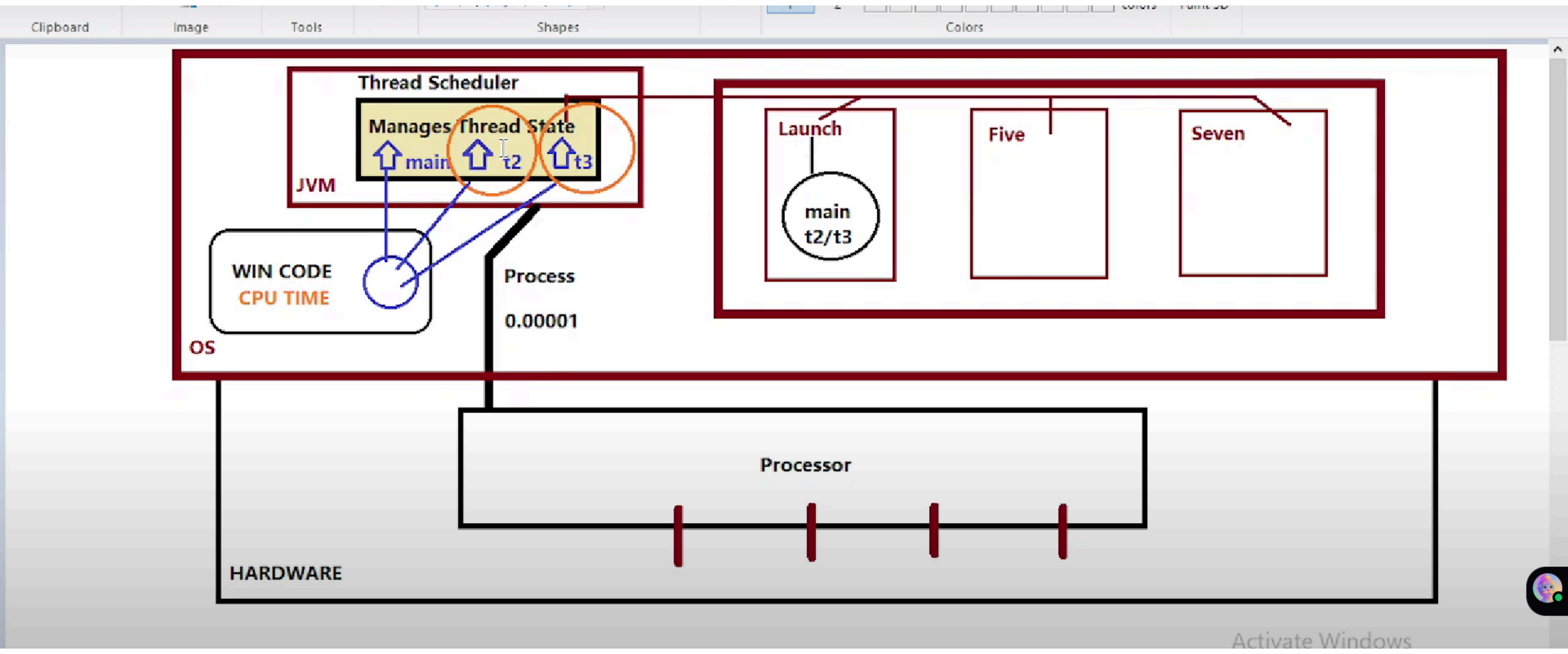
Lec 31 Multi-Threading Part 2 :

- JVM’s ThreadScheduler manages state of threads.



This lesson explains how Java handles multithreading, including what threads are, how they are created, and how they are managed by the JVM and operating system. It shows how to create and control threads in Java, both one-per-class and multiple threads per class, and describes how thread scheduling and execution are determined behind the scenes.

Introduction to Multithreading Concepts

Multithreading allows a program to perform multiple tasks at once. In Java, a "thread" is a lightweight process managed by the Java Virtual Machine (JVM), while a "process" is a heavier, independent running program managed by the operating system. Multitasking and multiprocessing are related but distinct concepts.

Thread Creation Basics in Java

Threads in Java are created by the JVM, but the programmer writes the logic. Threads enable multitasking, allowing multiple tasks to run seemingly at the same time. Java provides two main ways to create a thread: by extending the Thread class or by implementing the Runnable interface.

Thread Creation: Thread vs. Runnable

Extending the Thread class requires inheritance, while implementing Runnable allows for multiple inheritance since Java supports only single class inheritance but multiple interface implementations. Both methods let you define the thread’s behavior in a run() method.

The Main Thread in Java Applications

Every Java program starts with a main thread, created automatically by the JVM. This main thread manages the application’s flow and is associated with the main class and its main() method.

Sequential vs. Parallel Execution Example

If two classes (e.g., Five and Seven) each print a multiplication table, calling their methods sequentially in main() results in one table printing after the other. If an exception occurs and is not handled, the second table may not print.

Achieving Parallelism with Threads

To run both tables at the same time, separate threads are created for each class. The JVM manages these threads, enabling both tasks to appear to run simultaneously, even on a single-core processor, by rapidly switching between them (context switching).

Role of the Operating System and JVM

The operating system creates processes and manages time slicing and context switching between them. Within a process, the JVM manages threads, using a thread scheduler to decide which thread runs when.

JVM’s Thread Scheduler

The thread scheduler is a special part of the JVM that registers and manages all threads in a Java application. It decides which thread is ready to run, which is running, and which is waiting or terminated, but the actual switching is done by the OS.

Thread Life Cycle and States

Threads have several states: new (created), runnable (ready to run), running, waiting (or time-waited), and terminated. The thread scheduler manages transitions between these states, but the OS performs the actual execution.

OS and JVM Collaboration in Thread Execution

The OS gives CPU time to the Java process, and within that, the JVM’s thread scheduler manages which threads get to run. The main thread is created first, and additional threads are created as needed. The OS ultimately runs the threads based on the scheduler’s instructions.

Thread Execution Order and Non-Determinism

The order in which threads run is not guaranteed and can vary between runs or operating systems. The thread scheduler and OS decide the order based on internal logic, priorities, and other factors, making output unpredictable in multithreaded programs.

Sequential vs. Multithreaded Execution Clarified

Sequential execution runs tasks one after another, while multithreading allows tasks to overlap. Even with priorities, the actual order is not fixed, and it depends on the scheduler's implementation and system conditions.

One Class, One Thread Pattern

You can create one thread per class by extending the Thread class and overriding the run() method with the task logic. Objects are created for each class, and calling start() on each object starts the threads, which the scheduler then manages.

Runnable Interface and Functional Interfaces

The Runnable interface has a single abstract method, run(), making it a functional interface. The Thread class implements Runnable internally, allowing the run() method to be overridden for custom behavior.

How to Start Threads and Use run() Logic

To start a thread, you create an object of your class (extending Thread), then call start(), which triggers the run() method. Each thread runs its run() logic independently, and switching between threads is managed by the scheduler.

Thread Naming and Identification

Threads can be named for easier identification. By default, threads are named Thread-0, Thread-1, etc., but you can assign custom names using setName(). Thread logic can use the thread’s name to control behavior or output.

One Class, Multiple Threads Pattern

Multiple threads can be created from a single class by creating multiple objects and starting each as a thread. Each thread can have its own logic based on its name or other properties, allowing for flexible task assignment within the same class.

Using Thread Names for Conditional Logic

The run() method can use the thread's name to decide which task to perform, enabling different behaviors for different threads even within the same class. This is useful for assigning specific jobs to specific threads.

Summary and Practice Advice

Understanding the basics and internals of multithreading is crucial before moving to advanced topics like synchronization or deadlocks. Practicing thread creation with both one-class-one-thread and one-class-multiple-threads patterns builds a strong foundation.