**The Producer-Consumer Problem** is a classic example of a **multithreading** or **inter-thread communication** problem in Java. It involves two types of threads:

* **Producer**: Produces data and puts it into a shared resource (like a queue).
* **Consumer**: Consumes data from the shared resource.

**The Problem**

* If the **buffer is full**, the producer must wait.
* If the **buffer is empty**, the consumer must wait.
* Proper **synchronization** is required to prevent issues like:
  + Race conditions
  + Data inconsistency
  + Deadlocks

**Real-Time Analogy:**

Imagine a **restaurant kitchen**:

* The **chef** (producer) prepares dishes and places them on the counter (buffer).
* The **waiter** (consumer) picks up dishes from the counter and serves them.
* If the counter is full, the chef waits.
* If the counter is empty, the waiter waits.

**Java Implementation using wait() and notify()**

import java.util.LinkedList;

import java.util.Queue;

class SharedBuffer {

private Queue<Integer> buffer = new LinkedList<>();

private int capacity = 5;

public synchronized void produce(int item) throws InterruptedException {

while (buffer.size() == capacity) {

wait(); // Wait until space is available

}

buffer.add(item);

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

notify(); // Notify consumer

}

public synchronized int consume() throws InterruptedException {

while (buffer.isEmpty()) {

wait(); // Wait until item is available

}

int item = buffer.poll();

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

notify(); // Notify producer

return item;

}

}

class Producer extends Thread {

private SharedBuffer buffer;

public Producer(SharedBuffer buffer) {

this.buffer = buffer;

}

public void run() {

int i = 1;

while (true) {

try {

buffer.produce(i++);

Thread.sleep(500);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

}

class Consumer extends Thread {

private SharedBuffer buffer;

public Consumer(SharedBuffer buffer) {

this.buffer = buffer;

}

public void run() {

while (true) {

try {

buffer.consume();

Thread.sleep(800);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

}

public class ProducerConsumerExample {

public static void main(String[] args) {

SharedBuffer buffer = new SharedBuffer();

new Producer(buffer).start();

new Consumer(buffer).start();

}

}

**Concepts Used:**

* synchronized methods
* wait() and notify()
* Thread communication
* Shared buffer with size control

In **Java multithreading**, lock() is a method used to **explicitly acquire a lock** on a critical section of code or resource, ensuring that **only one thread can access it at a time**.

It is part of the **java.util.concurrent.locks** package and provides more **flexibility** and **control** than the traditional synchronized keyword.

**What is Lock?**

Lock is an **interface** in Java introduced in Java 1.5 in the package:

import java.util.concurrent.locks.Lock;

import java.util.concurrent.locks.ReentrantLock;

**Common Implementation:**

* **ReentrantLock** is the most commonly used implementation of Lock.

**Example: Bank Account Withdrawal**

import java.util.concurrent.locks.Lock;

import java.util.concurrent.locks.ReentrantLock;

class BankAccount {

private int balance = 1000;

private final Lock lock = new ReentrantLock();

public void withdraw(String name, int amount) {

lock.lock(); // Acquire lock

try {

if (balance >= amount) {

System.out.println(name + " is withdrawing: " + amount);

balance -= amount;

System.out.println("Remaining balance: " + balance);

} else {

System.out.println(name + " tried to withdraw " + amount + " but insufficient funds.");

}

} finally {

lock.unlock(); // Always release lock

}

}

}

class User extends Thread {

private BankAccount account;

private String name;

public User(BankAccount account, String name) {

this.account = account;

this.name = name;

}

public void run() {

account.withdraw(name, 600);

}

}

public class LockExample {

public static void main(String[] args) {

BankAccount account = new BankAccount();

User u1 = new User(account, "Alice");

User u2 = new User(account, "Bob");

u1.start();

u2.start();

}

}

**Why Use Lock Instead of synchronized?**

| **Feature** | **synchronized** | **Lock (ReentrantLock)** |
| --- | --- | --- |
| Flexibility | Less (automatic) | More (manual control) |
| Interruptible lock | ❌ Not possible | ✅ Yes (lockInterruptibly()) |
| Try to acquire lock | ❌ No | ✅ Yes (tryLock()) |
| Fairness policy | ❌ No | ✅ Yes (optional fairness) |
| Condition support | ❌ No | ✅ Yes (Condition object) |

**Best Practices**

* Always place lock.unlock() inside a finally block.
* Use tryLock() for timeout-based locking.
* Avoid deadlocks by consistent lock ordering.

**Real-Time Example: Warehouse Inventory ManagementScenario:**

* Multiple workers try to **pick items** from inventory.
* If inventory is low, another worker can **restock**.
* Lock is used to avoid **race conditions** on shared inventory.

**✅ Java Code with ReentrantLock:**

import java.util.concurrent.locks.Lock;

import java.util.concurrent.locks.ReentrantLock;

class Warehouse {

private int inventory = 10;

private final Lock lock = new ReentrantLock();

public void pickItem(String workerName, int quantity) {

lock.lock(); // Lock before accessing shared resource

try {

System.out.println(workerName + " trying to pick " + quantity + " items.");

if (inventory >= quantity) {

inventory -= quantity;

System.out.println(workerName + " picked " + quantity + " items. Inventory left: " + inventory);

} else {

System.out.println(workerName + ": Not enough inventory to pick " + quantity + " items.");

}

} finally {

lock.unlock(); // Ensure lock is always released

}

}

public void restock(String workerName, int quantity) {

lock.lock();

try {

inventory += quantity;

System.out.println(workerName + " restocked " + quantity + " items. Total inventory: " + inventory);

} finally {

lock.unlock();

}

}

}

class Worker extends Thread {

private Warehouse warehouse;

private String task;

private String name;

private int quantity;

public Worker(Warehouse warehouse, String name, String task, int quantity) {

this.warehouse = warehouse;

this.name = name;

this.task = task;

this.quantity = quantity;

}

public void run() {

if (task.equalsIgnoreCase("pick")) {

warehouse.pickItem(name, quantity);

} else if (task.equalsIgnoreCase("restock")) {

warehouse.restock(name, quantity);

}

}

}

public class WarehouseLockExample {

public static void main(String[] args) {

Warehouse warehouse = new Warehouse();

// Workers trying to pick or restock items

Worker w1 = new Worker(warehouse, "Alice", "pick", 4);

Worker w2 = new Worker(warehouse, "Bob", "pick", 7);

Worker w3 = new Worker(warehouse, "Charlie", "restock", 5);

Worker w4 = new Worker(warehouse, "Diana", "pick", 3);

w1.start();

w2.start();

w3.start();

w4.start();

}

}

**Key Concepts:**

* **ReentrantLock**: Ensures only one thread can modify the inventory at a time.
* **Race condition prevention**: Threads won’t read/update inventory at the same time.
* **Fairness and control**: You could also enable fairness using new ReentrantLock(true).

**🧪 Output Sample (May vary):**

Alice trying to pick 4 items.

Alice picked 4 items. Inventory left: 6

Bob trying to pick 7 items.

Bob: Not enough inventory to pick 7 items.

Charlie restocked 5 items. Total inventory: 11

Diana trying to pick 3 items.

Diana picked 3 items. Inventory left: 8

**ReentrantLock in Java**

ReentrantLock is a class in the java.util.concurrent.locks package that provides an advanced and flexible locking mechanism, **beyond what the synchronized keyword offers**.

**Definition**

ReentrantLock is a **mutual exclusion (mutex) lock** with the ability to:

* Re-acquire the same lock multiple times by the same thread
* Support fairness policies
* Interrupt waiting threads
* Use tryLock() for timed locking

**Common Methods in ReentrantLock**

| **Method** | **Description** |
| --- | --- |
| lock() | Acquires the lock, blocks if not available |
| unlock() | Releases the lock |
| tryLock() | Tries to acquire lock, returns true or false |
| tryLock(timeout, unit) | Tries to acquire lock within given time |
| lockInterruptibly() | Allows the thread to be interrupted while waiting for the lock |
| isLocked() | Checks if the lock is currently held |
| newCondition() | Creates a condition variable (advanced inter-thread communication) |

**Why "Reentrant"?**

It allows the **same thread** to acquire the lock **multiple times** **without causing a deadlock**.

Basic Syntax:

import java.util.concurrent.locks.ReentrantLock;

ReentrantLock lock = new ReentrantLock();

lock.lock(); // Acquire the lock

try {

// critical section

} finally {

lock.unlock(); // Always release the lock

}

**Example:**

lock.lock();

try {

// some code

lock.lock(); // Reentrant acquisition by the same thread

try {

// nested critical section

} finally {

lock.unlock();

}

} finally {

lock.unlock();

}