Multi-Array Queue

Vít Procházka

May 27, 2024

Abstract

A new Queue data structure is introduced that inherits the positive properties of array-based Queues while removing their main drawback: a fixed size.

1 Introduction

Queues are important components of many computer programs. Two major groups of Queues exist: Queues that use linked lists and Queues that use arrays.

Queues based on **linked lists** are e.g. the `LinkedBlockingQueue` or `ConcurrentLinkedQueue` from the Java built-in package `java.util.concurrent`. These Queues face some opposition due to the nodes of the linked lists being scattered in memory (being cache-unfriendly, see e.g. [1]) and - more importantly - that these Queues usually allocate memory and produce garbage (being detrimental to low-latency applications, see e.g. [2]).

Queues based on **arrays** are represented by e.g. the `ArrayBlockingQueue` from `java.util.concurrent` or by the LMAX Disruptor (that is actually more than just a Queue, see [3]). These Queues have, on the other hand, the disadvantage of a fixed capacity that must be allocated up-front.

There also exists a quest for an array-based Queue with auto-extension capability, also without any of the mentioned drawbacks. The auto-extension should, of course, be smarter than just "allocate a bigger array, copy data to it and garbage the old array".

The materials (source codes) referred to are hosted under:
https://github.com/MultiArrayQueue/MultiArrayQueue

2 The idea

Imagine a kid playing with a model railway: The kid initially has a railway ring of some size, and she/he then puts more and more coaches onto it. At once the train is so long that the engine is immediately behind the last coach. What to do now? What about building an additional ring with twice the size and putting a diversion from the old ring to the new ring? So that the engine does not hit the last coach "from behind" but diverts to the new ring! Now the kid can add further coaches to the train as there is now space. If that space is exhausted again, then yet another new ring of quadruple size and a corresponding new diversion shall be added. And so on. The train can grow pretty long! If all the rail work is done correctly, the train can even move, over all three rings.

If the kid now starts taking coaches away, the train becomes shorter and eventually does not need all three rings anymore. However the rings and diversions, once they are laid, stay. So a shorter train will now run over a long path consisting of all three rings. But: should the train start growing again, all is prepared! No need to build new rails (= allocate new memory).

This is the key idea of a new data structure: A Multi-Array Queue.

An **interactive Web-based simulator** exists to get you more familiar with the idea:
https://MultiArrayQueue.github.io/Simulator_MultiArrayQueue.html

More formally:

- Train moving means enqueueing and dequeueing, so optically the train "moves" (let's now abandon the metaphor that the kid drives the train by hand - it makes of course no sense to shift Object references around in memory).

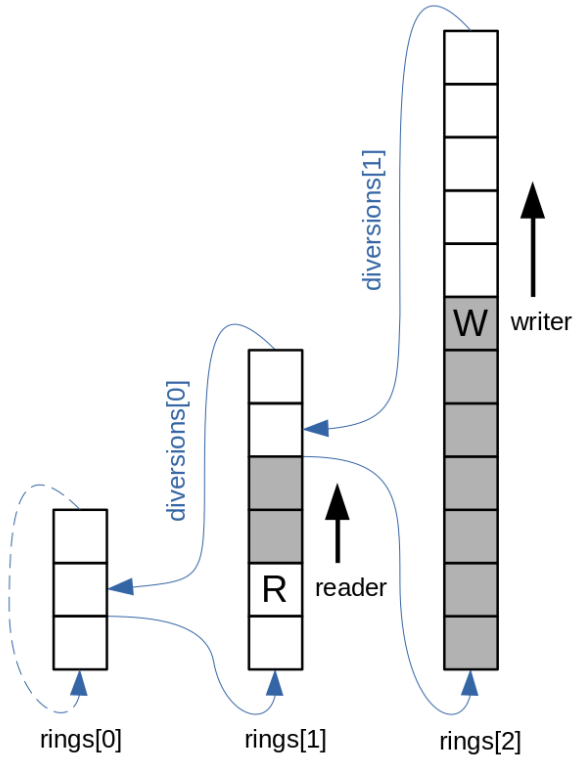

Figure 1: Multi-Array Queue after two extensions

- The engine side of the train is the **writer position**, or semantically the tail of the Queue (where new coaches are added (enqueued)).
- The last couch side of the train is the **reader position**, or semantically the head of the Queue (where the oldest coaches are removed (dequeued)).
- The railway **rings** are arrays of Objects.
- The **diversions** are `long`s (positions) that say: divert immediately before it (and on the return path go back exactly onto it).

The consecutive additions of exponentially growing arrays of Objects allow a capacity that goes beyond of what one single Java array can offer. E.g. if the size of the first array of Objects is 31, then 26 subsequent arrays could be added, the last having size 2 080 374 784 (which is still below $2^{31} - 1$ (the theoretical limit of array size in Java)). But the cumulative size of all 27 arrays is 4 160 749 537.

3 Implementation

Referring to the above example:

It makes sense to put the references of the 27 arrays of Objects into a "super-array": the `rings` array (Java type `Object[][]`).

In case of 27 arrays, there will be 26 diversions. In makes sense to put them into an array as well: array `diversions` (Java type `long[]`). The position of the diversion that leads to `rings[1]` will be in `diversions[0]`, and so on.

Please note that while the diversion to `rings[1]` must be located in `rings[0]`, the diversion to `rings[N]` can be located in any "lower" `rings[M]`. The exact positions of the diversions depend on where exactly the writer has hit the reader "from behind" (and created the diversions). In other words: The structure depends on the actual sequence of the enqueues and dequeues during the "extension phase" of the Queue.

Why is the position of a diversion a Java `long` (actually also the writer and reader positions are Java `long`)? This is because for building of the concurrent version of the Queue, the state must be packed

into a type that is atomically updatable, and Java `long` is that (via class `AtomicLong`). For the exact occupation of the available 64 bits see `ConcurrentMultiArrayQueue.java`, but briefly: 31 bits are used for the array index, 5 bits for the index in the `rings` array, 1 bit for the extension-in-progress flag and the remaining 27 bits are used for the round number to prevent the ABA problem.

3.1 Creation and Uniqueness of diversions

It is important that no two (or more) diversions exist on any given position, because otherwise it couldn't be concluded from the position alone "where we are". Therefore, let's now discuss the actual process of how diversions are created:

If the writer hits the reader "from behind" on a place where there is no diversion, then it will create it there, unless the Queue is already at its maximum capacity.

If on that place already exists a diversion and the writer is entering it (let's call it the **entry side of a diversion**), then instead of hitting the reader it diverts to the beginning of the bigger array. If that place contains another diversion, then the writer diverts further, eventually up to the last (biggest) array.

These two cases actually explain why the Queue can extend to its maximum size from any array onwards, even from an array with only one element: A diversion will be created in that one element that leads to a bigger array. The bigger array will contain two elements of which both can (but one would suffice) become hosts of diversions to yet bigger arrays, and so on till the maximum capacity.

A problem however occurs if the writer is leaving an array via the diversion that led to there (lets call this the **return path of a diversion**). If on exactly that spot sits the reader, then the writer is stuck until the reader moves forward (because it cannot create a second diversion there and it also cannot move back). This is of course not acceptable!

To resolve this, the writer must be **forward-looking**: If it sees that this situation could materialize in the next writer step, it must create a new diversion **now**, also at a time when it is able to. Such diversion creation might of course appear unnecessary in hindsight in case the reader has moved away from the critical spot shortly thereafter (and resolved the situation on its own), but this cannot be relied upon.

3.2 The Algorithms

Let's now put together the pseudocode of the writer:

```
Start a "prospective move forward" at the current writer position
Move forward by one in the array of Objects
if this prospective move goes "beyond" the end of the array then
  if in rings[0] then
    Move to rings[0][0]
    (do not break here because from rings[0][0] eventually diversion(s) shall be followed forward)
  else
    (i.e. we are in a "higher" rings[N])
    Follow diversion[N-1] back
    if the reader is there then
      return  Queue is full
    else
      (we are on the return path of a diversion (no second diversion can exist there))
      (this means: the prospective move forward is done)
      Continue processing at (preparations are done, start the actual work)
    end if
  end if
end if
if the prospective move reaches (an entry side of) a diversion then
  Follow it (to the beginning of respective rings[X])
  (another diversion may sit there, so then continue following)
end if
if the prospective move has hit the reader (that is in the previous round) "from behind" then
  if Queue extension is possible then
    extendQueue = true
```

```
    else
      return  Queue is full
    end if
  else
    if the next writer step could hit the reader on the return path of a diversion (the forward-looking
    check) then
      if Queue extension is possible then
        extendQueue = true
      end if
    end if
  end if
  (preparations are done, start the actual work)
  if extendQueue then
    Allocate a new array of Objects and put its reference into rings
    Put Object into the first array element of the new array of Objects
    Put into diversions the new diversion = the prospective position
    new writer position = first array element of the new array of Objects
    return  Success
  else
    new writer position = the prospective position
    Enqueue the new Object at that position
    return  Success
  end if
```

The pseudocode of the reader is analogous and simpler (the reader does not create the diversions). For full details on both the writer and the reader see the Java source codes and/or the JavaScript code of the Web-based simulator.

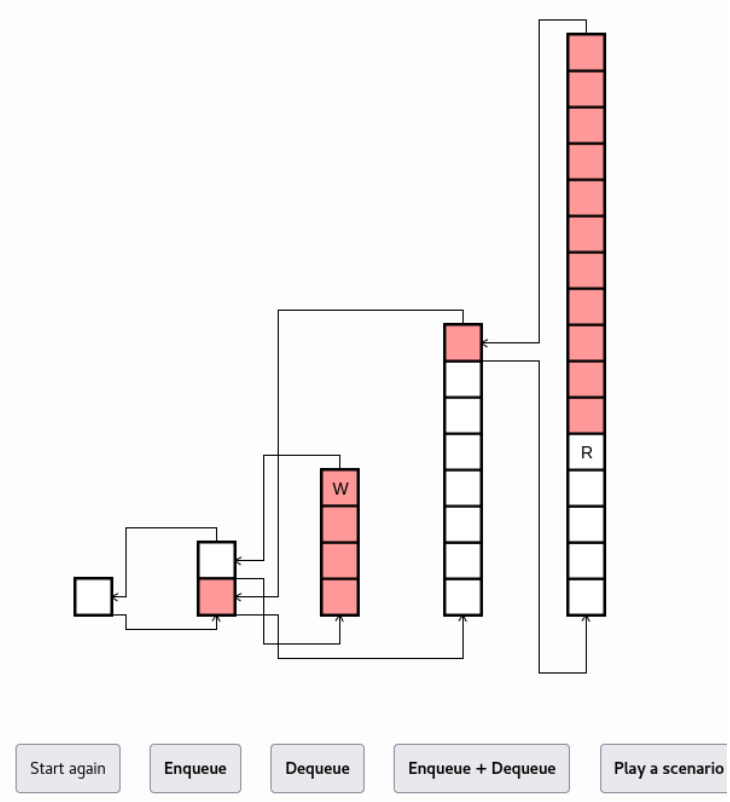

Figure 2: A more complex scenario in the Multi-Array Queue Simulator

It might be easier to read `BlockingMultiArrayQueue.java`, as it is free of the temporal intricacies that must handled by the ConcurrentMultiArrayQueue.

3.3 Cost of reading the diversions

Both the writer and the reader have to search the `diversions` array to check whether they have reached an entry side of a diversion. This is an extra cost.

Let's look again at the example of the unbounded Queue with size 31 of the first array (when the Queue is fully-extended): No diversions can exist in `rings[26]` (size 2 080 374 784), so no extra cost there. But `rings[25]` (size 1 040 187 392) can (but does not have to) contain `diversions[25]`, so one comparison there. Then `rings[24]` (size 520 093 696) can (but does not have to) contain `diversions[25]` and/or `diversions[24]`, so two comparisons there. And so on. Summarized in table 1.

Table 1: Number of comparisons per array

array	size	comparisons
rings[0]	31	26
rings[1]	62	25
rings[2]	124	24
rings[3]	248	23
rings[4]	496	22
rings[5]	992	21
rings[6]	1984	20
rings[7]	3968	19
rings[8]	7936	18
rings[9]	15 872	17
rings[10]	31 744	16
rings[11]	63 488	15
rings[12]	126 976	14
rings[13]	253 952	13
rings[14]	507 904	12
rings[15]	1 015 808	11
rings[16]	2 031 616	10
rings[17]	4 063 232	9
rings[18]	8 126 464	8
rings[19]	16 252 928	7
rings[20]	32 505 856	6
rings[21]	65 011 712	5
rings[22]	130 023 424	4
rings[23]	260 046 848	3
rings[24]	520 093 696	2
rings[25]	1 040 187 392	1
rings[26]	2 080 374 784	0

The weighted average number of comparisons is 1 (mathematically: infinite sum of $1/2+1/4+1/8... = 1$). This might appear favourable, but the distribution is skewed: 26 comparisons are necessary in `rings[0]`.

The mitigation factors however are: The comparisons utilize a tight loop running linearly over the `diversions` array. This array changes (i.e. new diversions are added) only when the Queue extends, and these should be rare occurrences (especially when the array is already big enough). Hence, from performance and scalability points of view, the `diversions` array can be seen as static (i.e. causes no invalidations of its cache line), which is favourable, see also [4].

Due to the worst case being a linear search over only 26 longs, no optimization measures were considered here, as it is known that a break-even point between a linear search and a binary search, for example, occurs at a surprisingly high number, much above the 26 (see e.g. [5]).

4 Thread-safe Program Codes

Handling of multi-threaded operations is not strictly part of the Multi-Array Queue as such, but rather a feature on top of it.

BlockingMultiArrayQueue uses `ReentrantLock` for serializing of the enqueue and dequeue operations, so the program codes guarded by the lock are always executed by only one thread. The use of

`ReentrantLock` allows for waiting (if the Queue is empty on dequeue or full on enqueue).

ConcurrentMultiArrayQueue is a truly concurrent program code based on atomic Compare-And-Swap (CAS) instructions. See [6] for an introduction to this area.

The pseudocode of the concurrent writer from the **temporal perspective** is:

Atomic read of the **writer position**
if the writer position contains the extension-in-progress flag **then**
  `Thread.yield()` and start anew
end if
(TLWACCH[1], especially concurrent readers can move forward the reader position)
Atomic read of the **reader position**
(TLWACCH, especially concurrent writers can extend the Queue)
Atomic read of the **maximum index** of the `rings` array
On local variables: Prospectively move forward, decide about extendQueue, eventually return if Queue is full. This step involves reading of the `diversions` array up to the maximum index obtained in the previous step.
(TLWACCH, especially concurrent writers can move forward the writer position (possibly by extending the Queue))
if extendQueue **then**
  CAS(writer position, "our" writer position, "our" writer position + extension-in-progress flag)
  if CAS succeeded (i.e. the writer position has not been modified yet by other writers) **then**
    (spot C relevant to lock-freedom (see 4.1))
    (other writers are now "locked out")
    (readers can continue their work but once they deplete the Queue, they cannot go past the writer position)
    Extend the Queue: Add to the `rings` and `diversions` arrays + atomically increment the maximum index
    (time lag when the new diversion is already visible, but not yet the new writer position)
    Atomically set new writer position (without the extension-in-progress flag)
    return Success
  else
    CAS failed (i.e. lost the race against other writers) → Start anew
  end if
else
  (i.e. no extendQueue)
  if the reader has not yet cleared the old Object from our prospective writer position (i.e. the reader is in spot B) **then**
    Wait with `Thread.yield()` (but stop waiting and start anew if the writer position has moved forward meanwhile)
  end if
  CAS(writer position, "our" writer position, "our" prospective writer position)
  if CAS succeeded (i.e. the writer position has not been modified yet by other writers) **then**
    (spot A relevant to lock-freedom (see 4.1))
    (the writer position is now "ours")
    write the Object to the writer position
    return Success
  else
    CAS failed (i.e. lost the race against other writers) → Start anew
  end if
end if

Again, the pseudocode of the reader is analogous and simpler.
Please read the comments in `ConcurrentMultiArrayQueue.java` for detailed treatment of:

- Prevention of reads and writes from getting re-ordered
- Argumentation about correctness of the order of reads

[1] Time Lag When Anything Concurrent Can Happen

- Usages of the 27-bit round number (implicit by the AtomicLong CAS + explicit in the program code)
- Other technical handlings and details not covered here

For prevention of the ABA problem the writer and reader positions contain a 27-bit round number that is incremented on each passing of `rings[0][0]`. However there still exists a (miniature) chance for the ABA problem to occur: If a thread gets preempted for such an excessive time during which the 27-bit round number would roll over back to its old value.

4.1 Lock-freedom

The ConcurrentMultiArrayQueue does not fulfill the strict requirement for "lock-free" that "in a bounded number of my steps somebody makes progress" (see [6]) because there exist three spots in the program code (A and B tiny and C the extension operation which is not so tiny) where preemption of a thread could block other threads for beyond "bounded number of my steps". A theoretical termination of a thread in one of these spots would leave the Queue in a blocked state.

Talking about the "tiny" spots A and B (B is in the reader): They result from the impossibility to perform two actions atomically, namely moving forward the writer/reader position and writing/clearing the Object reference to/from that position.

A double-location CAS (see [7]) would help here, but it is not supported by any widespread CPUs.

There exist techniques for pinning of threads to CPU cores, see e.g. [8]. If it can be achieved that the relevant threads never get preempted, then the "lock-free" property could be approached from that direction.

Leaving the double-location CAS possibility aside, positions in the arrays of Objects must have the following four-state diagram:

1. Writer position has moved forward to the position (a short-lived state in spot A of the writer)
2. Object reference was written to the position
3. Reader position has moved forward to the position (a short-lived state in spot B of the reader)
4. Object reference was cleared from the position

5 Performance

Referring to [4] again: If there is write sharing then the system ungracefully degrades, the more threads we add, the slower it becomes.

The obvious "hot spots" are the `ReentrantLock` in the BlockingMultiArrayQueue and of course the writer and reader positions (which are `long` in the BlockingMultiArrayQueue and `AtomicLong` in the ConcurrentMultiArrayQueue).

It is expected that these will be the performance-limiting factors and that other Queues - unless they are based on different principles - will suffer similarly. Therefore, the performance figures should be similar (or at least in similar ranges).

The performance was measured on a notebook with Intel(R) Core(TM) i5-7200U CPU @ 2.50GHz, with 2 cores with 2 physical threads each. The notebook runs Windows 10, also many software threads are already underway on it. It is clear that such machine is not suitable for rigorous measurements in the sense of obtaining absolute values, but comparative measurements to the following Java Queues should be meaningful nevertheless:

- `LinkedBlockingQueue` from `java.util.concurrent`
- `ConcurrentLinkedQueue` from `java.util.concurrent`
- `ArrayBlockingQueue` from `java.util.concurrent`

The number of enqueue/dequeue pairs per second were measured depending on how many threads do the enqueueing/dequeueing concurrently.

The results do not contrast with the "write shared" results in [4]. No Queue overperforms or underperforms the others by orders of magnitude. All Queues perform best in single-threaded regime, in multi-threaded regime the overall performance falls by around one half.

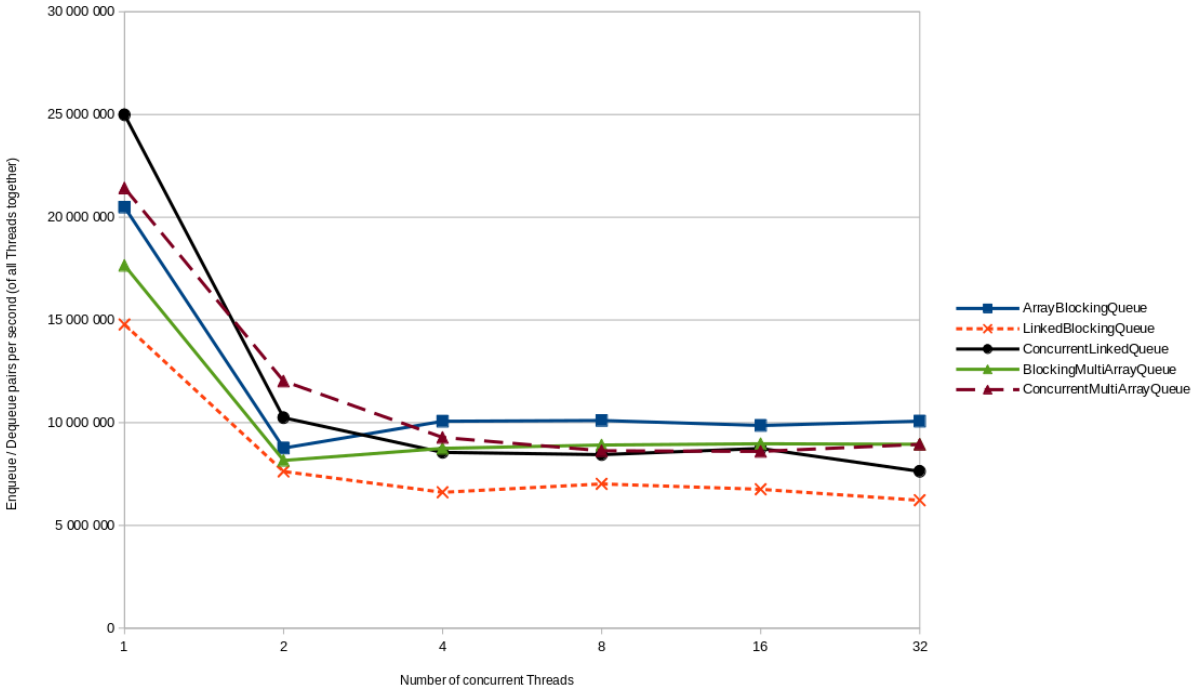

Figure 3: Performance measurements of Java built-in Queues and the new Multi-Array Queues

In the range above 4 concurrent threads the Multi-Array Queues underperform the `ArrayBlockingQueue` by roughly 10% but outperform the `LinkedBlockingQueue`. Given the more complex program logic of the Multi-Array Queues, their underperformance against the `ArrayBlockingQueue` appears logical.

6 Conclusion

The new Multi-Array Queue has been introduced, developed and measured.

As of publication date (2024) the idea and the program codes are in their early stages and only for academic interest, not for production use. Reviews, tests and comments are welcome (via Issues in the GitHub repository).

References

[1] Martin Thompson. *Mechanical Sympathy - Memory Access Patterns Are Important*. 2012. URL: *https://mechanical-sympathy.blogspot.com/2012/08/memory-access-patterns-are-important.html*.

[2] Benoît Jardin. *ZeroGC — Low Latency Java*. 2016. URL: *https://benoitjardin.wordpress.com/2016/09/22/avoiding-garbage-collection*.

[3] Martin Fowler. *The LMAX Architecture*. 2011. URL: *https://martinfowler.com/articles/lmax.html*.

[4] Dmitry Vyukov. *1024cores*. 2024. URL: *https://www.1024cores.net/home/lock-free-algorithms/first-things-first*.

[5] Dirty hands coding. *Performance comparison: linear search vs binary search*. 2017. URL: *https://dirtyhandscoding.github.io/posts/performance-comparison-linear-search-vs-binary-search.html*.

[6] Michael Scott. *Nonblocking data structures — SPTDC 2019*. 2019. URL: *https://www.youtube.com/watch?v=9XAx279s7gs*.

[7] Wikipedia. *Double compare-and-swap*. 2022. URL: *https://en.wikipedia.org/wiki/Double_compare-and-swap*.

[8] Manuel Bernhardt. *On pinning and isolating CPU cores*. 2023. URL: *https://manuel.bernhardt.io/posts/2023-11-16-core-pinning*.