Obstruction-free synchronization: Double-ended queues as an example [1]

Maurice Herlihy, Victor Luchangco and Mark Moir

ICDCS 2003

presented by: Martin Aigner

Department of Computer Sciences, University of Salzburg

November 25, 2010

Welcome! Why are we here today?

Why do we need concurrent programs?

It is unlikely that chips will get faster clocks (form factor, heat, energy). Today the industry focuses on speedup by parallel execution on multi-processor systems.

n threads on n cpus. Where is the problem?

No problem for independent threads. But:
Similar to interleaved execution on single-cpu systems: Concurrent access to shared data structures may cause *conflicts*.

Definition: concurrent system

A concurrent system consists of a collection of n sequential threads (processes) that communicate through shared objects.

Example: a bottleneck on shared resources

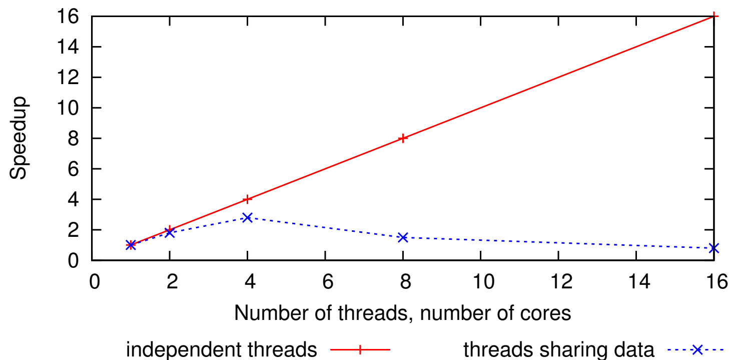


Figure: Speedup degradation on shared data with increasing number of threads

The cause of the bottleneck

Concurrent access needs to be managed!

Threads need some sort of synchronization to avoid race conditions.
In 1965, Dijkstra proposed semaphores. It solved the producer-consumer problem and enabled synchronization by critical sections.

Synchronization creates the bottleneck

Only one thread at a time may modify a memory location. If a second thread attempts to do the same, the OS preempts the thread and puts it to blocked state. This creates a massive amount of overhead. (context switch, scheduler overhead...)

Let us summarize the problems with locks

Terminology

- A *(b)locking* thread holds a lock.
- A *blocked* thread waits for a lock to be released.

What's wrong with using locks?

Locks cause blocking: Some other (blocked) threads have to wait until a (set of) lock(s) is released.

Deadlocks: The locking thread dies/loops/blocks... $\Rightarrow$ blocked threads may wait forever

Priority inversion: Blocked high-priority threads have to wait for a locking lower-priority thread.

Many more: Scheduler overhead, starvation, livelocks...

The problems with locks

The common source of the problems: (simplified)

Blocking threads hinder blocked threads to make progress.

We want (guaranteed) progress!

Intuitively: “No matter how my threads are interleaved, I want my system to make progress.”

What IS progress?

- *Progress* is a property defined by the developer.
- *Progress guarantee* is a property of a synchronization technique/algorithm.

blocking vs. non-blocking

Definition: Non-blocking algorithm

A synchronization algorithm is non-blocking if threads do not have their execution indefinitely postponed by mutual exclusion.

Example

Thread A holds a lock and dies. The other threads will never be able to enter the critical section.

What to use instead of locks?

The GNU C-library provides non-blocking versions of the mutex_lock and unlock functions. However, literature often refers to low-level operations such as compare and swap (cas) or load-locked/store-conditional.

non-blocking synchronization primitives [1] [2]

Listing 1: C implementation for cas on x86

```
static inline int cas(
    volatile int *atomic, int oldval, int newval)
{

    int result;

    asm volatile("lock; cmpxchgl %2, %1"
            : "=a" (result), "=m" (*atomic)
            : "r" (newval), "m" (*atomic), "0" (oldval)
            );

    return result;
}
```

[1] http://www.intel.com/products/processor/manuals/index.htm

[2] http://www.ibiblio.org/gferg/ldp/GCC-Inline-Assembly-HOWTO.html

Lockfreedom

Definition: Lockfreedom[2]

Some thread is guaranteed to complete an operation after the whole system took finitely many steps.

In other words

If you schedule enough steps across *all* threads, *one* of them will make progress.

In the sense of fault tolerance

Some thread will always make progress despite arbitrary failures/halting of other threads.

Waitfreedom

Definition: Waitfreedom [2]

Each thread is guaranteed to complete an operation after the whole system took finitely many steps.

In other words

If you schedule enough steps of *any* thread, *it* will make progress.

In the sense of fault tolerance

All non-halted thread will make progress despite arbitrary failures/halting of other threads.

Why do we need a weaker progress guarantee?

Not using locks $\not\Rightarrow$ guaranteed progress

Let thread A execute the following C code where *addr represents a shared object:

```
do {
  oldval = *addr;                             //read memory
  newval = calulation(oldval);                //perform action
} while (!cas(addr, oldval, newval));         //try to store newval
```

Now always preempt A while working on **calculation()**. Let thread B change ***addr** to any value $\neq$ oldval. Thread A will never finish the loop.

How is progress guarantee achieved?

Progress guarantees typically rely on complex (and inefficient) “helping” mechanisms; the strategy in the case that **cas** fails.

Conceptual problems and ideas

Mixing up correctness and progress

Authors opinion: "Ensuring progress should be a problem of engineering, not of mathematics."
The resulting complexity creates a barrier to the widespread of nonblocking synchronization.
A clean separation between correctness and progress promises simpler (more efficient, more effective) algorithms.

Tradeoff: complexity vs. progress guarantee

Weakening the guarantee of progress leads to less complexity.

Obstructionfreedom

Definition: Obstructionfreedom

A synchronization technique is obstruction-free if it guarantees progress for *any* thread that eventually executes in *isolation*.

In other words

If you let *any* thread run *alone for enough steps*, it will make progress.

In the sense of fault tolerance

Same as with lockfreedom: A thread will always make progress despite arbitrary failures/halting of other threads.

Where is the catch?

There is no progress guarantee in the presence of contention!

How to implement obstruction-free algorithms

Problem: make progress in the presence of contention

We need to provide a mechanism to deal with contention, e.g. randomized exponential backoff, queuing and timestamping approaches...
The threads agree amongst themselves to some sort of order.

Wait a minute... isn't that the same as using locks?

For simplistic applications: yes! But:
We have much more freedom to design sophisticated contention control mechanisms **without** jeopardizing correctness by interrupting an operation at any time.

The resulting algorithm will look like:

- a lightweight synchronization part
- a (maybe) heavyweight contention control part

Assumptions about the system

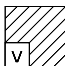
Case: No contention

In the absence of contention the heavyweight part will rarely be invoked. In contrast, in lock-free and wait-free implementations the mechanisms used for progress guarantees are always invoked.

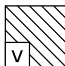
Case: High contention

Obstruction-free, wait-free and lock-free operations may cause similar overhead.

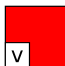
Deque data representation



LN: left-NULL object



RN: right-NULL object



data object

Figure: Array invariant: at least one LN followed by zero or more data fields followed by at least one RN

Examples for a valid deque

Figure: Array of size MAX+2 can contain MAX data objects. A[0] is always LN and A[MAX+1] is always RN

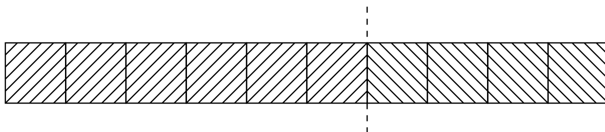


Figure: The double ended queue is empty

The oracle-function

An oracle(side) function to simplify the presentation

oracle(side) takes as a parameter either *left* or *right* and returns an array index. In case of left the function returns the rightmost left-NULL position and vice versa.

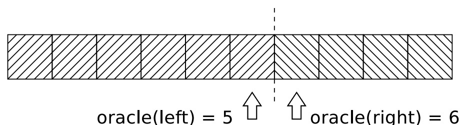


Figure: Semantics of the oracle-function

oracle may be incorrect

One assumption: oracle(left) always returns an index $\in$ [0,MAX] and oracle(right) return an index $\in$ [1,MAX+1]

Deque operations: rightpush(n)

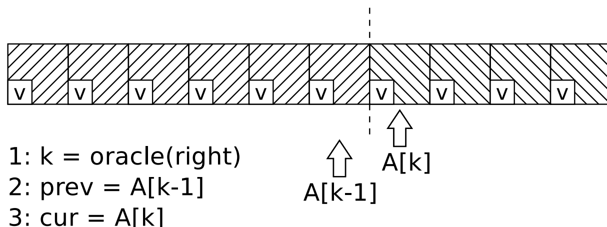
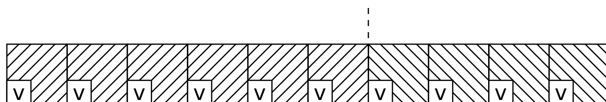


Figure: Ask oracle where to insert n

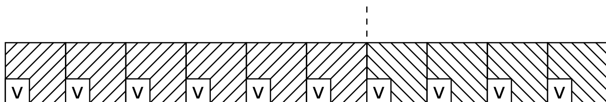
Deque operations: rightpush(n)




```
1: k = oracle(right)
2: prev = A[k-1]
3: cur = A[k]
4: if (prev.data != RN and cur.data == RN) {
5:   //oracle was right
6:} else {
7:  //oracle was wrong; loop
8:}
```

Figure: Check if oracle's answer was correct

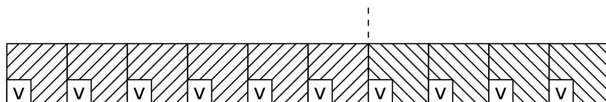
Deque operations: rightpush(n)




```
1: k = oracle(right)
2: prev = A[k-1]
3: cur = A[k]
4: if (prev.data != RN and cur.data == RN) {
5:   if (k = MAX+1) return "full";
6:}
```

Figure: Check the boundaries

Deque operations: rightpush(n)



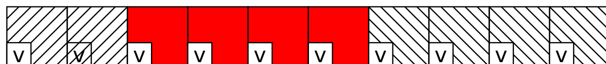
A[k]

A[k-1]

```
1: k = oracle(right)
2: prev = A[k-1]
3: cur = A[k]
4: if (prev.data != RN and cur.data == RN) {
5:   if (k = MAX+1) return "full";
6:   if CAS(&A[k-1], prev, <prev.data,prev.v+1>)
7:     if CAS(&A[k], cur, <n,cur.v+1>
8:       return "OK";
9:}
```

Figure: Perform synchronized push

Deque operations: rightpop()



1: k = oracle(right)
2: cur = A[k-1]
3: next = A[k]

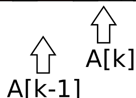
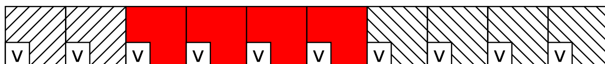


Figure: Ask oracle where the leftmost RN is

Deque operations: rightpop()

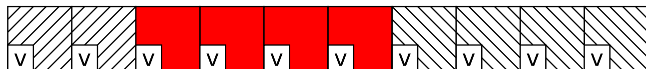


A[k]
A[k-1]

```
1: k = oracle(right)
2: cur = A[k-1]
3: next = A[k]
4: if (cur.data != RN and next.data == RN) {
5:   //oracle was right
6: } else {
7:   //oracle was wrong; loop
8: }
```

Figure: Check if oracle's answer was correct

Deque operations: rightpop()



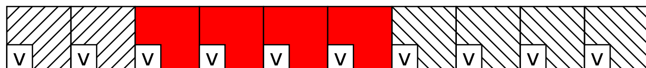
A[k]

A[k-1]

```
1: k = oracle(right)
2: cur = A[k-1]
3: next = A[k]
4: if (cur.data != RN and next.data == RN)
5:   if (cur.data == LN and A[k-1] == cur) return "empty";
```

Figure: Check for emptiness

Deque operations: rightpop()



A[k-1] A[k]

```
1: k = oracle(right)
2: cur = A[k-1]
3: next = A[k]
4: if (cur.data != RN and next.data == RN) {
5:   if (cur.data == LN and A[k-1] == cur) return "empty";
6:   if CAS(&A[k], next, <RN,next.v+1>)
7:     if CAS(&A[k-1], cur, <RN,cur.v+1>
8:       return cur.data;
9:}
```

Figure: Perform synchronized pop

Correctness considerations

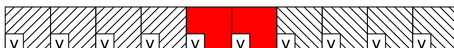
Basic Idea for correctness:

Version numbering in adjacent locations: We check that the neighbouring location contains the appropriate value, and we increment its version number between reading the location to change and attempting to change it.

Correctness considerations

Claim 1: rightpush(n)

At the moment that line 7 successfully changes A[k].data from RN to n, A[k-1].data contains a non-RN value (i.e. either LN or any data value)



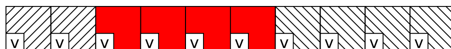
A[k-1] A[k]

```
1: k = oracle(right)
2: prev = A[k-1]
3: cur = A[k]
4: if (prev.data != RN and cur.data == RN) {
5:   if (k = MAX+1) return "full";
6:   if CAS(&A[k-1], prev, <prev.data,prev.v+1>)
7:     if CAS(&A[k], cur, <n,cur.v+1>
8:       return "OK";
9:}
```

Correctness considerations

Claim 2: rightpop

At the moment that line 7 successfully changes A[k-1].data from some value (cur.data) to RN, A[k].data contains RN.



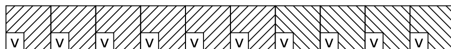
A[k-1] A[k]

```
1: k = oracle(right)
2: cur = A[k-1]
3: next = A[k]
4: if (cur.data != RN and next.data == RN) {
5:   if (cur.data == LN and A[k-1] == cur) return "empty";
6:   if CAS(&A[k], next, <RN,next.v+1>)
7:     if CAS(&A[k-1], cur, <RN,cur.v+1>
8:       return cur.data;
9:}
```

Correctness considerations

Claim 3: rightpop

If a rightpop returns “empty”, then at the moment it executed line 3, A[k].data == RN and A[k-1].data == LN held.



A[k-1] A[k]

```
1: k = oracle(right)
2: cur = A[k-1]
3: next = A[k]
4: if (cur.data != RN and next.data == RN) {
5:   if (cur.data == LN and A[k-1] == cur) return "empty";
6:   if CAS(&A[k], next, <RN,next.v+1>)
7:     if CAS(&A[k-1], cur, <RN,cur.v+1>
8:       return cur.data;
9:}
```

The oracle function

Requirements for linearizability:

oracle returns an index of the appropriate range depending on the parameter.
oracle(left) always returns an index $\in$ [0,MAX] and oracle(right) return an index $\in$ [1,MAX+1]

Requirements for obstructionfreedom:

We require that oracle is *eventually accurate if repeatedly invoked in the absence of interference*.

When is oracle accurate?

Oracle will return the right result, if any of the operations executes one iteration without interference.

Obstructionfreedom revisited

Definition: Obstructionfreedom

A synchronization technique is obstruction-free if it guarantees progress for *any* thread that eventually executes in *isolation*.

In other words

If you let *any* thread run *alone for enough steps*, it will make progress.

Are we convinced this can work?

A single iteration of any deque operation is quite short. It will be efficient in the absence of heavy contention. In case of heavy contention, one can experiment with the contention management without modifying the non-blocking algorithm.

References

[1] Maurice Herlihy, Victor Luchangco, and Mark Moir. Obstruction-free synchronization: Double-ended queues as an example. *Distributed Computing Systems, International Conference on*, 0:522, 2003. ISSN 1063-6927.

[2] Maurice Herlihy. A methodology for implementing highly concurrent data objects. *ACM Trans. Program. Lang. Syst.*, 15:745–770, November 1993. ISSN 0164-0925.