Contents

[USCSP301\_USCS303\_Operating System (OS) Practical-07 2](#_Toc80804522)

[Practical 07: Synchronization 2](#_Toc80804523)

[Practical Date: 25 August 2021 2](#_Toc80804524)

[Practical Aim: Bounded-Buffer Problem, Readers-Writers, Sleeping Barber Problem 2](#_Toc80804525)

[Synchronization 2](#_Toc80804526)

[Bounded Buffer Problem 2](#_Toc80804527)

[Reader’s Writers Problem 10](#_Toc80804528)

[Sleeping Barber Problem 14](#_Toc80804529)

# USCSP301\_USCS303\_Operating System (OS) Practical-07

## Practical 07: Synchronization

### Practical Date: 25 August 2021

### Practical Aim: Bounded-Buffer Problem, Readers-Writers, Sleeping Barber Problem

### Synchronization

To minimize the amount of waiting time for threads that share resources and operate at the same average speeds, we implement a bounded buffer.

To avoid starvation and deadlock situation, we use the concept of synchronization or locks.

One can also determine whether a process’s request for allocation of resources be safely granted immediately.

### Bounded Buffer Problem

**The** producer-consumer problem**, also known as** Bounded Buffer Problem**, illustrates the need for synchronization in systems where many processes share a resource.**

In the problem, two processes **share a fixed-size buffer.**

One process produces information and puts it in the buffer, while the other process consumes information from the buffer.

These processes do not take turns accessing the buffer, they both work concurrently. Here lies the problem.

**In order to synchronize these processes, we will block the producer when the buffer is full, and we will block the consumer when the buffer is empty.**

**QUESTION 1:**

Write a java program for Bounded Buffer Problem using synchronization.

//Name:Ritika Sahu

//Batch : B1

//PRN:2020016400783543

//Date: 25 August, 2021.

//Practical 7: Synchronization

public interface P7\_Q1\_Buffer\_RS

{

public void set(int value) throws InterruptedException;

public int get() throws InterruptedException;

}

//Name:Ritika Sahu

//Batch : B1

//PRN:2020016400783543

//Date: 25 August, 2021.

//Practical 7: Synchronization

public class P7\_Q1\_CircularBuffer\_RS implements P7\_Q1\_Buffer\_RS

{

private final int[] buffer = {-1,-1,-1}; // sharedBuffer

private int occupiedCells = 0; // count number of buffers used

private int writeIndex = 0; // index of next element to write to

public synchronized void set(int value) throws InterruptedException

{

while(occupiedCells == buffer.length)

{

System.out.printIn("Buffer is full. Producer waits.");

wait();

}

buffer[writeindex]=value;

writeIndex = (writeIndex + 1) % buffer.length;

++occupiedCells;

displayState("Producer write "+value);

notifyAll();

} // set () ends

public synchronized int get() throws InterruptedException

{

while(occupiedCells == 0)

{

System.out.printIn("Buffer is empty.Consumer waits.");

wait();

}

int readValue = buffer[readIndex];

readIndex = (readIndex + 1) % buffer.length;

--occupiedCells;

displayState("Consumer reads "+readValue);

notifyAll();

return readValue;

} // get() ends

public void displayState(String operation)

{

System.out.printIn(%s%s%d)\n%s",operation," (buffer cells occupied:", occupiedCells,"buffer cells: ");

for(int value : buffer)

System.out.printf(" %2d ",value);

System.out.print("\n ");

for(int i=0;i<buffer.length;i++)

System.out.print(" --- ");

System.out.print("\n ");

for(int i=0;i<buffer.length;i++)

{

if(i == writeIndex && i == readIndex)

System.out.print(" WR");

else if(i == writeIndex)

System.out.print(" W");

else if(i == readIndex)

System.out.print(" R");

else

System.out.print(" ");

}

System.out.printIn("\n");

} // displayState() ends

} // CircularBuffer class ends

//Name:Ritika Sahu

//Batch : B1

//PRN:2020016400783543

//Date: 25 August, 2021.

//Practical 7: Synchronization

import java.util.Random;

public class P7\_Q1\_Producer\_RS

{

private final static Random generator = new Random();

private final P7\_Q1\_Buffer\_RS sharedLocation;

public P7\_Q1\_Producer\_RS(P7\_Q1\_Buffer\_RS shared)

{

sharedLocation = shared;

}

public void run()

{

for(int count = 1;count <= 10;count++)

{

try {

Thread.sleep(generator.nextInt(3000));

sharedLocation.set(count);

}catch(InterruptedException e) {

e.printStackTrace();

}

}

System.out.printIn("Producer done producing. Terminating Producer");

} // run() ends

} // Producer class ends

//Name:Ritika Sahu

//Batch : B1

//PRN:2020016400783543

//Date: 25 August, 2021.

//Practical 7: Synchronization

import java.util.Random;

public class P7\_Q1\_Consumer\_RS implements Runnable

{

private final static Random generator = new Random();

private final P7\_Q1\_Buffer\_RS sharedLocation;

public P7\_Q1\_Consumer\_RS(P7\_Q1\_Buffer\_RS shared)

{

sharedLocation = shared;

}

public void run()

{

int sum =0;

for(int count = 1;count <=0;count++)

{

try {

Thread.sleep(generator.nextInt(3000));

sum += sharedLocation.get();

}catch(InterruptedException e) {

e,printStackTrace();

}

}

System.out.printf("\n%s %d\n%s\n","Consumer read values totaling", sum,"Terminating Consumer");

} // run() ends

} // Consumer class ends

//Name:Ritika Sahu

//Batch : B1

//PRN:2020016400783543

//Date: 25 August, 2021.

//Practical 7: Synchronization

import java.util.concurrent,\*;

public class P7\_Q1\_Test\_RS

{

public static void main(String[] args)

{

ExecutorService application = executors.newCachedThreadPool();

P7\_Q1\_CircularBuffer\_RS sharedLocation = new P7\_Q1\_CircularBuffer\_RS();

sharedLocation.displayState("Initial State");

application.execute(new P7\_Q1\_Producer\_RS(sharedLocation));

application.execute(new P7\_Q1\_Consumer\_RS(sharedLocation));

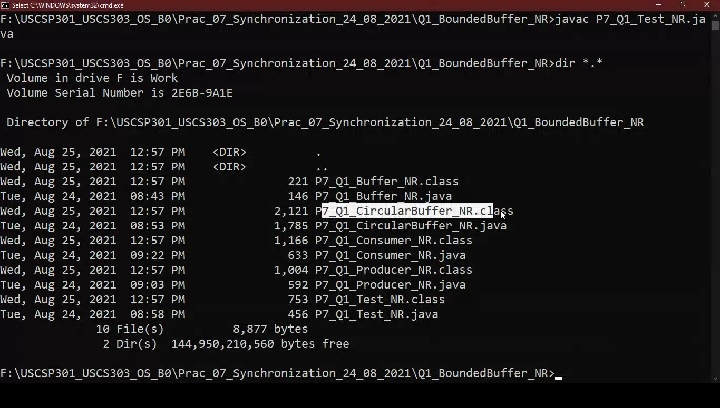
application.shutdown();

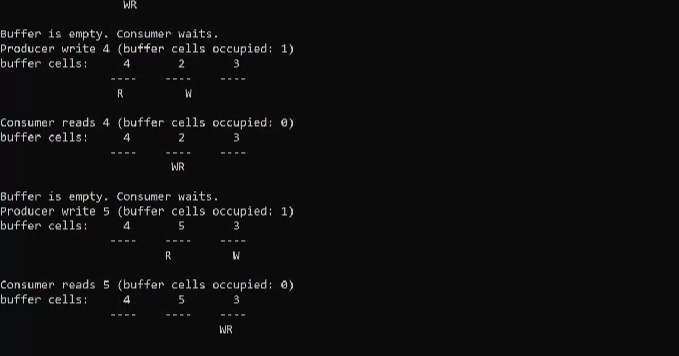
}

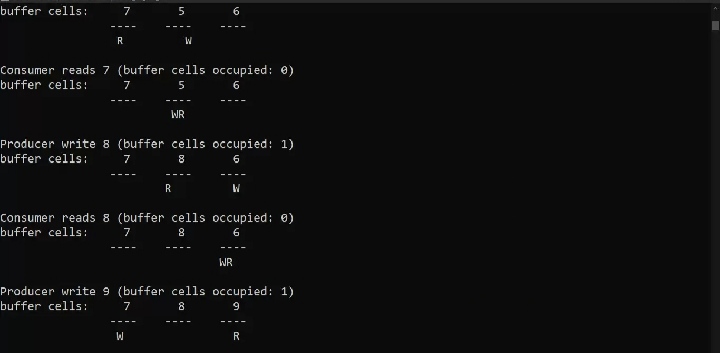
}

S

**OUTPUT:**







### Reader’s Writers Problem

1. In computer science, the **readers-writers problems** are example of a common computing problem in concurrency.
2. Here many threads (small processes which share data) try to access the same thread resource at one time.
3. Some threads may read and some may write, with the constraint that no process may access the shared resource for either reading or writing while another process is in the act of writing to it.
4. (In particular, we want to prevent more than one thread modify the shared resource simultaneously and allowed for two or more readers to access the shared rsource at the same time).
5. A readers-writer lock is a data structure that solves one or more of the readers-writers problems.

**QUESTION 2:**

Write a java program for Readers Writers Problem using semaphore.

//Name:Ritika Sahu

//Batch : B1

//PRN:2020016400783543

//Date: 25 August, 2021.

//Practical 7: Synchronization

import java.util.concurrent.Semaphore;

class P7\_Q2\_ReaderWriter\_RS

{

static Semaphore readLock = new Semaphore(1, true);

static Semaphore writeLock = new Semaphore(1, true);

static int readCount = 0;

static class Read implements Runnable {

@Override

public void run() {

try {

// Acquire Section

readLock.acquire();

readCount++;

if (readCount == 1) {

writeLock.acquire();

}

readLock.release();

// Reading section

System.out.print("Thread "+Thread.currentThread().getName() + " is READING");

Thread.sleep(1500);

System.out.print("Thread "+Thread.currentThread().getName() + " has FINISHED READING");

//Releasing section

readLock.acquire();

readCount --;

if(readCount == 0) {

writeLock.release();

} // try ends

catch(InterruptedException e) {

System.out.printIn(e.getMessage());

}

}// run() ends

}

// static class Read ends

static class Write implements Runnable {

@Override

public void run() {

try {

writeLock.acquire();

System.out.print("Thread "+Thread.currentThread().getName() + " is WRITING");

Thread.sleep(2500);

System.out.print("Thread "+Thread.currentThread().getName() + " has FINISHED WRITING");

writeLock.release();

} catch(InterruptedException e) {

System.out.printIn(e.getMessage());

}

}// run() ends

} // class Write ends

public static void main(String[] args)

throws Exception {

Read read = new Read();

Write write = new Write();

Thread t1 = new Thread(read);

t1.setName("thread1");

Thread t2 = new Thread(read);

t2.setName("thread2");

Thread t3 = new Thread(read);

t3.setName("thread3");

Thread t4 = new Thread(read);

t4.setName("thread4");

t1.start();

t3.start();

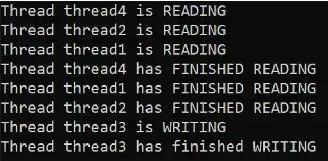
t2.start();

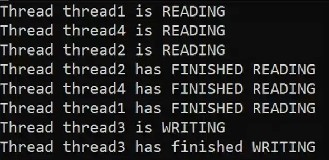
t4.start();

} // main ends

} // class P7\_Q2\_ReaderWriter\_RS ends

**OUTPUT:**





NOTE: The output keeps on changing based on the run due to random calls made to Threads.

### Sleeping Barber Problem

1. A barber shop consist of awaiting room with n chairs and a barber room with one barber chain.
2. If there are no customers to be served, the barber goes to sleep.
3. If a customer enters the barbershop and all chairs are occupied, then the customer leaves the shop.
4. If the barber is busy but chairs are available, then the customer sits in one of the free chairs. If the barber is asleep, the customer wakes u the barber.

//Name: Ritika Sahu

//Batch : B1

//PRN:2020016400783543

//Date: 25 August, 2021.

//Practical 7: Synchronization

import java.util.concurent.\*;

import java.util.concurrent.atomic.AtomicInteger;

import java.util.Random;

public class P7\_Q3\_Barber\_RS implements Runnable

{

private AtomicInteger spaces;

private Semaphore bavailable;

private Semaphore cavailable;

private Random ran = new Random();

public class P7\_Q3\_Barber\_RS (AtomicInteger spaces, Semaphore bavailable, Semaphore cavailable) {

this.spaces = spaces;

this.bavailable = bavailable;

this.cavailable = cavailable;

}

@Override

public void run() {

while (true) {

try {

cavailable.acquire();

// Space freed up in waiting area

System.out.printIn("Consumer getting hair cut");

Thread.sleep(ThreadLocalRandom.current().nextInt(1000, 10000 + 1000));

// Sleep to imitate length of time to cut hair

System.out.printIn("Conumer Pays and leaves");

bavailable.release();

} catch (InterruptedException e) {}

} // while ends

} // run ends

} // class ends

//Name:Ritika Sahu

//Batch : B1

//PRN:2020016400783543

//Date: 25 August, 2021.

//Practical 7: Synchronization

import java.util.concurent.\*;

import java.util.concurrent.atomic.AtomicInteger;

import java.util.Random;

public class P7\_Q3\_Customer\_RS implements Runnable

{

private AtomicInteger spaces;

private Semaphore bavailable;

private Semaphore cavailable;

private Random ran = new Random();

public class P7\_Q3\_Customer\_RS (AtomicInteger spaces, Semaphore bavailable, Semaphore cavailable) {

this.spaces = spaces;

this.bavailable = bavailable;

this.cavailable = cavailable;

}

@Override

public void run() {

try {

cavailable.release();

if(bavailable.hasQueuedThreads()) {

spaces.decrementAndGet();

System.out.printIn("Customer in waiting area");

bavailable.acquire();

spaces.incrementAndGet();

}

else

{

bavailable.acquire();

}

} catch (InterruptedException e) {}

} // run() ends

} // P7\_Q3\_Customer\_RS class

//Name:Ritika Sahu

//Batch : B1

//PRN:2020016400783543

//Date: 25 August, 2021.

//Practical 7: Synchronization

import java.util.concurrent.atomic.AtomicInteger;

import java.util.concurrent.\*;

public class P7\_Q3\_BarberShop\_RS {

public static void main(String[] args)

{

AtomicInteger spaces = new atomicInteger(15);

final Semaphore barbers = new Semaphore (3, true);

final Semaphore customers = new Semaphore (0, true);

ExecutorService openUp = exceution.newFixedThreadPool(3);

P7\_Q3\_Barber\_RS[] employees = new P7\_Q3\_Barber\_RS[3];

System.out.printIn("Opening up shop");

for(int i = 0; i<3;i++) {

employee[i] = new P7\_Q3\_Barber\_RS(spaces, barbers,customers);

openUp.execute(employees[i]);

}

while(true)

{

try {

Thread.sleep(ThreadLocalRandom.current().nextInt(100, 1000 + 100)); // Sleep until next person gets in

}

catch (InterruptedException e) {}

System.out.printIn("Customer walks in");

if(spaces.get() >= 0) {

new Thread(new P7\_Q3\_Customer\_RS(spaces, barbers, customers)).start();

}

else {

System.out.printIn("Customer walks out, as no seats are available");

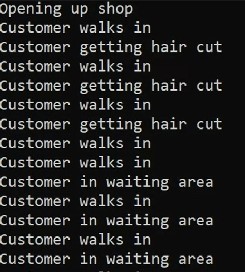
}

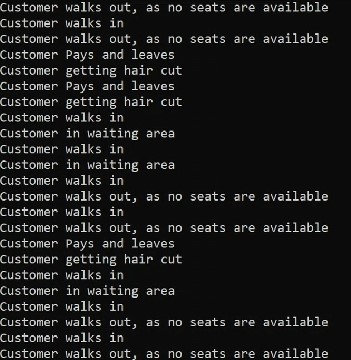
} // while ends

} // main ends

} // P7\_Q3\_BarberShop\_RS class ends

**OUTPUT:**

****

****