Concurrency - Processes & Threads

1. Understanding How Computer Applications Run

Computer applications operate by interacting with both hardware and software components. The operating system (OS) manages their execution using fundamental concepts such as processes, threads, CPU scheduling, and multitasking.

Programs and Processes

- **Program:** A set of instructions written to perform specific tasks.
- **Process:** An instance of a program that is actively executing. Multiple processes can run simultaneously on an OS.

Memory Allocation

- When a program is started, the OS assigns memory for its execution.
- This memory contains the program's code, data, and other runtime information.

Processor Execution

- The **CPU (Central Processing Unit)** fetches and executes instructions from memory.
- The OS uses **CPU Scheduling** techniques to allocate CPU time to different processes.

Context Switching

- The CPU rapidly switches between different processes.
- The OS saves the current state of a process and loads the state of another.

Multitasking and Parallel Execution

- **Multitasking:** The OS allows multiple processes to run seemingly at the same time.
- **Parallel Execution:** On systems with multiple CPU cores, different processes can run simultaneously.

Threads

- A **thread** is a smaller execution unit within a process.
- Threads within a process share the same resources but execute independently.
- **Multithreading** allows a process to run multiple threads concurrently.

Synchronization

- When multiple threads share resources, **synchronization mechanisms** prevent conflicts and ensure data consistency.

CPU Scheduling Algorithms

1. **First-Come-First-Serve (FCFS)** – Processes execute in the order they arrive.
2. **Shortest Job Next (SJN)** – The process with the shortest execution time runs first.
3. **Round Robin (RR)** – Each process gets a fixed time slice before moving to the back of the queue.
4. **Priority Scheduling** – Processes are assigned priorities, and higher-priority processes execute first.

Interrupts

- The CPU can be interrupted to handle external events (e.g., user input or network data).
- Interrupts help maintain system responsiveness.

Process Termination

- A process terminates when it completes its task or is forcibly closed.
- The OS reclaims memory and other resources assigned to it.

2. Concurrency: Real-World Applications of Threads

Concurrency enables software to perform multiple tasks efficiently. Real-world applications include:

Google Docs

- **Collaborative Editing:** Each user's actions are processed by a separate thread.
- **Conflict Resolution:** Threads synchronize changes.
- **UI Updates:** A separate thread continuously updates the document interface.

Music Players (Spotify, iTunes)

- **Playback Thread:** Manages audio playback.
- **UI Thread:** Handles user interactions (e.g., browsing songs).
- **Parallel Execution:** Threads allow playback and interaction simultaneously.

Adobe Lightroom

- **Image Processing Threads:** Handle different parts of an image concurrently.
- **Background Tasks:** Importing photos runs in separate threads.
- **UI Responsiveness:** Ensures smooth editing experience.

3. Processes and Threads

Process Lifecycle

1. **Creation:** Program is loaded into memory and a process starts.
2. **Execution:** Process runs its instructions.
3. **Termination:** Process completes execution or is terminated.

Benefits of Multithreading

- **Performance:** Enables faster execution by utilizing CPU cores efficiently.
- **Responsiveness:** Keeps applications responsive during long-running tasks.
- **Efficiency:** Reduces overhead compared to multiple processes.
- **Resource Sharing:** Threads share memory, improving efficiency.

Challenges

- **Data Synchronization:** Shared data must be synchronized properly.
- **Deadlocks:** Threads waiting for each other can lead to unresolvable states.

4. Concurrent vs Parallel Execution

Feature	Concurrent Execution	Parallel Execution
Execution	Tasks overlap in time but don't necessarily run at the same time	Tasks run at the same time on different processors
CPU Usage	Utilizes a single processor with time-sharing	Requires multiple processors or cores
Example	Running multiple applications on a single-core processor	Image processing on a multi-core processor

5. Multithreading in Java

Creating Threads in Java

Way 1: Subclassing the Thread Class

```
public class NewThread extends Thread {
   public void run() {
      System.out.println("Thread is running");
   }
}
public class Main {
```

```
    public static void main(String[] args) {
        NewThread t1 = new NewThread();
        t1.start();
    }
}
```

Way 2: Implementing Runnable (Preferred Method)

```
class SimpleRunnable implements Runnable {
    public void run() {
        System.out.println("Running in a separate thread");
    }
}
public class Main {
    public static void main(String[] args) {
        Thread t = new Thread(new SimpleRunnable());
        t.start();
    }
}
```

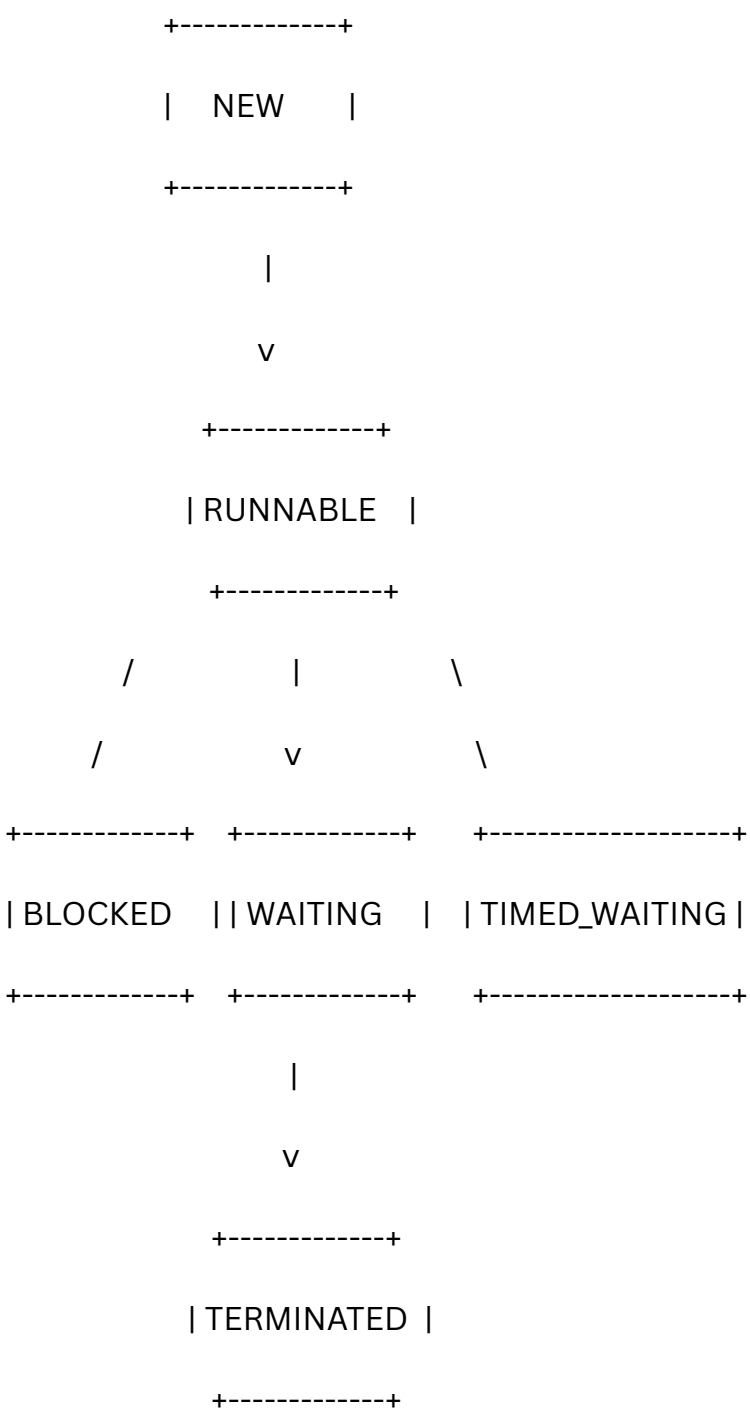
- Runnable is preferred because it supports **composition over inheritance**.

Java 8: Using Lambda Expressions

```
Thread t = new Thread(() -> System.out.println("Lambda Thread"));
t.start();
```

Thread Lifecycle & States

1. **NEW:** Thread created but not started.
2. **RUNNABLE:** Thread is ready to execute but waiting for CPU.
3. **BLOCKED:** Thread waiting for access to a locked resource.
4. **WAITING:** Thread waiting indefinitely for another thread's signal.
5. **TIMED_WAITING:** Thread waiting for a specified time period.
6. **TERMINATED:** Thread has completed execution.



Common Thread Methods

- start(): Begins execution of the thread.
- run(): Contains the logic for execution.
- sleep(ms): Pauses execution for a specified time.
- join(): Waits for the thread to finish before continuing execution.
- interrupt(): Interrupts the execution of a thread.

6. Problem Statements

Problem 1: Number Printer

Create 100 threads to print numbers from 1 to 100 in any order.

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

Problem 2: Factorial Computation

Compute factorial for a list of numbers using separate threads.

```
public class FactorialThread extends Thread {
   private long number;
   public FactorialThread(long number) { this.number = number; }
   public void run() { System.out.println("Factorial of " + number + " is " + factorial(number)); }
   private BigInteger factorial(long n) {
      BigInteger result = BigInteger.ONE;
      for(long i = 2; i <= n; i++) {
```

```
            result = result.multiply(BigInteger.valueOf(i));
        }
        return result;
    }
}
```

Executors & Callables

1. Executor Framework

Overview

- The Executor Framework simplifies concurrent task execution.
- It abstracts thread management, making code more readable and maintainable.
- It allows executing tasks asynchronously using thread pools.

Using Executor Framework

- Part of java.util.concurrent package.
- Introduced in Java 5.
- Handles thread creation and management efficiently.

Example:

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;


public class ExecutorDemo {
    public static void main(String[] args) {
        ExecutorService executor = Executors.newFixedThreadPool(10);
        for(int i=0; i<100;i++){
            executor.execute(new NumberPrinter(i));
        }
```

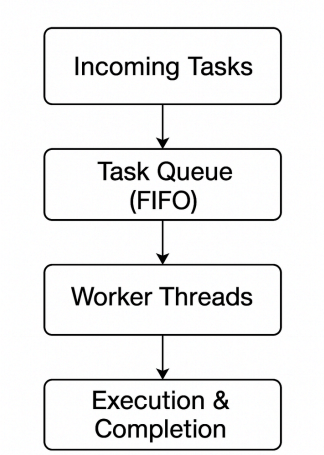
```
        executor.shutdown();

    }

}
```

Thread Pool

- A collection of threads that execute tasks efficiently.
- Components:
  1. Worker Threads: Execute tasks.
  2. Task Queue: Holds submitted tasks (FIFO order).
  3. Thread Pool Manager: Allocates tasks and manages synchronization.



Types of Thread Pools

1. **FixedThreadPool**
   - Reuses a fixed number of threads.
   - Unused threads are reused.
   - Example: ExecutorService executor = Executors.newFixedThreadPool(nThreads);
2. **CachedThreadPool**
   - Dynamically adjusts the number of threads.
   - Suitable for short-lived tasks.
   - Example: ExecutorService executor = Executors.newCachedThreadPool();
3. **SingleThreadExecutor**
   - Executes tasks sequentially.
   - Example: ExecutorService executor = Executors.newSingleThreadExecutor();
4. **ScheduledThreadPool**
   - Supports scheduled execution.

    - Example: ScheduledExecutorService executor = Executors.newScheduledThreadPool(nThreads);
5. **WorkStealingPool** (Java 8+)
    - Uses multiple worker threads.
    - Example: ExecutorService executor = Executors.newWorkStealingPool();

Benefits of Executor Framework

- Simplifies task execution.
- Efficient resource utilization.
- Provides better control and flexibility.
- Enhances scalability.
- Supports task scheduling.

2. Callable & Future

Overview

- Unlike Runnable, Callable returns a result.
- Uses Future to retrieve results asynchronously.

Example:

```
import java.util.concurrent.*;


ExecutorService executor = Executors.newCachedThreadPool();

Future<Integer> future = executor.submit(() -> 2 + 3);

Integer result = future.get();
```

- Future.get() blocks until the result is available.
- Future.cancel() cancels execution.

3. Coding Problems

3.1 Multi-threaded Merge Sort

Solution:

```
import java.util.concurrent.*;

import java.util.*;


public class Sorter implements Callable<List<Integer>> {
```

```
    private List<Integer> arr;
    private ExecutorService executor;

    Sorter(List<Integer> arr, ExecutorService executor){
        this.arr = arr;
        this.executor = executor;
    }
    @Override
    public List<Integer> call() throws Exception {
        if (arr.size() <= 1) return arr;
        int mid = arr.size() / 2;
        List<Integer> leftArr = arr.subList(0, mid);
        List<Integer> rightArr = arr.subList(mid, arr.size());
        Future<List<Integer>> leftFuture = executor.submit(new Sorter(leftArr, executor));
        Future<List<Integer>> rightFuture = executor.submit(new Sorter(rightArr, executor));
        return merge(leftFuture.get(), rightFuture.get());
    }
}
```

3.2 Download Manager

- Uses Executor Framework to download files concurrently.
- Tracks progress for each file.

Solution:

```
import java.util.concurrent.*;
import java.util.*;

class DownloadTask implements Runnable {
```

```
    private String fileUrl;
    public DownloadTask(String fileUrl) { this.fileUrl = fileUrl; }
    @Override
    public void run() {
        System.out.println("Downloading: " + fileUrl);
        for (int i = 0; i <= 100; i += 10) {
            System.out.println("Progress: " + fileUrl + " - " + i + "%");
            try { Thread.sleep(500); } catch (InterruptedException e) { }
        }
    }
}

public class DownloadManager {
    private ExecutorService executorService;
    public DownloadManager(int threads) { executorService =
Executors.newFixedThreadPool(threads); }
    public void downloadFiles(List<String> urls) {
        for (String url : urls) { executorService.submit(new DownloadTask(url)); }
    }
    public void shutdown() { executorService.shutdown(); }
}
```

3.3 Image Processing

- Uses four threads to repaint a 2D array.

Solution:

```
import java.util.concurrent.*;
```

```
class ArrayRepainterTask implements Runnable {
   private int[][] array; int startRow, endRow, startCol, endCol;
   public ArrayRepainterTask(int[][] array, int startRow, int endRow, int startCol, int endCol) {
      this.array = array; this.startRow = startRow; this.endRow = endRow;
      this.startCol = startCol; this.endCol = endCol;
   }
   @Override
   public void run() {
      for (int i = startRow; i <= endRow; i++)
         for (int j = startCol; j <= endCol; j++)
            array[i][j] *= 2;
   }
}
```

3.4 Scheduled Executor

- Uses ScheduledExecutorService to execute tasks at fixed intervals.

Solution:

```
import java.util.concurrent.*;


public class ScheduledExecutorExample {
   public static void main(String[] args) {
      ScheduledExecutorService scheduler = Executors.newSingleThreadScheduledExecutor();
      scheduler.scheduleAtFixedRate(() -> System.out.println("Hello"), 0, 5,
TimeUnit.SECONDS);
   }
}
```

4. Synchronization

- When multiple threads access the same resource, synchronization is needed.

Example: Adder & Subtractor

```
class Count { int value = 0; }


class Adder implements Runnable {
   private Count count;
   public Adder(Count count) { this.count = count; }
   @Override
   public void run() {
      for (int i = 1; i <= 100; i++) count.value += i;
   }
}


class Subtractor implements Runnable {
   private Count count;
   public Subtractor(Count count) { this.count = count; }
   @Override
   public void run() {
      for (int i = 1; i <= 100; i++) count.value -= i;
   }
}
```

Observations:

- Without synchronization, results are inconsistent.
- Solution: Use synchronized keyword or ReentrantLock.

Synchronization

Introduction

- **Topics Covered**:
  - Synchronization problems
  - Mutex locks
  - Synchronized keyword
  - Atomic data types
  - Volatile keyword
  - Concurrent HashMap

Synchronization Problem

Adder-Subtractor Problem:

- Demonstrates the need for synchronization.
- Two threads modifying a shared count variable concurrently.
- Without synchronization, the final value is unpredictable due to race conditions.

Conditions for Synchronization Problems:

1. **Critical Section**: Code that must be executed by only one thread at a time.
2. **Race Conditions**: The output depends on the scheduling order of threads.
3. **Preemption**: Interrupting one thread's execution can lead to inconsistencies.

Properties of a Good Synchronization Solution:

1. **Mutual Exclusion**: Only one thread accesses the critical section at a time.
2. **Progress**: Threads waiting should get a chance to execute.
3. **Bounded Waiting**: Threads should not be indefinitely delayed.
4. **No Deadlocks**: No circular waits that cause indefinite blocking.
5. **Efficiency**: Minimal overhead on system resources.
6. **Adaptability**: Should work across different workloads.
7. **Fairness**: All threads should get equal opportunities.

Solutions to Synchronization Problems

1. Mutex Locks

- Ensures mutual exclusion.
- A thread must acquire a lock before accessing the critical section and release it afterward.

- Example with ReentrantLock:

```
Lock lock = new ReentrantLock();

lock.lock();

try {

   count += 1;

} finally {

   lock.unlock();

}
```

2. Synchronized Keyword

- Ensures only one thread can execute a synchronized method/block at a time.
- Example:

```
public synchronized void increment() {

   count++;

}
```

3. Atomic Data Types

- Provide atomic operations without explicit synchronization.
- Example with AtomicInteger:

```
AtomicInteger counter = new AtomicInteger(0);

counter.incrementAndGet();
```

4. Volatile Keyword

- Ensures visibility of changes across threads but does not guarantee atomicity.
- Example:

```
volatile boolean flag = false;
```

Concurrent Data Structures

1. Synchronized HashMap

- Thread-safe wrapper over HashMap using Collections.synchronizedMap().
- Example:

Map<String, Integer> syncMap = Collections.synchronizedMap(new HashMap<>());

2. Concurrent HashMap vs. Hashtable

Feature	Hashtable	ConcurrentHashMap
Synchronization	Entire table locked	Fine-grained locking at bucket level
Performance	Slower	Faster
Usage	Legacy applications	Modern concurrent applications

Coding Projects

1. Ticket Booking System

- Uses ReentrantLock to ensure thread safety when reserving seats.

```
Lock lock = new ReentrantLock();

lock.lock();

try {

   availableSeats--;

} finally {

   lock.unlock();

}
```

2. Thread-safe Bank Transactions

- Ensures deposits and withdrawals are atomic.

```
Lock lock = new ReentrantLock();
```

```
lock.lock();
try {
    balance += amount;
} finally {
    lock.unlock();
}
```

Synchronization Using Semaphores

Introduction

Semaphores are synchronization primitives used to control access to shared resources in concurrent programming. They prevent race conditions and ensure orderly execution of processes.

Key Concepts

- **Mutex Semaphore (mutex)**: Ensures mutual exclusion, allowing only one process to access the critical section.
- **Counting Semaphores (empty, full)**: Used to track the number of available and occupied slots in a resource (e.g., a buffer).

Producer-Consumer Problem (T-Shirt Store Example)

The **Producer-Consumer** problem is a classic synchronization problem. In a **T-Shirt store**, producers add T-shirts to the store, while consumers buy them.

Java Implementation Using Semaphores

java

CopyEdit

```
import java.util.concurrent.Semaphore;


public class TShirtStore {
```

```
    private static final int STORE_CAPACITY = 5;

    private static Semaphore mutex = new Semaphore(1); // Controls access to critical section

    private static Semaphore empty = new Semaphore(STORE_CAPACITY); // Empty slots in
store

    private static Semaphore full = new Semaphore(0); // Filled slots in store

    private static int tShirtCount = 0;


    static class Producer implements Runnable {

        @Override

        public void run() {

            try {

                while (true) {

                    empty.acquire(); // Wait for an empty slot

                    mutex.acquire(); // Enter critical section


                    // Produce a T-shirt

                    System.out.println("Producer produces a T-shirt. Total: " + ++tShirtCount);


                    mutex.release(); // Exit critical section

                    full.release(); // Signal that a T-shirt is ready

                    Thread.sleep(1000); // Simulate production time

                }

            } catch (InterruptedException e) { e.printStackTrace(); }

        }

    }
```

```
static class Consumer implements Runnable {
    @Override
    public void run() {
        try {
            while (true) {
                full.acquire(); // Wait for a T-shirt
                mutex.acquire(); // Enter critical section

                // Consume a T-shirt
                System.out.println("Consumer buys a T-shirt. Remaining: " + --tShirtCount);

                mutex.release(); // Exit critical section
                empty.release(); // Signal that space is available
                Thread.sleep(1500); // Simulate consumption time
            }
        } catch (InterruptedException e) { e.printStackTrace(); }
    }
}

public static void main(String[] args) {
    Thread producerThread = new Thread(new Producer());
    Thread consumerThread = new Thread(new Consumer());

    producerThread.start();
    consumerThread.start();
}
```

}

Explanation

1. **Producer**
   - Waits for an empty slot (empty.acquire()).
   - Locks the critical section (mutex.acquire()).
   - Produces a T-shirt and increments count.
   - Releases the lock (mutex.release()) and signals availability (full.release()).
2. **Consumer**
   - Waits for a filled slot (full.acquire()).
   - Locks the critical section (mutex.acquire()).
   - Buys a T-shirt and decrements count.
   - Releases the lock (mutex.release()) and signals a free slot (empty.release()).

Deadlocks

A **deadlock** occurs when multiple processes block each other by waiting for resources held by the other processes.

Deadlock Conditions

1. **Mutual Exclusion** – Resources are non-shareable.
2. **Hold and Wait** – A process holds resources while waiting for others.
3. **No Preemption** – Resources cannot be forcibly taken away.
4. **Circular Wait** – A circular chain exists where each process waits for a resource held by another.

Example of a Deadlock Situation

java

CopyEdit

```
class DeadlockExample {

   static final Object resource1 = new Object();

   static final Object resource2 = new Object();



   public static void main(String[] args) {

      Thread t1 = new Thread(() -> {

         synchronized (resource1) {
```

```
            System.out.println("Thread 1 locked resource 1");

            try { Thread.sleep(100); } catch (InterruptedException e) {}

            synchronized (resource2) {
                System.out.println("Thread 1 locked resource 2");
            }
        }
    });

    Thread t2 = new Thread(() -> {
        synchronized (resource2) {
            System.out.println("Thread 2 locked resource 2");

            try { Thread.sleep(100); } catch (InterruptedException e) {}

            synchronized (resource1) {
                System.out.println("Thread 2 locked resource 1");
            }
        }
    });

    t1.start();
    t2.start();
  }
}
```

Explanation

- **Thread 1** locks resource1 and waits for resource2.
- **Thread 2** locks resource2 and waits for resource1.
- Both threads are waiting indefinitely, leading to a deadlock.

Deadlock Prevention

1. **Avoid Hold & Wait** – Acquire all required resources at once.
2. **Resource Preemption** – Allow resource release before acquiring another.
3. **Circular Wait Prevention** – Impose an ordering on resource acquisition.

Producer-Consumer Using wait() & notify()

Concept

- wait() releases the lock and waits until another thread calls notify().
- notify() wakes up a waiting thread.

Java Implementation

java

CopyEdit

```
import java.util.LinkedList;


class SharedBuffer {
    private final LinkedList<Integer> buffer = new LinkedList<>();
    private final int capacity;

    public SharedBuffer(int capacity) { this.capacity = capacity; }

    public synchronized void produce() throws InterruptedException {
        while (buffer.size() == capacity) {
            wait(); // Wait if buffer is full
        }
```

```
        int newItem = (int) (Math.random() * 100);
        buffer.add(newItem);
        System.out.println("Produced: " + newItem);

        notify(); // Notify consumer
    }

    public synchronized void consume() throws InterruptedException {
        while (buffer.isEmpty()) {
            wait(); // Wait if buffer is empty
        }

        int item = buffer.removeFirst();
        System.out.println("Consumed: " + item);

        notify(); // Notify producer
    }
}

class Producer implements Runnable {
    private final SharedBuffer sharedBuffer;
    public Producer(SharedBuffer sharedBuffer) { this.sharedBuffer = sharedBuffer; }

    @Override
    public void run() {
```

```
        try {
            while (true) {
                sharedBuffer.produce();
                Thread.sleep(1000);
            }
        } catch (InterruptedException e) { e.printStackTrace(); }
    }
}

class Consumer implements Runnable {
    private final SharedBuffer sharedBuffer;
    public Consumer(SharedBuffer sharedBuffer) { this.sharedBuffer = sharedBuffer; }

    @Override
    public void run() {
        try {
            while (true) {
                sharedBuffer.consume();
                Thread.sleep(1500);
            }
        } catch (InterruptedException e) { e.printStackTrace(); }
    }
}

public class ProducerConsumerExample {
    public static void main(String[] args) {
```

```
        SharedBuffer sharedBuffer = new SharedBuffer(5);


        Thread producerThread = new Thread(new Producer(sharedBuffer));

        Thread consumerThread = new Thread(new Consumer(sharedBuffer));


        producerThread.start();

        consumerThread.start();

    }

}
```

Key Differences from Semaphore Approach

- Uses **intrinsic locks (synchronized)** instead of semaphores.
- Producer waits when the buffer is full (wait()).
- Consumer waits when the buffer is empty (wait()).
- notify() signals waiting threads to continue execution.

Conclusion

- **Semaphores** are powerful tools for synchronization, allowing controlled access to resources.
- **Deadlocks** occur due to improper handling of resource allocation.
- **Using wait() and notify()**, we can achieve efficient producer-consumer communication.