**ICS 2304**

**Programming Paradigms**

**Context**: we are exploring the fundamental *concepts* and *practical* **implementation** of threads.

Submitted: November 5(to be edited), 2024

Threads Implementation

Group A7:

|  |  |
| --- | --- |
| Registration Number | Student Name |
| SCT211-0848/2018 | Jany Muong |
| SCT211-0079/2022 | Joram Kireki |
| SCT211-0504/2021 | Gatmach Yuol Nyuon |
| SCT211-0003/2022 | Josphat Waweru Thumi |
| SCT211-0535/2022 | Akech Atem |
| SCT211-0070/2022 | Vincent Ohanga |

Abstract:

This assignment explores the fundamental concepts and practical implementation of **threads**. It covers aspects of thread management, including lifecycle control, synchronization, communication, and **thread-safe** programming techniques. We will touch on the Java Virtual Machine (JVM) and other uses of threads in real-world applications, such as web servers and distributed systems. We will majorly focus on how **threads** are **managed**, **synchronized**, and made to **communicate** within a **Java** program and handling exceptions.

Content:

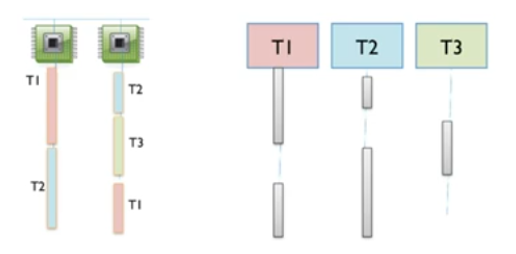
Thread implementation and covers the key aspects of it in Java programming language. We are covering these core areas:

1. **Thread Lifecycle Management**: explore the creation, execution, suspension, resumption, and termination of threads.
2. **Thread Synchronization**: implement a **shared resource** and demonstrate synchronization.
3. **Thread Communication**: develop a MessageQueue class that enables threads to send and receive messages.
4. **Thread-Safe Programming**: refactoring the **shared resource** to use ReentrantLock and Condition, and create a thread-safe caches.
5. **Thread Pools**
6. **Real-World Applications**: discuss the use of threads in the Java Virtual Machine (JVM) and others.

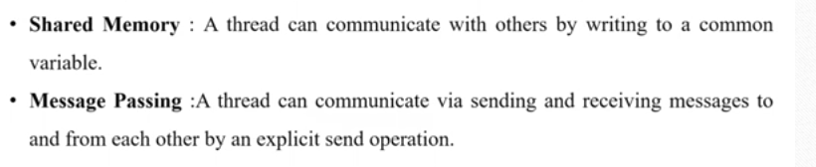
Threads - Definition:

A thread is a smaller unit of execution within a **process** (a program in execution). Threads are effectively **lightweight processes** that have a **single flow of control in a running program**. Threads share **all the memory in the process** – thus communications is effectively easier.

Threads are used to achieve **concurrency** in programming i.e. multiple computations happening at the same time. In concurrent programs, more than one instruction may be eligible for execution at any given moment. In Java, threading enables the CPU to handle multiple tasks efficiently, allowing for faster processing and more responsive applications. Concurrency is simulated via time slicing by processors – i.e. the processor switches between threads:



Concurrent programming is achieved via:

 Part 1: Thread Lifecycle Management

In this part, the focus is not thread creation so we will only explore one aspect of it - **thread lifecycle management**. We implement a custom GroupA7Thread class that extends the Thread class and demonstrates the various states a thread can transition through during its lifetime: e.g. **starting**, **suspending**, **resuming** and **stopped** (finished)

Implementing the Class – Demo for Lifecycle of a Thread:

In this example, we start a GroupA7Thread instance, and then wait for it to finish using the join() method. We want to see how a thread goes through the lifecycle.

Code Snippet:

public class GroupA7Thread extends Thread {

    private final String name;

    private volatile boolean suspended = false;

    public GroupA7Thread(String name) {

        this.name = name;

    }

    public void suspendThread() {

        suspended = true;

    }

    public synchronized void resumeThread() {

        suspended = false;

        notify(); // wake up the thread if it's waiting

    }

    @Override

    public void run() {

        System.out.println("Thread " + name + " started.");

        try {

            // simulate some work

            Thread.sleep(2000);

            // suspend the thread using a flag

            System.out.println("Thread " + name + " suspending itself.");

            synchronized (this) {

                while (suspended) {

                    wait();

                }

            }

            // Resume the thread after 1 second

            System.out.println("Thread " + name + " resuming.");

            Thread.sleep(2000);

        } catch (InterruptedException e) {

            System.out.println("Thread " + name + " interrupted.");

            Thread.currentThread().interrupt(); // set the interrupt flag

            return;

        }

        System.out.println("Thread " + name + " finished.");

    }

    public static void main(String[] args) throws InterruptedException {

        GroupA7Thread thread = new GroupA7Thread("A7");

        thread.start(); // start the thread

        Thread.sleep(1000); // let the thread start and work a bit

        thread.suspendThread(); // suspend the thread

        Thread.sleep(3000); // wait while the thread is suspended

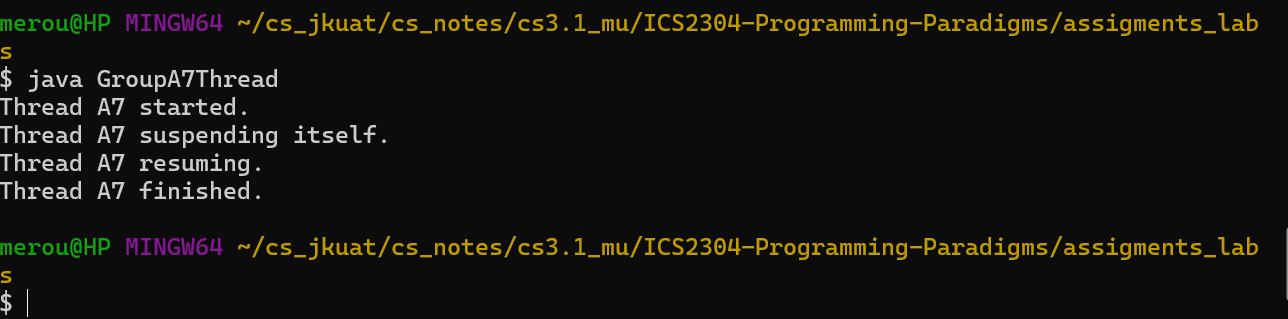
        thread.resumeThread(); // resume the thread

    }

}

Explanation: in the run() method, we simulate some work by having the thread sleep for specified seconds. Then, we suspend the thread, wait for it, and resume the thread before simulating more work.

Output:



Part 2: Thread Synchronization

Synchronization is needed when working with **shared resources** to prevent **race conditions**. It can be used to control the order in which operations occur in different threads.

1. Implementing A Shared Counter Class:

This is a class that will be accessed by multiple threads and it will showcase the use of **synchronized** methods.

Code Snippet:

public class Counter {

    private int count = 0;

    public synchronized void increment() {

        count++;

    }

    public synchronized int getValue() {

        return count;

    }

    public static void main(String[] args) throws InterruptedException {

        Counter sharedCounter = new Counter();

        Thread thread1 = new Thread(() -> {

            for (int i = 0; i < 1000; i++) {

                sharedCounter.increment();

            }

        });

        Thread thread2 = new Thread(() -> {

            for (int i = 0; i < 1000; i++) {

                sharedCounter.increment();

            }

        });

        thread1.start();

        thread2.start();

        thread1.join();

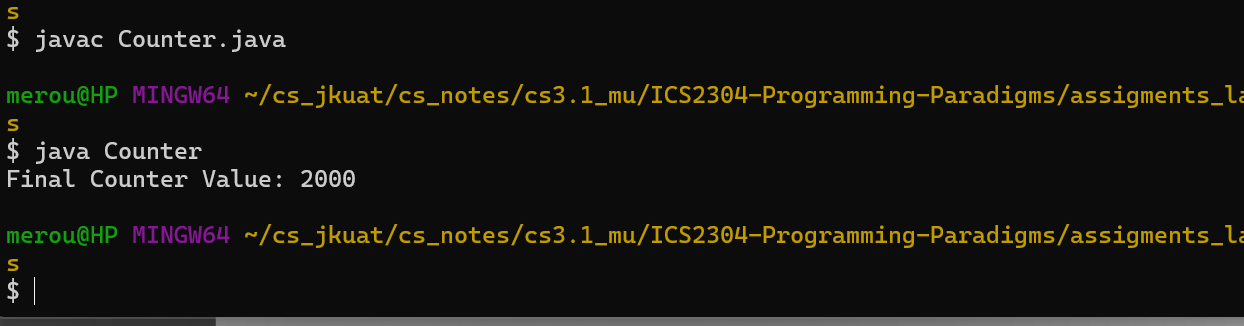
        thread2.join();

        System.out.println("Final Counter Value: " + sharedCounter.getValue());

    }

}

Output:



**Explanation:**

This code defines a Counter class with synchronized methods to safely increment and retrieve the count value across multiple threads. Two threads both increment the counter - 1,000 times each. The synchronized keyword ensures thread-safe access, producing the expected final count of 2,000.

1. Advanced Synchronization with ReentrantLock

For more flexible synchronization, we can use some classes from the java.util.concurrent.locks package. We will use this along with other methods as a **scheduler-based synchronization** technique.

import java.util.concurrent.locks.Condition;

import java.util.concurrent.locks.ReentrantLock;

public class ThreadSafeCounter {

    private final ReentrantLock lock = new ReentrantLock();

    private final Condition counterCondition = lock.newCondition();

    private int count = 0;

    public void increment() {

        lock.lock();

        try {

            count++;

            counterCondition.signalAll();

        } finally {

            lock.unlock();

        }

    }

    public int getCount() {

        lock.lock();

        try {

            return count;

        } finally {

            lock.unlock();

        }

    }

    public void waitUntilCountReaches(int target) {

        lock.lock();

        try {

            while (count < target) {

                counterCondition.await();

            }

        } catch (InterruptedException e) {

            Thread.currentThread().interrupt();

        } finally {

            lock.unlock();

        }

    }

    public static void main(String[] args) throws InterruptedException {

        ThreadSafeCounter counter = new ThreadSafeCounter();

        Thread incrementingThread = new Thread(() -> {

            for (int i = 0; i < 1000; i++) {

                counter.increment();

            }

        });

        Thread waitingThread = new Thread(() -> {

            counter.waitUntilCountReaches(1000);

            System.out.println("Count reached 1000!");

        });

        incrementingThread.start();

        waitingThread.start();

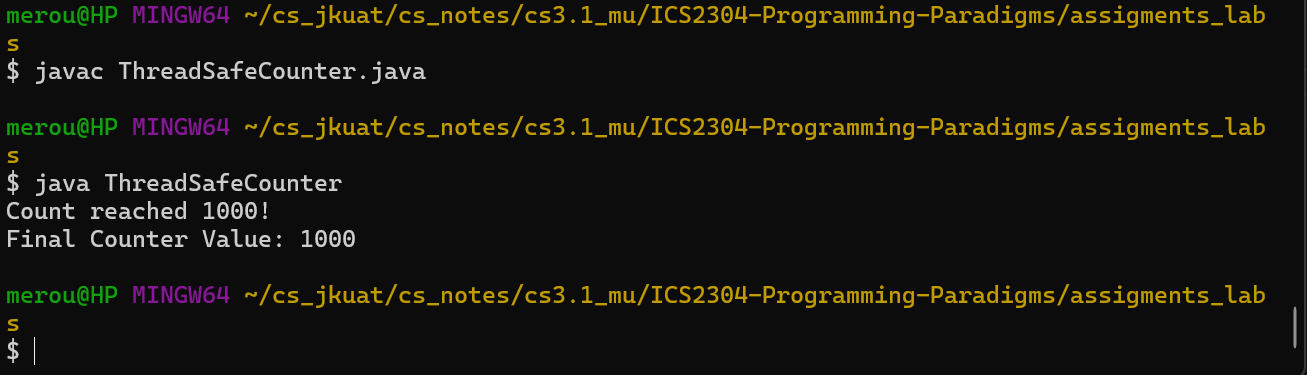
        incrementingThread.join();

        waitingThread.join();

        System.out.println("Final Counter Value: " + counter.getCount());

    }

}



In this version, we use a ReentrantLock and a **Condition** to control access to the shared counter. This uses ReentrantLock for flexible synchronization, allowing precise control over locking and unlocking. The increment() method increments count and signals any waiting threads. The waitUntilCountReaches() method makes a thread wait until count reaches a specified target, demonstrating the use of conditions for thread coordination.

Part 3: Inter-Thread Communication

The need for this communication is the exchange of data and we will use the **MessageQueue** class that enables threads to send and receive messages using methods within this class for communication.

This is the code snippet to demonstrate this:

import java.util.concurrent.BlockingQueue;

import java.util.concurrent.LinkedBlockingQueue;

public class MessageQueue {

    private final BlockingQueue<String> queue = new LinkedBlockingQueue<>();

    public void sendMessage(String message) {

        // send message to other threads

        try {

            queue.put(message);

            System.out.println("Sent message: " + message);

        } catch (InterruptedException e) {

            Thread.currentThread().interrupt();

        }

    }

    public String receiveMessage() {

        // received message to be printed to the console

        try {

            String message = queue.take();

            System.out.println("Received message: " + message);

            return message;

        } catch (InterruptedException e) {

            Thread.currentThread().interrupt();

            return null;

        }

    }

    public static void main(String[] args) {

        MessageQueue messageQueue = new MessageQueue();

        Thread sender = new Thread(() -> {

            messageQueue.sendMessage("Hello, world");

            messageQueue.sendMessage(" from Programming Paradigms");

        });

        Thread receiver = new Thread(() -> {

            String message1 = messageQueue.receiveMessage();

            String message2 = messageQueue.receiveMessage();

            System.out.println("Received messages: " + message1 + " - " + message2);

        });

        sender.start();

        receiver.start();

        try {

            sender.join();

            receiver.join();

        } catch (InterruptedException e) {

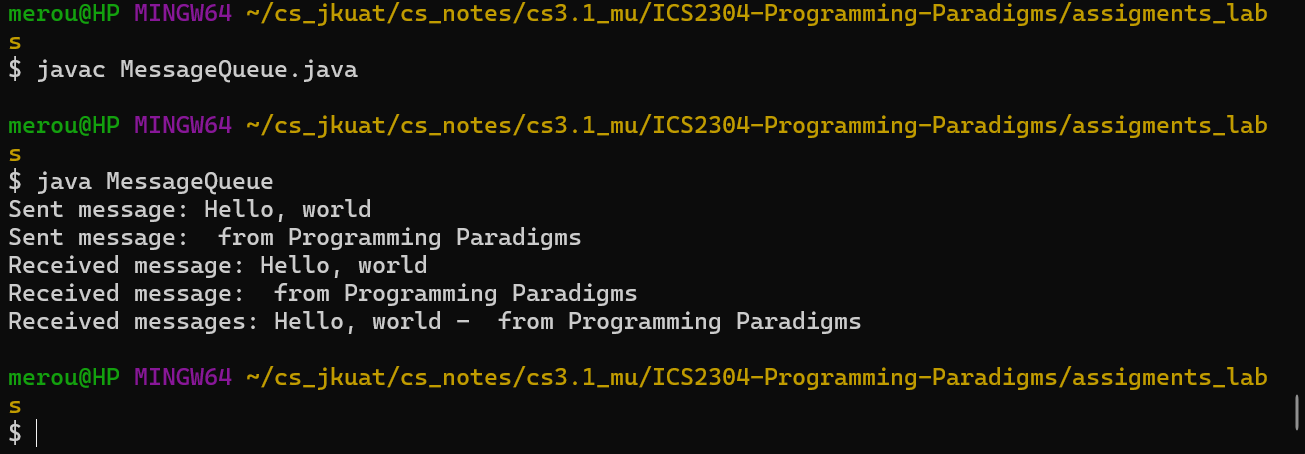
            e.printStackTrace();

        }

    }

}

Ouput:

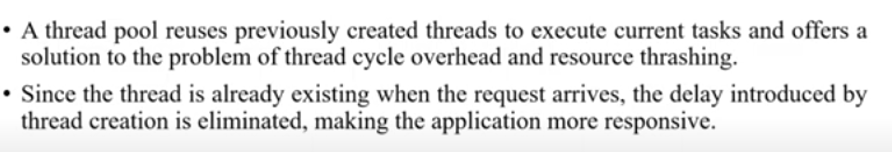


Explanation:

Inter-thread communication is done by using a MessageQueue class based on a BlockingQueue, which safely handles message passing between threads. The sender thread places messages into the queue, while the receiver thread retrieves them, showcasing coordinated data exchange.

Part 4: Thread Pooling

A thread pool represents a group of threads that are waiting for a job to execute and reuse many times. A thread in this case is pooled out from the pool, and assigned a job by the service provider.



Java's **ExecutorService** provides an easy way to manage a pool of threads as demonstrated below.

In this example, we create a thread pool with 3 threads using Executors.newFixedThreadPool(3). We then submit 5 tasks to the pool, and the threads in the pool execute these tasks.

// Thread Pooling

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

public class ThreadPooling {

    public static void main(String[] args) {

        ExecutorService executor = Executors.newFixedThreadPool(3);

        // tasks with a thread pool

        for (int i = 0; i < 5; i++) {

            executor.submit(() -> {

                System.out.println("Task executed by: " + Thread.currentThread().getName());

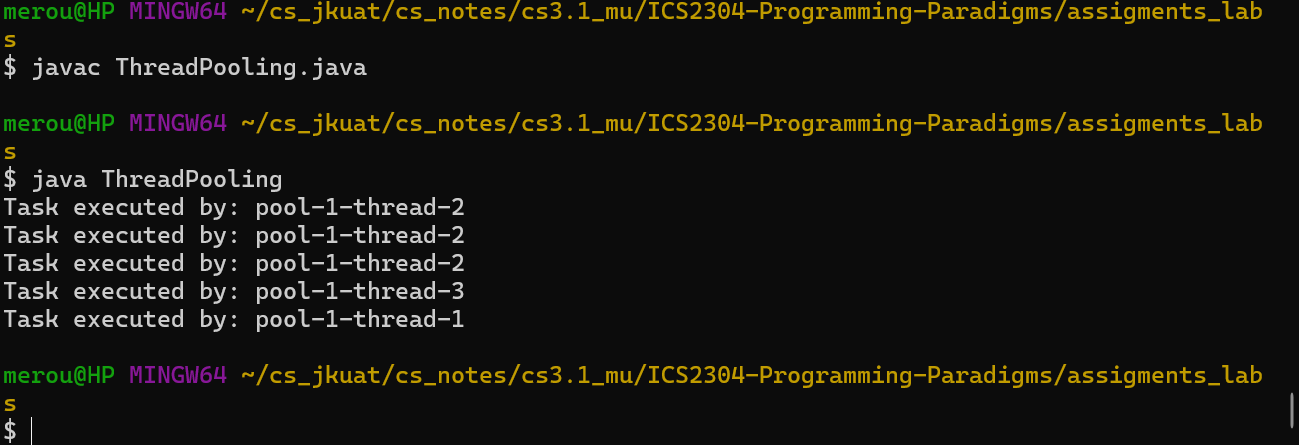
            });

        }

        executor.shutdown();

    }

}



Output Explanation:

Notice how the tasks are executed by the same set of 3 threads, demonstrating the reuse of thread resources.

The shutdown() method is called to gracefully terminate the thread pool after all tasks have been completed.

Part 5: Piece it Together – An Example of Producer-Consumer Problem

Bringing together the concepts covered earlier, implementing a complete producer-consumer scenario that showcases thread synchronization, communication, and the use of a thread pool.

In this demonstration, we have a Producer thread that generates items and adds them to a BlockingQueue, and a Consumer thread that retrieves items from the queue and processes them.

The ExecutorService is used to manage the producer and consumer threads, creating a thread pool with 2 threads. This ensures efficient resource usage and allows the threads to be reused for multiple production and consumption cycles.

import java.util.concurrent.BlockingQueue;

import java.util.concurrent.LinkedBlockingQueue;

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

class Item {

    private final int value;

    public Item(int value) {

        this.value = value;

    }

    public int getValue() {

        return value;

    }

}

class Producer implements Runnable {

    private final BlockingQueue<Item> queue;

    public Producer(BlockingQueue<Item> queue) {

        this.queue = queue;

    }

    @Override

    public void run() {

        for (int i = 0; i < 10; i++) {

            try {

                Item item = new Item(i);

                queue.put(item);

                System.out.println("Produced item: " + item.getValue());

            } catch (InterruptedException e) {

                Thread.currentThread().interrupt();

            }

        }

    }

}

class Consumer implements Runnable {

    private final BlockingQueue<Item> queue;

    public Consumer(BlockingQueue<Item> queue) {

        this.queue = queue;

    }

    @Override

    public void run() {

        while (true) {

            try {

                Item item = queue.take();

                System.out.println("Consumed item: " + item.getValue());

            } catch (InterruptedException e) {

                Thread.currentThread().interrupt();

                return;

            }

        }

    }

}

public class ProducerConsumerItem {

    public static void main(String[] args) {

        BlockingQueue<Item> queue = new LinkedBlockingQueue<>(5);

        ExecutorService executor = Executors.newFixedThreadPool(2);

        executor.submit(new Producer(queue));

        executor.submit(new Consumer(queue));

        executor.shutdown();

    }

}

Output:



Part 6: JVM and other Applications

Threads are used in modern applications for parallelism, responsive UIs, and efficient task management. We have listed out below some of a few key areas where threads are applied:

1. **Java Virtual Machine (JVM):** the JVM itself uses threads extensively. The Garbage Collector (GC), for instance, operates on a separate thread to free memory by removing unreachable objects. Java's Just-In-Time (JIT) compiler also runs on a separate thread, optimizing bytecode execution in the background.
2. **Web Servers and Application Servers:** Java-based servers like Apache Tomcat use thread pools to handle concurrent client requests. Each incoming request can be processed on a separate thread, allowing efficient handling of multiple users.
3. **Android Development:** threads are crucial for Android apps to keep the user interface responsive. The main UI thread handles UI updates, while background threads are used for network requests, file I/O, and other long-running tasks to avoid freezing the UI.
4. **Data Processing and Machine Learning:** multithreading is widely used in data processing and machine learning applications where large data sets or computations are divided among multiple threads to enhance speed and efficiency.