**Producer-Consumer**

**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.

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

**package com.producerConsumer;**

**import java.util.LinkedList;**

**import java.util.Queue;**

**public class ClassicProducerConsumerExample {**

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

**Buffer buffer = new Buffer(2);**

**Thread producerThread = new Thread(new Runnable() {**

**@Override**

**public void run() {**

**try {**

**buffer.produce();**

**} catch (InterruptedException e) {**

**e.printStackTrace();**

**}**

**}**

**});**

**Thread consumerThread = new Thread(new Runnable() {**

**@Override**

**public void run() {**

**try {**

**buffer.consume();**

**} catch (InterruptedException e) {**

**e.printStackTrace();**

**}**

**}**

**});**

**producerThread.start();**

**consumerThread.start();**

**producerThread.join();**

**consumerThread.join();**

**}**

**static class Buffer {**

**private Queue<Integer> list;**

**private int size;**

**public Buffer(int size) {**

**this.list = new LinkedList<>();**

**this.size = size;**

**}**

**public void produce() throws InterruptedException {**

**int value = 0;**

**while (true) {**

**synchronized (this) {**

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

**// wait for the consumer**

**wait();**

**}**

**list.add(value);**

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

**value++;**

**// notify the consumer**

**notify();**

**Thread.sleep(1000);**

**}**

**}**

**}**

**public void consume() throws InterruptedException {**

**while (true) {**

**synchronized (this) {**

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

**// wait for the producer**

**wait();**

**}**

**int value = list.poll();**

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

**// notify the producer**

**notify();**

**Thread.sleep(1000);**

**}**

**}**

**}**

**}**

**}**

**Exampe 2: Using Blocking queue.**

**package** com.producerConsumer;

**import** java.util.concurrent.BlockingQueue;

**import** java.util.concurrent.LinkedBlockingDeque;

**public** **class** ProducerConsumerUsingBlockedQueue {

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

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

Thread producerThread = **new** Thread(() -> {

**try** {

**int** value = 0;

**while** (**true**) {

blockingQueue.put(value);

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

value++;

Thread.*sleep*(1000);

}

} **catch** (InterruptedException e) {

e.printStackTrace();

}

});

Thread consumerThread = **new** Thread(() -> {

**try** {

**while** (**true**) {

**int** value = blockingQueue.take();

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

Thread.*sleep*(1000);

}

} **catch** (InterruptedException e) {

e.printStackTrace();

}

});

producerThread.start();

consumerThread.start();

producerThread.join();

consumerThread.join();

}

}

**Example 3:**

## 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 come 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.

package ProducerConsumer;

import java.util.concurrent.BlockingQueue;

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

import java.util.concurrent.LinkedBlockingDeque;

public class ProducerConsumerExecutorService {

public static void main(String[] args) {

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

ExecutorService executor = Executors.newFixedThreadPool(2);

Runnable producerTask = () -> {

try {

int value = 0;

while (true) {

blockingQueue.put(value);

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

value++;

Thread.sleep(1000);

}

} catch (InterruptedException e) {

e.printStackTrace();

}

};

Runnable consumerTask = () -> {

try {

while (true) {

int value = blockingQueue.take();

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

Thread.sleep(1000);

}

} catch (InterruptedException e) {

e.printStackTrace();

}

};

executor.execute(producerTask);

executor.execute(consumerTask);

executor.shutdown();

}

}

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.