# The Evolution of the Producer-Consumer Problem in Java

**Problem Statement**

There are two processes, a producer and a consumer, that share a common buffer with a limited size. The producer “produces” data and stores it in the buffer, and the consumer “consumes” the data, removing it from the buffer. Having two processes that run in parallel, we need to make sure that the producer will not put new data in the buffer when the buffer is full and the consumer won’t try to remove data from the buffer if the buffer is empty.

**Solution**

For solving this concurrency problem, the producer and the consumer will have to communicate with each other. If the buffer is full, the producer will go to sleep and will wait to be notified. After the consumer will remove some data from the buffer, it will notify the producer, and then, the producer will start refilling the buffer again. The same process will happen if the buffer is empty, but in this case, the consumer will wait to be notified by the producer.

If this communication is not done properly, it can lead to a deadlock where both processes will wait for each other.

**Classic Approach**

Let’s see a typical Java solution to this problem.

1

package ProducerConsumer;

2

​

3

import java.util.LinkedList;

4

import java.util.Queue;

5

​

6

public class ClassicProducerConsumerExample {

7

​

8

public static void main(String[] args) throws InterruptedException {

9

Buffer buffer = new Buffer(2);

10

​

11

Thread producerThread = new Thread(new Runnable() {

12

@Override

13

public void run() {

14

try {

15

buffer.produce();

16

} catch (InterruptedException e) {

17

e.printStackTrace();

18

}

19

}

20

});

21

​

22

Thread consumerThread = new Thread(new Runnable() {

23

@Override

24

public void run() {

25

try {

26

buffer.consume();

27

} catch (InterruptedException e) {

28

e.printStackTrace();

29

}

30

}

31

});

32

​

33

producerThread.start();

34

consumerThread.start();

35

​

36

producerThread.join();

37

consumerThread.join();

38

}

39

​

40

static class Buffer {

41

​

42

private Queue<Integer> list;

43

private int size;

44

​

45

public Buffer(int size) {

46

this.list = new LinkedList<>();

47

this.size = size;

48

}

49

​

50

public void produce() throws InterruptedException {

51

int value = 0;

52

while (true) {

53

synchronized (this) {

54

while (list.size() >= size) {

55

// wait for the consumer

56

wait();

57

}

58

​

59

list.add(value);

60

​

61

System.out.println("Produced " + value);

62

​

63

value++;

64

​

65

// notify the consumer

66

notify();

67

​

68

Thread.sleep(1000);

69

}

70

}

71

}

72

​

73

public void consume() throws InterruptedException {

74

while (true) {

75

synchronized (this) {

76

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

77

// wait for the producer

78

wait();

79

}

80

​

81

int value = list.poll();

82

​

83

System.out.println("Consume " + value);

84

​

85

// notify the producer

86

notify();

87

​

88

Thread.sleep(1000);

89

}

90

}

91

}

92

}

93

}

Here we have two threads, a producer and a consumer thread, which share a common buffer. The producer thread starts producing new elements and stores them in the buffer. If the buffer is full, it goes to sleep and will wait to be notified. Otherwise, it will put a new element in the buffer and notify the consumer. Like I said before, the same process applies to the consumer. If the buffer is empty, the consumer will wait to be notified by the producer. Otherwise, it will remove an element from the buffer and it will notify the consumer.

As you can see, in the previous example, both jobs are managed by the buffer object. The threads are just calling  buffer.produce()  and buffer.consume(),  and everything is done by these two methods.

This is a debatable subject, but in my opinion, the buffer shouldn’t be responsible for creating or removing the elements. Of course, that depends on what you want to achieve, but in this case, the buffer should be responsible just for storing and pooling elements in a thread-safe manner, not for producing the elements.

So, let’s move the produce and consume logic out of the buffer object.

1

package ProducerConsumer;

2

​

3

import java.util.LinkedList;

4

import java.util.Queue;

5

​

6

public class ProducerConsumerExample2 {

7

​

8

public static void main(String[] args) throws InterruptedException {

9

Buffer buffer = new Buffer(2);

10

​

11

Thread producerThread = new Thread(() -> {

12

try {

13

int value = 0;

14

while (true) {

15

buffer.add(value);

16

​

17

System.out.println("Produced " + value);

18

​

19

value ++;

20

​

21

Thread.sleep(1000);

22

}

23

} catch (InterruptedException e) {

24

e.printStackTrace();

25

}

26

});

27

​

28

Thread consumerThread = new Thread(() -> {

29

try {

30

while (true) {

31

int value = buffer.poll();

32

​

33

System.out.println("Consume " + value);

34

​

35

Thread.sleep(1000);

36

}

37

​

38

} catch (InterruptedException e) {

39

e.printStackTrace();

40

}

41

});

42

​

43

producerThread.start();

44

consumerThread.start();

45

​

46

producerThread.join();

47

consumerThread.join();

48

}

49

​

50

static class Buffer {

51

​

52

private Queue<Integer> list;

53

private int size;

54

​

55

public Buffer(int size) {

56

this.list = new LinkedList<>();

57

this.size = size;

58

}

59

​

60

public void add(int value) throws InterruptedException {

61

synchronized (this) {

62

while (list.size() >= size) {

63

wait();

64

}

65

list.add(value);

66

notify();

67

}

68

}

69

​

70

public int poll() throws InterruptedException {

71

synchronized (this) {

72

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

73

wait();

74

}

75

​

76

int value = list.poll();

77

notify();

78

return value;

79

}

80

}

81

}

82

}

That’s better. Now, the buffer is responsible for storing and removing the elements in a thread-safe manner.

**Blocking Queue**

However, we can further improve this. In the previous example, we’ve created a buffer that, when storing an element, waits for a slot to become available in case there is no more space, and, on pooling, in case that the buffer is empty, it waits for an element to become available, making the storing and removing operations thread-safe.

But, Java already has a collection for this. It’s called a BlockingQueue and, as it is described [here](http://tutorials.jenkov.com/java-util-concurrent/blockingqueue.html), this is a queue that is thread-safe to put into and take instances from. It does exactly what we want. So, if we use a BlockingQueue in our example, we don’t have to implement the waiting and notifying mechanism.

Let’s see how it looks.

1

package ProducerConsumer;

2

​

3

import java.util.concurrent.BlockingQueue;

4

import java.util.concurrent.LinkedBlockingDeque;

5

​

6

public class ProducerConsumerWithBlockingQueue {

7

​

8

public static void main(String[] args) throws InterruptedException {

9

BlockingQueue<Integer> blockingQueue = new LinkedBlockingDeque<>(2);

10

​

11

Thread producerThread = new Thread(() -> {

12

try {

13

int value = 0;

14

while (true) {

15

blockingQueue.put(value);

16

​

17

System.out.println("Produced " + value);

18

​

19

value++;

20

​

21

Thread.sleep(1000);

22

}

23

} catch (InterruptedException e) {

24

e.printStackTrace();

25

}

26

});

27

​

28

Thread consumerThread = new Thread(() -> {

29

try {

30

while (true) {

31

int value = blockingQueue.take();

32

​

33

System.out.println("Consume " + value);

34

​

35

Thread.sleep(1000);

36

}

37

​

38

} catch (InterruptedException e) {

39

e.printStackTrace();

40

}

41

});

42

​

43

producerThread.start();

44

consumerThread.start();

45

​

46

producerThread.join();

47

consumerThread.join();

48

}

49

}

The runnables look exactly as they did before. They produce and consume elements in the same way. The only difference is that here we use a blockingQueue instead of our buffer object.

**Some Details About the Blocking Queue**

There are two types of [BlockingQueue](https://www.baeldung.com/java-blocking-queue" \t "_blank):

* Unbounded queue
* Bounded queue

An unbounded queue can grow almost indefinitely, and the add operations are not blocking. You can create an unbounded queue like this:

1

BlockingQueue<String> blockingQueue = new LinkedBlockingDeque<>();

In this case, since the add operations are not blocking, the producer doesn’t have to wait when adding new elements. Every time when the producer wants to add a new element, the queue will store it. But, there is catch here. If the consumer doesn’t remove elements faster than the producer is adding new elements, then the memory will fill up and we will get an OutOfMemory exception.

The bounded queue, instead, has a fixed size. You can create one like this:

1

BlockingQueue<String> blockingQueue = new LinkedBlockingDeque<>(10);

The main difference is that using a bounded queue, if the queue is full and the producer tries to store another element, depending on what method is used for adding, the queue will block until it will be enough space.

There are four methods for adding elements in a blocking queue:

* add() – returns *true*if the insertion was successful, otherwise, it will throw an  IllegalStateException
* put() – inserts an element into the queue and waits for a free slot if necessary
* offer()  – returns *true*if the insertion was successful, otherwise, it returns *false*
* offer(E e, long timeout, TimeUnit unit) – inserts an element into the queue if it is not full, or waits for an available slot within a specified timeout

So, if you use the put() method and the queue is full, the producer will have to wait until there is a free slot. That’s what we used in the previous example, and this will work in the same way as ProducerConsumerExample2.

**Using a Thread Pool**

What else can we improve here? Let’s analyze what we did. We’ve instantiated two threads, one that puts some elements in the blocking queue, the producer, and another that retrieves elements from the queue, the consumer.

But, good software techniques suggest that creating and destroying threads manually is bad practice. Thread creation is an expensive task. Each thread creation implies the following steps:

* It allocates memory for a thread stack
* The OS creates a native thread corresponding to the Java thread
* Descriptors relating to the thread are added to the JVM internal data structures

Don’t get me wrong. There is nothing wrong with having more threads. That’s how parallelism works. The problem here is that we’ve created them “manually." That’s the bad practice. If we create them manually, besides the creation’s cost, another problem is that we don’t have control over how many of them are running at the same time. For example, if millions of requests are reaching a server app, and for each request, a new thread is created, then millions of threads will run in parallel and this could lead to a [thread starvation](https://en.wikipedia.org/wiki/Starvation_(computer_science)).

So, we need a way to strategically manage threads. And here comes the thread pools.

Thread pools handle the threads' life cycle based on a selected strategy. It holds a limited number of idle threads and reuses them when it needs to solve tasks. This way, we don’t have to create a new thread every time for a new request, and therefore, we can avoid reaching a thread starvation,

The Java thread pool implementation consists of:

* A task queue
* A collection of worker threads
* A thread factory
* Metadata for managing thread pool state.

For running some tasks concurrently, you have to put them in the task queue. Then, when a thread is available, it will receive a task and run it. The more available threads, the more tasks that are executed in parallel.

Beside the thread lifecycle management, another advantage when working with a thread pool is that when you plan on how to split the work to be executed concurrently, you can think in a more functional way. The unit of parallelism is not the thread anymore; it’s the task. You design some tasks that are executed concurrently, and not some threads that share a common memory and are running in parallel. Thinking in a functional way can help us avoid some common multithreading problems, like deadlocks or data races. Nothing can stop us from reaching into these problems again, but, because using the functional paradigm, we don’t imperatively synchronize the concurrent computation (with locks). This is far less happening than working directly with threads and shared memory. This is not the case in our example since the tasks share a blocking queue, but I just wanted to highlight this advantage.

[Here](https://allegro.tech/2015/05/thread-pools.html)and [here](https://dzone.com/articles/getting-the-most-out-of-the-java-thread-pool) you can find more details about thread pools.

With all of this being said, let’s see how our example looks using a thread pool.

1

package ProducerConsumer;

2

​

3

import java.util.concurrent.BlockingQueue;

4

import java.util.concurrent.ExecutorService;

5

import java.util.concurrent.Executors;

6

import java.util.concurrent.LinkedBlockingDeque;

7

​

8

public class ProducerConsumerExecutorService {

9

​

10

public static void main(String[] args) {

11

BlockingQueue<Integer> blockingQueue = new LinkedBlockingDeque<>(2);

12

ExecutorService executor = Executors.newFixedThreadPool(2);

13

​

14

Runnable producerTask = () -> {

15

try {

16

int value = 0;

17

while (true) {

18

blockingQueue.put(value);

19

​

20

System.out.println("Produced " + value);

21

​

22

value++;

23

​

24

Thread.sleep(1000);

25

}

26

} catch (InterruptedException e) {

27

e.printStackTrace();

28

}

29

};

30

​

31

Runnable consumerTask = () -> {

32

try {

33

while (true) {

34

int value = blockingQueue.take();

35

​

36

System.out.println("Consume " + value);

37

​

38

Thread.sleep(1000);

39

}

40

​

41

} catch (InterruptedException e) {

42

e.printStackTrace();

43

}

44

};

45

​

46

executor.execute(producerTask);

47

executor.execute(consumerTask);

48

​

49

executor.shutdown();

50

}

51

}

The difference here is that, instead of manually creating and running the consumer and producer threads, we build a thread pool, and it will receive two tasks, a producer and a consumer task. The producer and consumer tasks are actually the same runnables that were used in the previous example. Now, the executor (the thread pool implementation) receives the tasks, and its working threads will execute them.

In our simple case, everything will work the same as before. Just like in previous examples, we still have two threads, and they still produce and consume elements in the same way. So, we don’t have a performance improvement here, but the code looks cleaner. We no longer build the threads manually, but, instead, we just specify what we want. And, we want a way to execute some tasks in parallel.

So, when you use a thread pool, you don’t have to think to threads as the unit of parallelism, but instead, you think to some tasks that are executed concurrently. That’s what you need to know, and the executor will handle the rest. It will receive some tasks, and then, it will execute them using the available working threads.

**Summary**

First, we saw the “traditional” solution of a consumer-producer problem. We try to not reinvent the wheel when is not necessary, but instead, we want to reuse already tested solutions. So, instead of writing down a wait/notify system, why not use the Java blocking queue that already offers that? And also, we can get rid of manually creating threads when Java provides us with a thread pool that manages thread lifecycle very efficiently already. With these improvements, the solutions of the consumer-producer problem look more reliable and understandable.