Fast Thread-safe Priority Queues

Yong Wan, Hualong Zhu, Zhihao Zhang

Abstract

This report indicates fast thread-safe priority queues using in Java. Although Java provides PriorityBlockingQueue which is a kind of priority queues for the implementation of thread-safe concurrency, it cannot meet the requirements of fast multithreaded processing. In this report, we achieve a comparison between LockFreePriorityQueue, Pipelined PriorityQueue, and PriorityBlockingQueue. This report also expounds the basic knowledge used in these priority queues, implementation details, the methodology for implementation. The testing and evaluation of our implementation are illustrated. For future works, we will do more research for different priority queues which can do parallel computing and test their actual efficiency.

1. Introduction

Priority queues are playing an important role in algorithm design, which is provided by Java SDK. Meanwhile, priority queues are logically implemented using a heap structure (complete binary tree), which is physically implemented using a dynamic array. Priority queues use key-value pairs to implement priority operations since the priority is represented by the key. To insert or delete elements in a priority queue, it supports insert and delete operations, which are all based on the key (priority) of the element.

Java also provides a priority queue called PriorityBlockingQueue to implement the thread safe parallelization. PriorityBlockingQueue is an unbounded concurrent security priority queue based on the priority heap. However, when using PriorityBlockingQueue, other concurrent operations cannot make any progress while the access to the shared resource is blocked by the lock. PriorityBlockingQueue can lead to deadlock and priority inversion.

Consequently, we explored two alternative fast thread-safe priority queues: LockFreePriorityQueue and Pipelined PriorityQueue.

LockFreePriorityQueue aims to implement the increasement of searching efficiency by using the skip-list data structure. That means there is no need to use locks in the multithreaded program. Furthermore, thanks to some kinds of the synchronization primitives, LockFreePriorityQueue implement the supporting of thread or process synchronization. We will explain this in the next sections.

Unlike PriorityBlockingQueue using a global lock to keep data safe when reading and writing, instead Pipelined PriorityQueue uses a double-level lock to implement a pipelined queue, in which only a part of data in the queue would be locked when r/w operation occurs[3].

In our project, we will analyze and implement both LockFreePriorityQueue and Pipelined PriorityQueue. After that, these three queues including PriorityBlockingQueue will be compared together on the performance.

2. Basic knowledge

In this section, the basic knowledge is introduced including binary heap, SkipList and lock, which is used in our project.

A. Binary Heap

The binary heap is a special kind of heap, which is a complete binary tree or an approximate complete binary tree. Furthermore, it satisfies the structure and sequence of the tree. The tree mechanism characteristic is the structure which the complete binary tree should have. The heap order is that the key values of the parent node always remain in a fixed order in relation to the key value of any child node. Also, the left subtree and right subtree of each node is a binary heap. It has two manifestations: the smallest heap (as shown in Figure 2.1) and the largest heap (as shown in Figure 2.2).

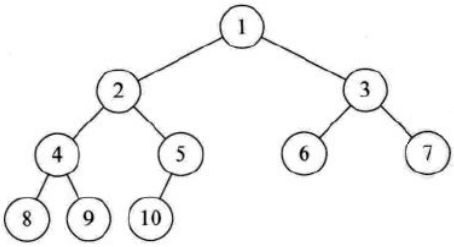

Figure 2.1 The smallest heap tree

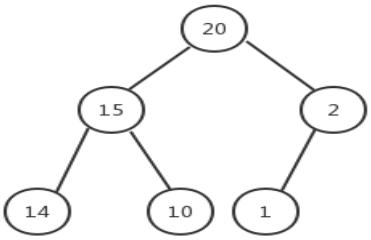

Figure 2.2 The largest heap tree

As shown in the figures, the smallest heap follows the principle that the parent node's key value is always greater than or equal to the key value of any one of the child node(s). Also, the largest heap is the opposite.

B. Skip-list

Skip-list is a widely used data structure which supports fast searching. According to William's works, a skip list saves elements in a hierarchical list in an orderly fashion[1].

I. The data structure of skip-list

Skip list uses randomization and has a probabilistic time complexity of O(log N) where N is the maximum number of elements in the list[2]. The data structure based on a list which contains ordered data, then add shortcuts which randomly distributed for improving the whole search efficiency on this struct. In general, the level of shortcuts (the maximum height) of the data structure is log N. The highest priority node is located first after node head in the list.

II. An example of skip-list

As shown in Figure 2.3, there are three levels including Height 2, Height 1 and Next. While H means Head and T means Tail, these two nodes are fixed in every skip list and other nodes are

normal data nodes. We can see the figure that the nodes which are H, 5 and T in the height 2 level. At the same time, the height 1 level has nodes H, 1, 3, 5 and T. The lowest level 'Next' goes through all nodes.

For example, when we do a search for a node equals 4. Firstly, we start from the Highest level which is Height 2. Comparing the first node after H which is 5 and it is great than 4, we go down to a lower level which is Height 1. Secondly, continue to compare node 1 which after the first node H. Obviously, 1 is less than 4 then we go to the next node 3 which is also less than 4. By using the same principle, we arrive node 5 (greater than 4) so we should go back to node 3. Finally, lower down to the level named 'Next' and we get node 4.

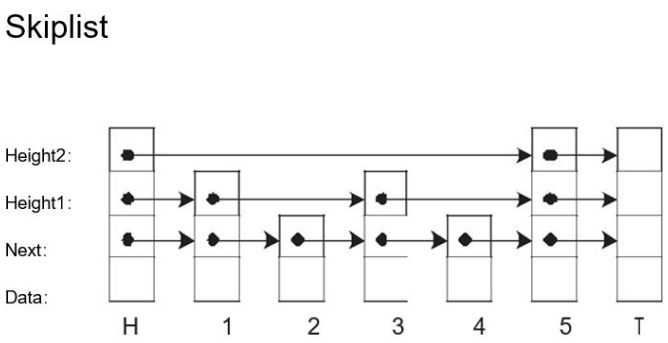

Figure 2.3 The structure of a skip list[2]

III. How skip-list works on multi-threads

Three kinds of the standard atomic synchronization primitives, which named Test-And-Set(TAS), Fetch-And-Add(FAA) and Compare-And-Swap(CAS), are implemented on the current popular computer system. A skip-list inserts a node or deletes a node by using these three atomic methods[2].

Especially, for Java, we should use AtomicMarkableReference. An AtomicMarkable-Reference maintains an object reference along with a mark bit, that can be updated atomically. The main methods of AtomicMarkableReference include compareAndSet() ,set(), isMarked() etc.

C. Pipelining

Pipelining (As shown in Figure 2.4) is a particularly efficient way of organizing concurrent activity in a computer system.

In the field of computing, a pipeline, which is also known as data pipeline, is a set of data processing elements connected in series. The elements of a pipeline are often executed in parallel or in time-sliced fashion.

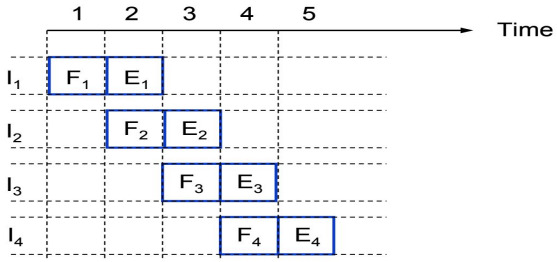

Figure 2.4 The basic idea of instruction pipelining

3. Implementation

A. The Implementation of LockFree PriorityQueue

LockFreePriorityQueue was initially introduced by Hakan and Philippas and implemented in C++. Then Raja implemented it in java. LockFreePriorityQueue uses skip-list to stores data in a list structure and uses different levels pointers to improve the searchment efficiency. Comparing PriorityBlockingQueue, LockFreePriorityQueue does not use a lock to support multi-threads, atomical updating methods are widely used in inserting and deleting operation by LockFreePriorityQueue to support multi-threads environment. In detail, it always uses compareAndSet() method to set IsMarked on nodes which need to be updated first, this operation is atomic by operation system. After that, other threads would not access this node if they get IsMarked status, the node's IsMarked become false until the whole updating operation finished. Then other threads can through this node to any nodes they will go to.

B. The Implementation of Pipelined PriorityQueue

The Pipelined PriorityQueue, which is a little similar to the to the PriorityBlockingQueue, also has a binary heap array which stores data in the tree structure. But there are some differences, as the figure 3.1 shows, Pipelined PriorityQueue owns two arrays, instead of one like PriorityBlockingQueue. The additional array is called Token array, which can store the node needed to be inserted, the node contained the the priority value, the location of the node in the binary heap array which is compared with, the index of the token array equals the level of the binary heap array. Another different point from PriorityBlockingQueue is that the binary heap array is not locked with all the nodes when there is any w/r operation , instead most of time, just two levels of nodes would be locked.

The node in binary heap array has two essential attributes except for the node value. One is state, which can be active and inactive, when the state is inactive, it means the node is empty. The other attribute is capacity, which records the total inactive nodes of its sub-nodes including itself. For example, as figure 3.1 shows, node 2 in Binary Heap Array is active and its capacity is 2 because there is data in this node and it has two inactive sub-nodes which are node 12 and node 13.

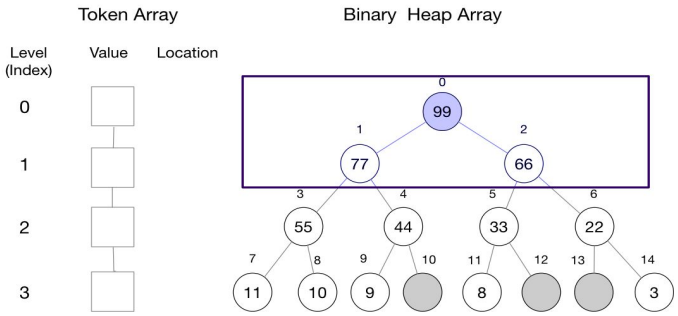

Figure 3.1 The data structure of pipelined priority queue

As the figure 3.2 shows that the UML class graph of Pipelined PriorityQueue, Class TokenArrayElement represents the node data in the Token Array. Class BinaryArrayElement represents the node data in Binary Heap Array. Class Pipelined PriorityQueue inherited AbstractQueue which contains many essential functions such as Enqueue and Dequeue.

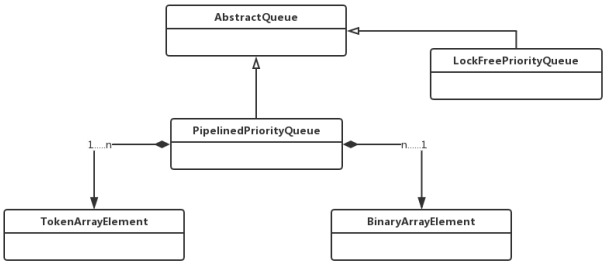

Figure 3.2 UML Graph of Pipelined PriorityQueue

4. Methodology

A. Insert Node by Lockfree PriorityQueue

The main algorithm steps, for inserting a new node at a rodam position in our skip list has two steps:

I. From the highest level node of heat, comparing the priority, if the number of priority less than next node, lowing the level, compare again until the lowest level to find a right position to add a new node[2].

II. From the lowest level of to-be-previous nodes of the new node,update their next pointers atomically ,also include the new node's next pointer[2].

B. Poll Node by Lockfree PriorityQueue

The main steps of the algorithm for deleting a node at a random position has four steps:

The first step is to find the appropriate position for deleting, the method likes the first step of inserting a node[2].

Then the to-be-deleted node should be written a deletion indication to avoid being used by another thread[2].

After that, the deletion marks also should be set on the next pointers of the to-be-deleted node, from the lowest level to the highest level which the node hold[2].

The last step has two part. The first part, in the lowest level, atomically update the next pointers of the previous nodes of the to-be-deleted node; after then, unlike the previous steps, it should update atomically the next pointers from the most topmost level of its parent to the second lowest level[2].

C. Insert Node By Pipelined PriorityQueue

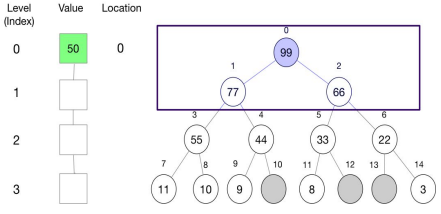

Figure 4.1.1

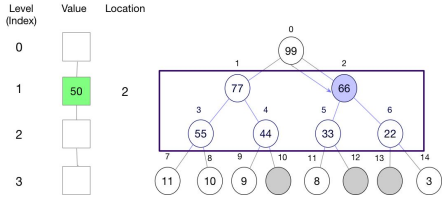

Figure 4.1.1

To insert a node to the Pipelined PriorityQueue, the new node should be inserted into the first index of the Token Array (T array) firstly, the the node should compare its value with the top node of the Binary Heap Array (B array), because the top node is active and its value is larger than the value of the new node, so that the node should compare with the nodes in next level.

Because the capacity of right-side sub-node is larger than that of left side, so the right-side sub-node will be chosen as next base node. Also, the lock will move to lock the next level nodes but release the nodes in the first level.

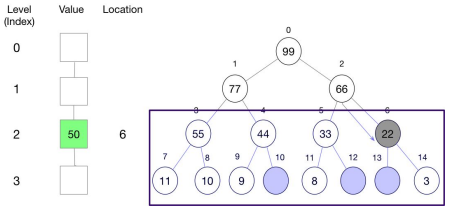

Figure 4.1.3

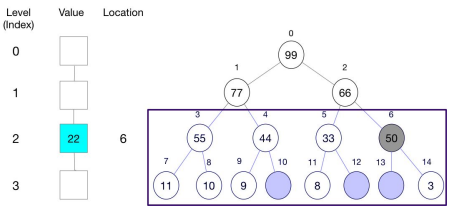

Figure 4.1.4

Until the level 2, when the value of the new node is larger than the one's in the B array, then both nodes should exchange the position. As there is still a node in the T array, the whole operation should continue to find an inactive node to inject the node or to compare with.

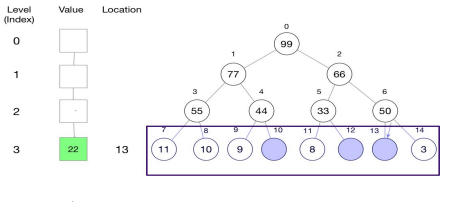

Figure 4.1.5

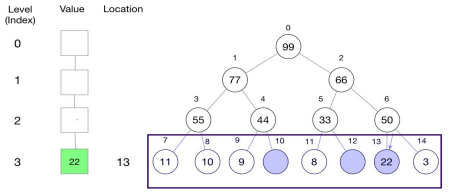

Figure 4.1.6

Until that the node in T array finds an available node to inject, the whole insert operation could finally be over.

D. Poll node by Pipelined PriorityQueue

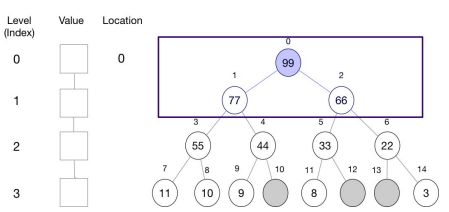

Figure 4.2.1

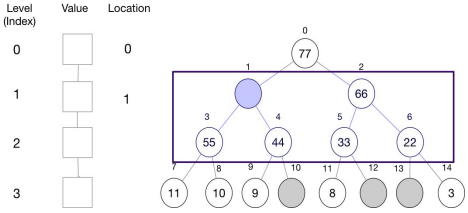

Figure 4.2.2

The operation to dequeue from Pipelined PriorityQueue is distinct from PriorityBlockingQueue which starts from the bottom of the tree, it starts from the top node because top node in Pipelined PriorityQueue owns the highest priority. Firstly, the top node would be dequeued, then its two sub-nodes will compared the node value with each other, the higher one will move up.

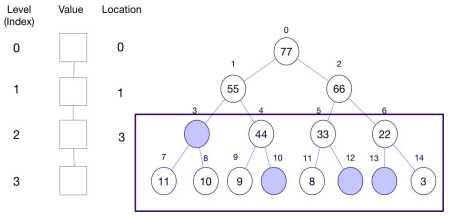

Figure 4.2.3

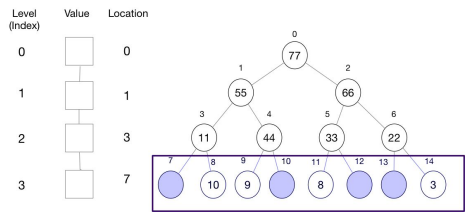

Figure 4.2.4

Following this rule, its sub-nodes continue the similar operation, until that there is no sub-nodes needed to be moved up.

5. Testing and Evaluation

The setting of our benchmark environment is that Intel® Core™ i5-8300H processor with 4 cores. As we can see from the figures (Figure 5.1, Figure 5.2 and Figure 5.3), we tested three different operations in the same environment.

At the begin, when there is only a one thread, the PriorityBlockingQueue is the most efficient because processing single thread doesn't have mutually exclusive or deadlock situations. In contrast, Pipelined PriorityQueue and LockFreePriorityQueue have a lot of extra action to avoid blocking. In fact, there's no blocking happening. Furthermore, when the processor need to handle 4 threads. Pipelined PriorityQueue and LockFreePriorityQueue tend to have the similar performance.

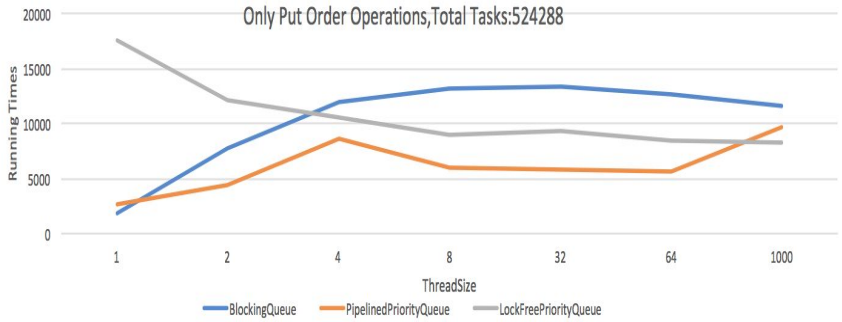

Figure 5.1 Ordered put operation

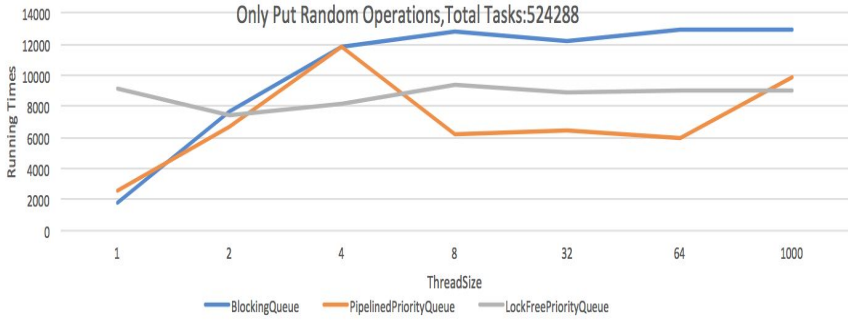

Figure 5.2 Random put operation

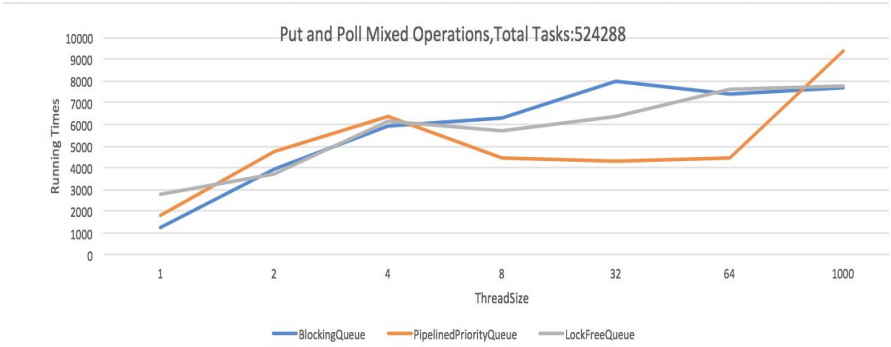

Figure 5.3 Random mixed operation

6. Future work

According to what we discussed in previous sections, we found that the increase in efficiency is not great as we expected. For this reason, we decide to test on extra cpu with more cores to and optimize our program.

7. Conclusion

To sum up, in terms of our research, PriorityBlockingQueue uses a lock to block related operations like put and poll. Contrastly, Pipelined PriorityQueue only locks two levels including the level of current node and the next level and LockFreePriorityQueue doesn't use a lock, it updates data by atomic methods. Based on our test, when there is only one thread, we use PriorityBlockingQueue to get the highest efficiency. For multithreaded process, we will choose using LockFreePriorityQueue or Pipelined PriorityQueue depending on the thread size.

References

[1]Pugh W. (1989) Skip lists: A probabilistic alternative to balanced trees. In: Dehne F., Sack J.R., Santoro N. (eds) Algorithms and Data Structures. WADS 1989. Lecture Notes in Computer Science, vol 382. Springer, Berlin, Heidelberg

[2]Sundell, H., & Tsigas, P. (2005). Fast and lock-free concurrent priority queues for multi-thread systems. Journal of Parallel and Distributed Computing, 65(5), 609-627.

[3] Bhagwan, R., & Lin, B. (2000). Fast and scalable priority queue architecture for high-speed network switches. In INFOCOM 2000. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE (Vol. 2, pp. 538-547). IEEE.

Contribution Table

Members' Name	Username	Presentation	Implementation	Report Writing
Yong Wan	wanyong117	33.3%	33.3%	33.3%
Hualong Zhu	Long315	33.3%	33.3%	33.3%
Zhihao Zhang	zzh1227	33.3%	33.3%	33.3%