EXPERIMENT NO : 3

**Problem Statement:** Write a program to solve Classical Problems of Synchronization using Mutex and Semaphore.

1.Bounded-buffer (or Producer-Consumer) Problem: 2.Dining-Philosophers Problem:

3.Reader and Writers Problem:

## Bounded-buffer (or Producer-Consumer) Problem:

**Source Code:**

**package** lab;

//Java implementation of a producer and consumer

//that use semaphores to control synchronization.

**import** java.util.concurrent.Semaphore;

**class** Q {

// an item

**int** item;

// semCon initialized with 0 permits

// to ensure put() executes first

**static** Semaphore *semCon* = **new** Semaphore(0);

**static** Semaphore *semProd* = **new** Semaphore(1);

// to get an item from buffer

**void** get()

{

**try** {

}

// Before consumer can consume an item,

// it must acquire a permit from semCon

*semCon*.acquire();

**catch** (InterruptedException e) { System.***out***.println("InterruptedException caught");

}

// consumer consuming an item System.***out***.println("Consumer consumed item : " + item);

// After consumer consumes the item,

// it releases semProd to notify producer

*semProd*.release();

}

// to put an item in buffer

**void** put(**int** item)

{

**try** {

}

// Before producer can produce an item,

// it must acquire a permit from semProd

*semProd*.acquire();

**catch** (InterruptedException e) { System.***out***.println("InterruptedException caught");

}

// producer producing an item

**this**.item = item;

System.***out***.println("Producer produced item : " + item);

// After producer produces the item,

// it releases semCon to notify consumer

*semCon*.release();

}

}

//Producer class

**class** Producer **implements** Runnable { Q q;

Producer(Q q)

{

**this**.q = q;

**new** Thread(**this**, "Producer").start();

}

**public void** run()

{

**for** (**int** i = 0; i < 5; i++)

// producer put items q.put(i);

}

}

//Consumer class

**class** Consumer **implements** Runnable { Q q;

Consumer(Q q)

{

**this**.q = q;

**new** Thread(**this**, "Consumer").start();

}

**public void** run()

{

**for** (**int** i = 0; i < 5; i++)

// consumer get items q.get();

}

}

//Driver class

**class** PC {

**public static void** main(String args[])

{

// creating buffer queue Q q = **new** Q();

// starting consumer thread

**new** Consumer(q);

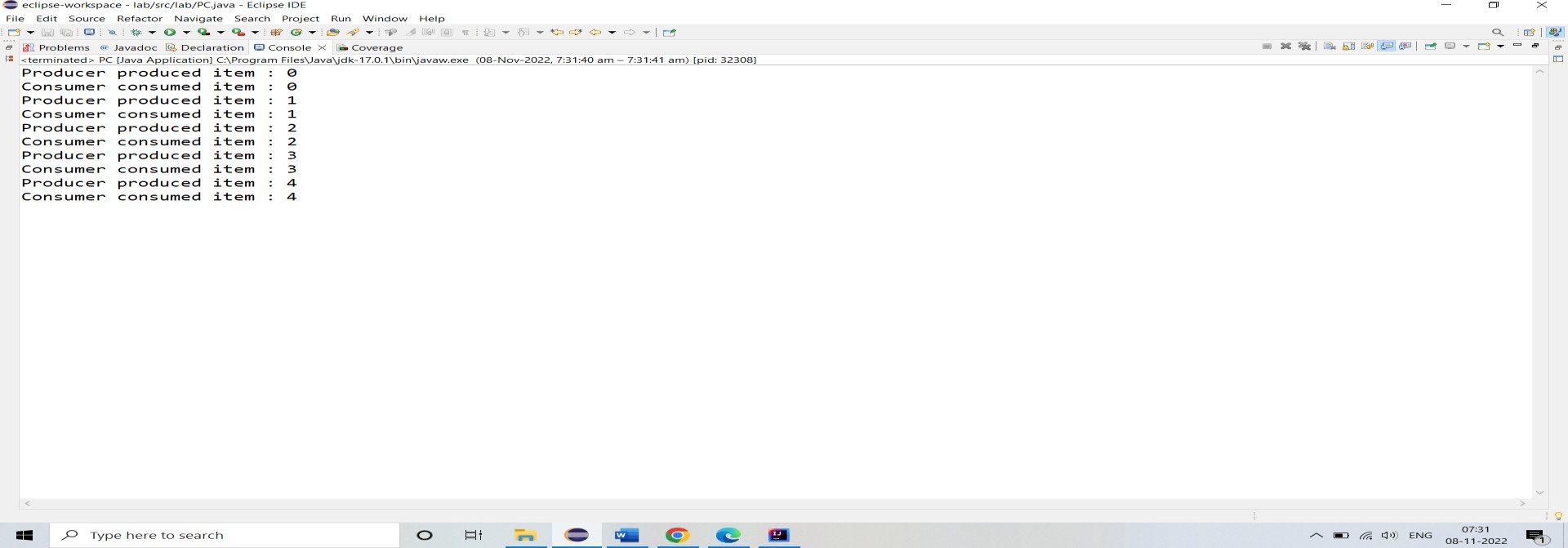
// starting producer thread

**new** Producer(q);

}

}

# OUTPUT:



## Dining-Philosophers Problem:

**Source Code:**

**package** lab;

**import** java.util.concurrent.Semaphore;

**import** java.util.concurrent.ThreadLocalRandom;

**public class** Diningphilosopher

{

//defining the number of philosophers

**static int** *philosopher* = 5;

//initializing an array of philosophers with the number of philosophers

**static** Philosopher *philosophers*[] = **new** Philosopher[*philosopher*];

//initializing an array of chosticks with the number of philosophers **static** Chopstick *chopsticks*[] = **new** Chopstick[*philosopher*]; **static class** Chopstick

{

//creating a constructor of the Semaphore class that accepts the number permits

**public** Semaphore mutex = **new** Semaphore(1);

//the method grabs the chopstick

**void** grab()

{

## try

{

//acquires a permit from the semaphore mutex.acquire();

}

**catch** (Exception e)

{

e.printStackTrace(System.***out***);

}

}

//release the chopstick

**void** release()

{

//releases an acquire a permit and increases the number of available permits by one mutex.release();

}

//checks if the chopstick is free or not

**boolean** isFree()

{

//the method returns the current number of permits available in the semaphore

//returns true if available permits is greater than 0, else returns false

**return** mutex.availablePermits() > 0;

}

} //end of Chopstick class

**static class** Philosopher **extends** Thread

{

**public int** number;

//represents left chopstick

**public** Chopstick leftchopstick;

//represents right chopstick

**public** Chopstick rightchopstick;

//creating a constructor of the Philosopher class Philosopher(**int** num, Chopstick left, Chopstick right)

{

number = num; leftchopstick = left; rightchopstick = right;

}

**public void** run()

{

## while (true)

{

//philosopher grabs the chopsticks if both are free leftchopstick.grab();

System.***out***.println("Philosopher " + (number+1) + " grabs left chopstick."); rightchopstick.grab();

System.***out***.println("Philosopher " + (number+1) + " grabs right chopstick.");

//hunger philosopher starts eating eat();

//releases left and right chopsticks when philosopher is not hunger leftchopstick.release();

System.***out***.println("Philosopher " + (number+1) + " releases left chopstick."); rightchopstick.release();

System.***out***.println("Philosopher " + (number+1) + " releases right chopstick.");

} //end of while loop

} //end of run() method

//the method invokes after grabbing both the chopsticks (left and right)

**void** eat()

{

## try

{

//determines the pseudorandom number between 0 to 1000 that represents the sleep time in milli seconds

**int** sleepTime = ThreadLocalRandom.*current*().nextInt(0, 1000); System.***out***.println("Philosopher " + (number+1) + " eats for " + sleepTime +"ms");

//sleeps the thread for a specified time Thread.*sleep*(sleepTime);

}

**catch** (Exception e)

{

e.printStackTrace(System.***out***);

}

}// end of eat() method

}

**public static void** main(String args[])

{

//loop iterates over chopsticks

**for** (**int** i = 0; i < *philosopher*; i++)

{

*chopsticks*[i] = **new** Chopstick();

} //end of for loop

//loop iterates over philosopher

**for** (**int** i = 0; i < *philosopher*; i++)

{

*philosophers*[i] = **new** Philosopher(i, *chopsticks*[i], *chopsticks*[(i + 1) % *philosopher*]);

//begins the execution of the thread

*philosophers*[i].start();

} //end of for loop

## while (true)

{

## try

{

//thread sleep for 1 sec or 1000ms Thread.*sleep*(1000);

//check for deadlock condition

**boolean** deadlock = **true**;

//for each loop iterates over chopsticks

**for** (Chopstick cs : *chopsticks*)

{

//checks if chopstick is free or not

**if** (cs.isFree())

{

deadlock = **false**; **break**;

} //end of if

} //end of for loop

//deadlock occurs if sleep time is 1000ms it means each philosopher is eating

**if** (deadlock)

{

Thread.*sleep*(1000); System.***out***.println("Everyone Eats"); **break**;

} //end of if

}

**catch** (Exception e)

{

e.printStackTrace(System.***out***);

}

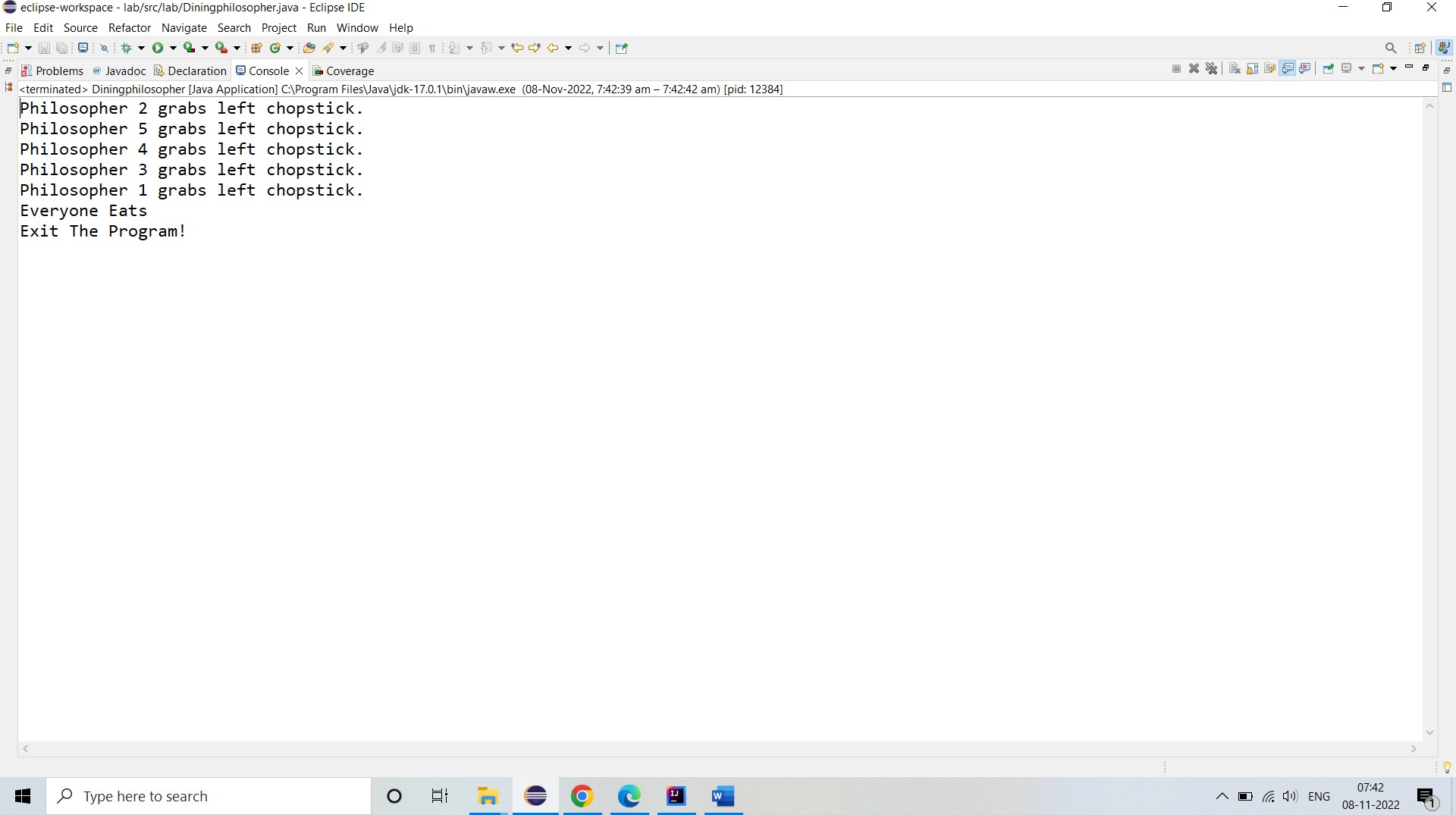
}

System.***out***.println("Exit The Program!"); System.*exit*(0);

}

}

## OUTPUT:



1. **Reader and Writers Problem:**

## Source Code:

**package** lab;

**import** java.util.concurrent.Semaphore;

**class** ReadersWriters {

**static** Semaphore *readLock* = **new** Semaphore(1); **static** Semaphore *writeLock* = **new** Semaphore(1); **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();

} **catch** (InterruptedException e) {

System.***out***.println(e.getMessage());

}

}

}

**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());

}

}

}

**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();

}

}

# OUTPUT:

