**Documentation for Project**

Solution pseudocode:

Producer

WHILE (an empty slot) is true

synchronized (queue) {

While (queue size ==max size) is true

Wait until consumer consume an item

End While

Add next produced to the buffer

queue. Notify

}

ENDWHILE

1. first producer cheek if there is at least one empty slot.
2. Then it decrements the empty queue because, there will now be one less empty slot, since the producer is going to insert data in one of those slots.
3. then synchronized the buffer, so that the consumer cannot access the buffer until producer completes its operation
4. After performing the insert operation, the value of full is incremented because the producer has just filled a slot in the buffer.

5.finally waking up threads that are waiting for access this object

Consumer

Consumer

WHILE (one full slot) is true

synchronized (queue) {

While (queue size ==0) is true

Wait until producer produce an item

End While

remove an item from buffer

queue. Notify

}

ENDWHILE

1. first consumer cheek if there is at least one full slot in the buffer.

2. decrements the full queue because the number of occupied slots will be decreased by one, after the consumer completes its

operation

3. the consumer synchronized the buffer

4. the consumer completes the removal operation so that the data from one of the full slots is removed.

5. the empty semaphore is incremented by 1, because the consumer has just removed data from an occupied slot, thus making

it empty.

6.finally waking up threads that are waiting for access this object

What is a Deadlock ?

The term Deadlock refers to a situation in multithreading, where two or more threads are blocked forever, waiting for each other.

It is a simply a condition where a set of threads are blocked because each thread is holding a resource and waiting for another resource acquired by some other thread.

Since, all the threads are waiting for each other to release the resource, the condition is called Deadlock.

we use synchronized keyword to make the class or method thread-safe.The keyword ensures that only one thread can access the resource at a given point of time. A synchronized block is synchronized on some object. The block ensures that only one thread executes inside it at a time. All other threads attempting to enter it are blocked until the thread inside it exits.

If we use two synchronized block might cause deadlock

Example of deadlock:

If we created tow constant string variables resourse1, resourse2 to use them to produce or consume , in run function we created objects of two threads t1, t2 and use two synchronized block on resourse1, resourse2

For producer

Thread t1 = new Thread() {

synchronized ( resource 1 )

......

synchronized ( resource 2 )

{

For consumer

Thread t2 = new Thread() {

synchronized ( resource 2 )

................

synchronized ( resource 1 )

{

what is happening here –

1. Thread 1 starts and acquires lock on resource 1.

2. Thread 2 starts and acquires lock on resource 2.

3. Thread 1 prints "Thread 1: I have locked Resource 1" and Thread 2 prints "Thread 2: I have locked Resource 2 ".

4. Thread 1 tries to take object lock of resource 2 but it is already acquired by Thread 2 so it waits till it become free. It will not release lock of Resource 1 until it gets lock of Resource 2.

5. And same happens with thread 2. It tries to take object lock of resource 1 but it is already acquired by Thread 1 so it waits till it become free. It will not release lock of Resource 2 until it gets lock of Resource 1.

6. Now, both threads are in wait state, and are waiting for each other to release the locks.

7. As none of the thread is ready to release lock, so this is the Dead Lock condition.

8. And hence when we run the program, it seems as if the program execution has stopped.

How did solve deadlock:

To resolve this deadlock, we can simply change the order of acquiring the locks.

For producer

Thread t1 = new Thread() {

synchronized (resource 2)

......

synchronized (resource 1)

{

For consumer

Thread2 tries to lock resource2 and then resoursce1

Thread t2 = new Thread() {

synchronized ( resource 2 )

................

synchronized ( resource 1 )

we used just one synchronized because it will prevent deadlock

.

Examples of starvation:

What is a Starvation ?

Starvation of thread in java is said to occur when a particular thread does not get access to the object or the resource which leads to an increase in waiting and execution time.

Starvation is said to occur when two or more threads are allocated to the CPU (Central Processing Unit) and takes a lot of time in execution, due to which other waiting threads cannot get the CPU for its execution to carry on.

**Causes Of Starvation:**

There are many reasons for causes of starvation of threads in java, some of them are described below:

**-1High Priority Running Thread:** There may be a case where a high priority thread is running by occupying the CPU and it needs heavy processing which requires a lot of time in completion, so for this work to get completely executed the other threads which have a low priority order have to wait for a long time which leads to starvation.

**-2Synchronized Block:** There may be a case where the order in which the threads are allowed to enter the **synchronized block** is granted the resources in the same order as they are programmed to be scheduled, which results in waiting for the resources and the objects by another thread leading to starvation, where the other threads other than a particular thread are given the CPU for its execution.

Example of Starvation:

Class MyThread extends Thread {

Public void run() {

String threadName = Thread.currentThread().getName();

System.out.println(threadName + “ Started”);

Synchronized(MyThread.class) { // lock

// doing some useful work

Try {

Thread.sleep(2000); // 2 sec

} catch (InterruptedException ie){}

}

System.out.println(threadName + “ End”);

}

}

Public class Test {

Public static void main(String[] args) {

System.out.println(“Start of Main thread”);

MyThread mt[] = new MyThread[10];

For (int i=0; i<mt.length; i++) { Mt[i] = new MyThread(); // create thread

Mt[i].start();

}

System.out.println(“End of Main thread”);

In this example, the main thread created 10 child threads. To execute some portion of the run() method (synchronized block) each child thread needs the lock of the current class. At a time only one thread can get the lock of one object. And to complete execution, each thread required more than 2 seconds time.

Among these 10 threads, there will be some threads executing at last. They were waiting for a long period of time because the thread was unable to gain regular access to shared resources (lock of current class) and was unable to make progress. In the above output, thread-8 waited for a long period of time

**Another Example Of Starvation:**

class Starvation extends Thread {   
static int count = 1;   
public void run() {   
System.out.println(count + " Thread execution starts");   
System.out.println("Thread execution completes");   
count++;   
}   
public static void main(String[] args) throws InterruptedException {   
System.out.println("Parent thread execution starts");   
  
/\* Priority of each thread given. \*/  
/\* Thread 1 with priority 7. \*/  
Starvation thread1 = new Starvation();   
thread1.setPriority(7);   
/\* Thread 2 with priority 6. \*/  
Starvation thread2 = new Starvation();   
thread2.setPriority(6);   
/\* Thread 3 with priority 5. \*/  
Starvation thread3 = new Starvation();   
thread3.setPriority(5);  
/\* Thread 4 with priority 4. \*/  
Starvation thread4 = new Starvation();   
thread4.setPriority(4);   
/\* Thread 5 with priority 3. \*/  
Starvation thread5 = new Starvation();   
thread5.setPriority(3);   
  
thread1.run();   
thread2.run();   
thread3.run();   
thread4.run();   
thread5.run();   
  
System.out.println("Parent thread execution completes");   
}   
}

Here thread 5 have to wait because of the other threads

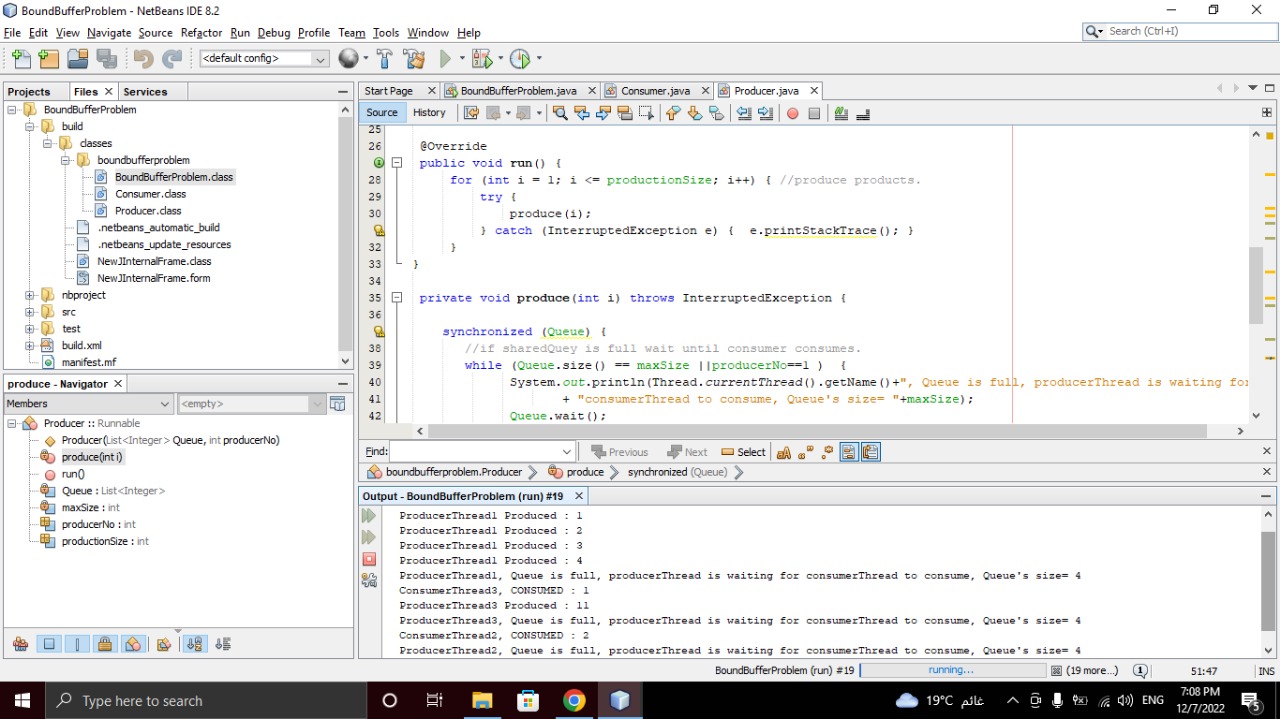
Here thread 5 have to wait because of the other threads have high. **Priority more than thread 5 .**

How did solve Stavation:

Some of the important points to remove starvation of threads are given as follows:

1-By implementation of the thread.yeid()  method, so that when the thread in the process after releasing the lock gets a fair chance to occupy the C.P.U. and can get some time to complete its execution till the original thread again gets the control over the C.P.U.

2-One can also use the thread.sleep()   method to given chance to other Threads for execution. (As We Used)

* Producer on certain condition is not returning produced buffer to ready buffer queue and continuing to wait for empty buffer to produce..
* Then eventually this kind of situation will lead to starvation.
* How did solve starvation:
* Consumer blocking till there is ready buffer to consume.
* Producer blocking till there is empty buffer to produce.
* Producer after producing immediately moving buffer to ready buffer queue.
* Consumer after consuming immediately moving buffer to empty buffer queue.

Explanation for real world application and how did apply the problem:

Real world application works like producer and consumer problem: Table restaurant

reservation

Bounded buffer problem (producer consumer problem): is one of the classic

problems of synchronization.

//Producer

The producer’s job is to generate data, put it into the buffer, and start again.

//Consumer

The consumer is consuming the data (i.e., removing it from the buffer), one piece at a

time.

//Problem

To make sure that the producer won’t try to add data into the buffer if it’s full and

that the consumer won’t try to remove data from an empty buffer.

Solution

The producer is to either go to sleep or discard data if the buffer is full. The next time

the consumer removes an item from the buffer, it notifies the producer, who starts to

fill the buffer again. In the same way, the consumer can go to sleep if it finds the

buffer to be empty. The next time the producer puts data into the buffer, it wakes up

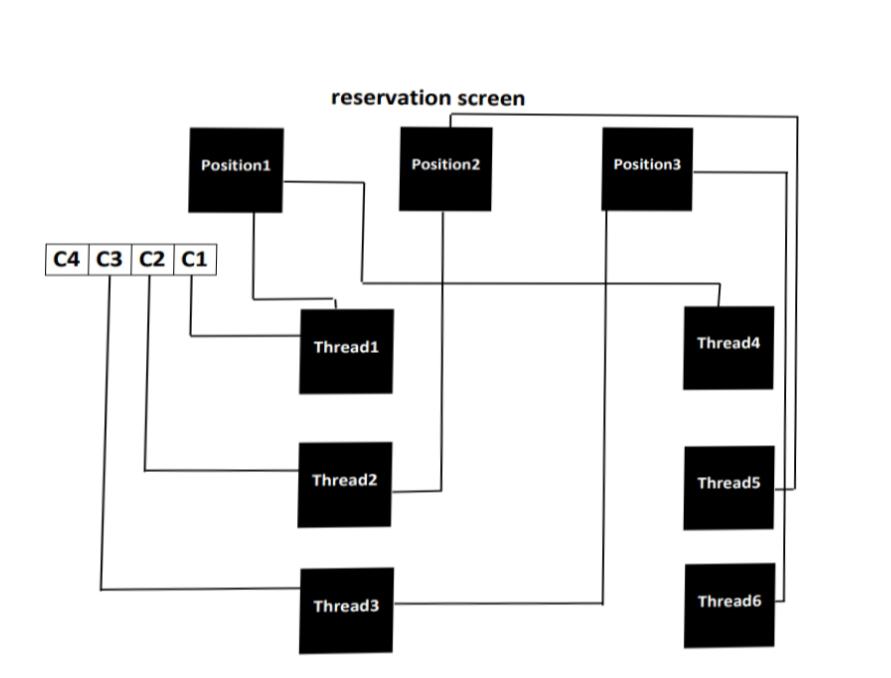
the sleeping consumer.

An inadequate solution could result in a deadlock where both processes are waiting

to be awakened.

\*We meet such a problem in our daily life, and we will discuss it in this example

**Tables restaurant reservation**



in this example:

Reservation Screen: Refer to the buffer (shared memory)

1-waiting line of customer: refer to waiting queue in the problem

-we have customer want to book tables

they are waiting in Line.

Customer: Refer to processes.

//Producer

-when First customer(c1) in waiting queue have to book table:

Employee1: work as thread 1

The first employee takes the first customer to the place of reservation

when Second customer in waiting queue have to book table:

Employee2: work as thread 2

The Second employee takes the first customer to the place of reservation

-when Third customer in waiting queue have to book table:

Employee3: work as thread 3

The Third employee takes the first customer to the place of reservation

//Consumer

-when employee1(thread1) produced customer1(process1) in the first available

position in buffer

Thread 4 consumed one of available position(decremented no of counter).

-when employee2(thread2) produced customer2(process2) in the first available

position in buffer

Thread 5 consumed one of available position(decremented no of counter).

-when employee6(thread6) produced customer3(process3) in the first available

position in buffer

Thread 6 consumed one of available position(decremented no of counter).