Lock-Free Techniques for Concurrent Access to Shared Objects

Dominique Fober, Yann Orlarey, Stéphane Letz

This is a revised version of the previously published paper. It includes a contribution from Shahar Frank who raised a problem with the *fifo-pop* algorithm.
Revised version date: sept. 30 2003.

Lock-Free Techniques for Concurrent Access to Shared Objects

Dominique Fober **Yann Orlarey** **Stephane Letz**
Grame - Centre National de Création Musicale
9, rue du Garet BP 1185
69202 LYON CEDEX 01
Tél +33 (0)4 720 737 00 Fax +33 (0)4 720 737 01
[fober, orlarey, letz]@grame.fr

Abstract

Concurrent access to shared data in preemptive multi-tasks environment and in multi-processors architecture have been subject of many works. Proposed solutions are commonly based on semaphores which have several drawbacks. For many cases, lock-free techniques constitute an alternate solution and avoid the disadvantages of semaphore based techniques. We present the principle of these lock-free techniques with the simple example of a LIFO stack. Then, based on Michael-Scott previous work, we propose a new algorithm to implements lock-free FIFO stacks with a simple constraint on the data structure.

1. Introduction

A shared data structure is lock-free if its operations do not require mutual exclusion: if a process is interrupted in the middle of an operation, it will not prevent the other processes from operating on that object. Lock-free techniques avoid common problems associated with conventional locking techniques:

- priority inversion: occurs when a high-priority process requires a lock holded by a lower-priority process,
- convoying: occurs when a process holding a lock is descheduled by exhausting its quantum, by a page fault or by some other kind of interrupt. In this case, running processes requiring the lock are unable to progress.
- deadlock: can occur if different processes attempt to lock the same set of objects in different orders.

In particular locking techniques are not suitable in a real-time context and more generally, they suffer significant performance degradation on multiprocessors systems.
A lot of works have investigated lock-free concurrent data structures implementations [1, 2, 3, 4]. Advantages and limits of these works are discussed in [5]. We propose a new lock-free FIFO queue algorithm. It has been initially designed to be part of a multi-tasks, real-time MIDI operating system [6] in order to support an efficient inter-applications communication mechanism. Its implementation is based on Michael-Scott [4] but removes the necessary node allocation when enqueing a value, by introducing a simple constraint on the value data type structure.
The rest of this paper is organized as follow: section 2 introduces lock-free techniques with the example of a LIFO stack, section 3 presents our proposed lock-free FIFO queue algorithm, section 4 discuss the correctness of the FIFO operations and section 5 is dedicated to performances issues.

2. Lock-free LIFO stacks

A LIFO stack is made up of linked cells. A *cell* can be anything provided it starts with a pointer available to link together the cells of the stack (figure 1) and the structure of a LIFO is a simple pointer to the top of the stack (figure 2). The last cell of the LIFO always points to NULL.

```
structure cell {
        next:       a pointer to next cell
        value:      any data type
}
```

Figure 1: a cell structure

```
structure lifo {
        top:        a pointer to a cell
}
```

Figure 2: a lifo structure

Common operations on a LIFO are:

- lifo-init: to initialize the LIFO stack by setting the top pointer to NULL.
- lifo-push: to push a new cell on top of the stack
- lifo-pop: to pop the top cell of the stack

A naive and unsafe implementation of the push operation is presented in figure 3.

```
lifo-push (lf: pointer to lifo, cl: pointer to cell)
A1:     cl->next = lf->top          # set the cell next pointer to top of the lifo
A2:     lf->top = cl                # set the top of the lifo to cell
```

Figure 3: non-atomic lifo-push

Obviously, if a process trying to enqueue a new cell is preempted after A1 and if the top pointer has been modified when it resumes at A2, the push operation will not operate correctly.

2.1. Atomic operations implementation

To guaranty the correctness of the lifo operations, they should appear as taking instantaneously effect, as if they couldn't be interrupted. We'll further talk of "*atomic operation*" to refer to this property. A common approach is to make use of an atomic primitive such as *compare-and-swap* which takes as argument the address of a memory location, an expected value and a new value (figure 4). If the location holds the expected value, it is assigned the new value atomically. The returned boolean value indicates whether the replacement occurred.

```
compare-and-swap (addr: pointer to a memory location, old, new: expected and new values): boolean
    x = read (addr)
    if x == old
        write (addr, new)
        return true
    else
        return false
    endif
```

Figure 4: atomic compare-and-swap

The *compare-and-swap* primitive was first implemented in hardware in the IBM System 370 architecture [7]. More recently, it can be found on the Intel i486 [8] and on the Motorola 68020 [9]. A variation of the *compare-and-swap* primitive can also operate in memory on double-words. To differenciate between the two primitives in the following examples we'll refer to them with:

`CAS (mem, old, new)` for single word operations

where *mem* is a pointer to a memory location
old and *new* are the expected and the new value

and

`CAS2 (mem, old1, old2, new1, new2)` for double word operations

where *mem* is a pointer to a memory location
old1, *old2* and *new1*, *new2* are the expected and the new values

On PowerPC architecture, the *compare-and-swap* primitive may be implemented using the *load-and-reserve* instruction associated with a *store-conditional* instruction [10].
Using compare-and-swap, the operations on the stack are now implemented as shown in figure 5 and 6 and appear like atomic operations.

```
lifo-push (lf: pointer to lifo, cl: pointer to cell)
B1:     loop
B2:         cl->next = lf->top              # set the cell next pointer to top of the lifo
B3:         if CAS (&lf->top, cl->next, cl) # try to set the top of the lifo to cell
B4:             break
B5:         endif
B6:     endloop
```

Figure 5: lifo-push

```
lifo-pop (lf: pointer to lifo): pointer to cell
C1:     loop
C2:         head = lf->top                  # get the top cell of the lifo
C3:         if head == NULL
C4:             return NULL                 # LIFO is empty
C5:         endif
C6:         next = head->next               # get the next cell of cell
C7:         if CAS (&lf->top, head, next)   # try to set the top of the lifo to the next cell
C8:             break
C9:         endif
C10:    endloop
C11:    return head
```

Figure 6: lifo-pop

2.2. The ABA problem

However, the above implementation of the LIFO pop operations doesn't catch the ABA problem. Assume that a process is preempted while dequeing a cell after C6: severall concurrent push and pop operations may result in a situation where the top cell remains unchanged but points to a different next cell as shown in figure 7.

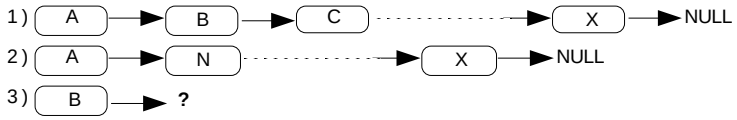

Figure 7: 1) state at the beginning of the pop operation,
2) state after preemption,
3) state after pop completion

The LIFO change won't prevent the CAS operation to operate in C7, allowing to put a wrong cell on top of the stack. The solution to the ABA problem consists in adding a count of the cells popped from the stack to the LIFO structure as shown in figure 8 and to make use of the CAS2 primitive.

```
structure lifo {
        top:        a pointer to a cell
        ocount:     total count of pop operations
}
```

Figure 8: extended lifo structure

The push operation remains unchanged and the pop operation is now implemented as shown in figure 9: it checks both for lifo top and output count changes when trying to modify the lifo top.

```
lifo-pop (lf: pointer to lifo): pointer to cell
        SC1:   loop
        SC2:       head = lf->top                          # get the top cell of the lifo
        SC2:       oc = lf->ocount                         # get the pop operations count
        SC3:       if head == NULL
        SC4:           return NULL                         # LIFO is empty
        SC5:       endif
        SC6:       next = head->next                       # get the next cell of cell
        SC7:       if CAS2 (&lf->top, head, oc, next, oc + 1)   # try to change both the top of the lifo and pop count
        SC8:           break
        SC9:       endif
        SC10: endloop
        SC11: return head
```

Figure 9: lifo-pop catching the ABA problem

3. Lock-free FIFO stacks

The FIFO queue is implemented as a linked list of cells with *head* and *tail* pointers. Each pointer have an associated counter, *ocount* and *icount*, wich maintains a unique modification count of operations on *head* and *tail*. The cell structure is the same as above (figure 1) and the fifo structure is shown in figure 10.

```
structure fifo {
        head:       a pointer to head cell
        ocount:     total count of pop operations
        tail:       a pointer to tail cell
        icount:     total count of push operations
}
```

Figure 10: the fifo structure

As in Michael-Scott [4] and Valois [3], the FIFO always contains a dummy cell, only intended to maintain the consistency. An empty FIFO contains only this dummy cell which points to an *end fifo marker* unique to the system: a trivial solution consists in using the FIFO address itself as a unique marker. All along the operations, *head* always points to the dummy cell which is the first cell in the list and *tail* always points to the last or the second last cell in the list. The double-word *compare-and-swap* increments the modification counters to avoid the ABA problem.

The queue consistency is maintained by *cooperative concurrency*: when a process trying to enqueue a cell detects a pending enqueue operation (tail is not the last cell of the list), it first tries to complete the pending operation before enqueing the cell. The dequeue operation also ensures that the *tail* pointer does not point to the dequeued cell and if necessary, tries to complete any pending enqueue operation. Figure 11 to 13 presents the commented pseudo-code for the fifo queue operations.

```
fifo-init (ff: pointer to fifo, dummy: pointer to dummy cell)
        dummy->next = NULL                          # makes the cell the only cell in the list
        ff->head = ff->tail = dummy                 # both head and tail point to the dummy cell
```

Figure 11: the fifo initialization operation

```
fifo-push (ff: pointer to fifo, cl: pointer to cell)
E1:     cl->next = ENDFIFO(ff)                      # set the cell next pointer to end marker
E2:     loop                # try until enqueue is done
E3:         icount = ff->icount                     # read the tail modification count
E4:         tail = ff->tail                         # read the tail cell
E5:         if CAS (&tail->next, ENDFIFO(ff), cl)   # try to link the cell to the tail cell
E6:             break;                              # enqueue is done, exit the loop
E7:         else    # tail was not pointing to the last cell, try to set tail to the next cell
E8:             CAS2 (&ff->tail, tail, icount, tail->next, icount+1)
E9:         endif
E10:    endloop
E11:    CAS2 (&ff->tail, tail, icount, cl, icount+1)    # enqueue is done, try to set tail to the enqueued cell
```

Figure 12: the fifo push operation

```
fifo-pop (ff: pointer to fifo): pointer to cell
D1:     loop                # try until dequeue is done
D2:         ocount = ff->ocount                     # read the head modification count
D3:         icount = ff->icount                     # read the tail modification count
D4:         head = ff->head                         # read the head cell
D5:         next = head->next                       # read the next cell
D6:         if  ocount == ff->oc                    # ensures that next is a valid pointer
                                                    # to avoid failure when reading next value
D7:             if  head == ff->tail                # is queue empty or tail falling behind ?
D8:                 if  next == ENDFIFO(ff)         # is queue empty ?
D9:                     return NULL                 # queue is empty: return NULL
D10:                endif
                    # tail is pointing to head in a non empty queue, try to set tail to the next cell
D11:                CAS2 (&ff->tail, head, icount, next, icount+1)
D12:            else if next <> ENDFIFO(ff)         # if we are not competing on the dummy next
D13:                value = next->value             # read the next cell value
D14:                if CAS2 (&ff->head, head, ocount, next, ocount+1) # try to set head to the next cell
D15:                    break                       # dequeue done, exit the loop
D16:            endif
D17:        endif
D18:    endloop
D19:    head->value = value                         # set the head value to previously read value
D20:    return head                                 # dequeue succeed, return head cell
```

Figure 13: the fifo pop operation

4 Correctness of the FIFO operations

Traditional sequential programs may be viewed as functions from inputs to outputs which may be specified as a pair consisting of a precondition describing the allowed inputs and postcondition describing the desired results for these inputs. However for concurrent programs, this approach is too limited and numerous work has been done for formal verification of concurrent systems. Although informal, two properties introduced by Lamport [11] are required for correctness of concurrent programs:

- *safety property*: states that “something bad never happens”,
- *liveness property*: states that “something good eventually happens”.

Formalizing this classification has been a main motivation for much of the work done on specification and verification of concurrent systems [12]. Formal methods successfully applied to sequential programs have also been extended to consider concurrent programming: Herlihy proposed a correctness condition for concurrent objects called “*Linearizability*” [13, 14]. It states that a concurrent computation is linearizable if it is equivalent to a legal sequential computation. An object (viewed as the agregate of a type, which defines a set of possible values, and a set of primitive operations), is linearizable if each operation appears to take effect instantaneaously at some point between the operation’s invocation and response. It implies that processes appear to be interleaved at the granularity of complete operations and that the order of non-overlapping operations is preserved.

Correctness of the FIFO operations formal proof is beyond the scope of this paper, however it will be examined according to the properties mentionned above.

3.1 Linearizability

The algorithm is linearizable because each operation takes effect at an atomic specific point: E5 for enqueue and D14 for dequeue. Therefore, the queue will never enter any transient unsafe state: along any concurrent implementation history, it can only swing between the two different states S0 and S1 illustrated in figure 14 and 15, which are acceptable and safe states for the queue:

Assuming a queue in state S0:

1) consider an *push* operation : as the queue state is S0, the atomic operation in E5 will succeed and the queue swings to S1 state. Then the atomic operation in E10 is executed: in case of success, the queue swings back to S0, in case of failure a successfull concurrent operation occurs on a S1 state and therefore by 3) and 4), the queue state should be S0.
2) consider a *pop* operation : if the queue is empty the operation returns in D9 and the state remains unchanged, otherwise the operation atomically executes D14: in case of success, the queue state remains in S0, in case of failure, a concurrent dequeue occured and as it has successfully operated on a S0 queue (by hypothesis) the final state remains also in S0.

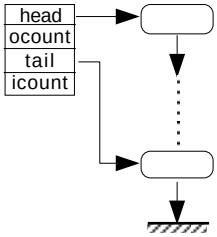

Figure 14: FIFO state S0

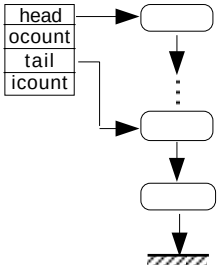

Figure 15: FIFO state S1

Assuming a queue in state S1:

3) consider an *enqueue* operation: as the queue state is S1, the operation atomically executes E8 and then loops. In case of success, the queue swings to S0 otherwise a concurrent dequeue or enqueue successfully occured and the operation loop should operate on a queue back to S0.
4) consider a *dequeue* operation: it is concerned by S1 only if *tail* and *head* points to the same cell which is only possible with a queue containing a single cell linked to the dummy cell. In this case, the operation atomically executes D11 and then loop. In case of success, the queue swings to S0 state. A failure means that a concurrent dequeue or enqueue successfully occured: a successfull dequeue swing the queue to S0 (but it is now empty) and a successfull enqueue too (by 3).

3.2 Safety

The main difference with the Michael-Scott algorithm [4] relies on the cells structure constraint, which allows to avoid nodes allocation and release. In fact, the cells memory management is now in charge of the FIFO clients and may be optimised to the clients requirements but it doesn't introduce any change in the algorithm functionning. Another difference is the modification counts to take account of the ABA problem: they are now associated only to the *head* and *tail* pointers to ensures atomic modifications of these pointers.
The safety properties satisfied by the Michel-Scott algorithm continue to hold ie:

- the linked list is always connected,
- cells are only inserted after the last cell in the linked list,
- cells are only deleted from the beginning of the linked list,
- head always points to the first node in the linked list,
- tail always points to a node in the linked list.

3.3 Liveness

The lock-free algorithm is non-blocking. This is asserted similarly to [4].
Assume a process attempting to operate on the queue:

- the process tries to enqueue a new cell: a failure means that the process is looping thru E8 and then another process must have succeeded in completing an enqueue operation or in dequeuing the tail cell.
- the process tries to dequeue a cell: a failure means that the process is looping thru D11 or D14. A failure in D11 means that another process must have succeeded in completing an enqueue operation or in dequeuing the tail cell. A failure in D14 means that another process must have succeeded in completing a dequeue operation.

5 Performances

Performances have been measured both for the lock-free LIFO compared to a lock-based implementation and for the lock-free FIFO algorithm compared to a lock-based implementation and to the Michael Scott algorithm. The bench has been made on a Bi-Celeron 500MHz SMP station running a 2.4.8 Linux kernel. It measures the time required for 1 to 8 concurrent threads to perform 500 000 x 6 concurrent push and pop operations on a shared LIFO or FIFO queue. The code executed by each thread is shown in Figure 16. The lock-based implementation makes use of the pthread mutex API with a statically allocated mutex.

```
long stacktest (long n) {
        cell*    tmp[6]; long i; clock_t t0, t1;

        t0 = clock();
        while (n--) {
                for (i=0; i<6; i++) tmp[i] = pop(&gstack);
                for (i=0; i<6; i++) push(&gstack, tmp[i]);
        }
        t1 = clock();
        return t1-t0;
}
```

Figure 16: the bench task.

The integrity of the queue was checked after the threads had completed their operations. Results are presented by figures 17 and 18 as average time (in µs) to perform a paired pop and push operations.

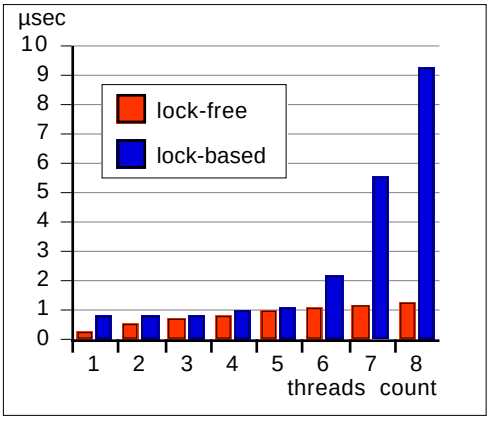

Figure 17: lock-free LIFO compared to lock-based.

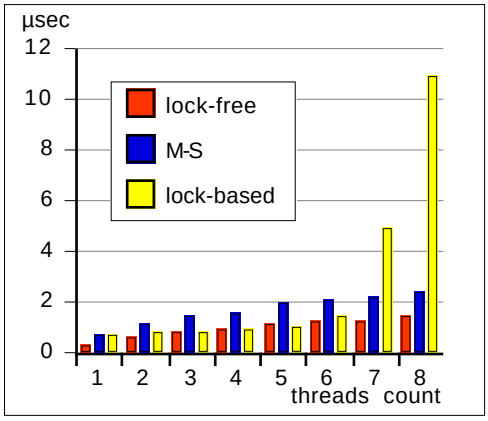

Figure 18: lock-free FIFO compared to Michael-Scott and lock-based.

In the Michael-Scott implementation, nodes allocation is performed using a statically allocated set of nodes and an index atomically incremented to access the next free node in the table (figure 19). The node table size prevents multiple node allocation. A node release is implicit and needs no additionnal operation.

```
node_t * new_node() {
        static long index = 0;
        long next, i;
        do {
                i= index;
                next = (i >= MAXNODES) ? 0 : i+1;
        } while (!CAS(&index, i, next));
        return &nodes[next];
}
```

Figure 19: node allocation in Michael Scott implementation

Comparison between the lock-free and the lock-based operations shows the following:

- in lack of concurrency (single thread), the lock-based operations are more than 2 times more expensive than the lock-free operations,
- performances are roughly the same for a few concurrency (2 to 5 threads),
- lock-based operations cost dramatically increases in medium-high concurrency to reach more than 7 times the lock-free cost for 8 concurrent threads.

Comparison between our lock-free FIFO algorithm and the Michael-Scott algorithm shows the following:

- for a single thread, the Michael-Scott operations cost is roughly 2 times more expensive
- when the concurrency increases, this cost is converging to 1.6 times our solution cost.

This behavior may be explained by the necessity to allocate the nodes pushed on the stack and to handle additionnal concurrency while performing the allocation.

7. Conclusion

Lock-free techniques are clearly more suited to real-time applications than lock-based techniques. They are more efficient and avoid priority inversion which is a major drawback in a real-time context. We have showed how to apply this technique to simple objects like LIFO and FIFO queues associated with basic operations. Finaly, our proposed new algorithm for FIFO operations improves existing algorithms with a simple constraint on the value data structure which allows more efficient specialized implementations. Although limited to LIFO and FIFO queues, the presented lock-free techniques may be very useful to solve situations commonly encountered in the musical domain where events have frequently to be queued while waiting for their deadline.

8. Aknowledgements

Thanks to Shahar Frank <fesh@exanet.com> who reported the *fifo-pop* problem and for its suggested solution.

References

[1] James H. Anderson, Srikanth Ramamurthy and Kevin Jeffay. “Real-time computing with lock-free shared objects.” ACM Transactions on Computer Systems Vol. 15, No. 2, May 1997, pp. 134 - 165

[2] M. Herlihy. “A methodology for implementing highly concurrent data objects.” ACM Trans. Program. Lang. Syst. 1993, Vol. 15, No.5, pp. 745–770.

[3] John D. Valois. “Implementing Lock-Free Queues.” Proceedings of the Seventh International Conference on Parallel and Distributed Computing Systems, Las Vegas, October 1994, pp. 64-69

[4] M. M. Michael and M. L. Scott. “Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms.” 15th ACM Symp. on Principles of Distributed Computing (PODC), May 1996. pp. 267 - 275

[5] M. M. Michael and M. L. Scott. “Nonblocking Algorithms and Preemption-Safe Locking on Multiprogrammed Shared Memory Multiprocessors.” Journal of Parallel and Distributed Computing, 1998, pp. 1-26.

[6] Y. Orlarey, H. Lequay. “MidiShare : a Real Time multi-tasks software module for Midi applications” Proceedings of the International Computer Music Conference 1989, Computer Music Association, San Francisco, pp.234-237

[7] International Business Machines Corp. “System / 370 Principles of Operation” 1983

[8] Intel Corporation. “i486 Processor Programmer’s reference Manual” Intel, Santa Clara, CA, 1990

[9] Motorola. “MC68020 32-Bit Microprocessor User’s Manual” Prentice-Hall, 2nd edition, 1986

[10] IBM Microelectronics, Motorola. “PowrPC 601 RISC Microprocessor User’s Manual”, 1993

[11] L. Lamport. “Proving the Correctness of Multiprocess Programs.” IEEE Transactions on Software Engineering SE-3, 2 (March 1977), 125-143.

[12] R. Cleaveland, S.A. Smolka & al. “Strategic Directions in Concurrency Research.” ACM Computing Surveys, Vol. 28, No. 4, December 1996, pp. 607-625

[13] M. P. Herlihy, J. M. Wing. “Axioms for concurrent objects.” In Proceedings of the 14th ACM Symposium on Principles of Programmmg Languages,Jan. 1987, pp. 13-26.

[14] M. P. Herlihy, J. M. Wing. “Linearizability: A Correctness Condition for Concurrent Objects.” ACM Transactions on Programming Languages and Systems, Vol. 12, No. 3, July 1990, pp. 463-492.