Smt. Chandibai Himathmal Mansukhani College

Contents

USCSP301-USCS303: Operating system (OS) Practical-07 2

Practical-07: Synchronization (Bounded Buffer, Readers - Writers Problem, Sleeping-Barber Problem). 2

Practical Date: 27– 08 – 2021 2

Practical Aim: Bounded Buffer Problem, Readers - Writers Problem, Sleeping-Barber Problem. 2

Synchronization: 2

1. Bounded Buffer Problem. 2

a) Definition:- 2

b) Question - 01: 4

c) Implementation 4

d) Output: 14

2. Readers - Writers Problem. 16

a) Definition 16

b) Question-02: 16

c) Implementation 16

d) Output: 19

3. Sleeping-Barber Problem. 20

a) Definition 20

b) Question-03: 20

c) Implementation: 20

d) Output: 26

NAME :- Bimal Lugun BATCH NO :- B2

USCSP301-USCS303: Operating system (OS) Practical-07

Practical-07: Synchronization (Bounded Buffer, Readers - Writers Problem, Sleeping-Barber Problem).

Practical Date: 27– 08 – 2021

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

Synchronization:

1. **Bounded Buffer Problem.**
    a) **Definition:-**

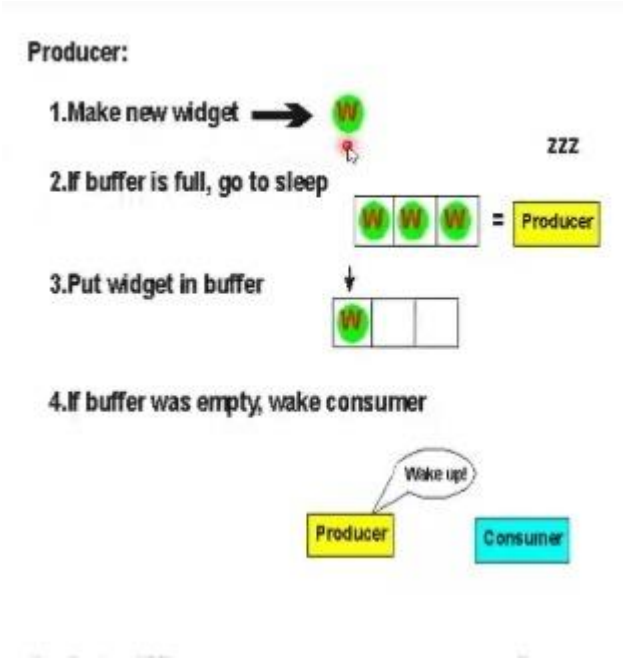
The producer-consumer problem, also known as Bounded Buffer Problem, illustrates the need for synchronization in systems where many processes share a resource. In the problem, two processes buffer share a fixed-size. One process produces information and puts it in the buffer, while the other process consumes information from the buffer.These processes do not take turns accessing the buffer, they both work concurrently. Herein lies the problem.

•What happens if the producer tries to put an item into a full buffer?

•What happens if the consumer tries to take an item from an empty buffer?

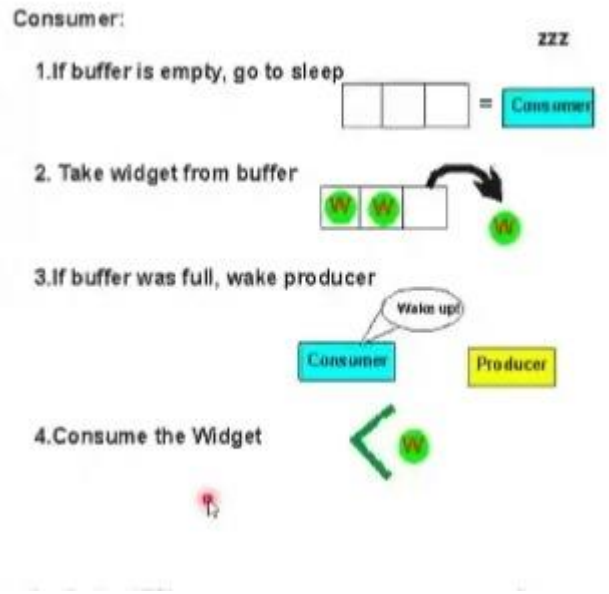
In order to synchronize these processes, we will block the producer when the buffer is full, and we will block the consumer when the buffer is empty.

1. So the two processes, Producer and Consumer, should work as follows:
   i) The producer must first create a new widget.
2. ii)Then, it checks to see if the buffer is full. If it is, the producer will put itself to sleep until the consumer wakes it up. A "wakeup" will come if the consumer finds the buffer empty.
3. iii) Next, the producer puts the new widget in the buffer. If the producer goes to sleep in step (ii), it will not wake up until the buffer is empty, so the buffer will never overflow.
4. iv) Then, the producer checks to see if the buffer is empty. If it is, the producer assumes that the consumer is sleeping, and so it will wake the consumer. Keep in mind that between any of these steps, an interrupt might occur, allowing the consumer to run.



So the two processes, Producer and Consumer, should work as follows:

1. The consumer checks to see if the buffer is empty. If so, the consumer will put itself to sleep until the producer wakes It up. A "wakeup" will occur if the producer finds the buffer empty after it puts an item into the buffer.
2. Then, the consumer will remove a widget. from the buffer. The consumer will never try to remove a widget from an empty buffer because it will not wake up until the buffer is full. If the buffer was full before it removed the widget, the consumer will wake the producer.
3. If the buffer was full before it removed the widget, the consumer will wake the producer.
4. Finally, the consumer will consume the widget. As was the case with the producer, an interrupt could occur between any of these steps, allowing the producer to run.



b) Question - 01:

Write a java program for Bounded Buffer Problem using synchronization

c) Implementation

Code 1:

//Name: ABHISHEK NIKAM

// Batch: B2

// PRN: 2020016400805951

// Date: 27 August 2021

// Prac-07: Synchronization

public interface P7_Q1_Buffer_AN

{

```
    public void set(int value) throws InterruptedException;

    public int get() throws InterruptedException;

}
```

Code 2:

```
//Name: ABHISHEK NIKAM
// Batch: B2
// PRN: 2020016400805951
// Date: 27 August 2021
// Prac-07: Synchronization


public class P7_Q1_CircularBuffer_BL implements P7_Q1_Buffer_BL
{

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

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

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

        private int readIndex = 0; // index of next element to read

public synchronized void set(int value) throws InterruptedException
{
    while(occupiedCells == buffer.length)
    {
    System.out.println("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.println("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.printf("%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.println("\n");
} // displayState() ends

}// CircularBuffer class ends
```

Code 3:

```
//Name: ABHISHEK NIKAM
// Batch: B2
// PRN: 2020016400805951
// Date: 27 August 2021
// Prac-07: Synchronization

import java.util.Random;
public class P7_Q1_Consumer_AN implements Runnable
{
        private final static Random generator = new Random();
         private final P7_Q1_Buffer_BL sharedLocation;
        public P7_Q1_Consumer_BL(P7_Q1_Buffer_BL shared)
        {
                sharedLocation=shared;
        }
public void  run()
 {
      int sum=0;
      for(int count = 1;count <= 10;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
```

Code 4:

```
//Name: ABHISHEK NIKAM
// Batch: B2
// PRN: 2020016400805951
// Date: 27 August 2021
// Prac-07: Synchronization

import java.util.Random;
public class P7_Q1_Producer_BL implements Runnable
{
        private final static Random generator=new Random();
        private final P7_Q1_Buffer_BL sharedLocation;

        public P7_Q1_Producer_BL(P7_Q1_Buffer_BL 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.println("Producer done producing.Terminating Producer");
  }//run() ends
}//Producer class ends
```

Code 5:

```
//Name: ABHISHEK NIKAM
// Batch: B2
// PRN: 2020016400805951
// Date: 27 August 2021
// Prac-07: Synchronization

import java.util.concurrent.*;
public class P7_Q1_Test_BL
{
        public static void main(String args[])
        {
                ExecutorService application = Executors.newCachedThreadPool();
                P7_Q1_CircularBuffer_BL sharedLocation = new
P7_Q1_CircularBuffer_BL();
                sharedLocation.displayState("Initial State");
                application.execute(new P7_Q1_Producer_BL(sharedLocation));
                application.execute(new P7_Q1_Consumer_BL(sharedLocation));
                 application.shutdown();
        }
}
```

d) Output:

```
Initial State(buffer cells occupied: 0)
buffer cells:  -1   -1   -1
     ----  ----  ----
        WR

Producer write 1(buffer cells occupied: 1)
buffer cells:   1   -1   -1
     ----  ----  ----
        R   W

Producer write 2(buffer cells occupied: 2)
buffer cells:   1    2   -1
     ----  ----  ----
        R      W

Consumer reads 1(buffer cells occupied: 1)
buffer cells:   1    2   -1
     ----  ----  ----
           R   W

Consumer reads 2(buffer cells occupied: 0)
buffer cells:   1    2   -1
     ----  ----  ----
             WR
```

```
Producer write 3(buffer cells occupied: 1)
buffer cells:   1    2    3
     ----  ----  ----
        W     R

Consumer reads 3(buffer cells occupied: 0)
buffer cells:   1    2    3
     ----  ----  ----
        WR

Buffer is empty. Consumer waits.
Producer write 4(buffer cells occupied: 1)
buffer cells:   4    2    3
     ----  ----  ----
        R   W

Consumer reads 4(buffer cells occupied: 0)
buffer cells:   4    2    3
     ----  ----  ----
         WR

Producer write 5(buffer cells occupied: 1)
buffer cells:   4    5    3
     ----  ----  ----
         R   W

Producer write 6(buffer cells occupied: 2)
buffer cells:   4    5    6
     ----  ----  ----
        W   R

Consumer reads 5(buffer cells occupied: 1)
buffer cells:   4    5    6
     ----  ----  ----
        W     R

Consumer reads 6(buffer cells occupied: 0)
buffer cells:   4    5    6
     ----  ----  ----
        WR
```

```
Producer write 7(buffer cells occupied: 1)
buffer cells:   7    5    6
     ----  ----  ----
         R   W

Consumer reads 7(buffer cells occupied: 0)
buffer cells:   7    5    6
     ----  ----  ----
           WR

Producer write 8(buffer cells occupied: 1)
buffer cells:   7    8    6
     ----  ----  ----
           R   W

Consumer reads 8(buffer cells occupied: 0)
buffer cells:   7    8    6
     ----  ----  ----
              WR

Producer write 9(buffer cells occupied: 1)
buffer cells:   7    8    9
     ----  ----  ----
         W      R

Consumer reads 9(buffer cells occupied: 0)
buffer cells:   7    8    9
     ----  ----  ----
         WR

Buffer is empty. Consumer waits.
Producer write 10(buffer cells occupied: 1)
buffer cells:  10    8    9
     ----  ----  ----
         R   W

Consumer reads 10(buffer cells occupied: 0)
buffer cells:  10    8  Producer done producing.Terminating Producer
  9
     ----  ----  ----
           WR


Consumer read values totaling  55
Terminating Consumer

C:\SYCS\USCSP301_USCS303_OS_B2\Prac_07_Bimal_BL_27 aug 2021\Q1_Bounded buffer_BL>
```

2. Readers - Writers Problem.

a) Definition

In computer science, the readers-writers problems are examples of a common computing problem in concurrency. Here many threads (small processes which share data) try to access the same shared resource at one time. 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. (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 resource at the same time).A readers-writer lock is a data structure that solves one or more of the readers-writers problems.

b) Question-02:

Write a java program for Readers - Writers Problem using semaphore.

c) Implementation

Code:

```
//Name: ABHISHEK NIKAM
// Batch: B2
// PRN: 2020016400805951
// Date: 27 August 2021
// Prac-07: Synchronization

import java.util.concurrent.Semaphore;

class P7_Q2_ReaderWriter_AN
{
        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.println("Thread" + Thread.currentThread().getName()+ "is
READING");
                Thread.sleep(1500);
                System.out.println("Thread"+Thread.currentThread().getName() + " has
FINISHED READING");
            //Releasing section
                readLock.acquire();
                readCount--;
                if(readCount == 0) {
                    writeLock.release();
                }
                readLock.release();
                }//try ends
                catch (InterruptedException e){
                    System.out.println(e.getMessage());
                }
            }//run() ends
        }//static class Read ends
static class Write implements Runnable{
        @Override
        public void run(){
```

```
        try {
                writeLock.acquire();
                System.out.println("Thread"+Thread.currentThread().getName()+"is WRITING");
                Thread.sleep(2500);
                System.out.println("Thread"+Thread.currentThread().getName()+"has finished
WRITING");
                writeLock.release();
                }catch (InterruptedException e){
                        System.out.println(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(write);
                t3.setName("thread3");
                Thread t4 = new Thread(read);
                t4.setName("thread4");
                t1.start();
                t3.start();
                t2.start();
                t4.start();
        }//main ends
}//class P7_Q2_ReadWriter_BL ends
```

d) Output:

```
Threadthread1is READING
Threadthread4is READING
Threadthread2is READING
Threadthread4 has FINISHED READING
Threadthread2 has FINISHED READING
Threadthread1 has FINISHED READING
Threadthread3is WRITING
Threadthread3has finished WRITING
```

3. Sleeping-Barber Problem.

a) Definition

- A barber shop consists of awaiting room with n chairs and a barber room with one barber chair.
- If there are no customers to be served, the barber goes to sleep.
- If a customer enters the barbershop and all chairs are occupied, then the customer leaves the shop.
- 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 up the barber.

b) Question-03:

Write a program to coordinate the barber and the customer using Java synchronization.

c) Implementation:

Code 1:

```
//Name: ABHISHEK NIKAM
// Batch: B2
// PRN: 2020016400805951
// Date: 27 August 2021
// Prac-07: Synchronization

import java.util.concurrent.*;
import java.util.concurrent.atomic.AtomicInteger;
import java.util.Random;

public class P7_Q3_Barber_BL implements Runnable
{
        private AtomicInteger spaces;
```

```
        private Semaphore bavailable;
        private Semaphore cavailable;
        private Random ran = new Random();
        public P7_Q3_Barber_BL(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.println("Customer getting hair cut");

        Thread.sleep(ThreadLocalRandom.current().nextInt(1000,10000+1000));
                                //Sleep to imitate length of time to cut hair
                                System.out.println("Customer pays and leaves");
                                bavailable.release();
                        }catch(InterruptedException e){}
                }//while ends
        }//run ends
}//class ends
```

Code 2:

```
//Name: ABHISHEK NIKAM
// Batch: B2
// PRN: 2020016400805951
// Date: 27 August 2021
// Prac-07: Synchronization

import java.util.concurrent.atomic.AtomicInteger;
import java.util.concurrent.*;

class P7_Q3_BarberShop_BL{
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 = Executors.newFixedThreadPool(3);
        P7_Q3_Barber_BL[] employees = new P7_Q3_Barber_BL[3];
        System.out.println("Opening up shop");
        for(int i = 0; i < 3; i++){
                employees[i] = new P7_Q3_Barber_BL(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.println("Customer walks in");
                if(spaces.get() >= 0){
                        new Thread(new P7_Q3_Customer_BL(spaces,
barbers,customers)).start();
                }
                else{
                        System.out.println("Customer walks out, as no seats are
available");
                }
        }//while ends
        }//main ends
}//P7_Q3_BarberShop_BL class ends
```

Code 3:

```
//Name: ABHISHEK NIKAM
// Batch: B2
// PRN: 2020016400805951
// Date: 27 August 2021
// Prac-07: Synchronization

import java.util.concurrent.*;
import java.util.concurrent.atomic.AtomicInteger;
import java.util.Random;

public class P7_Q3_Customer_BL implements Runnable
{
        private AtomicInteger spaces;
        private Semaphore bavailable;
        private Semaphore cavailable;
        private Random ran = new Random();
        public P7_Q3_Customer_BL(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.println("Customer in waiting area");
                    bavailable.acquire();
                    spaces.incrementAndGet();
                }
                else
                {
                    bavailable.acquire();
                }
            }catch(InterruptedException e){}
        }//run ends
}//P7_Q3_Customer_BL class
```

d) Output:

```
Opening up shop
Customer walks in
Customer getting hair cut
Customer walks in
Customer getting hair cut
Customer walks in
Customer getting hair cut
Customer pays and leaves
Customer walks in
Customer getting hair cut
Customer walks in
Customer walks in
Customer in waiting area
Customer walks in
Customer in waiting area
Customer pays and leaves
Customer getting hair cut
Customer walks in
Customer in waiting area
Customer walks in
Customer in waiting area
Customer walks in
Customer in waiting area
Customer walks in
Customer in waiting area
Customer walks in
Customer in waiting area
Customer pays and leaves
Customer getting hair cut
Customer walks in
Customer in waiting area
```

```
Customer in waiting area
Customer pays and leaves
Customer getting hair cut
Customer walks in
Customer in waiting area
Customer walks in
Customer in waiting area
Customer pays and leaves
Customer getting hair cut
Customer walks in
Customer in waiting area
Customer walks in
Customer walks out, as no seats are available
Customer walks in
Customer walks out, as no seats are available
Customer walks in
Customer walks out, as no seats are available
Customer walks in
Customer walks out, as no seats are available
Customer pays and leaves
Customer getting hair cut
Customer walks in
Customer in waiting area
Customer pays and leaves
Customer getting hair cut
Customer walks in
Customer in waiting area
Customer walks in
Customer walks out, as no seats are available
Customer walks in
Customer walks out, as no seats are available
Customer pays and leaves
Customer getting hair cut
Customer walks in
Customer in waiting area
Customer pays and leaves
Customer getting hair cut
Customer walks in
Customer in waiting area
Customer walks in
Customer walks out, as no seats are available
Customer walks in
Customer walks out, as no seats are available
Customer walks in
Customer walks out, as no seats are available
Customer pays and leaves
Customer getting hair cut
Customer walks in
Customer in waiting area
Customer walks in
```