Lock-free Data Structures

Burak Ok
Friedrich-Alexander-Universität Erlangen-Nürnberg
burak.ok@fau.de

ABSTRACT

This paper lists the general properties of lock-free data structures. Furthermore, the advantages and disadvantages is getting discussed. To demonstrate how to verify a lock-free algorithm, we will take a look on the lock-free queue proposed by Michael and Scott[19]. Finally the queue gets compared to other newer lock-free queues and a simple lock-based queue to show the differences regarding the speed.

1 INTRODUCTION

Processors are getting more and more cores, but the clock speed and instructions per cycle increase has been at a low rate compared to the core count [5]. The programmers need to divide the work and roles and distribute it to different cores. Sometimes these cores need to communicate with each other to exchange data and the programmer has to ensure that the data does not get garbled up. Many applications use exclusive locks to serialize the access on the data. This is fine as long as the application is not critical for the user or any other processes. But if, for example, the application was a part of a nuclear reactor, then missing deadlines or a occurrence of a deadlock could lead to disastrous outcomes. Fortunately, lock-free data structures can solve both of these problems.

2 LOCK-FREE PROGRAMMING

In order for a algorithm to qualify as lock-free, it must allow a thread to complete its task regardless of the state of other threads. In order to get an overview about lock-free algorithms, the basic primitives behind the algorithms must be understood. Furthermore, these algorithms do not only have advantages, but also certain disadvantages.

2.1 Primitives

A small number of primitives exists, which helps to program lock-free algorithms. Most of the algorithms uses read-modify-write primitives. A selection of these primitives gets named and described briefly.

Memory barriers Modern compilers and processors can reorder loads and stores to boost performance. Memory barriers tells both, the compiler and the processor, to stop reordering loads and stores for a specific critical section. [1] [6] [2]

test-and-set The primitive takes a pointer to a *boolean* variable atomically, sets it to *true* and returns the old value. [6]

fetch-and-add This primitive adds a number to a specified variable and returning the old value. [6]

linked-load and store-conditional Linked-load reads from a memory address. A subsequent store-conditional to the same address only stores the value if the supplied address did not got written onto. [1]

compare-and-swap This is a most used primitive. It only stores a value to an address if it has a given value. [6]

2.2 Advantages

Lock-free algorithms can have a significantly higher throughput than their lock based counterparts when being under heavy load. If a thread is holding a lock for a critical section and gets preempted, every other thread can not progress further. All threads have to wait until the lock-owning thread gets scheduled again and hopefully unlocks the lock. In the lock-free algorithm, other threads can try to change the shared variable and succeed if no other thread did before. The original thread that wanted to modify the variable now has to loop at least once more and try it again.

There can not be any deadlocks by definition[4]. This is really important for any critical task and distributed system. It easily solves the issue when a thread hangs, gets killed or a computing node in a distributed system is not accessible any more while holding a remote lock. There are many possibilities to create deadlocks when working with locks. The problems ranges from the basic philosophers problem[11] to more complex problems like priority inversion[16]. Finding and implementing a smart deadlock prevention algorithm for these issues is not easy and requires additional developing time, runtime and memory. Even then the prevention algorithm might find deadlocks but can create another ”kind” of deadlock: the livelock [23]. A livelock happens when two threads are aware that a deadlock could occur, thus trying to dodge the possibility of a deadlock and which results in another deadlock situation and so on. That is why most of the noncritical applications are take the little risk of a deadlock instead of dealing with the problem. This is known as the Ostrich Algorithm[21]. Even the UNIX and Windows operating systems are not handling all possible deadlocks that can occur[3].

2.3 Disadvantages

Lock-free does not mean wait-free. Therefore a slower thread can starve if other faster ones are progressing. Lock-free has only the property of a guaranteed global throughput making slow threads suffer from this, whereas wait-free guarantees per-thread progress[4]

Since CAS is the most used primitive in lock-free algorithms [10] you probably have to deal with the ABA-Problem. An example helps to describe the underlying issue. Both graphics in Figure 1 are visualizing a stack. The `Top of Stack` points to the first node of the stack and every node points to its successor. If thread 1 wants to `pop()` from the stack, it needs to copy the `Top of Stack` pointer and its successor, as seen in 1) of Figure 1. Then the thread is getting preemptively unscheduled. Meanwhile another thread can `pop()` and `push()` as many times as it wants. After all these operations,

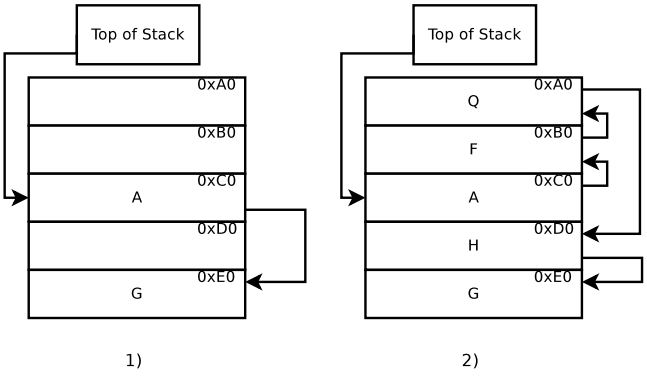


Figure 1: ABA-Problem in a stack implementation

there is a possibility that the stack could look like 2) in Figure 1. When thread 1 gets scheduled again and runs, it would compare its saved value of `Top of Stack` to the actual `Top of Stack`, to check if the stack got modified. Since `Top of Stack` points to the same address in 1) and 2), the thread does not know that the stack got changed. It swings the `Top of Stack` pointer to node `G` and the nodes `Q`, `F` and `H` will get lost.

Writing lock-free code is difficult. There are many primitives to choose from and everyone of them has its own hazards [7]. Even Michael and Scott found several bugs in some of the lock-free queues while benchmarking their queue against alternative ones [19].

3 MICHAEL-SCOTT QUEUE

One of the queues proposed by Michael and Scott is lock-free. The pseudo code is divided into three segments. Listing 9 shows the structures and the initialization for the queue, Listing 19 inserts a new element into the tail of the queue and Listing 23 dequeues the head and returns it, if one exists.

To solve the ABA-Problem the structure **pointer_t** in listing 9 is storing a modification counter next to the pointer. The algorithm needs either an access to a double-word CAS to swap both values atomically or the pointer and counter have to share a single word. When the modification counter is combined with the pointer, it implies that the pointer can not be a "real" pointer. The counter gets incremented every time a new value got written into the pointer. This does not solve the ABA-Problem but it makes it really unlikely to occur since the counter has to completely wrap up to get the same bit pattern.

Internally the queue uses a singly linked list, with pointers to the head and tail, to store the data. To avoid special cases with an empty list, the linked-list contains always at least one node. When dequeuing, the algorithm actually removes the front node but writes the value from the former second node into `pvalue` **(D12)**.

For this algorithm, the authors has noted that the `free()` used in line **(D19)** should not be compared to the release of dynamic memory provided by operating systems. It has to release the node to a special maintained list for the linked-list otherwise a use-after-free bug will follow [19]. The issue with memory reclamation and the ABA-Problem got elegantly solved with Hazard-Pointers in a later paper [18]. In short, Hazard-Pointers are a per-thread list of currently used pointers. If a thread wants to delete a node and reclaim its memory, it has to look into every other Hazard-Pointer-List whether the list contains the pointer or not.

3.1 Linearizability

A data structure is linearizable when the changes to the structure appears to be instantaneous for an external observer [12]. The presented queue is linearizable, because there is a specific operation in both methods, `enqueue()` and `dequeue()`, where the queue changes its state.

For `enqueue()` the operation is at line **(E9)**. Even though the `CAS` line does not change the `Tail` to the newest element, it points the `next` pointer to the newly enqueued node. Every operation that requires the correct `Tail` will advance it to the next element in the linked list until the *next* pointer of a node is `NULL`. So the enqueue seems to happen instantaneous.

The operation in `dequeue(` is at line **(D13)**. Here the *Head* pointer is getting pointed at the node which the *next* pointer of the dequeued node pointed to.

3.2 Safety

The safety property states that something bad never happens and the data structure never enters an invalid state [20]. It is one of the two properties of lock-free algorithms mentioned in the paper. For this queue, Michael and Scott set five requirements which needs to be satisfied [19]:

(1) The linked list is always connected.
(2) Nodes are only inserted after the last node in the linked list.
(3) Nodes are only deleted from the beginning of the linked list.
(4) `Head` always points to the first node in the linked list.
(5) `Tail` always points to a node in the linked list.

After the initial call to `initialize()` in Listing 9, all of the requirements are satisfied. To show that `enqueue()` and `dequeue()` do not break any requirements, Michael and Scott shows by induction that all of them are still satisfied.

(1) The linked list is always connected because the `next` pointer of a node is only set once when a new node is inserted after it **(2)**. No node is getting deleted besides the one pointed to by `Head` **(3)**
(2) A new node will only get inserted at the back of the linked list. `Tail` always points to a node in the linked list **(5)**. The new node will get linked after the node who has a `NULL` `next` pointer **(2)**
(3) Nodes are only deleted from the beginning of the linked list. The node pointed to by `Head` is the first node in the linked list**(4)**.
(4) `Head` always points to the first node in the linked list. The only line that changes `Head` is line **(D13)** and the new node it is pointing to is its successor `next`. `Head` also never points to `NULL`, because if the linked list has only one node **(D6)** the `dequeue()` function will return without deleting anything **(D8)**

```
structure pointer_t       {ptr: pointer to node_t, count: unsigned integer}
structure node_t          {value: data_type, next: pointer_t}
structure queue_t         {Head: pointer_t, Tail: pointer_t}

initialize(Q: pointer to queue_t)
      node = new_node()                                   # Allocate a free node
      node->next.ptr = NULL                               # Make it the only node in the linked list
      Q->Head = Q->Tail = <node, 0>                       # Both Head and Tail point to it
```

Listing 1: Definitions and Initialization

The structures and initialization for the MSQ

```
enqueue(Q: pointer to queue_t, value: data type)
E1:   node = new_node()                                   # Allocate a new node from the free list
E2:   node->value = value                                 # Copy enqueued value into node
E3:   node->next.ptr = NULL                               # Set next pointer of node to NULL
E4:   loop                                                # Keep trying until Enqueue is done
E5:       tail = Q->Tail                                  # Read Tail.ptr and Tail.count together
E6:       next = tail.ptr->next                           # Read next ptr and count fields together
E7:       if tail == Q->Tail                              # Are tail and next consistent?
E8:           if next.ptr == NULL                         # Was Tail pointing to the last node?
E9:               if CAS(&tail.ptr->next, next, <node, next.count+1>)   # Try link node at the end of the linked list
E10:                  break                               # Enqueue is done. Exit loop
E11:              endif
E12:          else                                        # Tail was not pointing to the last node
E13:              CAS(&Q->Tail, tail, <next.ptr, tail.count+1>)         # Try to swing Tail to the next node
E14:          endif
E15:      endif
E16:  endloop
E17:  CAS(&Q->Tail, tail, <node, tail.count+1>)           # Try swing Tail to inserted Node
```

Listing 2: Enqueue

The function enqueues a new node into the back of the MSQ

```
dequeue(Q: pointer to queue_t, pvalue: pointer to data type): boolean
D1:   loop                                                # Keep trying until Dequeue is done
D2:       head = Q->Head                                  # Read Head
D3:       tail = Q->Tail                                  # Read Tail
D4:       next = head.ptr->next                           # Read Head.ptr−>next
D5:       if head == Q->Head                              # Are head, tail, and next consistent?
D6:           if head.ptr == tail.ptr                     # Is queue empty or Tail falling behind?
D7:               if next.ptr == NULL                     # Is queue empty?
D8:                   return FALSE                        # Queue is empty, couldn't dequeue
D9:               endif
D10:              CAS(&Q->Tail, tail, <next.ptr, tail.count+1>)         # Tail is falling behind. Try to advance it
D11:          else                                        # No need to deal with Tail
                  # Read value before CAS, otherwise another dequeue might free the next node
D12:              *pvalue = next.ptr->value
D13:              if CAS(&Q->Head, head, <next.ptr, head.count+1>)      # Try to swing Head to the next node
D14:                  break                               # Dequeue is done. Exit loop
D15:              endif
D16:          endif
D17:      endif
D18:  endloop
D19:  free(head.ptr)                                      # It is safe now to free the old dummy node
D20:  return TRUE                                         # Queue was not empty, dequeue succeeded
```

Listing 3: Dequeue

The function dequeues a node from the front of the MSQ

(5) `Tail` always points to a node in the linked list. It never lags behind `Head` **(D6)**. It also never points behind the end of the linked list. **(E13)** advances `Tail` only if it already checked that there is a valid `next` pointer for the node **(E8)**. **(E17)** only happens when an enqueue was successful **(E10)**. **(D10)** only gets executed if `Head` and `Tail` points to the same node **(D6)** and that same node has a predecessor **(D7)**

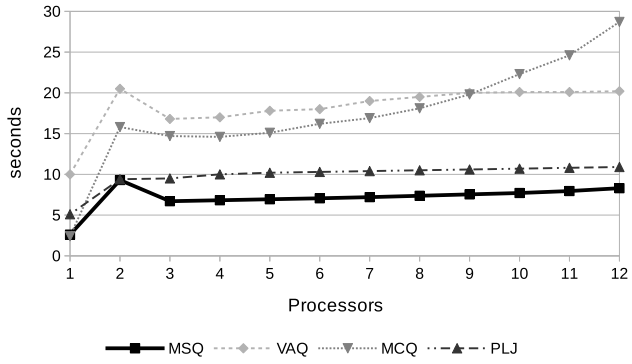
3.3 Liveness

Liveness is described as that something desirable has to happen sometime [20]. In order to show that `enqueue()` and `dequeue()` is lock-free, it is necessary to prove that a thread only has to loop a finite amount of times even if another thread has modified the queue and successfully queued or dequeued an item. Therefore, all conditions which can lead to a loop needs to be verified.

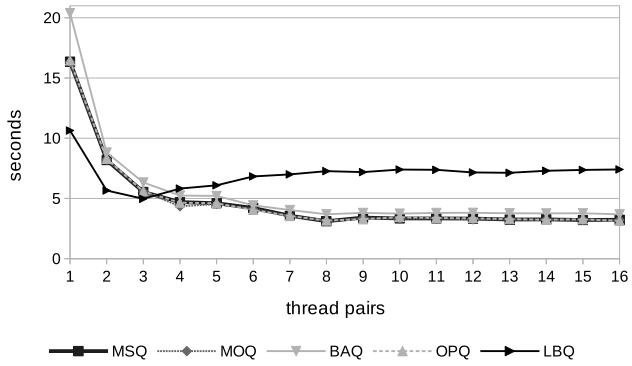
- **(E7)** The condition only fails if the local variable `tail` is not pointing to the queues `Tail` anymore. This happens when the `Tail` gets swung to the next node which implies that another thread must have succeeded.
- **(E8)** The condition fails if the `next` node has a successor, which means another thread has successfully enqueued a new node. If that happens **(E13)** will try to swing `Tail` to the next node. This will prevent the thread from waiting for another thread to update the `Tail` pointer.
- **(E9)** The `CAS` only fails if another thread queued a node before it.
- **(D5)** The condition only fails if `Head` is pointing to another node as when it was read in **(D2)**. This means that another thread has successfully dequeued a node.
- **(D6)** The condition only succeeds if `Head` and `Tail` are pointing to the same node. There are two possible situations where this can occur. First possibility is that the queue could be empty, what means that there is only one item in the list **(D7)**. In that case the algorithm returns **FALSE**. If this is not the case then the **Tail** pointer must be lagging behind. In order to fix this the `Tail` is getting set to the `next` pointer of the current node. This only happens if there was an enqueue but the `Tail` did not get updated yet **(E17)**
- **(D13)** The `CAS` only fails if the `Head` pointer is not pointing to the value read in **(D2)**. Therefore another thread had successfully dequeued an item.

4 EVALUATION

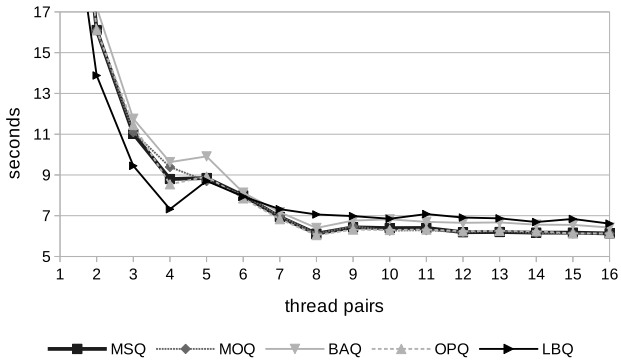
I benchmarked the Michael-Scott Queue (MSQ) against newer lock-free queues. These queues consists of the Moir Queue (MOQ) [17], which is a slightly optimized MSQ, the Basket Queue (BAQ) [13], which has buckets as nodes which can store multiple items, and the Optimistic Queue (OPQ) [15], which implements a doubly-linked list and enqueues to the head instead of the tail. All of these queue implementations are provided by the libcds library written by Max Khizhinsky [14]. The evaluation also covers a lock-based queue (LBQ), whose implementation uses a simple mutex wrapper around the `std::deque` from *C++*.



(a) Original test with 2 processes per processor



(b) *iteration_count* = 150



(c) *iteration_count* = 400
The y-axis is cropped to 5 - 17

Figure 2: Benchmarks

For the benchmark i slightly modified the methodology of the original paper [19]. Every thread is enqueuing an item into the shared queue, then doing some ”other” work, dequeuing an item and finally doing the same ”other” work. Every thread executes the loop $\lfloor num\_items/thread\_count \rfloor$ times. The ”other” work consists of spinning in an empty loop $iteration\_count$ times. On one hand i chose 150 for the $iteration\_count$ to simulate a light workload and on the other hand i chose 400 to simulate a higher workload. High values are not interesting since the benchmark is supposed to stress enqueues and dequeues. To achieve a high runtime, the number of

items was set to 20′000′000. In the original benchmark, the total runtime of the ”other” work was subtracted from the total runtime of the benchmark. Since it is very hard to accurately measure the runtime of small and fast code segments, I choose not to subtract it.

The processor used in this benchmark is an Intel® Xeon® E3-1230 v2, which is a 4 core processor with Hyper-Threading. The tests got compiled with g++ 6.2 with the -03 flag and were run on Ubuntu 16.10.

The graphs in Figure 2 shows benchmarks results, where 2a is one of the three original graphs from the paper which had slightly different settings than this benchmark. The ”other” work consisted of spinning in a loop for about 6 µs, the item count was 1′000′000 and the tests ran on a 12 core processor. The MSQ was tested against three lock-free queue implementations: Valois queue(VAQ) [22], Mellor-Crummey queue (MCQ) [8] and the queue proposed by Prakash, Lee and Johnson(PLJ). For a better comparison to my benchmarks, I picked the graph which is running two processes per processor. This assures that the workload is not profiting from core exclusivity nor is overloading them. The proposed MSQ outperformed the other queues in every aspect, be it on a single core or on twelve. The huge performance impact that every queue got on two cores from one is because of all the cache misses that the processors are experiencing now [19]. Before that, all queues were completely cached on the cores L1-Cache and did not have to synchronize with other caches. However the algorithm run-time drops back to 6.7 seconds at three cores, since it already experienced cache problems and no other performance impact happens. The completion time keeps rising only a little bit and scales linealy to 12 processors. The PLJ and VAQ are analogue to the ms queue. They just have a higher constant addend in the run time complexity. Only the MC queue did not scale lineally, but instead seems to rise exponentially.

Note that in the following two graphs, 2b and 2c, the workload did not get subtracted. Figure 2b shows time consumption for a light workload. The lock-based queue is more than 30% faster than the lock-free ones when only 1 to 2 threads are running. The reason behind it being faster is the much lower operation count required for the thread to perform an enqueue or dequeue. It only has to lock a shared mutex, enqueue/dequeue the item and finally unlock the mutex, whereas the lock-free variants do not use mutexes, but have to check multiple conditions and handle them properly. Additionally if one or more conditions fail, the thread has to loop and start over to try another time.

Every algorithm scales well and almost has a perfect speedup up to 8 threads. However the time to completion for the LBQ does not decrease after using 3 thread pairs. Instead it takes more time to complete the test, since the contention for the lock is bottlenecking more than the speedup of parallelism is benefiting. This problem is compelling the completion time of the simple_lock queue to rise after 3 thread pairs and stagnates at around 7.3 seconds, which is over twice the time of all lock-free queues.

When the workload increases, as shown in Figure 2c, the differences are getting smaller, since the thread spents the majority of the time on ”other” work instead of modifying the queue. The lock-based queue is once again faster when the contention for the lock is low. The LBQ gets overtaken by the lock-free queues at 6 thread pairs instead of 4. In general less contention for the lock, the faster is the lock-based one, because of fewer operations.

Two spikes occur during the test. The first spike takes place between 4 and 5 thread pairs, after the processor has to run the thread on a hyper-threaded core instead of a ”real” core. The second spike is after 8 cores when there are no more logical cores left to run all threads parallel. The benchmarks were run with up to 16 thread pairs to see how the algorithms are reacting when the thread is not dedicated to a logical core.

All in all, the MSQ is a relatively fast lock-free data structure despite its age. In comparison, the modern queues are only as good as the MSQ. The bad performance of modern queues is because the benchmark only simulates a specific workload. For example, the maximum of the queues size is only as large as the thread-count, the enqueuers and dequeuers were mixed and balanced. The modern queues have better performance.

The BAQ always has a higher runtime than the other lock-free queues. A basket is an ordered list of groups [13]. When there is high contention there is a possibility that many threads can work on other baskets, which decreases the contention for the single items, resulting in faster operations. The basket-queue has a dynamic number of baskets, which increases if a contention was detected and decreases when there are no more items left in that basket. Since there are very few baskets in the test, the advantages it gets from it could not overcome the overhead of the baskets management.

The OPQ is as fast as the MSQ. To get better performance, the OPQ enqueues on the `Head` and dequeues from the `Tail`. This way it saves one CAS, which takes an order of magnitude longer than a simple store, because it needs to take exclusive ownership of a cache line and completely flushes the write buffer of a processor [15]. To fix the reverse direction of the dequeues, the nodes need to have a `prev` pointer to the predecessor. The `prev` pointer is not stored atomically, so there is the possibility of corruption. If a *prev* pointer is found to be inconsistent a `fixList` method will resolve the issue by iterating from the `Head` to the `Tail` and setting the correct `prev` pointers. The result will be the corrected queue, since the *next* pointers are guaranteed to be consistent. Since `fixList` needs to iterate through the whole list, the cost of calling this function is relatively high, but it is compensated by the fact that the calling frequency to the method is very low. `prev` only gets inconsistent when there are long delays. The test can contain some delays and this is why the OPQ is not faster than the MSQ. The authors of the OPQ paper observed the same result in this workload scenario [15].

The MOQ is a simple ms-queue where the CAS is replaced by special load-links and store-conditionals. The x86 Platform does not have these primitives, thus it has to get emulated with CAS [9]. Even then, the performance is up par with the ms-queue

5 SUMMARY

Many people claim that lock-free algorithms can solve all their problems, but it is by no means a silver bullet. All in all the lock-free data structures are a great tool to avoid serious hazards when dealing with parallel algorithms. They are also performing better than lock-based algorithms in some scenarios as seen in the evaluation where a simple lock-based queue outperformed all lock-free implementations up to 6 thread pairs. This is why someone should always measure before replacing lock-based algorithms and even

then the right type of lock-free algorithm should be chosen, since many are specialized to a specific use-case.

In summary the Michael-Scott queue is an elegant and easy to understand lock-free queue. Despites its age, it is still faster than newer lock-free queues in a workload with balanced enqueuers and dequeuers.

REFERENCES

[1] ARM Architecture Reference Manual. https://www.scss.tcd.ie/~waldroj/3d1/arm_arm.pdf. Accessed: 2017-01-16.

[2] C++ International Standard. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4296.pdf. Accessed: 2017-01-16.

[3] Chapter 03. http://users.cis.fiu.edu/~sadjadi/Teaching/Operating%20Systems/Lectures/Chapter-03.ppt. Accessed: 2017-01-16.

[4] Definitions of Non-blocking, Lock-free and Wait-free. https://www.justsoftwaresolutions.co.uk/threading/non_blocking_lock_free_and_wait_free.html. Accessed: 2017-01-17.

[5] The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. http://www.gotw.ca/publications/concurrency-ddj.htm. Accessed: 2017-01-16.

[6] Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2 (2A, 2B, 2C & 2D): Instruction Set Reference, A-Z. http://www.intel.de/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-instruction-set-reference-manual-325383.pdf. Accessed: 2017-01-16.

[7] Lock-Free Code: A False Sense of Security. http://www.drdobbs.com/cpp/lock-free-code-a-false-sense-of-security/210600279. Accessed: 2017-01-09.

[8] 1987. Definitions of Non-blocking, Lock-free and Wait-free. https://www.cs.rice.edu/~johnmc/papers/cqueues-mellor-crummey-TR229-1987.pdf. Accessed: 2017-01-17.

[9] James H. Anderson and Mark Moir. 1995. Universal Constructions for Multi-object Operations. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing (PODC '95)*. ACM, New York, NY, USA, 184–193. DOI: http://dx.doi.org/10.1145/224964.224985

[10] David Dice, Danny Hendler, and Ilya Mirsky. 2013. Lightweight Contention Management for Efficient Compare-and-swap Operations. In *Proceedings of the 19th International Conference on Parallel Processing (Euro-Par'13)*. Springer-Verlag, Berlin, Heidelberg, 595–606. DOI: http://dx.doi.org/10.1007/978-3-642-40047-6_60

[11] E. W. Dijkstra. 1971. Hierarchical ordering of sequential processes. *Acta Informatica* 1, 2 (1971), 115–138. DOI: http://dx.doi.org/10.1007/BF00289519

[12] Maurice P. Herlihy and Jeannette M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (July 1990), 463–492. DOI: http://dx.doi.org/10.1145/78969.78972

[13] Moshe Hoffman, Ori Shalev, and Nir Shavit. 2007. *The Baskets Queue.* Springer Berlin Heidelberg, Berlin, Heidelberg, 401–414. DOI: http://dx.doi.org/10.1007/978-3-540-77096-1_29

[14] Max Khizhinsky. A C++ library of Concurrent Data Structures. commit-id: 03601c4b049873992f30f269b131e0f0f19742e5. https://github.com/khizmax/libcds. Accessed: 2016-12-26.

[15] Edya Ladan-Mozes and Nir Shavit. 2008. An optimistic approach to lock-free FIFO queues. *Distributed Computing* 20, 5 (2008), 323–341. DOI: http://dx.doi.org/10.1007/s00446-007-0050-0

[16] Butler W. Lampson and David D. Redell. 1980. Experience with Processes and Monitors in Mesa. *Commun. ACM* 23, 2 (Feb. 1980), 105–117. DOI: http://dx.doi.org/10.1145/358818.358824

[17] Victor Luchangco, Mark Moir, and Nir Shavit. 2003. *On the Uncontended Complexity of Consensus.* Springer Berlin Heidelberg, Berlin, Heidelberg, 45–59. DOI: http://dx.doi.org/10.1007/978-3-540-39989-6_4

[18] Maged M. Michael. 2004. Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects. *IEEE Trans. Parallel Distrib. Syst.* 15, 6 (June 2004), 491–504. DOI: http://dx.doi.org/10.1109/TPDS.2004.8

[19] Maged M. Michael and Michael L. Scott. 1996. Simple, Fast, and Practical Non-blocking and Blocking Concurrent Queue Algorithms. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing (PODC '96)*. ACM, New York, NY, USA, 267–275. DOI: http://dx.doi.org/10.1145/248052.248106

[20] Susan Owicki and Leslie Lamport. 1982. Proving Liveness Properties of Concurrent Programs. *ACM Trans. Program. Lang. Syst.* 4, 3 (July 1982), 455–495. DOI: http://dx.doi.org/10.1145/357172.357178

[21] Andrew Tanenbaum. 2009. Modern operating systems. (2009).

[22] John David Valois. 1996. *Lock-free Data Structures.* Ph.D. Dissertation. Troy, NY, USA. UMI Order No. GAX95-44082.

[23] Dieter Zöbel. 1983. The Deadlock Problem: A Classifying Bibliography. *SIGOPS Oper. Syst. Rev.* 17, 4 (Oct. 1983), 6–15. DOI: http://dx.doi.org/10.1145/850752.850753