

Multithreading

- **Monitor lock:** The lock acquired by running thread on the object.

```
class MonitorLockExample{ 2 usages new *
    public synchronized void task1(){ 1 usage new *
        try{
            System.out.println("Inside task1");
            Thread.sleep( millis: 10000);
        } catch(Exception e){

        }
    }

    public void task2(){ 1 usage new *
        System.out.println("Inside task2, before synchronized");
        synchronized (this){ //this indicated the same object on which monitor lock is already acquired by thread1.
            System.out.println("Inside task2, synchronized");
        }
    }

    public void task3(){ 1 usage new *
        System.out.println("Inside task3");
    }
}
```

```
public class Demo {

    public static void main(String[] args) { new *
        MonitorLockExample lockExample = new MonitorLockExample();
        Thread t1 = new Thread(()->lockExample.task1());
        Thread t2 = new Thread(()->lockExample.task2());
        Thread t3 = new Thread(()->lockExample.task3());

        t1.start(); //puts monitor lock on lockExample object.
        t2.start();
        t3.start();
    }
}
```

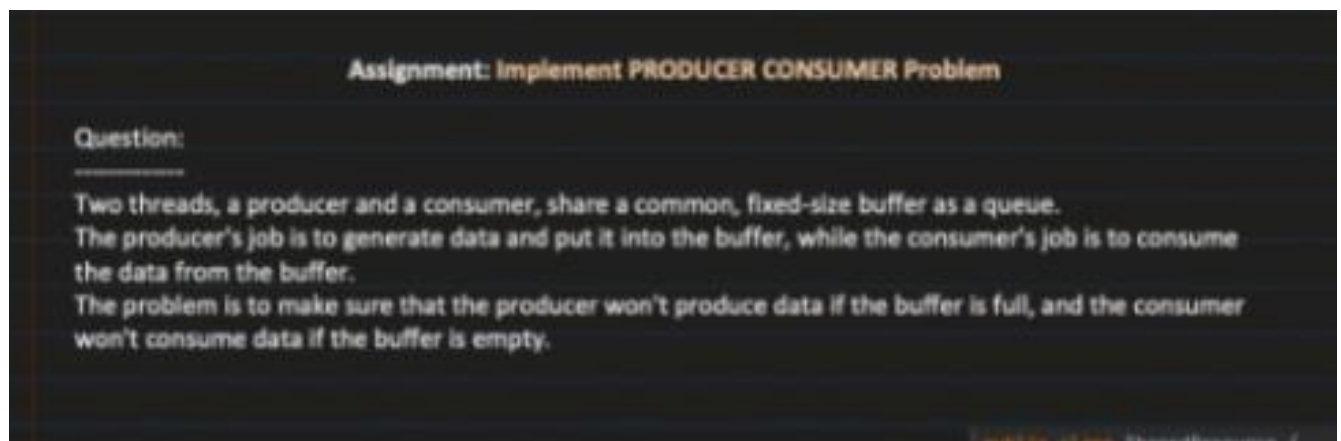
When `thread1.start()` is called, `thread1` will put monitor lock on the object (`lockExample`) as the `task1` method is synchronized. As we are calling `Thread.sleep(10000)`, the lock will be acquired on the object for 10 more seconds. Meanwhile, `thread2` will start and call `task2`. It will print the first line, but it will not go inside synchronized block because it cannot put lock on a object as lock is

already acquired by thread1. Meanwhile, thread3 will get executed. Now, after 10 seconds, when thread1 releases monitor lock, then thread2 will put monitor lock and will execute synchronized block.

Output –

```
> Task :org.practice.trialerror.Demo.main()
Inside task1
Inside task2, before synchronized
Inside task3
Calling synchronized block... (waiting for thread1 to release the lock...)
Inside task2, synchronized
```

Producer Consume problem –



The image is a screenshot of a presentation slide with a dark blue background and light blue text. The title 'Assignment: Implement PRODUCER CONSUMER Problem' is at the top in a bold, sans-serif font. Below the title, the word 'Question:' is followed by a horizontal line. The main text describes the producer-consumer problem: 'Two threads, a producer and a consumer, share a common, fixed-size buffer as a queue. The producer's job is to generate data and put it into the buffer, while the consumer's job is to consume the data from the buffer. The problem is to make sure that the producer won't produce data if the buffer is full, and the consumer won't consume data if the buffer is empty.' In the bottom right corner, there is a small, faint text that reads 'Copyright © 2000, SharedResource, Inc.'

Assignment: Implement PRODUCER CONSUMER Problem

Question:

Two threads, a producer and a consumer, share a common, fixed-size buffer as a queue. The producer's job is to generate data and put it into the buffer, while the consumer's job is to consume the data from the buffer. The problem is to make sure that the producer won't produce data if the buffer is full, and the consumer won't consume data if the buffer is empty.

Copyright © 2000, SharedResource, Inc.

```

public class SharedResource {

    private Queue<Integer> sharedBuffer;
    private int bufferSize;

    public SharedResource(int bufferSize){
        sharedBuffer = new LinkedList<>();
        this.bufferSize = bufferSize;
    }

    public synchronized void produce(int item) throws InterruptedException {
        //if buffer is full, wait for consumer to consume first.
        while(sharedBuffer.size()==bufferSize){
            System.out.println("buffer is full, producer is waiting for consumer");
            wait();
        }
        sharedBuffer.add(item);
        System.out.println("Produced -> " + item);
        notify();
    }

    public synchronized int consume() throws InterruptedException {
        //if buffer is empty wait for producer to produce the item
        while(sharedBuffer.isEmpty()){
            System.out.println("buffer is empty, waiting for producer to produce the item.");
            wait();
        }
        int item = sharedBuffer.poll();
        System.out.println("Consumed -> " + item);
        notify();
        return item;
    }
}

```

```

public class Main {

    public static void main(String[] args) {
        SharedResource sharedBuffer = new SharedResource(3);
        Thread producerThread = new Thread(() -> {
            try {
                for (int i = 0; i <= 6; i++) {
                    sharedBuffer.produce(i);
                }
            } catch (Exception e) {
                throw new RuntimeException(e);
            }
        });

        Thread consumerThread = new Thread()->{
            try {
                for (int i = 0; i <= 6; i++) {
                    sharedBuffer.consume();
                }
            } catch (Exception e) {
                throw new RuntimeException(e);
            }
        });
        producerThread.start();
        consumerThread.start();
    }
}

```

Why Stop, Resume, Suspended method is deprecated?

✓ **STOP** : Terminates the thread abruptly, No lock release, No resource clean up happens.

✓ **SUSPEND**: Put the Thread on hold (suspend) for temporarily. No lock is release too.

RESUME: Used to Resume the execution of Suspended thread.

Both this operation could led to issues like deadlock.

JOIN:

- When JOIN method is invoked on a thread object. Current thread will be blocked and waits for the specific thread to finish.
- It is helpful when we want to coordinate between threads or to ensure we complete certain task before moving ahead.

THREAD PRIORITY:

- Priorities are integer ranging from 1 to 10.

1 -> low priority

10 -> highest priority

- Even we set the thread priority while creation, its not guranttred to follow any specific order, its just a hint to thread scheduler which to execute next. (but its not strict rule)
- When new thread is created, it inherit the priority of its Parent thread.
- we can set custom priority using "**setPriority(int priority)**" method

Daemon thread –

In Java, a **daemon thread** is a special type of thread that runs in the background and is primarily used for tasks that support the application but don't need to be completed before the application terminates. **A daemon thread does not prevent the JVM from exiting. The JVM will terminate as soon as all the non-daemon threads (user threads) have completed their execution, even if the daemon threads are still running.** Their key feature is that they don't keep the application alive once all user threads have finished executing.

Use Cases: Common use cases for daemon threads include:

- Garbage collection, AutoSave
- Background tasks that periodically check conditions
- Logging or monitoring tasks that run indefinitely

In Java, you can create a daemon thread by setting the thread as a daemon before starting it. This is done using the **setDaemon(true)** method.

```
public static void main(String[] args) { new *
    Thread daemonThread = new Thread(() -> {
        while (true) {
            System.out.println("Daemon thread running...");
            try {
                Thread.sleep(1000); // Simulate some work
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }
    });

    // Set the thread as a daemon thread
    daemonThread.setDaemon(true);

    // Start the daemon thread
    daemonThread.start();

    // Main program ends, the daemon thread continues running in the background
    System.out.println("Main thread finished.");
}
```

So, daemon thread is alive only till any user thread is alive.

Custom locks and their types –

Reentrant lock:

- The lock is not dependent on object.
- Even if different object is created and trying to access a particular method then also it cannot be accessed until lock is released.

```
6 public class SharedResource { 4 usages new *
7
8
9     boolean isAvailable = false; 1 usage
10
11     public void produce(ReentrantLock reentrantLock) { 2 usages new *
12         try{
13             reentrantLock.lock();
14             System.out.println("lock acquired by -> " + Thread.currentThread().getName());
15             isAvailable = true;
16             Thread.sleep(4000);
17         } catch(Exception e){
18
19         }finally {
20             reentrantLock.unlock();
21             System.out.println("lock released by -> " + Thread.currentThread().getName());
22         }
23     }
24 }
```

```
6 public static void main(String[] args) { new *
7
8     ReentrantLock reentrantLock = new ReentrantLock();
9
10     SharedResource sharedResource1 = new SharedResource();
11     Thread t1 = new Thread(() -> {
12         sharedResource1.produce(reentrantLock);
13     });
14
15     SharedResource sharedResource2 = new SharedResource();
16     Thread t2 = new Thread(() -> {
17         sharedResource2.produce(reentrantLock); //same lock passed to different object
18     });
19
20     t1.start();
21     t2.start();
22 }
```

```
> Task :org.practice.multithreading.Main.main()
lock acquired by -> Thread-0
lock acquired by -> Thread-1
lock released by -> Thread-0
lock released by -> Thread-1
```

In the above example, both thread calling produce method using different objects but still t2 is not able to progress because lock is in place. After 4 seconds when lock is released by t1 then t2 can progress. So, locking happened on the basis of lock object and not on the basis of resource object.

Read write lock:

Read-Write Lock is a synchronization mechanism that allows multiple threads to read data concurrently while ensuring exclusive access to write operations. The idea is that reading does not modify the shared resource, so multiple threads can safely read it at the same time. However, if a thread needs to write to the resource, it must acquire exclusive access, meaning no other threads can read or write at the same time.

Read Lock (Shared Lock):

- Multiple threads can acquire the read lock simultaneously, as long as no thread holds the write lock.
- It is suitable when threads need to **read** shared data without modifying it.

Write Lock (Exclusive Lock):

- Only one thread can acquire the write lock at a time, and no other threads (neither readers nor writers) can access the shared resource when a thread holds the write lock.
- This is suitable when a thread needs to **modify** the shared data.

```

public class Main {
    private static final ReadWriteLock lock = new ReentrantReadWriteLock();
    private static String sharedData = "Initial Data";

    // Method to read data
    public static void readData() throws InterruptedException {
        System.out.println("Inside readData");
        lock.readLock().lock(); // Acquire the read lock
        try {
            System.out.println("Reading data by : " + Thread.currentThread().getName() + " value is : " + sharedData);
            Thread.sleep(8000);
        } finally {
            System.out.println("Released the read lock by : " + Thread.currentThread().getName());
            lock.readLock().unlock(); // Release the read lock
        }
    }

    // Method to write data
    public static void writeData(String newData) {
        System.out.println("Inside writeData");
        lock.writeLock().lock(); // Acquire the write lock
        try {
            sharedData = newData;
            System.out.println("Data written by : " + Thread.currentThread().getName() + " value is : " + newData);
        } finally {
            System.out.println("Released the write lock by : " + Thread.currentThread().getName());
            lock.writeLock().unlock(); // Release the write lock
        }
    }

    public static void main(String[] args) {
        // Simulating multiple threads
        Thread writer = new Thread(() -> writeData("Updated Data"));
        Thread reader1 = new Thread(() -> {
            try {
                Main.readData();
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }
        });
        Thread reader2 = new Thread(() -> {
            try {
                Main.readData();
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }
        });

        reader1.start();
        reader2.start();
        writer.start();
    }
}

```



```

> Task :org.practice.multithreading.Main.main()
Inside readData
Inside readData
Inside writeData
Reading data by : Thread-2 value is : Initial Data
Reading data by : Thread-1 value is : Initial Data
Released the read lock by : Thread-2
Released the read lock by : Thread-1
Data written by : Thread-0 value is : Updated Data
Released the write lock by : Thread-0

```

StampedLock:

Optimistic lock –

It did not put an actual lock on any object. It first put read lock which will return the current state. When we are writing/updating then we need to pass the state (which was returned by read lock). If the state is still same then only we can update the value otherwise it will rollback and we need to read the value again before attempting to update the value.

```

public class Main {
    public static void main(String args[]) {
        SharedResource resource = new SharedResource();

        Thread th1 = new Thread() -> {
            resource.producer();
        };

        Thread th2 = new Thread() -> {
            resource.consumer();
        };

        th1.start();
        th2.start();
    }
}

public class SharedResource {
    int a = 10;
    StampedLock lock = new StampedLock();

    public void producer(){
        long stamp = lock.tryOptimisticRead();
        try {
            System.out.println("Taken optimistic lock");
            a = 11;
            Thread.sleep( time: 6000);
            if(lock.validate(stamp)){
                System.out.println("updated a value successfully");
            }
            else {
                System.out.println("rollback of work");
                a = 10; //rollback
            }
        }
        catch (Exception e) {
        }
    }

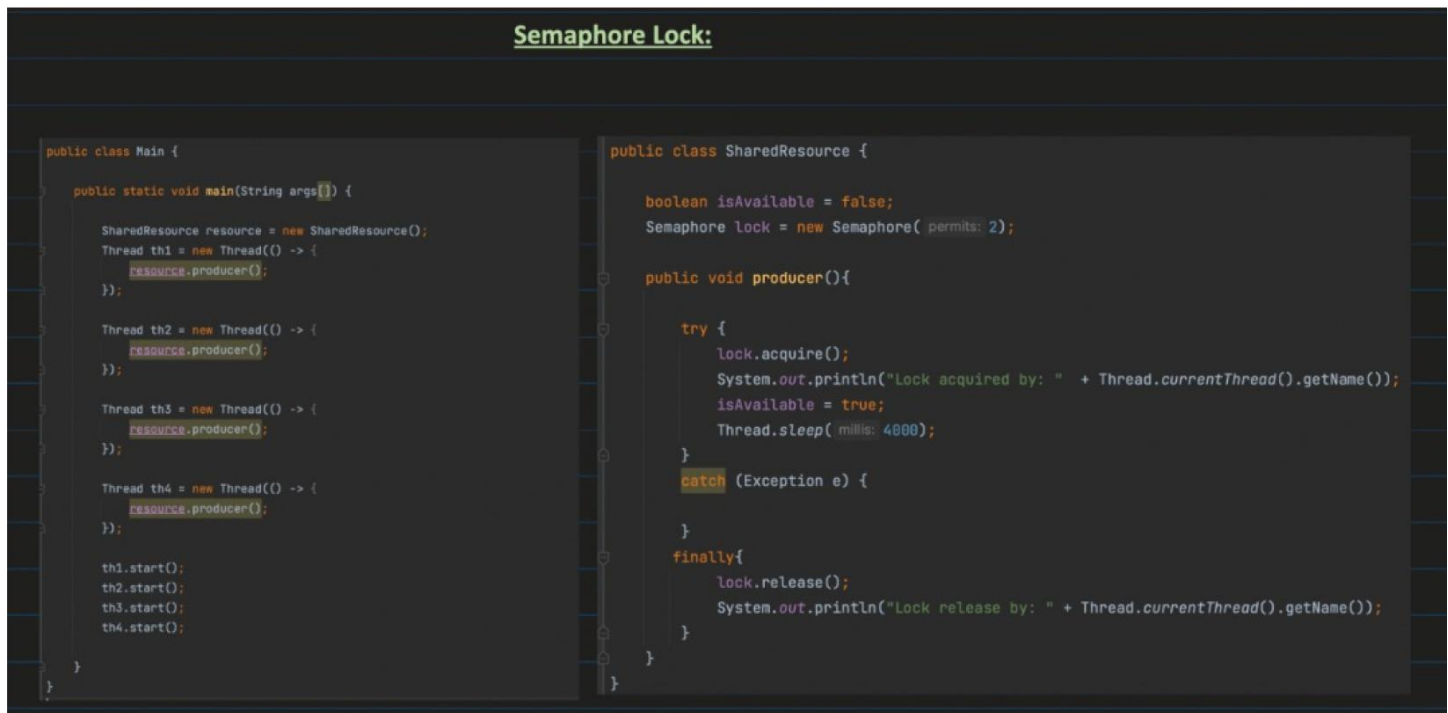
    public void consumer(){
        long stamp = lock.writeLock();
        System.out.println("write lock acquired by : " + Thread.currentThread().getName());

        try {
            System.out.println("performing work");
            a = 9;
        }
        finally {
            lock.unlockWrite(stamp);
            System.out.println("write lock released by : " + Thread.currentThread().getName());
        }
    }
}

```

Semaphore lock:

It is same as Reentrant lock with the additional control over several threads which can access a piece of code. While creating object of Semaphore lock, we can pass a number which is nothing but number of threads which can simultaneously access the code.



In the above example, there are 4 threads trying to access the producer method, but we are allowing only 2 threads at a time to access the method. Until thread-0 and thread-1 are not released, next 2 threads cannot access the method.

Lock Condition:

In case of custom locks `notify()` and `wait()` methods will not work hence we need to use –

await() = wait()
signal() = notify()

Volatile:

It is a modifier that ensures the most up-to-date value of a variable is always visible across all threads.

- **Every read** of the volatile variable will fetch the most recent value from the main memory (not from the thread's local cache).
- **Every write** to a volatile variable will be written directly to the main memory, making it immediately visible to other threads.

This is different from non-volatile variables, where each thread may maintain a cached copy of the variable (in the thread's local memory or cache), leading to potential inconsistencies.