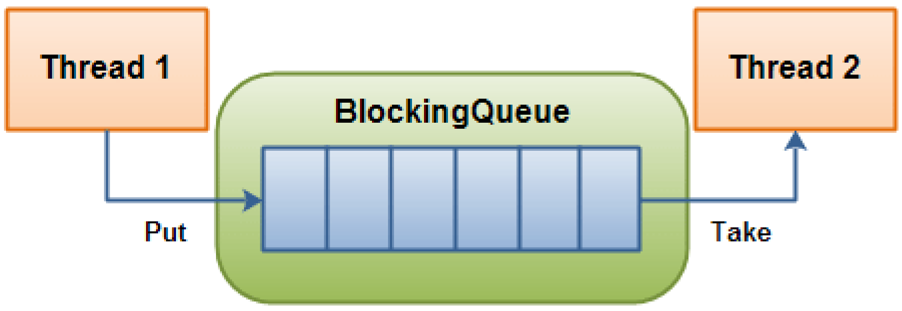
Producer Consumer Problem Using BlockingQueue

`BlockingQueue` is excellent when you want to skip the complexity involved in `wait–notify` statements. This `BlockingQueue` can be used to solve the producer-consumer problem as well as given below example. As this problem is well known to every programmer, I am not going into detail of the problem description.

How BlockingQueue fit into Solution

Any effective solution of producer consumer problem has to control the invocation of produce's put() method which generates the resource – and consumer's take() method which consumes the resource. Once you achieve this control of blocking the methods, you have solved the problem.

Java provides out of the box support to control such method invocations where one thread is creating resources and other is consuming them- through `BlockingQueue`. The Java `BlockingQueue` interface in the `java.util.concurrent` package represents a queue which is thread safe to put into, and take instances from.



BlockingQueue is a construct where one thread puts resources into it, and another thread takes from it.

This is exactly what is needed to solve the producer consumer problem. Let’s solve the problem now !!

Using BlockingQueue to solve Producer Consumer problem

Producer

Below code is for producer thread

```
class Producer implements Runnable
{
  protected BlockingQueue<Object> queue;

  Producer(BlockingQueue<Object> theQueue) {
    this.queue = theQueue;
  }

  public void run()
  {
    try
    {
      while (true)
      {
        Object justProduced = getResource();
        queue.put(justProduced);
        System.out.println("Produced resource - Queue size now =
"  + queue.size());
      }
    }
    catch (InterruptedException ex)
    {
      System.out.println("Producer INTERRUPTED");
    }
  }

  Object getResource()
  {
    try
    {
      Thread.sleep(100); // simulate time passing during read
    }
    catch (InterruptedException ex)
    {
      System.out.println("Producer Read INTERRUPTED");
    }
    return new Object();
  }
}
```
.

Here, producer thread creates a resource (i.e. Object) and put it in the queue. If the queue is already full (max size is 20); then it will wait – until the consumer thread pulls a resource out of it. So the queue size never goes beyond maximum i.e. 20.

Consumer

Below code is for consumer thread.

```java
class Consumer implements Runnable
{
  protected BlockingQueue<Object> queue;

  Consumer(BlockingQueue<Object> theQueue) {
    this.queue = theQueue;
  }

  public void run() {
    try
    {
      while (true)
      {
        Object obj = queue.take();
        System.out.println("Consumed resource - Queue size now =
"  + queue.size());
        take(obj);
      }
    }
    catch (InterruptedException ex)
    {
      System.out.println("CONSUMER INTERRUPTED");
    }
  }

  void take(Object obj)
  {
    try
    {
      Thread.sleep(100); // simulate time passing
    }
    catch (InterruptedException ex)
    {
      System.out.println("Consumer Read INTERRUPTED");
    }
    System.out.println("Consuming object " + obj);
  }
}
```

Consumer thread pulls a resource from the queue if it's there otherwise it will wait and then check again when the producer has put something into it.

Testing Producer Consumer Solution

Now let’s test out producer and consumer components written above.

```java
public class ProducerConsumerExample
{
  public static void main(String[] args) throws
InterruptedException
  {
    int numProducers = 4;
    int numConsumers = 3;

    BlockingQueue<Object> myQueue = new LinkedBlockingQueue<>(20);

    for (int i = 0; i < numProducers; i++){
      new Thread(new Producer(myQueue)).start();
    }

    for (int i = 0; i < numConsumers; i++){
      new Thread(new Consumer(myQueue)).start();
    }

    // Let the simulation run for, say, 10 seconds
    Thread.sleep(10 * 1000);

    // End of simulation - shut down gracefully
    System.exit(0);
  }
}
```

When you run the code, you find output similar to below:

```
Consumed resource - Queue size now = 1

Produced resource - Queue size now = 1

Consumed resource - Queue size now = 1
Consumed resource - Queue size now = 1

Produced resource - Queue size now = 1

Produced resource - Queue size now = 1

Produced resource - Queue size now = 1

Consuming object java.lang.Object@14c7f728

Consumed resource - Queue size now = 0

Consuming object java.lang.Object@2b71e323

Consumed resource - Queue size now = 0

Produced resource - Queue size now = 0

Produced resource - Queue size now = 1

Produced resource - Queue size now = 2

Consuming object java.lang.Object@206dc00b

Consumed resource - Queue size now = 1

Produced resource - Queue size now = 2

Produced resource - Queue size now = 3

Consuming object java.lang.Object@1a000bc0

Consumed resource - Queue size now = 2

Consuming object java.lang.Object@25b6183d
```

```
Consumed resource - Queue size now = 1

Produced resource - Queue size now = 2

Produced resource - Queue size now = 3

...

...

Produced resource - Queue size now = 20

Consuming object java.lang.Object@2b3cd3a6

Consumed resource - Queue size now = 19

Produced resource - Queue size now = 20

Consuming object java.lang.Object@3876982d

Consumed resource - Queue size now = 19

Produced resource - Queue size now = 20
```

IN CONCLUSION

Output clearly shows that queue size never grows beyond 20, and consumer threads are processing the queue resources put by producer threads. It’s very simple.