SSC - Concurrency and Multi-threading
Producer Consumer Design Pattern and Thread Coordination

Shan He

School for Computational Science
University of Birmingham

Module 06-19321: SSC

Outline of Topics

Producer consumer problem

Producer consumer implementations

Blocking queue and Producer Consumer model

Summary

Producer Consumer problem

- Producer Consumer problem: also known as bounded-buffer problem)
  - Two threads: the producer and the consumer
  - A shared buffer: a fixed-size queue.
- The producer: generating a piece of data, putting it into the buffer and start again.
- The consumer: removing the data continuously from the buffer one piece at a time
- **Requirements**:
  - the producer won't try to add data into the buffer if it's full
  - the consumer won't try to remove data from an empty buffer.
- Everyday examples everywhere: rotating sushi bar

Producer Consumer problem: solutions

- Three situations:
  - The buffer is full: the producer stops producing, i.e., sleep
  - The buffer is empty: the consumer stops removing, i.e., sleep
  - The buffer is neither full or empty: the producer and the consumer continue working or notify the sleeping producer/consumer to resume
- **Key principle**: Synchronisation is required for the **shared buffer** to avoid thread safety problem, e.g., thread interference problem which might cause deadlock
- Deadlock: both threads are waiting to be awakened by the other.
- Once you have found a good solution, it becomes a design pattern: **Producer Consumer Design Pattern**

Producer Consumer design pattern

- Producer Consumer design pattern: a classic concurrency or threading programming design pattern
- Usages:
    - to separate work that needs to be done from the execution of that work.
    - to decouple threads that produce and consume data in different rates
- Example: application accepts data while processing them in the order they were received.
    - Producer: Producing the data, e.g., queueing up the received data in order - fast
    - Consumer: Consuming the data, e.g., processing the data - slow

Producer Consumer design pattern: Guarded block

- Shared buffer: we use `Queue` interface in `java.util` package to implement a queue
- **Key principle**: Synchronisation is required for the **shared buffer** to avoid thread safety problem, e.g., thread interference problem which might cause deadlock
- Synchronisation: We will use Synchronized keyword
- Thread coordination is required for the **bounded queue**:
    - Guarded block, e.g., wait for a particular condition to become true and only in that case the actual execution of the thread resumes
    - `wait()/notifyAll()`
- Java example

Producer Consumer design pattern: Semaphore

- We need to use two Semaphores:
  - `prodSemaphore` : the number of available spaces in the buffer where the producer can put in
  - `consSemaphore` : is the number of items already in the buffer and available for the consumer to get
- Producer put a new item into the buffer: increases `consSemaphore` by `release()` , and decreases `prodSemaphore` by `acquire()`
- Consumer get a item from the buffer: decreases `consSemaphore` by `acquire()` , and increases `prodSemaphore` by `release()`
- Question: What are the initial values of the two Semaphores?

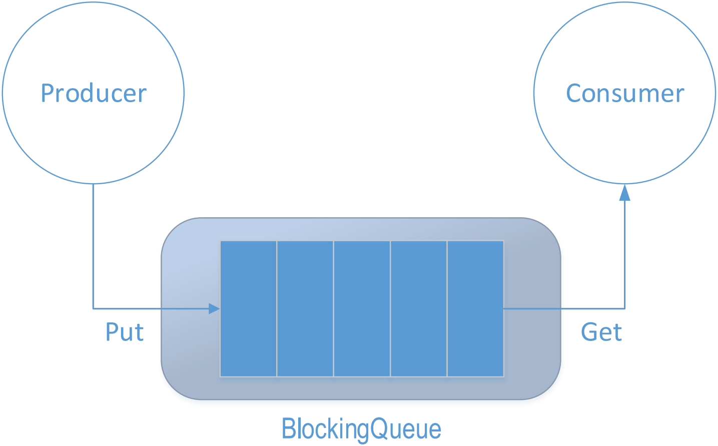
`java.util.concurrent` and Blocking queue

- Utility classes commonly useful in concurrent programming
- Provides the following classes:
  - Executors: a simple standardized interface for defining custom thread-like subsystems
  - **Queues**: thread-safe non-blocking FIFO queue.
  - Timing: multiple granularities (including nanoseconds) for specifying and controlling time-out based operations.
  - Synchronizers: special-purpose coordination (synchronization) idioms such as Semaphore, CountDownLatch and CyclicBarrier
  - **Concurrent Collections**: collections such as Hashmap and ListMap in multithreaded contexts, e.g., `ConcurrentHashMap` and `ConcurrentSkipListMap` .

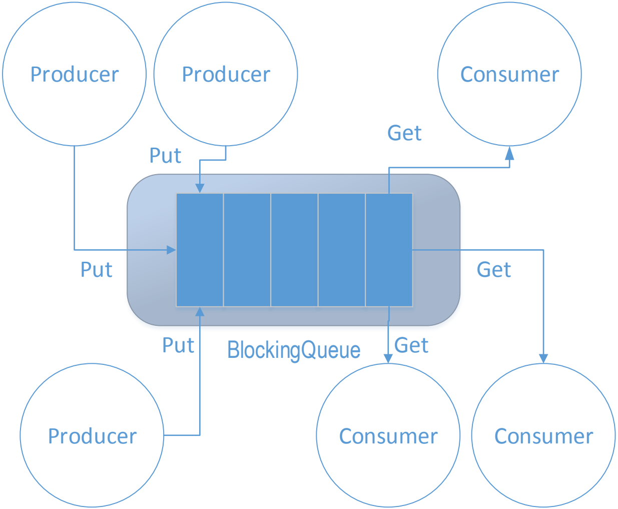
Blocking queue

- `BlockingQueue` : an interface in the `java.util.concurrent` class represents a queue which is thread safe to put into, and take instances from.
- Designed for Producer Consumer model: "*FIFO data structure that blocks or times out when you attempt to add to a full queue, or retrieve from an empty queue.*"
- Can also be used for multiple producers and multiple consumers.

Blocking queue: single prducer/consumer



Blocking queue: multiple prducers/consumers



How to use `BlockingQueue`

- Three operations: inserting, removing and examining the elements in the queue
- If the attempted operation is not possible immediately, but may be satisfied at some point in the future, there are 4 different ways of handling operations:
  - Throws Exception: an exception is thrown.
  - Special Value: a special value is returned, e.g., null or false.
  - Blocks: blocks the current thread indefinitely until the operation can succeed
  - Times Out: blocks the current thread for only a given maximum time limit before giving up. Returns a special value telling whether the operation succeeded or not (typically true / false).

Blocking queue

	Throws Exception	Special Value	Blocks	Times Out
Insert	add(e)	offer(e)	put(e)	offer(e, timeout, timeunit)
Remove	remove()	poll()	take()	poll(timeout, timeunit)
Examine	element()	peek()	N/A	N/A

How to use `BlockingQueue` : Implementations

- `BlockingQueue` is an interface, requires its implementations to use it.
- Java Classes implemented `BlockingQueue`
  - `ArrayBlockingQueue` : a bounded, blocking queue that stores the elements internally in an array
  - `LinkedBlockingQueue` : keeps the elements internally in a linked structure
  - `PriorityBlockingQueue` : an unbounded concurrent queue of which the elements are ordered according to their natural ordering,
  - `DelayQueue` : an unbounded concurrent queue keeps the elements internally until a certain delay has expire
- Java example: Producer Consumer Model using `BlockingQueue`

Concurrent design patterns

- Q: What is a design pattern?
- A: “a general reusable solution to a commonly occurring problem within a given context in software design” – provides a tested, proven development paradigm.
- Q: Why we need design patterns?
- A: Threads usually shared resources, it is difficult to managed them when concurrent programmes become complex.
- Other concurrent design patterns:
  - Active Object: decouples method execution from method invocation
  - Leader/Follower: multiple threads take turns to share a set of event sources
- For more information, you can read this paper.

More complex concurrent programming: Actor model

- Mathematical model of concurrent computation: Actor model
- Actor: universal primitives of concurrent computation, which can response to a message that it receives by:
  - making local decisions
  - creating more actors,
  - sending more messages
  - determining how to respond to the next message received.
- Don't re-invent the wheel:
  - Vert.x: a lightweight, high performance application platform for the JVM that's designed for modern mobile, web, and enterprise applications.
  - akka: toolkit and runtime for building highly concurrent, distributed, and resilient message-driven applications on the JVM.