The bounded-buffer problem

Introduction:

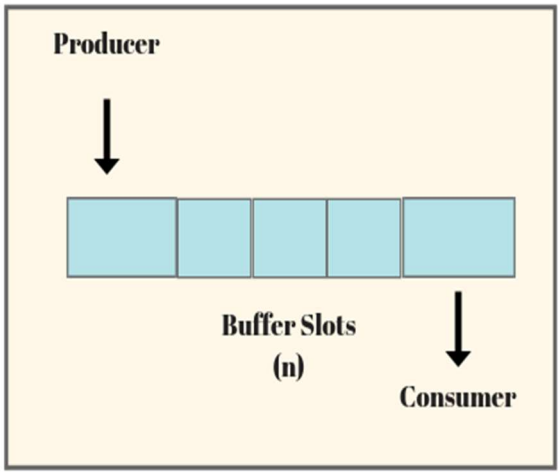
- The bounded-buffer problem which is also called (**producer consumer problem)**, It is problem based on synchronization
- This problem is generalized in terms of the Producer-Consumer problem

Description:

There is a buffer With capacity N can store data items, The places used to store the data items inside the bounded buffer are called slots(N), each slot is capable of storing one unit of data.

We have 2 Process (**Producer and consumer**) each of them operating on the buffer.

A producer tries to insert data into an empty slot of the buffer and consumer tries to remove data from a filled slot in the buffer



Without proper synchronization the following errors may occur:

- The producer tries to insert data when the buffer is full
- The consumer tries to consume date from empty slot
- The producer and consumer insert and remove data in the same time
- 2 producer writes into the same slot
- 2 consumer reads the same slot

Pseudocode (using wait() and notify):

Shared variables :-

```
private List<Integer> sharedQueue;
maxSize=4;
```

*Shared buffer has a limited size and can hold a num of items

Producer:

```
synchronized (sharedQueue) { // is used to lock an object for any shared resource
            while (sharedQueue is full){
             sharedQueue.wait(); // wait an action to happen
          }
sharedQueue add Item; // else add item
sharedQueue.notify(); // wake up another process
}
```

Consumer:

```
synchronized (sharedQueue) {
            while (sharedQueue is empty){
             sharedQueue.wait();
          }
sharedQueue remove Item;
sharedQueue.notify();
}
```

Pseudocode explanation:

Producer:

-producerThread will enter run method and call produce() method. There it will check for sharedQueue's size if size is equal to 4 (i.e. maximum number of products which sharedQueue can hold at a time), wait for consumer to consume using wait() method.

-if size is less than 4, producer will start producing and use notify() method to wake up another process

Consumer:

-consumerThread will enter run method and call consume() method. There it will check for sharedQueue's size.
-if size is equal to 0 that means producer hasn't produced any product, wait for producer to produce using wait() method.

- if size is greater than 0, consumer will start consuming and use notify() method

Notes:

- The synchronized keyword is used for exclusive accessing.
- wait() instructs the calling thread to shut down the monitor and sleep until another thread enters the monitor and calls notify().
- notify() wakes up the first thread that called wait() on the same object.
- When a thread invokes a synchronized method, it automatically acquires the lock for that object and releases it when the thread completes its task.

Deadlock:

Overview:

All the processes in a system require some resources such as central processing unit(CPU), file storage, input/output devices, etc to execute it. Once the execution is finished, the process

releases the resource it was holding. However, when many processes run on a system they also compete for these resources they require for execution. This may arise a deadlock situation.

A deadlock is a situation in which more than one process is blocked because it is holding a resource and also requires some resource that is acquired by some other process. Therefore, none of the processes gets executed.

A set of processes are waiting for each other in a circular fashion(**queue)**. For example, lets say there are a set of processes {producer0(p0), producer1(p1), consumer0(c0), consumer1(c1)} such as p0 depends on p1, p1 depends on c0, c0 depends on c1, c1 depends on p0.etc… This creates a circular relation between all these processes and they have to wait forever to be executed

---

If we remove **notify()** from either producer or consumer .

All process of consumer or producer is blocked and keep waiting for other to complete and none get executed

EX (Consumer):-

```
private void consume() throws InterruptedException

   synchronized (sharedQueue) {

      while (sharedQueue.size() == 0) {

         System.out.println(Thread.currentThread().getName()+", Queue is empty, consumerThread is waiting for "

                     + "producerThread to produce, sharedQueue's size= 0");

         sharedQueue.wait();

      //if sharedQuey is empty wait until producer produces.

       }


      Thread.sleep((long)(Math.random() * 2000));
```

```
        System.out.println(Thread.currentThread().getName()+", CONSUMED : "+
sharedQueue.remove(0));

        }

}
```

Output:-

```
ProducerThread0 Produced : 1

ProducerThread0 Produced : 2

ProducerThread0 Produced : 3

ProducerThread0 Produced : 4

ProducerThread0, Queue is full, producerThread is waiting for consumerThread to consume, sharedQueue's size= 4

ConsumerThread1, CONSUMED : 1

ConsumerThread1, CONSUMED : 2

ConsumerThread1, CONSUMED : 3

ConsumerThread1, CONSUMED : 4

ConsumerThread1, Queue is empty, consumerThread is waiting for producerThread to produce, sharedQueue's size= 0

ConsumerThread0, Queue is empty, consumerThread is waiting for producerThread to produce, sharedQueue's size= 0

ProducerThread1 Produced : 6

ProducerThread1 Produced : 7

ProducerThread1 Produced : 8

ProducerThread1 Produced : 9

ProducerThread1, Queue is full, producerThread is waiting for consumerThread to consume, sharedQueue's size= 4

ProducerThread0, Queue is full, producerThread is waiting for consumerThread to consume, sharedQueue's size= 4

ConsumerThread0, CONSUMED : 6

ConsumerThread0, CONSUMED : 7

ConsumerThread0, CONSUMED : 8

ConsumerThread0, CONSUMED : 9

ConsumerThread0, Queue is empty, consumerThread is waiting for producerThread to produce, sharedQueue's size= 0

ConsumerThread1, Queue is empty, consumerThread is waiting for producerThread to produce, sharedQueue's size= 0
```

as we see that the program stuck at waiting producer to produce but it cant because we removed notify() that can waking up the thread in waiting state

Solution: USE notify()

Starvation:

Overview:

Generally, Starvation occurs in Priority Scheduling or Shortest Job First Scheduling. In the Priority scheduling technique, we assign some priority to every process we have, and based on that priority, the CPU will be allocated, and the process will be executed. Here, the CPU will be allocated to the process that has the highest priority. Even if the burst time is low, the CPU will be allocated to the highest priority process Starvation is very bad for a process in an operating system, but we can overcome this starvation problem with the help of Aging.

Starvation or indefinite blocking is a phenomenon associated with the Priority scheduling algorithms, in which a process (producer or consumer) ready for the CPU (resources) can wait to run indefinitely because of low priority. In a heavily loaded computer system, a steady stream of higher-priority processes can prevent a low-priority process from ever getting the CPU.

If we remove **wait()** from either producer or consumer .

All process of consumer or producer keep executing and other processes are being blocked

EX (Producer):-

```
private void produce(int i) throws InterruptedException {


   synchronized (sharedQueue) {

     while (sharedQueue.size() == maxSize) {

         System.out.println(Thread.currentThread().getName()+", Queue is full,
producerThread is waiting for "

                 + "consumerThread to consume, sharedQueue's size= "+maxSize);


       }
```

```java
      int producedItem = (productionSize*producerNo)+ i;
      System.out.println(Thread.currentThread().getName() +" Produced : " +
producedItem);
      sharedQueue.add(producedItem);
       Thread.sleep((long)(Math.random() * 1000));
       sharedQueue.notify();
    }
 }
```

Output:-

```
ProducerThread0 Produced : 1
ProducerThread0 Produced : 2
ProducerThread0 Produced : 3
ProducerThread0 Produced : 4
ProducerThread0, Queue is full, producerThread is waiting for consumerThread to consume, sharedQueue's size= 4
ProducerThread0, Queue is full, producerThread is waiting for consumerThread to consume, sharedQueue's size= 4
ProducerThread0, Queue is full, producerThread is waiting for consumerThread to consume, sharedQueue's size= 4
ProducerThread0, Queue is full, producerThread is waiting for consumerThread to consume, sharedQueue's size= 4
ProducerThread0, Queue is full, producerThread is waiting for consumerThread to consume, sharedQueue's size= 4
ProducerThread0, Queue is full, producerThread is waiting for consumerThread to consume, sharedQueue's size= 4
ProducerThread0, Queue is full, producerThread is waiting for consumerThread to consume, sharedQueue's size= 4
ProducerThread0, Queue is full, producerThread is waiting for consumerThread to consume, sharedQueue's size= 4
ProducerThread0, Queue is full, producerThread is waiting for consumerThread to consume, sharedQueue's size= 4
ProducerThread0, Queue is full, producerThread is waiting for consumerThread to consume, sharedQueue's size= 4
ProducerThread0, Queue is full, producerThread is waiting for consumerThread to consume, sharedQueue's size= 4
ProducerThread0, Queue is full, producerThread is waiting for consumerThread to consume, sharedQueue's size= 4
ProducerThread0, Queue is full, producerThread is waiting for consumerThread to consume, sharedQueue's size= 4
ProducerThread0, Queue is full, producerThread is waiting for consumerThread to consume, sharedQueue's size= 4
ProducerThread0, Queue is full, producerThread is waiting for consumerThread to consume, sharedQueue's size= 4
ProducerThread0, Queue is full, producerThread is waiting for consumerThread to consume, sharedQueue's size= 4
```

ProducerThread0, Queue is full, producerThread is waiting for consumerThread to consume, sharedQueue's size= 4

ProducerThread0, Queue is full, producerThread is waiting for consumerThread to consume, sharedQueue's size= 4

ProducerThread0, Queue is full, producerThread is waiting for consumerThread to consume, sharedQueue's size= 4

ProducerThread0, Queue is full, producerThread is waiting for consumerThread to consume, sharedQueue's size= 4

ProducerThread0, Queue is full, producerThread is waiting for consumerThread to consume, sharedQueue's size= 4

ProducerThread0, Queue is full, producerThread is waiting for consumerThread to consume, sharedQueue's size= 4

ProducerThread0, Queue is full, producerThread is waiting for consumerThread to consume, sharedQueue's size= 4

ProducerThread0, Queue is full, producerThread is waiting for consumerThread to consume, sharedQueue's size= 4

ProducerThread0, Queue is full, producerThread is waiting for consumerThread to consume, sharedQueue's size= 4

.etc

as we see that the producer keep executing and the consumer being blocked

because we removed wait() that make the thread wait until the notify() method invoked

Solution: USE wait()

RealWorld Application:

•A factory that has an inventory which has a limited size contains num of Products, each product has a unique number.

•An importing Truck(producer) import products into inventory (buffer) that has a fixed size of slots n.

•when the inventory is full importing trucks cants import products and wait for exporting trucks to export products.

•when the inventory is empty exporting trucks cants export products and wait for importing trucks to import products.

•exporting truck cannot export the same product another truck exported