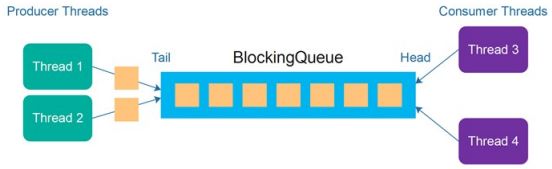
Source: https://jenkov.com/tutorials/java-concurrency/producer-consumer.html

The Producer Consumer Patter

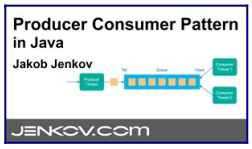
Jakob Jenkov
Last update: 2021-04-19

The producer consumer pattern is a concurrency design pattern where one or more producer threads produce objects which are queued up, and then consumed by one or more consumer threads. The objects enqueued often represent some work that needs to be done. Decoupling the detection of work from the execution of work means you can control how many threads at a time that are engaged in detecting or executing the work.



Producer Consumer Tutorial Video

If you prefer video, I have a video version of this producer consumer pattern tutorial here:
The Producer Consumer Pattern - in Java.



Use Cases

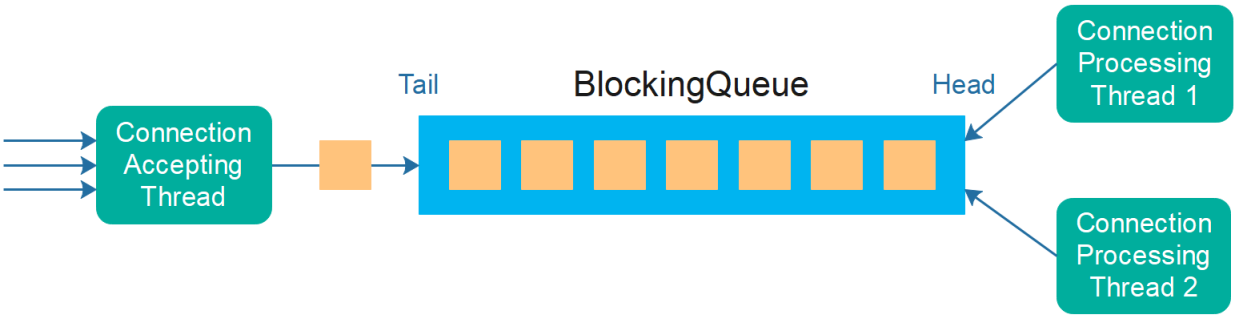
There are several different kinds of use cases for the producer consumer design pattern. Some of the most common are:

- Reduce foreground thread latency.
- Load balance work between different threads.
- Backpressure management.

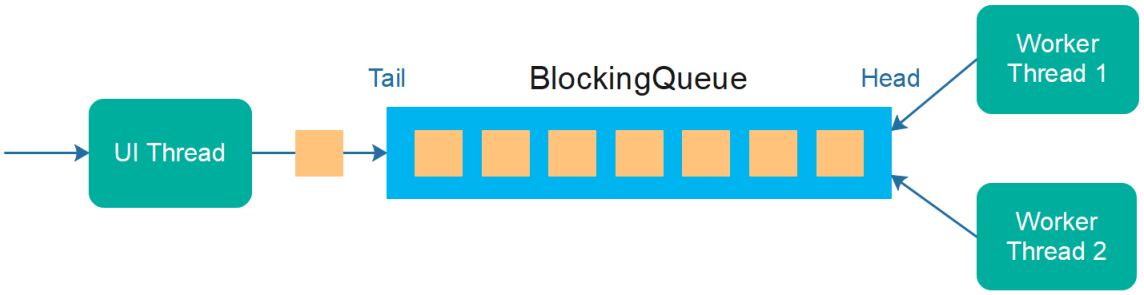
Reduce Foreground Thread Latency

In some systems you have a single foreground thread which has the communication with the outside world. In a server it could be the thread accepting the incoming connections from clients. In a desktop app that could be the UI thread.

In order to not make the foreground thread busy with tasks - whatever tasks the foreground thread receives from the outside world are offloaded to one or more background threads. In a server it could be processing the data that is received via the inbound connections.



In a desktop app the foreground thread (UI thread) could be responding to the user events - a.g. opening a file, or downloading a file, or saving a file etc. To avoid having the UI thread blocked so the whole GUI is unresponsive, the UI thread can offload long-running tasks to background worker threads.



Load Balance Work Between Threads

Another type of use case for the producer consumer pattern is to load balance work between a set of worker threads. Actually, this load balancing happens pretty much automatically, as long as the worker threads take new task objects from the queue as soon as they have time to process them. This will result in load balancing the tasks between the worker threads.

Backpressure Management

If the queue between the producer and consumer threads is a **Java BlockingQueue**, then you can use the queue for backpressure management. This is another built-in feature of the producer consumer pattern.

Backpressure means, that if the producer thread(s) produce more work than the consumer threads are able to handle - the tasks will queue up in the queue. At some point the BlockingQueue will become full, and the producer threads will be blocked trying to insert new tasks / work objects into the queue. This phenomenon is called backpressure. The system presses back against the producers - towards the beginning of the "pipeline" - preventing more work from coming in.

The backpressure will "spill out" of the queue, and slow down the producer thread(s). Thus, they too could propagate the pressure back up the work pipeline, if there are any earlier steps in the total pipeline.