage of **modern multi-core systems**

Creating Threads in Java – Approach 1 (Extending Thread)

- **Java provides two main ways to create threads:**
  - **Extend the Thread class** – *not recommended*
  - **Implement the Runnable interface** – *preferred*

Approach 1: Extending the Thread Class

- **Create a subclass of Thread**
- **Override the run() method** to define what the thread will do
- **Start the thread using start() (not run()!)**

```
public class MyT extends Thread {
    public void run() {
       System.out.println("Thread is running"); // Code for the thread
to execute
    }
}

MyT t = new MyT();   // Create a thread object
t.start();           // Start a new thread (runs run() in parallel)
```

Creating Threads in Java – Approach 1 (Extending Thread)

- **Notes:**
  - run() is what the thread does — called by the JVM when start() is used
  - start() creates a new thread and runs run() in parallel
  - **Avoid calling run() directly** – that runs on the main thread
- **Why Extending the Thread Class is *not* the preferred approach:**
  - Java only allows **single inheritance**, so extending Thread limits flexibility
  - Better to **separate "what to do" (Runnable)** from **"how to run it" (Thread)**

[Thread API]

See Examples: First see message package as an example of single threaded program, then see messageThreadExtends package (see *MyThreadExample* program last)

Approach 2: Using the Runnable Interface

- Define a class (worker) that implements the Runnable interface:

Runnable Interface API

- This class defines the task (in run()) to be executed in a separate thread.
- Two ways to use it:
- **Alternative 1**: Create a Thread object and pass the worker to its constructor.
- **Alternative 2**: Submit the worker to an **executor**

See Examples: `messageThreadRunnable` package (see *Runnable Example* program last)

Java Thread Lifecycle and States

- **Java threads** can be in one of the following states (as defined by Thread.State):
- **New**
  → Thread object is created but start() has not been called yet.
- **Runnable**
  → Thread is ready to run and waiting to be scheduled by the CPU.
  (May or may not be currently executing.)
- **Running** *(not an official state)*
  → When a **Runnable** thread is actually executing its code.
  (Internally still considered "Runnable" in Java.)
- **Blocked**
  → Thread is waiting to acquire a monitor lock (i.e., intrinsic lock for synchronization).
- **Waiting**
  → Thread is waiting indefinitely for another thread to perform a specific action (e.g., join(), wait()).
- **Timed Waiting**
  → Like **Waiting**, but with a time limit (e.g., sleep(ms), join(ms), wait(ms)).
- **Terminated**
  → Thread has completed execution or terminated due to an uncaught exception.

Java Thread Lifecycle and States

State Transitions

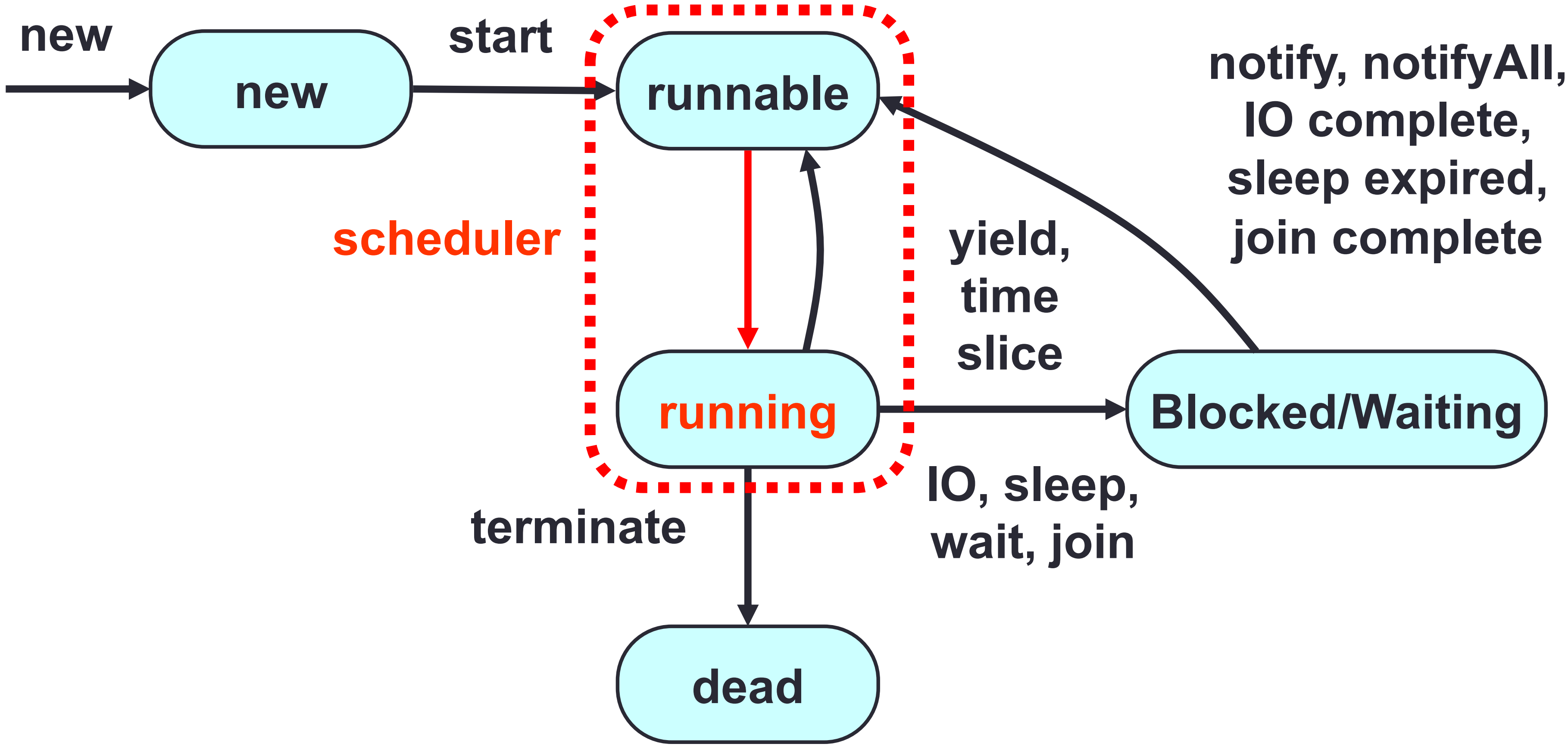
- Triggered by:
  - Method calls: start(), sleep(), wait(), notify(), join()...
  - JVM and system events: scheduling, I/O completion, run method returns…

Extra

- In Java, the states are defined by the Thread.State enum: Thread.State API Docs

Threads – Thread States

- State diagram



Running is a logical state → indicates runnable thread is actually running

Thread Scheduling in Java

- **Java thread scheduling is platform-dependent**
  - Behavior is determined by the **JVM** and **underlying OS**
  - Most modern systems use **preemptive scheduling**
- **What is Preemptive Scheduling?**
  - The **OS can interrupt** a running thread to switch to another thread
  - Enables **responsive multitasking**
  - Common on Windows, Linux, macOS
- **What is Non-Preemptive (Cooperative) Scheduling?**
  - A thread keeps running **until it voluntarily yields** control (e.g., via yield(), blocking I/O)
  - Less responsive, but simpler to implement
- **Java in Practice**
  - Most JVMs on modern systems use **preemptive scheduling**
  - **Java does not guarantee** a specific scheduling policy
  - Developers should write code that works **regardless of the scheduling style**

Waiting for a Thread to Finish: join()

- By default, threads run **independently** and **concurrently**.
- Sometimes, we want the **main thread to wait** for others to finish.
- Use join() to wait for a thread to complete
- Useful when a thread is doing work that the rest of the program depends on.
- Can throw InterruptedException → must handle or declare it.

Important

- You will limit concurrency if you do not start/join correctly.
- Suppose you want to run many threads **concurrently**:
  Start all the threads first, then **join on each one** afterward.
  Do not start one thread, join on it, start another thread, join on it, etc.
- The following is WRONG: t1.start() t1.join() t2.start() t2.join()
- Correct approach: t1.start() t2.start() t1.join() t2.join()

See `ThreadNoJoin followed by ThreadJoin`

Stopping a Thread in Java

- **Thread Lifecycle**:
  - A thread ends when the run() method completes.
- **Prematurely Stopping a Thread**:
  - Sometimes you may want to stop a thread before it finishes its task. For example:
    - If multiple threads are searching for a solution to a problem and one finds it, there’s no need for the others to keep running.
- **Deprecated Method: stop()**:
  - The stop() method is *deprecated* and should not be used, as it can lead to inconsistent thread states, resource leaks, and other issues.
- **Recommended Approach: interrupt()**:
  - The interrupt() method is a better, safer way to request that a thread stops its work.
  - Note that interrupt() doesn't force the thread to stop immediately—it signals that the thread should stop when it can.

Using the interrupt() Method

- **What interrupt() Does**:
  - The interrupt() method signals to the thread that it should consider stopping its work.
  - It doesn't stop the thread directly; instead, it sets an internal flag in the thread. The thread needs to check this flag to decide whether to stop.
- **Thread Behavior**:
  - It's the responsibility of the thread to **respond** to the interruption. The thread may ignore it or stop depending on its logic.
- **Checking for Interruption**:
  - The thread can check whether it has been interrupted using the Thread.interrupted() method. This returns true if the thread was interrupted.
  - A common pattern for checking interruptions is:

```
public void run() {
    while (!Thread.interrupted()) {
        // Perform work here
    }
    // Perform cleanup tasks, release resources
}
```

- The thread keeps running while Thread.interrupted() returns false.
- When the thread checks and finds the interruption flag set, it can exit the loop or perform cleanup operations.
- interrupted() clears the interrupt flag, so it's commonly called at the start of a loop or in the thread's primary task to handle an interruption promptly.

See: `ThreadInterruptExample`