	SHRI SANT GAJANAN MAHARAJ COLLEGE ENGG. SHEGAON		LABORATORY MANUAL
	PRACTICAL EXPERIMENT INSTRUCTION SHEET		
	EXPERIMENT TITLE: **Multithreading in Linux**		
EXPERIMENT NO.: SSGMCE/WI/ASH/01/1A6/01		ISSUE NO.:	**ISSUE DATE:**26.08.23
REV. DATE: 26.08.23	REV. NO.: 00	DEPTT. : INFORMATION TECHNOLOGY.	
LABORATORY: OPERATING SYSTEMS		SEMESTER : IV	PAGE NO. 01 OF 05

Date:

MULTITHREADING IN LINUX

01. AIM :

Write a program to implement multithreading.

02. FACILITIES :

Linux Operating System: The program is to be executed on a Linux machine.

Java Development Kit (JDK): The program uses Java's multithreading capabilities which are part of the standard Java library.

IDE or Command Line: You can run the program using a text editor or an IDE like IntelliJ IDEA or Eclipse, or directly from the terminal.

03. SCOPE :

Understanding Multithreading: This program helps in understanding the basics of multithreading in Java.

Thread Execution: The program demonstrates how threads are executed concurrently in a Java environment.

Simulating Parallel Tasks: The experiment simulates how multiple threads can perform parallel tasks in a single program, improving performance for independent tasks.

04 THEORY:

Multithreading:

Multithreading is a powerful programming concept that allows multiple threads to execute independently within a single process, improving performance and responsiveness. In a multithreaded environment, threads are the smallest units of a process, each capable of executing a part of the program simultaneously.

This capability is particularly useful in applications that require tasks to run concurrently, such as web servers, games, or any system where responsiveness and parallel processing are important.

Multithreading enhances the performance of programs by making better use of system resources, especially on multi-core systems, where different threads can run on different processors simultaneously. It also helps in improving the responsiveness

of applications by allowing long-running tasks to run in the background while the main thread continues its work.

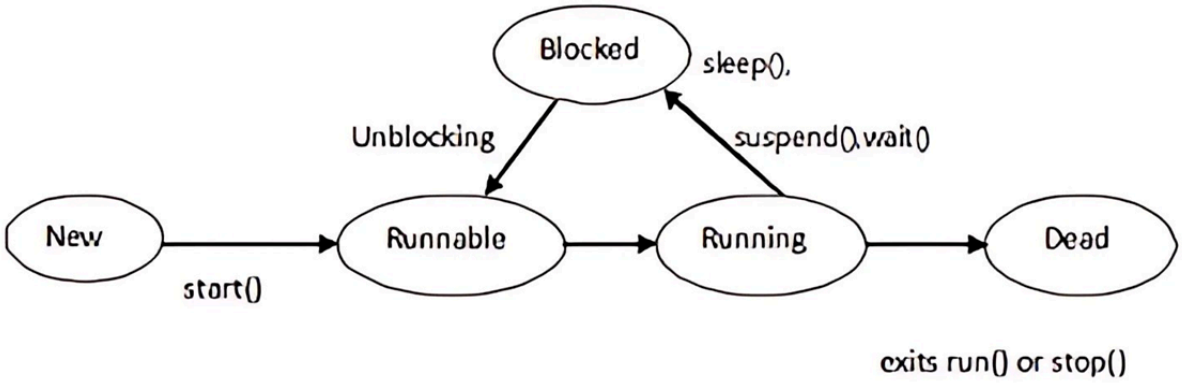
Thread Class:

The `Thread` class is a fundamental part of Java's multithreading model. It represents a single thread of execution and can be used to create new threads. The thread performs tasks defined in the `run()` method, which is executed when the thread starts.

Creating Threads:

A new thread can be created in Java in two primary ways:

- **Extending the Thread Class**: By extending the `Thread` class, you can create a custom thread and override its `run()` method to define the task that the thread should perform.
- **Implementing the Runnable Interface**: Alternatively, you can implement the `Runnable` interface and pass an instance of it to a `Thread` object. This is a more flexible approach because it allows your class to extend other classes while still being able to define its own thread logic.



run() Method:

The `run()` method is where the thread's work is defined. When `start()` is called, the `run()` method is invoked in a new thread of execution. If you don't override this method, the thread will simply do nothing.

start() Method:

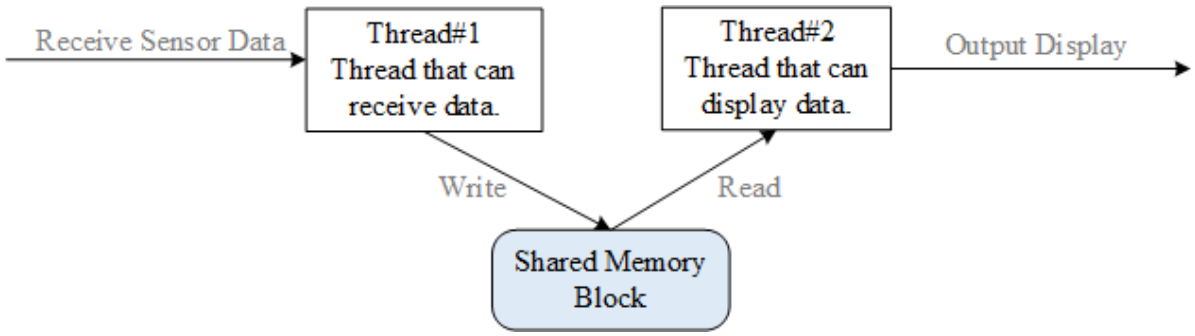
The `start()` method is used to begin the execution of the thread. It invokes the `run()` method in a separate thread. Unlike calling `run()` directly, `start()` allows the thread to be scheduled for execution by the Java runtime system.

Concurrency and Scheduling:

Threads can run **concurrently**, which means that the program allows multiple threads to progress at the same time. On a single-core system, this is achieved via **time-slicing**, where the CPU switches between different threads quickly. On a multi-core system, threads may be executed truly in parallel, with each core running a separate thread.

Thread Synchronization:

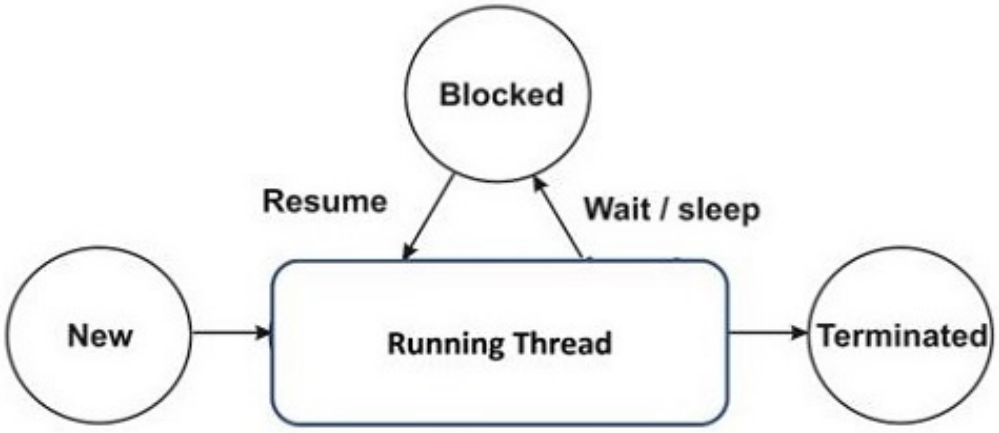
When multiple threads access shared resources or data, conflicts can occur (e.g., one thread updates a value while another is reading it). Java provides synchronization mechanisms (e.g., synchronized blocks or methods) to ensure that only one thread can access the shared resource at a time, thus avoiding race conditions.



Thread Lifecycle:

Threads in Java go through several states during their lifecycle:

- **New**: The thread is created but not yet started.
- **Runnable**: The thread is ready to run, and it is eligible for execution.
- **Blocked**: The thread is waiting for a resource.
- **Waiting**: The thread is waiting indefinitely for another thread to perform a particular action.
- **Terminated**: The thread has finished executing.



05 PROGRAM:

```
class MultithreadingDemo extends Thread {
    public void run() {
        try {
            // Print the ID of the thread executing the run() method
            System.out.println("Thread " + Thread.currentThread().getId() + " is running.");
        } catch (Exception e) {
            // Handle any exceptions that may occur
            System.out.println("Exception is caught.");
        }
    }
}
```

```
public class Multithread {
    public static void main(String[] args) {
        int n = 5; // Number of threads to be created
        for (int i = 0; i < n; i++) {
            MultithreadingDemo thread = new MultithreadingDemo();
            thread.start(); // Start the thread and invoke its run method
        }
    }
}
```

06 OUTPUT

```
localhost:~# javac Multithread.java
localhost:~# java Multithread
Thread 15 is running.
Thread 16 is running.
Thread 17 is running.
localhost:~#
```

08 CONCLUSION:

Multithreading in Java allows multiple tasks to be executed concurrently, improving program efficiency and responsiveness. The `Thread` class and its methods like `start()` and `run()` make it easy to implement multithreading. This experiment demonstrates how to create and manage multiple threads, providing a foundation for more complex multithreading techniques in the future.

09 VIVA QUESTIONS:

PREPARED BY **H. P. Amle**	**APPROVED BY** **HOD**