<http://tutorials.jenkov.com/java-concurrency/blocking-queues.html>

# Blocking Queues

A blocking queue is a queue that blocks when you try to dequeue from it and the queue is empty, or if you try to enqueue items to it and the queue is already full. A thread trying to dequeue from an empty queue is blocked until some other thread inserts an item into the queue. A thread trying to enqueue an item in a full queue is blocked until some other thread makes space in the queue, either by dequeuing one or more items or clearing the queue completely.

Here is a diagram showing two threads cooperating via a blocking queue:

|  |
| --- |
| A BlockingQueue with one thread putting into it, and another thread taking from it. |
| **A BlockingQueue with one thread putting into it, and another thread taking from it.** |

Java 5 comes with blocking queue implementations in the java.util.concurrent package. You can read about that class in my [**java.util.concurrent.BlockingQueue**](http://tutorials.jenkov.com/java-util-concurrent/blockingqueue.html) tutorial. Even if Java 5 comes with a blocking queue implementation, it can be useful to know the theory behind their implementation.

## Blocking Queue Implementation

The implementation of a blocking queue looks similar to a [**Bounded Semaphore**](http://tutorials.jenkov.com/java-concurrency/semaphores.html#bounded). Here is a simple implementation of a blocking queue:

public class BlockingQueue {

private List queue = new LinkedList();

private int limit = 10;

public BlockingQueue(int limit){

this.limit = limit;

}

public synchronized void enqueue(Object item)

throws InterruptedException {

while(this.queue.size() == this.limit) {

wait();

}

this.queue.add(item);

if(this.queue.size() == 1) {

notifyAll();

}

}

public synchronized Object dequeue()

throws InterruptedException{

while(this.queue.size() == 0){

wait();

}

if(this.queue.size() == this.limit){

notifyAll();

}

return this.queue.remove(0);

}

}

Notice how notifyAll() is only called from enqueue() and dequeue() if the queue size is equal to the size bounds (0 or limit). If the queue size is not equal to either bound when enqueue() or dequeue() is called, there can be no threads waiting to either enqueue or dequeue items

<https://www.javacodemonk.com/blocking-queue-implementation-in-java-044ee033>

# Blocking Queue implementation in Java

A blocking queue allows multiple threads to communicate with each other and pass data around. For example, a producer can put items to the queue while consumer can take items out from the queue.

|  |  |
| --- | --- |
|  | BlockingQueue implementations are designed to be used primarily for producer-consumer queues. |

A blocking queue has below characteristics:

1. It is always thread-safe
2. It can hold arbitrary data
3. Producer has to wait if the queue is already full, similarly consumer has to be wait if no item is present in the queue.

**Java implementation**

A trivial implementation of BlockingQueue using *intrinsic locking* using synchronized keyword will look like the following:

BlockingQueue Implementation using synchronization

class SimpleBlockingQueue {

final Object[] items = new Object[100];

int putptr, takeptr, count;

public synchronized void put(Object x) throws InterruptedException {

while (count == items.length)

wait();

items[putptr] = x;

if (++putptr == items.length) putptr = 0;

++count;

notifyAll();

}

public synchronized Object take() throws InterruptedException {

while (count == 0)

wait();

Object x = items[takeptr];

if (++takeptr == items.length) takeptr = 0;

--count;

notifyAll();

return x;

}

}

|  |  |
| --- | --- |
|  | 1. Max capacity of blocking queue is 100 |
|  | 1. We are waiting inside a while loop while queue capacity is full. While loop is required to avoid spurious wakeups. |
|  | 1. All other waiting threads are notified as soon as a new item is added to the queue. |
|  | 1. Consumer thread waits inside a while loop for arrival of new item, if queue is empty. while loop prevents spurious wakeup problem. |
|  | 1. Consumer thread notifies all waiting producer threads as soon as an item is removed from the queue. |

A realistic implementation would have much more methods (peek(), remove(), offer(), isFull(), etc.) to the queue, but for brevity we have implemented only two methods.

**A slightly improved version**

Improved version of Queue will use explicit locking, to improve the multi-threading performance. *Lock* and *Condition* interface provides much better flexibility compared to intrinsic locking mechanism, but this flexibility brings more responsibility as we have to take care of calling lock and unlock ourselves. Since one lock can be associated with multiple conditions (notFull & notEmpty in this case), this results in better throughput due to lesser thread contention.

|  |  |
| --- | --- |
|  | Lock is analogical to synchronized keyword and Condition is similar to wait/notify. |

SimpleBlockingQueue using Lock and Condition

class SimpleBlockingQueue {

final Lock lock = new ReentrantLock();

final Condition notFull = lock.newCondition();

final Condition notEmpty = lock.newCondition();

final Object[] items = new Object[100];

int putptr, takeptr, count;

public void put(Object x) throws InterruptedException {

lock.lock();

try {

while (count == items.length)

notFull.await();

items[putptr] = x;

if (++putptr == items.length) putptr = 0;

++count;

notEmpty.signal();

} finally {

lock.unlock();

}

}

public Object take() throws InterruptedException {

lock.lock();

try {

while (count == 0)

notEmpty.await();

Object x = items[takeptr];

if (++takeptr == items.length) takeptr = 0;

--count;

notFull.signal();

return x;

} finally {

lock.unlock();

}

}

}

Here we are using ReentrantLock along with Condition to explicitly define the lock. This reduces thread contention under heavy load circumstances.

We can do few further improvements to the above version, for example we can use LinkedList instead of array and optionally make it generic to support any item type.

Generic BlockingQueue using LinkedList under the hood

import java.util.LinkedList;

import java.util.Queue;

import java.util.concurrent.locks.Condition;

import java.util.concurrent.locks.Lock;

import java.util.concurrent.locks.ReentrantLock;

public class BlockingQueue<T> {

private Queue<T> queue = new LinkedList<T>();

private int capacity;

private Lock lock = new ReentrantLock();

private Condition notFull = lock.newCondition();

private Condition notEmpty = lock.newCondition();

public BlockingQueue(int capacity) {

this.capacity = capacity;

}

public void put(T element) throws InterruptedException {

lock.lock();

try {

while (queue.size() == capacity) {

notFull.await();

}

queue.add(element);

notEmpty.signal();

} finally {

lock.unlock();

}

}

public T take() throws InterruptedException {

lock.lock();

try {

while (queue.isEmpty()) {

notEmpty.await();

}

T item = queue.remove();

notFull.signal();

return item;

} finally {

lock.unlock();

}

}

}

Please be noted that all above implementations are for learning purpose only, not for production use. For production, you shall prefer to use *LinkedBlockingQueue* or any other equivalent implementation from java.util.concurrent package.

That’s all.