Concurrency-1 Introduction to Processes and Threads

In this tutorial, we will cover the following concepts.

- How Computer Applications Run
- Concurrency: Real-World Applications of Threads

  ```
  - Google Docs
  - Music Player
  - Adobe Lightroom
  ```

- Processes and Threads

  ```
  - Benefits of Multithreading
  - Challenges of Multithreading
  ```

- Concurrent vs Parallel Execution
- Multithreading in Java
  - Thread Creation

    ```
        - Subclass of Thread Class
        - Using Runnable
    ```

  - Starting a Thread
  - Problem Statement 1 : Number Printer
- Additional Concepts
  - Commonly used Methods on Threads
  - Problem Statement 2: Factorial Computation Task
  - Thread Lifecycle & States

Understanding How Computer Applications Run

Computer applications are complex systems that run on a computer's operating system, interacting with hardware and software components to perform various tasks. To comprehend how these applications operate efficiently, it's essential to delve into fundamental concepts like processes, threads, CPU scheduling, multithreading, and parallel execution. Let us understand how a program runs.

1. Programs / Processes

Programs: These are sets of instructions for the computer. Each application you use, such as a web browser or word processor, is a program. Processes: When you open a program, it becomes a process. A process is an instance of a program in execution.

2. Memory Allocation

When you start a program, the operating system (OS) allocates memory to it. This memory contains the program's code, data, and other necessary information.

3. Processor (CPU) Execution

The Central Processing Unit (CPU) is the brain of the computer. It fetches instructions from memory and executes them. Each process takes turns using the CPU. The OS manages this by employing a technique called CPU Scheduling.

4. Context Switching

The CPU rapidly switches between different processes. This is known as context switching. The OS saves the current state of a process, loads the state of the next process, and hands control to it.

5. Multitasking

The ability of a computer to execute multiple processes concurrently is called multitasking. While it may seem like everything is happening at once, the CPU is actually rapidly switching between processes.

6. Parallel Execution

In some systems, especially those with multiple processors or cores, true parallel execution can occur. This means multiple processes genuinely run simultaneously.

7. Threads

A process can be further divided into threads. Threads within a process share the same resources but can execute independently. Multithreading allows for parallel execution within a single process.

8. Synchronization

When multiple processes or threads share resources (like data), synchronization mechanisms are employed to avoid conflicts. This ensures that data remains consistent.We will discuss synchronization in great detail in coming lectures.

9. Task Management by the Operating System

The OS keeps track of all running processes and manages their execution. It assigns priorities, allocates resources, and ensures fair access to the CPU. This is done by scheduling algorithms. Some of the popular scheduling algorithms are as follows -

- First-Come-First-Serve (FCFS): Processes are executed in the order they arrive.
- Shortest Job Next (SJN): The process with the shortest execution time is selected.
- Round Robin (RR): Each process gets a fixed time slice, then moves to the back of the queue.
- Priority Scheduling: Processes are assigned priorities, and the highest priority process is executed first.

Modern CPUs often employ a mix of static and dynamic scheduling strategies. Advanced techniques, like predictive algorithms, may be used to anticipate the next process to run.

10. Interrupts

The CPU can be interrupted to handle external events, like input from a user or data arriving from a network.Interrupts are crucial for maintaining responsiveness in a multitasking environment.

11. Termination of Processes

When a program finishes its task or is closed by the user, the associated process is terminated. The OS reclaims the allocated resources and frees up memory.

12. Efficient Resource Utilization

The goal is to efficiently utilize the available resources, ensuring that each running application gets its fair share of CPU time. In summary, the execution of multiple applications or processes involves careful management by the operating system, with the CPU rapidly switching between tasks, allocating resources, and ensuring that everything runs smoothly. Multitasking and, in some cases, parallel execution contribute to the efficiency and responsiveness of modern computer systems.

Conclusion

Understanding how computer applications run involves grasping the intricacies of processes, threads, CPU scheduling, multithreading, and parallel execution. As technology evolves, mastering these concepts becomes increasingly important for developing efficient and responsive applications. Experimenting with these concepts in programming languages and frameworks will deepen your understanding and proficiency in building robust and high-performance software.

Concurrency: Real-World Applications of Threads

Concurrent programming, which involves the execution of multiple tasks simultaneously, is a fundamental concept in modern software development. One powerful mechanism for achieving concurrency is the use of threads. Let's explore how threads are employed in real-world applications, focusing on Google Docs, Music Players, and Adobe Lightroom.

Concurrent programming enables multiple operations to progress in overlapping time intervals. Threads, the smallest units of execution within a process, are instrumental in achieving concurrency. They allow different parts of a program to run concurrently, enhancing efficiency and responsiveness.

1. Google Docs: Collaborative Editing

Google Docs exemplifies the power of concurrency through its collaborative editing feature. When multiple users are editing a document simultaneously, threads come into play. Each user's edits are handled by a separate thread, ensuring that changes made by one user do not disrupt the editing experience of others.

Threads in Google Docs

- Thread per User: Each user's editing actions are processed by an individual thread.
- Conflict Resolution: Threads synchronize to resolve conflicts and merge edits seamlessly.
- Auto-Suggest/Auto-complete: A separate thread can run spell check for the words you write.
- UI Thread: A separate thread can continuously update UI for the users.

2. Music Players: Smooth Playback and User Interaction

In music players like Spotify or iTunes, threads are crucial for delivering a smooth user experience during playback while allowing users to interact with the application concurrently.

How Threads Work in Music Players

- Playback Thread: A dedicated thread manages audio playback, ensuring uninterrupted streaming.
- User Interface Thread: Another thread handles user interactions, such as browsing playlists or adjusting settings.
- Parallel Execution: Threads allow simultaneous playback and user interactions without one affecting the other.

3. Adobe Lightroom: Image Processing

In photo editing applications like Adobe Lightroom, where resource-intensive tasks like image processing are common, threads are employed to maintain responsiveness and reduce processing times.

How Threads Work in Lightroom

- Image Processing Threads: Multiple threads handle the processing of different parts of an image concurrently.
- Background Tasks: Threads enable background tasks like importing photos while allowing users to continue editing.
- Responsive UI: Threads ensure that the user interface remains responsive even during computationally intensive operations.

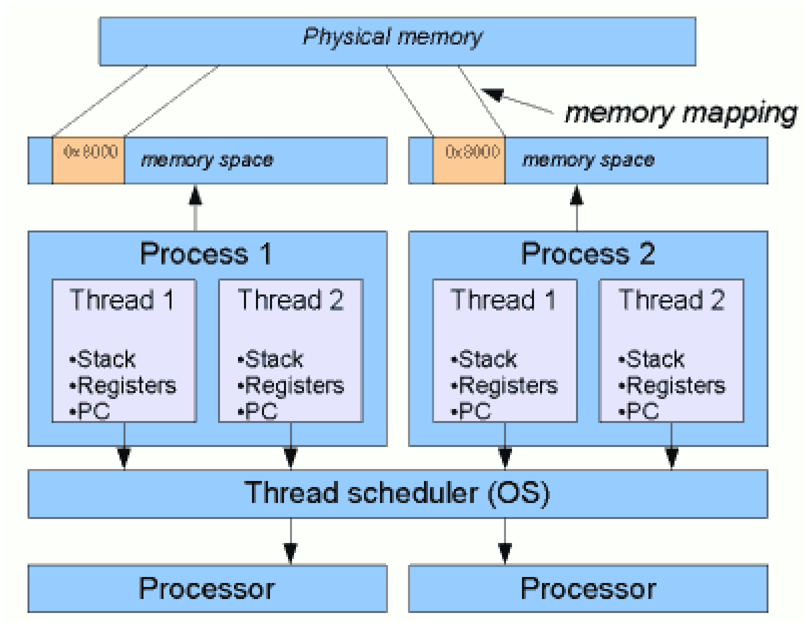
Processes and Threads - Deep Dive

A **process** is an independent program in execution. It has its own memory space called heap, code, data, and system resources. The heap isn't shared between two applications or two processes, they each have their own. The terms process and application are often used interchangeably. Processes enable multiple tasks to run concurrently, offering isolation and independence.

Process Lifecycle

- Creation: When a program is launched, it is loaded into memory, a process is created.
- Execution: The process runs its instructions.
- Termination: The process completes its execution or is terminated.

A **thread** is the smallest unit of execution within a process. Multiple threads can exist within a single process, sharing the same resources like memory but executing independently.



Benefits of Multithreading - Why use multiple threads?

- Performance: Threads can execute concurrently, enhancing performance.
- Responsiveness: Multithreading allows a program to remain responsive during time-consuming tasks. This is especially helpful in applications with user interfaces.
- Efficiency: Exploiting parallelism improves overall system performance.
- One of the most common reasons, is to offload long running tasks. Instead of tying up the main thread, we can create additional threads, to execute tasks that might take a long time. This frees up the main thread so that it can continue working, and executing, and being responsive to the user.
- You also might use multiple threads to process large amounts of data, which can improve performance, of data intensive operations.
- A web server, is another use case for many threads, allowing multiple connections and requests to be handled, simultaneously.
- Resource Sharing: Threads within a process share resources, reducing overhead.

Challenges

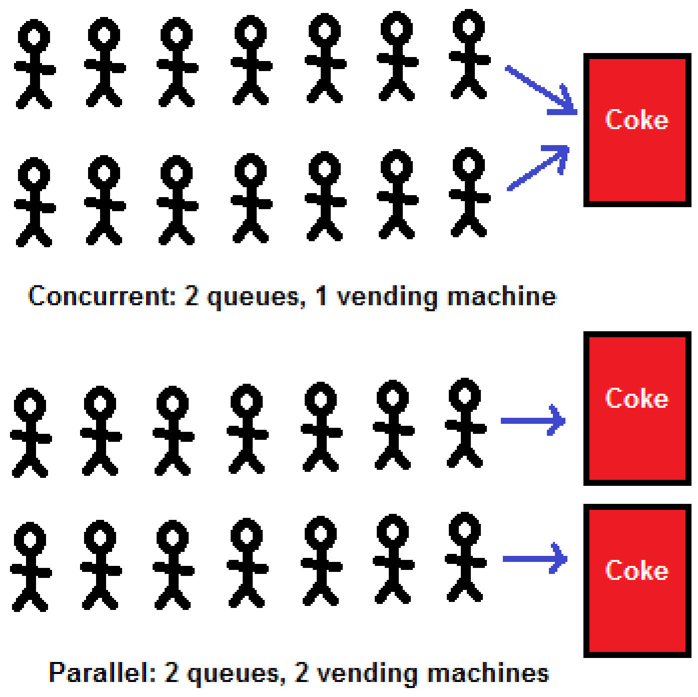
- Data Synchronization: Threads may need to synchronize access to shared data to prevent conflicts.
- Deadlocks: Concurrent threads might lead to situations where each is waiting for the other to release a resource.

We will address these challenges in the up-coming classes.

Concurrent Execution vs Parallel Execution

- Concurrent execution refers to the ability of a system to execute multiple tasks or processes at the same time, appearing to overlap in time. Concurrent Execution can happen on single core as well.
- Parallel execution involves the simultaneous execution of multiple tasks or processes using multiple processors or cores. Multiple cores are must for truly

parallel execution.


Coke
Concurrent: 2 queues, 1 vending machine
Coke
Coke
Parallel: 2 queues, 2 vending machines

Data Parallelism: Dividing a task into subtasks processed concurrently. Task Parallelism: Assigning multiple independent tasks to separate processors/cores.

Key Differences

Concurrent Execution - Tasks may overlap in time but not necessarily execute simultaneously. **Parallel Execution** - Tasks are actively running at the same time on separate processors.

Resource Utilization

Concurrent Execution - Utilizes a single processor by interleaving tasks. **Parallel Execution** - Utilizes multiple processors, ensuring more tasks are completed in the same time frame.

Hardware Requirement

Concurrent Execution - Can occur on a system with a single processor. **Parallel Execution** - Requires multiple processors or cores.

Example

Concurrent Execution - Multiple applications running on a single-core processor. **Parallel Execution** - Image processing tasks being performed simultaneously on different cores of a multi-core processor.

Multithreading in Java

In the Java, multithreading is driven by the core concept of a Thread. There are two ways to create Threads in Java.

Thread Class: Java provides the Thread class, which serves as the foundation for creating and managing threads.

Runnable Interface: The Runnable interface is often implemented to define the code that a thread will execute.

Lets write some logic that runs in a parallel thread by using the Thread framework. In the below code example we are creating two threads and running them concurrently.

Way-1 : Subclassing a Thread Class

```java
public class NewThread extends Thread {
    public void run() {
            // business logic
            ...
        }
    }
}
```

Class to initialize and start our thread.

```java
public class MultipleThreadsExample {
    public static void main(String[] args) {
        NewThread t1 = new NewThread();
        t1.setName("MyThread-1");
        NewThread t2 = new NewThread();
        t2.setName("MyThread-2");
        t1.start();
        t2.start();
    }
}
```

Way -2 : Using Runnable (Preferred Way)

```java
class SimpleRunnable implements Runnable {
    public void run() {
        // business logic
    }
}
```

```java
public class Main {
    public static void main(String[] args) {
        Thread t = new Thread(new SimpleRunnable());
        t.start();
    }
}
```

The above SimpleRunnable is just a task which we want to run in a separate thread. There’re various approaches we can use for running it; one of them is to use the Thread class.

Simply put, we generally encourage the use of Runnable over Thread: When extending the Thread class, we’re not overriding any of its methods. Instead, we override the method of Runnable (which Thread happens to implement).

- This is a clear violation of IS-A Thread principle
- Creating an implementation of Runnable and passing it to the Thread class utilizes composition and not inheritance – which is more flexible
- After extending the Thread class, we can’t extend any other class
- From Java 8 onwards, Runnables can be represented as lambda expressions

```java
public class ThreadWithLambdaExample {
    public static void main(String[] args) {
        // Creating a thread with a Runnable implemented as a lambda expression
        Thread myThread = new Thread(() -> {
                System.out.println(Thread.currentThread().getName());
                ...
            }
        });

         // Starting the thread
        myThread.start();
    }
}
```

We create a new Thread and pass a Runnable as a lambda expression directly to its constructor. The lambda expression defines the code to be executed in the new thread. In this case, it's a simple prints message that prints the name of Thread.

Starting a Thread - Behind the Scenes

When you call `thread.start()` in Java, it initiates the execution of the thread and invokes the run method of the thread. Here's a step-by-step explanation of what happens:

1. Thread Initialization

If you have a class that extends the Thread class, or if you have a class that implements the Runnable interface, you create an instance of that class, which represents the thread.

```java
Thread myThread = new MyThread(); // or Thread myThread = new Thread(new
MyRunnable());
```

The thread is in the "new" state after initialization. When you call start(), the thread transitions to the "runnable" state. It is ready to run but is waiting for its turn to be scheduled by the Java Virtual Machine (JVM).

```java
myThread.start();
```

2. Thread Scheduling:

The JVM's scheduler determines when the thread gets CPU time for execution. The actual timing is managed by the operating system, and it may vary.

3. run() Method Execution:

Once the thread is scheduled, the JVM calls the run method of the thread. The run method contains the code that will be executed in the new thread.

```java
class MyThread extends Thread {
    public void run() {
        // Code to be executed by the thread
    }
}
```

If you implemented Runnable instead:

```java
class MyRunnable implements Runnable {
    public void run() {
        // Code to be executed by the thread
    }
}
```

4. Concurrent Execution:

If there are multiple threads in the program, they may execute concurrently, with each thread running independently, potentially interleaving their execution.

5. Thread Termination:

The run method completes its execution, and the thread transitions to the "terminated" state. The thread is no longer active.

Important Notes

- Direct run Method Invocation: Calling the run method directly (myThread.run()) will not start a new thread; it will execute the run method in the current thread (ie main thread).
- One-Time Execution: The start method can only be called once for a thread. Subsequent calls will result in an IllegalThreadStateException.
- In summary, calling thread.start() initiates the execution of a new thread, and the JVM takes care of the thread scheduling and execution of the run method in a separate concurrent context.

Problem Statement - 1 Number Printer

Write a program to print numbers from 1 to 100 using 100 different threads. Since you can't control the order of execution of threads, it is okay to get these numbers in any order. Hint: Create a Runnable Task, which prints a single number.

Solution

NumberPrinter.java

```java
public class NumberPrinter implements Runnable {
    int number;
    NumberPrinter(int number){
        this.number = number;
    }
    @Override
    public void run(){
        System.out.println("Printing "+number + " from 
"+Thread.currentThread().getName());
    }
}
```

Main.java

```
public class Main {
    public static void main(String[] args) {
        for(int i=0; i<100;i++){
           Thread t = new Thread(new NumberPrinter(i));
           t.start();
        }
    }
}
```

Sample Output

```
Printing 3 from Thread-3
Printing 19 from Thread-19
Printing 14 from Thread-14
Printing 6 from Thread-6
Printing 21 from Thread-21
Printing 22 from Thread-22
Printing 0 from Thread-0
Printing 10 from Thread-10
...
Printing 94 from Thread-94
Printing 95 from Thread-95
Printing 96 from Thread-96
Printing 97 from Thread-97
Printing 98 from Thread-98
Printing 99 from Thread-99
```

Additonal Concepts (Optional)

Lets cover some more advanced concepts related to Threads.

Commonly used Methods on Threads

In Java, the Thread class provides several commonly used methods for managing and controlling threads. Here are some of the key methods:

1. start()

Initiates the execution of the thread, causing the run method to be called. Usage `myThread.start();`

2. run()

Contains the code that will be executed by the thread. This method needs to be overridden when extending the Thread class or implementing the Runnable interface. Usage: Defined by the user based on the specific task.

3. sleep(long milliseconds)

Description: Causes the thread to sleep for the specified number of milliseconds, pausing its execution. Usage: `Thread.sleep(1000);`

4. join()

Waits for the thread to complete its execution before the current thread continues. It is often used for synchronization between threads. Usage: `myThread.join();`

5. interrupt()

Interrupts the thread, causing it to stop or throw an InterruptedException. The thread must handle interruptions appropriately. Usage: `myThread.interrupt();`

6. isAlive():

Returns true if the thread has been started and has not yet completed its execution, otherwise returns false. Usage: `boolean alive = myThread.isAlive();`

7. setName(String name)

Sets the name of the thread. Usage: `myThread.setName("MyThread");`

8. getName()

Returns the name of the thread. Usage: `String threadName = myThread.getName();`

9. setPriority(int priority)

Sets the priority of the thread. Priorities range from Thread.MIN_PRIORITY to Thread.MAX_PRIORITY. Usage: `myThread.setPriority(Thread.MAX_PRIORITY);`

10. getPriority()

Returns the priority of the thread. Usage: `int priority = myThread.getPriority();`

11. currentThread()

Returns a reference to the currently executing thread object. Usage: `Thread currentThread = Thread.currentThread();`

These methods provide essential functionality for managing thread execution, synchronization, and interaction. When working with threads, it's crucial to understand and use these methods effectively to create robust and efficient concurrent programs.

Problem Statement - 2 Factorial Computation Task

Write a program that computes Factorial of a list of numbers. Each factorial should be computed on a separate thread. For each factorial calculation, do not wait for more than 2 seconds. Hint: Use the join() method on each factorial thread, before main starts executing again.

Solution

FactorialThread.java

```java
import java.math.BigInteger;

public class FactorialThread extends Thread {
    private long number;
    private BigInteger result;
    private boolean isFinished;

    FactorialThread(long number){
        this.number = number;
        result = BigInteger.valueOf(0); //Or BigInteger.ZERO;
        isFinished = false;
    }

    @Override
    public void run() {
        //Business Logic
        result = factorial(number);
        isFinished = true;
    }
    BigInteger factorial(long n){
        BigInteger ans = BigInteger.ONE;
        for(long i=2; i<=n; i++){
            ans = ans.multiply(BigInteger.valueOf(i));
        }
        return ans;
    }

    BigInteger getResult(){
        return result;
    }
    boolean isFinished(){
        return isFinished;
    }
}
```

```java
import java.math.BigInteger;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

public class Main {

    // Task calculate Factorial of List of Numbers
    public static void main(String[] args) throws InterruptedException {

        List<Long> inputNumbers = Arrays.asList(100000000L, 3435L, 35435L, 2324L,
4656L, 23L, 5556L);
        List<FactorialThread> threads = new ArrayList<>();
        for(long number:inputNumbers){
            FactorialThread t = new FactorialThread(number);
            //System.out.println(t.getState());
            threads.add(t);
        }

        for(Thread t:threads){
            t.start();
        }

        for(Thread t:threads){
            t.join(2000);
        }

        //--------------------//
        for(int i=0;i<inputNumbers.size();i++){
            FactorialThread t = threads.get(i); //ith Thread Object
            if(t.isFinished()){
                System.out.println(t.getResult());
            }
            else{
                System.out.println("Couldn't complete calc in 2s");
            }
        }
        System.out.println("Main is completed!");
    }
}
```

Thread Life Cycle

During thread lifecycle, threads go through various states. The `java.lang.Thread` class contains a static State enum – which defines its potential states. During any given point of time, the thread can only be in one of these states:

- **NEW** – a newly created thread that has not yet started the execution
- **RUNNABLE** – either running or ready for execution but it’s waiting for resource allocation
- **BLOCKED** – waiting to acquire a monitor lock to enter or re-enter a synchronized block/method
- **WAITING** – waiting for some other thread to perform a particular action without any time limit
- **TIMED_WAITING** – waiting for some other thread to perform a specific action for a specified period
- **TERMINATED** – has completed its execution

1.NEW

A NEW Thread (or a Born Thread) is a thread that’s been created but not yet started. It remains in this state until we start it using the start() method.

The following code snippet shows a newly created thread that’s in the NEW state:

```java
Runnable runnable = new NewState();
Thread t = new Thread(runnable);
System.out.println(t.getState());
```

Since we’ve not started the mentioned thread, the method `t.getState()` prints:

```
NEW
```

2. Runnable

When we've created a new thread and called the start() method on that, it's moved from NEW to RUNNABLE state. Threads in this state are either running or ready to run, but they're waiting for resource allocation from the system.

In a multi-threaded environment, the Thread-Scheduler (which is part of JVM) allocates a fixed amount of time to each thread. So it runs for a particular amount of time, then leaves the control to other RUNNABLE threads.

For example, let's add `t.start()` method to our previous code and try to access its current state:

```java
Runnable runnable = new NewState();
Thread t = new Thread(runnable);
t.start();
System.out.println(t.getState());
```

This code is most likely to return the output as:

```
RUNNABLE
```

Note that in this example, it's not always guaranteed that by the time our control reaches `t.getState()`, it will be still in the RUNNABLE state.

It may happen that it was immediately scheduled by the Thread-Scheduler and may finish execution. In such cases, we may get a different output.

3. BLOCKED

A thread is in the BLOCKED state when it's currently not eligible to run. It enters this state when it is waiting for a monitor lock and is trying to access a section of code that is locked by some other thread.

Let's try to reproduce this state:

```java
public class BlockedState {
    public static void main(String[] args) throws InterruptedException {
        Thread t1 = new Thread(new DemoBlockedRunnable());
        Thread t2 = new Thread(new DemoBlockedRunnable());

        t1.start();
        t2.start();

        Thread.sleep(1000); //pause so that t2 states changes during this time
        System.out.println(t2.getState());
        System.exit(0);
    }
}

class DemoBlockedRunnable implements Runnable {
    @Override
    public void run() {
        commonResource();
    }

    public static synchronized void commonResource() {
        while(true) {
            // Infinite loop to mimic heavy processing
            // 't1' won't leave this method
            // when 't2' try to enter this
        }
    }
}
```

In this code:

We’ve created two different threads – t1 and t2, t1 starts and enters the synchronized commonResource() method; this means that only one thread can access it; all other subsequent threads that try to access this method will be blocked from the further execution until the current one will finish the processing.

When t1 enters this method, it is kept in an infinite while loop; this is just to imitate heavy processing so that all other threads cannot enter this method

Now when we start t2, it tries to enter the commonResource() method, which is already being accessed by t1, thus, t2 will be kept in the BLOCKED state. Being in this state, we call `t2.getState()` and get the output as:

```
BLOCKED
```

4. WAITING

A thread is in WAITING state when it’s waiting for some other thread to perform a particular action. According to JavaDocs, any thread can enter this state by calling any one of the following three methods:

- object.wait()
- thread.join() or
- LockSupport.park()

Note that in wait() and join() – we do not define any timeout period as that scenario is covered in the next section.

In this example, thread-1 starts thread 2 and waits for thread-2 to finish using `thread.join()` method. During this time t1 is in `WAITING` state.

Simple Runnable.java - Thread 1

```
public class SimpleRunnable implements Runnable{

    @Override
    public void run(){
        Thread t2 = new Thread(new SimpleRunnableTwo());
        t2.start();
        try {
            t2.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

Simple Runnable 2 - Thread 2

```
public class SimpleRunnableTwo implements Runnable {
    @Override
    public void run() {
        try{
            Thread.sleep(5000);
        }
        catch(InterruptedException e){
            Thread.currentThread().interrupt();
            e.printStackTrace();
        }
    }
}
```

Main

```
public class Main {
    public static void main(String[] args) throws InterruptedException {
        Thread t1 = new Thread(new SimpleRunnable());
        t1.start();

        Thread.sleep(1000); //1ms pause
        System.out.println("T1 :"+ t1.getState()); //T1 is waiting state
        System.out.println("Main :" + Thread.currentThread().getState());
    }
}
```

5. TIMED WAITING

A thread is in `TIMED_WAITING` state when it’s waiting for another thread to perform a particular action within a stipulated amount of time.

According to JavaDocs, there are five ways to put a thread on TIMED_WAITING state:

- thread.sleep(long millis)
- wait(int timeout) or wait(int timeout, int nanos)
- thread.join(long millis)
- LockSupport.parkNanos
- LockSupport.parkUntil

Here, we’ve created and started a thread t1 which is entered into the sleep state with a timeout period of 5 seconds; the output will be `TIMED_WAITING`.

```
public class SimpleRunnable implements Runnable{
    @Override
    public void run() {
        try{
            Thread.sleep(5000);
        }
        catch(InterruptedException e){
            e.printStackTrace();
        }
    }
}
```

In Main, if you check the state of T1 after 2s it will be `TIMED WAITING`

```
public class Main {
    public static void main(String[] args) throws InterruptedException {
        Thread t1 = new Thread(new SimpleRunnable());
        t1.start();

        Thread.sleep(2000);
        System.out.println(t1.getState());
    }
}
```

6. TERMINATED

This is the state of a dead thread. It’s in the `TERMINATED` state when it has either finished execution or was terminated abnormally. There are different ways of terminating a thread.

Let’s try to achieve this state in the following example:

```
public class TerminatedState implements Runnable {
    public static void main(String[] args) throws InterruptedException {
        Thread t1 = new Thread(new TerminatedState());
        t1.start();

        // The following sleep method will give enough time for
        // thread t1 to complete
        Thread.sleep(1000);
        System.out.println(t1.getState());
    }

    @Override
    public void run() {
        // No processing in this block

    }
}
```

Here, while we’ve started thread t1, the very next statement Thread.sleep(1000) gives enough time for t1 to complete and so this program gives us the output as:

```
TERMINATED
```

--End---