Java Multithreading: Producer-Consumer, Synchronization, Barriers, and More

---

1. Producer-Consumer Problem (Classic Example)

The Producer-Consumer problem is a classic example of a multi-threading scenario where two threads (producer and consumer) share a common buffer. The producer adds items to the buffer, and the consumer removes them. Synchronization is required to avoid race conditions and ensure correct behavior.

Example: Using `wait()` and `notify()`

```
import java.util.*;
public class ProducerConsumer {
    int n = 2;
    LinkedList<Integer> buffer = new LinkedList<>();
    public void produce() throws InterruptedException {
        int value = 0;
        while(true){
            synchronized (this) {
                while(buffer.size() == n) {
                    System.out.println("Buffer is full, waiting for consumer to consume..."
                    wait();
                }
                buffer.add(value);
                System.out.println("Produced: " + value);
                value++;
                notifyAll();
            }
        }
    }
    public void consume() throws InterruptedException {
        while(true) {
            synchronized (this) {
                while(buffer.isEmpty()) {
                    System.out.println("Buffer is empty, waiting for producer to produce...
                    wait();
                }
                int value = buffer.removeFirst();
                System.out.println("Consumed: " + value);
                notifyAll();
            }
        }
    }
```

```java
    public static void main(String[] args) {
        ProducerConsumer pc = new ProducerConsumer();
        Thread producerThread = new Thread(() -> {
            try { pc.produce(); } catch (InterruptedException e) { e.printStackTrace(); }
        });
        Thread consumerThread = new Thread(() -> {
            try { pc.consume(); } catch (InterruptedException e) { e.printStackTrace(); }
        });
        producerThread.start();
        consumerThread.start();
    }
}
```

Common Errors: - Using `if` instead of `while` for buffer checks can cause spurious wakeups and bugs. - Forgetting to use `notifyAll()` (or `notify()`) after `wait()` can cause deadlocks. - Not synchronizing on the correct object.

---

2. Producer-Consumer with BlockingQueue

Java provides `BlockingQueue` to simplify producer-consumer logic and avoid manual synchronization.

```java
import java.util.concurrent.*;
public class BlockingQueueDemo {
    public static void main(String[] args) {
        BlockingQueue<Integer> queue = new ArrayBlockingQueue<>(2);
        Thread producer = new Thread(() -> {
            int value = 0;
            try {
                while (true) {
                    queue.put(value);
                    System.out.println("Produced: " + value);
                    value++;
                    Thread.sleep(500);
                }
            } catch (InterruptedException e) { e.printStackTrace(); }
        });
        Thread consumer = new Thread(() -> {
            try {
                while (true) {
                    int value = queue.take();
                    System.out.println("Consumed: " + value);
                    Thread.sleep(1000);
                }
            } catch (InterruptedException e) { e.printStackTrace(); }
```

```
        });
        producer.start();
        consumer.start();
    }
}
```

Why use BlockingQueue? - Handles all synchronization internally. - No need for explicit `wait()/notify()`.

---

3. CyclicBarrier: Thread Coordination

A `CyclicBarrier` allows a set of threads to all wait for each other to reach a common barrier point. Useful for parallel tasks that must synchronize at certain steps.

Example: Simple CyclicBarrier

```java
import java.util.concurrent.*;
public class CyclicBarrierDemo {
    public static void main(String[] args) {
        int numThreads = 3;
        CyclicBarrier barrier = new CyclicBarrier(numThreads, () -> {
            System.out.println("All threads reached the barrier. Continuing...");
        });
        for (int i = 1; i <= numThreads; i++) {
            int threadNum = i;
            new Thread(() -> {
                try {
                    System.out.println("Thread " + threadNum + " is doing work...");
                    Thread.sleep((long) (Math.random() * 3000));
                    System.out.println("Thread " + threadNum + " reached the barrier");
                    barrier.await();
                    System.out.println("Thread " + threadNum + " continues execution");
                } catch (Exception e) { e.printStackTrace(); }
            }).start();
        }
    }
}
```

Example: CyclicBarrier in a Race Simulation

```java
import java.util.concurrent.*;
class Car implements Runnable {
    private final int carId;
    private final CyclicBarrier barrier;
```

```
    private final int totalLaps;
    Car(int carId, CyclicBarrier barrier, int totalLaps) {
        this.carId = carId;
        this.barrier = barrier;
        this.totalLaps = totalLaps;
    }
    @Override
    public void run() {
        try {
            for (int lap = 1; lap <= totalLaps; lap++) {
                System.out.println("Car " + carId + " is racing in lap " + lap + "...");
                Thread.sleep((long) (Math.random() * 3000 + 1000));
                System.out.println("Car " + carId + " reached the checkpoint for lap " + lap
                barrier.await();
                if (lap < totalLaps) {
                    System.out.println("Car " + carId + " starts next lap!\n");
                } else {
                    System.out.println("Car " + carId + " finished the race!\n");
                }
            }
        } catch (Exception e) { e.printStackTrace(); }
    }
}
public class CyclicBarrierRace {
    public static void main(String[] args) {
        int numCars = 4;
        int totalLaps = 3;
        CyclicBarrier barrier = new CyclicBarrier(numCars, () -> {
            System.out.println("\n=== All cars reached checkpoint! Starting next lap ===\n")
        });
        for (int i = 1; i <= numCars; i++) {
            new Thread(new Car(i, barrier, totalLaps)).start();
        }
    }
}
```

---

4. Multithreading Concepts: Theory and Common Errors

Key Concepts

- **Thread**: Smallest unit of execution. Created by extending `Thread` or implementing `Runnable`/`Callable`.
- **Race Condition**: When two or more threads access shared data and try to change it at the same time.
- **Deadlock**: Two or more threads are blocked forever, each waiting for the

other.

- **Starvation**: A thread is unable to gain regular access to the resources it needs.
- **Livelock**: Threads are not blocked but keep changing state in response to each other and cannot proceed.
- **Synchronized**: Keyword to lock a method/block so only one thread can execute it at a time.
- **wait()/notify()/notifyAll()**: Used for inter-thread communication.
- **volatile**: Keyword to ensure visibility of changes to variables across threads.
- **Thread safety**: Code that functions correctly when accessed from multiple threads.
- **Thread pool**: A pool of worker threads to execute tasks, managed by `ExecutorService`.
- **Callable & Future**: For tasks that return results or throw exceptions.

Common Errors

- Not synchronizing shared data (leads to race conditions).
- Using `if` instead of `while` with `wait()` (spurious wakeups).
- Forgetting to release locks (deadlocks).
- Not handling `InterruptedException`.
- Using thread-unsafe collections (use `ConcurrentHashMap`, `CopyOnWriteArrayList`, etc. for safety).
-

  Not shutting down thread pools (`executor.shutdown()`).

5. More Examples: Thread Basics, Synchronization, and Errors

Example: Creating Threads

```
public class ThreadDemo {
    public static void main(String[] args) {
        Thread t1 = new Thread(() -> System.out.println("Hello from thread!"));
        t1.start();
    }
}
```

Example: Extending Thread

```
class MyThread extends Thread {
    public void run() {
```

```
        System.out.println("MyThread running");
    }
}
public class ThreadExtendsDemo {
    public static void main(String[] args) {
        new MyThread().start();
    }
}
```

Example: Synchronization

```
class Counter {
    private int count = 0;
    public synchronized void increment() { count++; }
    public int getCount() { return count; }
}
public class SyncDemo {
    public static void main(String[] args) throws InterruptedException {
        Counter counter = new Counter();
        Thread t1 = new Thread(() -> { for(int i=0;i<1000;i++) counter.increment(); });
        Thread t2 = new Thread(() -> { for(int i=0;i<1000;i++) counter.increment(); });
        t1.start(); t2.start();
        t1.join(); t2.join();
        System.out.println(counter.getCount()); // 2000
    }
}
```

Example: Deadlock

```
class DeadlockDemo {
    static final Object lock1 = new Object();
    static final Object lock2 = new Object();
    public static void main(String[] args) {
        Thread t1 = new Thread(() -> {
            synchronized(lock1) {
                try { Thread.sleep(100); } catch (Exception e) {}
                synchronized(lock2) { System.out.println("Thread 1 acquired both locks"); }
            }
        });
        Thread t2 = new Thread(() -> {
            synchronized(lock2) {
                try { Thread.sleep(100); } catch (Exception e) {}
                synchronized(lock1) { System.out.println("Thread 2 acquired both locks"); }
            }
        });
        t1.start(); t2.start();
```

```
    }
}
```

Common Error: - Acquiring locks in different orders can cause deadlock.
