Distributed Algorithms
Lecture 10

Lecture by Dr. Gadi Taubenfeld
Typeset by Steven Karas

2017-06-14
Last edited 17:58:11 2017-06-14

Disclaimer These lecture notes are based on the lecture for the course Distributed Algorithms, taught by Dr. Gadi Taubenfeld at IDC Herzliyah in the spring semester of 2017. Sections may be based on the lecture slides and accompanying book written by Dr. Gadi Taubenfeld.

Agenda

- Barrier synchronization
- Lock-free concurrent queue implementation

The next homework will be published either tonight or tomorrow.

1 Concurrent data structures

Using locks Traditional data structures need locks to restrict access. This simplifies the programming model, but creates a sequential bottleneck, even when there are theoretically no conflicts between accesses.

Freedom is measured on the scale of deadlock or livelock freedom, through starvation freedom, towards FIFO+deadlock freedom. However, this is always under a model where there are no failures.

Without locks Lock-free data structures are typically more complex, but have several advantages. They are resilient to failures, provide weak progress conditions, and allow concurrent access. However, they can be significantly more complex and typically take up much more space.

Freedom is measured on the scale from wait freedom, through non-blocking, towards obstruction freedom.

Wait-freedom Every non-failing process that begins an operation, will finish its operation.

Non-blocking For any set of processes that begin an operation, at least 1 will eventually finish the operation.

Obstruction-freedom I didn't catch the formal definition of this.

1.1 Consistency conditions for concurrent objects

For example, a queue has two conditions: Items are enqueued, and the item that has been in the queue the longest will be returned exactly once when dequeued. This implies that ever item that is enqueued is eventually dequeued.

Notation Invocation is when a process begins an operation.

The Linearization Point is when an operation is marked for purposes of linearizability.

Response is when the operation ends.

1.1.1 Linearizability

If we can provide a sequential specification of a concurrent execution that is valid, then the data structure provides sequential consistency.

Moreover, if we take the space of all valid sequential executions, we can map each concurrent execution to at least one valid sequential execution.

1.1.2 Sequential Consistency

Consistency model where the order of operations taken by each process are well defined, but the ordering between processes is not defined. In other words, the real time ordering of operations is not defined.

Interestingly, we can compose individual sets of otherwise sequentially consistent operations across multiple data structures to produce specifications that are not sequentially consistent. An example of this is given on slides 28-32.

1.2 Linearizable Non-Blocking Queue

We'll study the design presented by M. M. Michael and M. L. Scott in 1996. This is the design provided in the JDK concurrent package.

Impossibility with atomic registers We've proved that there is a wait-free consensus algorithm for 2 processes, using queues and atomic registers[1].

However, we've also proven that there is no wait-free (or non-blocking) consensus algorithm for 2 processes using only atomic registers.

This implies that it is impossible to implement a queue for two or more processes from atomic registers.

1.2.1 Primitive: Compare and Swap

$$\text{CAS}(A, B, C)$$

If $A = B$, then $A := C$; return true. Otherwise return false.

The ABA problem Examine the following program:

```
local := A
...local is changed to B...
...local is changed back to A...
CAS(A, local, 100)
```

In this case, the operation succeeds, even though we may not have wanted it to.

A simple solution to this is to add a tag to every CAS target, such that it has two fields: value and tag.

The algorithm The shared structure is two pointers: head and tail. A dummy node is used as the head, which is never deleted.

Enqueue

```
1  done = false
2  until done:
3    ltail = tail
4    lnext = ltail.next
5    if ltail = tail:
6      if lnext = nil:
7        if CAS(ltail.next, lnext, lnode):
8          done = true
9      else:
10       CAS(tail, ltail, lnext)
11 CAS(tail, ltail, lnext)
```

If we remove line 11, the algorithm stays valid. If we remove line 5, then the algorithm is not valid if cells can be released and reallocated. Otherwise the algorithm stays valid.

Dequeue

```
1  done = false
2  until done:
3    lhead = head
4    ltail = tail
```

[1] Roi Ramon's algorithm as presented in Lecture 7

```
5    lnext = lhead.next
6    if lhead = head:
7      if lhead = ltail:
8        if lnext = nil:
9          return nil
10       CAS(tail, ltail, lnext)
11     else:
12       lvalue = lnext.value
13       if CAS(head, lhead, lnext):
14         done = true
15 free(lhead)
16 return lvalue
```

Note that we return the value of the next node, because we reuse it as a dummy node.

As part of the proof, we need to show the linearization points and show that all linear specifications are valid with respect to those linearization points.

2 Memory Barriers

Example: Using flags Given that x and y are atomic bits, both initially 0.

Process A	Process B
write.x(1)	write.y(1)
read.y	read.x

Many hardware architectures do not support sequential consistency because they think it is too strong. As such, there exists an execution of the above program that reads 0 for both x and y.

The solution for this is to insert memory barriers between the write and read instructions.

Full memory barriers order both loads and stores. Read and write memory barriers each order only reads and writes, respectively.

Generally speaking, volatile variables will insert memory barriers.