Dartmouth College

Dartmouth Digital Commons

Dartmouth College Undergraduate Theses | Theses and Dissertations

6-3-2011

Obstruction-free Snapshot, Obstruction-free Consensus, and Fetch-and-add Modulo k

Jack R. Bowman
Dartmouth College

Follow this and additional works at: https://digitalcommons.dartmouth.edu/senior_theses

Part of the Computer Sciences Commons

Recommended Citation

Bowman, Jack R., "Obstruction-free Snapshot, Obstruction-free Consensus, and Fetch-and-add Modulo k" (2011). *Dartmouth College Undergraduate Theses*. 67.
https://digitalcommons.dartmouth.edu/senior_theses/67

This Thesis (Undergraduate) is brought to you for free and open access by the Theses and Dissertations at Dartmouth Digital Commons. It has been accepted for inclusion in Dartmouth College Undergraduate Theses by an authorized administrator of Dartmouth Digital Commons. For more information, please contact dartmouthdigitalcommons@groups.dartmouth.edu.

Dartmouth College Computer Science Technical Report TR2011-681

Obstruction-free Snapshot, Obstruction-free Consensus, and Fetch-and-add Modulo k

Jack Bowman

Advisor: Prasad Jayanti

June 3, 2011

Abstract

In this thesis we design algorithms for three problems: snapshot, consensus, and fetch-and-add modulo k. Our solutions for snapshot and consensus are non-anonymous and obstruction-free, and our solution for Fetch-and-add Modulo k is wait-free. We also conjecture an anonymous, obstruction-free solution to consensus.

1 Introduction

In this thesis we design algorithms for three problems: snapshot, consensus, and fetch-and-add modulo k. Our solutions for snapshot and consensus are non-anonymous and obstruction-free, and our solution for Fetch-and-add Modulo k is wait-free. We also conjecture an anonymous, obstruction-free solution to consensus. First we will describe the model of computation, the problems, and previous research. In the following sections we provide the algorithms and proofs of correctness.

We consider an *asynchronous system* with n processes where processes may run at any speed or crash. Each process has *local variables* which may only be accessed by that process. *Shared objects* also exist, which may be accessed by any process. In this thesis we may have *read-write registers* as well as *fetch-and-add objects*. Read-write registers store a single word supporting *read* and *write* operations. Fetch-and-add objects store a single word supporting the fetch-and-add operation, which adds an input value to the word and returns the value previously in the word.

There are two liveness properties discussed in this paper: *wait-freedom* and *obstruction-freedom*. An algorithm is wait-free if, regardless of the order or speed of processes running, every process completes the algorithm in a bounded number of its own steps. An algorithm is obstruction-free if at any point a process running alone will complete the algorithm in a bounded number of its own steps [8]. Wait-freedom is a stronger property to require than obstruction-freedom. Note that an obstruction-free algorithm permits processes to run forever if they "get in the way" of each other by running concurrently, whereas this is not permissible in a wait-free algorithm.

The correctness condition of a concurrent algorithm is typically *linearizability* [9]. An algorithm is linearizable if it appears to occur instantaneously even though it is actually executed over an interval of time. Formally, an algorithm is linearizable if each execution of the algorithm X occuring over the interval from t to t' by some process p has a *linearization point* $LP(X)$ such that $t \leq LP(X) \leq t'$ and all executions of the algorithm appear to occur instantly at their linearization point. To show that an algorithm is linearizable we typically define the linearization points of the algorithm and show that they are correct, i.e., the linearization points are indeed within the interval of execution and that when viewed as occuring instantly at the linearization points the executions are consistent.

An algorithm is *anonymous* if the actions of a process executing the algorithm do not depend on its name, or process id. Similarly, an algorithm is *non-anonymous* if the actions of a process do depend on its process id. With these definitions in hand, we can now define the problems.

A *Snapshot object* contains an array of some constant size m and supports two operations: SCAN and UPDATE(i, v). SCAN returns all of the values in the array. UPDATE(i, v) writes the value v to the ith word in the array. The Snapshot problem is to provide linearizable algorithms for SCAN and UPDATE [1, 2]. We design an obstruction-free, non-anonymous, linearizable Snapshot object in the next section. Note that wait-free implementations [10] as well as obstruction-free implementations [6] already exist. The goal of this paper's implementation is improved elegance.

The *consensus problem* requires processes to agree on a single value. Each process p_i begins with a *preference* v_i, unknown to the other processes. After executing the consensus algorithm, each process p_i *decides* on some value d_i. All of the d_i's must be the same, and furthermore $d_i = v_j$ for some j. Formally, a consensus algorithm satisfies three properties: validity, that the decision is some process's preference, agreement, that all processes decide the same value, and termination, that the algorithm is obstruction-free.

The consensus problem is a classic problem in concurrent computing [5]. Intriguingly, the wait-free consensus problem is impossible to solve using only read-write registers for even two processes [5]. Primitives such as fetch-and-add can solve the wait-free consensus problem for two processes, but we are at a loss with a basic system that only supports read-write registers. Herlihy showed that consensus can be used to compare the strength of different concurrent objects [7]. Herlihy formalized this notion with the *consensus hierarchy*, a hierarchy of concurrent objects, ranked by their *consensus number*–the number of processes for which wait-free consensus can be solved using only registers and the object [7]. Using a solution to the consensus problem for an arbitrary number of processes, we can create a universal construction for objects, a procedure by which we can design a linearizable algorithm for any concurrent object desired. [7] Because of the important impossibility results for consensus, the consensus hierarchy, and universal construction for objects from consensus, the problem has received much attention.

It has already been shown that one can implement an obstruction-free consensus object using only registers [4]. One implementation by Guerraoui and Ruppert for anonymous processes uses a snapshot object to accomplish this [6]. The goal of this paper's implementation is elegance and an anonymous solution that uses only registers.

A *Fetch-and-add Modulo k object* contains a single word which represents a value modulo k and supports one operation: F&A%K(v). This operation adds v to the word, modulo k, and returns the value previously in the word, modulo k. This operation is desirable to have. For example, Anderson's mutex algorithm [3] makes use of a fetch-and-increment object in combination with modulus operators. A fetch-

Shared variables: register X, array A

```
SCAN for process $i$
1.    do
2.        $X \leftarrow i$
3.        for each $i$:
4.            $B[i] \leftarrow A[i]$
5.    until $X == i$
6.    return $B$

UPDATE$(j, v)$
7.    $X \leftarrow \perp$
8.    $A[j] \leftarrow v$
```

Figure 1: An obstruction-free snapshot algorithm

and-add-mod k object could be used instead.

2 Obstruction-free Snapshot

We now describe the algorithm in Figure 1, which implements an obstruction-free, non-anonymous snapshot object from registers.

The algorithm given in Figure 1 works as follows. The goal of the process running SCAN is to try to read a consistent view of the array A without other processes interfering. The shared variable X is used to indicate who is trying to read the array. The process scanning first writes its process id to X. Then the process scanning will keep trying to read A until it sees that no one else has touched X. If this is the case, then it is assured that another process is not scanning when the process succeeds. To be further assured that there are not problems, the UPDATE procedure overwrites X so that no UPDATE will begin in the middle of a successful SCAN iteration, causing that SCAN to return inconsistent values.

Note that if we change line 5's condition to $X \neq \perp$, the resulting algorithm would be incorrect. Suppose in this modified algorithm that we have two SCANs, S and S', and two UPDATEs, U and U' which are all concurrent. Consider the following execution: U and U' execute line 7, S and S' execute line 2, S reads $A[1] = 0$, U

writes $A[1] \leftarrow 5$, S' reads $A[1] = 5$, S' reads $A[2] = 0$, U' writes $A[2] \leftarrow 17$, S reads $A[2] = 17$. S returns the values $[0, 17]$ and S' returns the values $[5, 0]$. This execution is unlinearizable, so this modified algorithm is incorrect.

Theorem 1. *The algorithm in Figure 1 gives an obstruction-free, non-anonymous implementation of a snapshot.*

Proof. UPDATE is trivially wait-free and thus obstruction-free. If one process runs in isolation SCAN will terminate, as the process will write its process id to X, read A, and still see its process id in X. Thus SCAN is also obstruction-free. Then it remains to be shown that SCAN and UPDATE are linearizable.

Let us denote the linearization point of a SCAN S by LP(S) and similarly denote the linearization point of an UPDATE U by LP(U).

Define a *successful iteration of a scan S* as the last execution of lines 2-5, i.e., the last iteration of the loop in S where S reads the values of A it returns in line 6.

We must show that there exist linearization points such that

1. For any SCAN S, LP(S) is within the interval of S.

2. For any UPDATE U, LP(U) is within the interval of U.

3. For any SCAN S, for each index i of the array A, let v be the value read by S from slot $A[i]$ on S's successful iteration and let u be the value written by U where U is the last UPDATE writing to $A[i]$ and $\mathrm{LP}(U) < \mathrm{LP}(S)$. Then $u = v$.

We say that a SCAN S *reads an* UPDATE *U's write* if U writes the value u to index i of A and no other process writes to $A[i]$ before S reads $A[i]$ in S's successful iteration.

Consider the following linearization points: For any SCAN S, LP(S) $=S$'s last execution of line 2. For any UPDATE U,

1. If U is not concurrent with any successful scan iteration, LP(U) = line 8 of U.

2. If $\exists S$ such that the successful iteration of S is concurrent with U and S reads U's write, $\mathrm{LP}(U) = \mathrm{LP}(S) - \varepsilon$.

3. If $\nexists S$ such that the successful iteration of S is concurrent with U and S reads U's write:

   (a) If $\exists S'$ such that S' reads U's write, LP(U) = line 8 of U.

   (b) If $\nexists S'$ such that S' reads U's write, LP(U) = line 8 of U.

where ε is some infinitesimal positive value.

Clearly these LPs satisfy requirement 1. For requirement 2, note that any UPDATE operation with interval $t < t'$ concurrent with a successful scan iteration with interval $s < s'$ must have begun before the successful scan iteration, i.e., $t < s$; otherwise, X would have been set to $\perp$ and the scan iteration would not have been successful. By definition of UPDATE being concurrent with the successful scan iteration, $t' > s$. Thus $\text{LP}(U) = \text{LP}(S) - \varepsilon$ is within the interval of U. Of course, in the other three cases requirement 2 is satisfied trivially. Thus we have requirements 1 and 2.

As for requirement 3, let S be any SCAN operation. Let v be the value that S reads in slot $A[i]$ during its successful iteration. If there are no UPDATEs concurrent with S's successful iteration, then clearly v is the value written by the last UPDATE U to write to that slot. In this case the linearization points for UPDATE given in cases (1) and (3a) will apply. Since these linearization points are at the moment UPDATE writes, v is indeed the value written by U.

Lastly we consider the case where there are UPDATES $U_1, ..., U_k$ concurrent with S's successful iteration. If S read U_i's write, then by case (2) of our linearization points, this U_i did write the value v. On the other hand, suppose S did not read any U_i's write. S must have either read no UPDATE's write or a write by an UPDATE U' not concurrent with S's successful iteration. If S read no UPDATE's write in $A[i]$, then we are assured by case (3b) that no UPDATE is linearized before S that would write to $A[i]$. If S read U''s write, then we are assured by case (3a) that U' is the last UPDATE linearized before S. Thus in every case we see that requirement 3 is met.

We have all the requirements, and hence the algorithm is linearizable.

□

3 Obstruction-free Consensus

The algorithm in Figure 2 works as follows. The processes involved in consensus "fight" over the A array to determine the value that all processes will decide on. They do this by writing what they think is the majority in A until some value wins. The central idea is that A has $2n$ slots. If A is filled with entirely one value the majority cannot change as even if the $n-1$ processes still fighting wrote non-majority values to A, there will still be $n+1$ of the original value.

Theorem 2. *The algorithm in Figure 2 gives an obstruction-free implementation of consensus.*

Shared variables: $A[1..2n]$ initialized to $\perp$, where n is the number of processes

```
CONSENSUS for process with preference $v$
1.    $A[1] \leftarrow v$
2.    do
3.       $B \leftarrow$ SCAN($A$)
4.       $m \leftarrow$ majority in $B$
           (with ties broken arbitrarily)
5.       $c_m \leftarrow$ number of times $m$ occurs in $B$
6.       if $c_m \neq |A|$:
7.          UPDATE($A, i, m$) where $i$ is the first non-$m$ slot in $B$
8.    until $c_m == |A|$
9.    decide $m$
```

Figure 2: An obstruction-free consensus algorithm from registers and snapshot

Proof. The three correctness conditions are validity, obstruction-free termination, and agreement. Validity is ensured by line 1, and termination are obvious, so it only remains to show agreement.

For agreement, without loss of generality, suppose process p decides v at time t. Let us call the time of p's last SCAN operation t'.

We claim that any process q deciding after t must decide v. For q to decide $v' \neq v$, $2n$ v''s would have to be written to A by processes executing line 5 after t'. However, at most $n-1$ v''s can be written to A without a process scanning A after t'. Any process scanning A after t' will see at least $2n-(n-1) = n+1$ v's in A, since p saw $2n$ v's and at most $n-1$ v''s have been written. Thus that process will conclude that the majority in A is v and will not be able to write a v'. Therefore at most $n-1$ v''s can be written after t', so q could not decide v'. Hence we have the claim.

Since any process q deciding after p agrees with p, we have agreement.

By having the validity, termination, and agreement properties, we have shown that the algorithm is correct. □

Shared variables: $A[1..2n]$ initialized to $\perp$, where n is the number of processes

```
CONSENSUS for process with preference $v$
1.    $A[1] \leftarrow v$
2.    **do**
3.        for each $i$:
4.            $B[i] \leftarrow A[i]$
5.        $m \leftarrow$ majority in $B$
            (with ties broken arbitrarily)
6.        $c_m \leftarrow$ number of times $m$ occurs in $B$
7.        if $c_m \neq |A|$:
8.            $A[i] \leftarrow m$ where $i$ is the first non-$m$ slot in $B$
9.    **until** $c_m == |A|$
10.   decide $m$
```

Figure 3: An anonymous, obstruction-free consensus algorithm from registers

4 A Better Consensus Algorithm

The algorithm in Figure 3 is very similar to the algorithm in Figure 2, except that the size of the A array is larger and instead of scanning A, we simply read A. The features of this algorithm are that it only uses registers and that it is anonymous. For these reasons it is very elegant; the other obstruction-free consensus algorithms in the literature do not achieve both of these features [4, 6].

Once again, the processes "fight" over A. We conjecture this algorithm to be correct.

Conjecture 3. *The algorithm in Figure 3 gives an obstruction-free implementation of consensus.*

It is likely then that we do not even need the snapshot object for obstruction-free consensus. It is beyond this paper to prove the correctness of this algorithm.

5 Fetch-and-Add Modulo k

We define the % operation by the "Euclidean definition" such that $a\%k = b$ where $0 \leq b < k$ and $a = q \cdot k + b$ for some integer q. We consider only positive k, and so

```
FETCH-AND-ADD$(A, x)$ modulo $k$
1.    $y \leftarrow$ F&A$(A, x\%k)$
2.    if $(y \geq 0)$
3.       F&A$(A, -k)$
4.    return $y\%k$
```

Figure 4: A wait-free fetch-and-add modulo k algorithm

note that the result of $a\%k$ is always positive. With this definition in mind, a fetch-and-add modulo k object supports the operation fetch-and-add-mod which takes one value x and returns the value y. The returned value y is the value previously in the object, $0 \leq y < k$, and the value x is added to the object modulo k.

Note that a fetch-and-add object is trivially a fetch-and-add mod k object on a real machine if k is a power of two, since any overflow from the object can be ignored. This algorithm is notable, then, because it does not require that k be a power of two.

The algorithm is obviously correct if we assume an unbounded amount of space; it uses a fetch-and-add object to store the value necessary. However, the algorithm does work in a bounded amount of space by subtracting k from the fetch-and-add object, keeping the possible values in A between $-nk$ and $n(k-1)$. Since there are only n processes, the value in A cannot be forced beyond some factor of n.

Theorem 4. *The algorithm in Figure 4 gives a wait-free implementation of fetch-and-add modulo k in linear space and constant time.*

Proof. The algorithm adds the appropriate value mod k in line 1 and returns the appropriate value mod k in line 4. Line 3 has no effect on the correctness of the algorithm since $-k \equiv 0 \bmod k$. Thus the algorithm is correct and runs in constant time.

We claim that the space required for the fetch-and-add object is linear in the number of processes. We will show the claim by showing the upper and lower bounds for the value in A.

We claim that the upper bound for the value in A is $n(k-1)$. The value in A will only increase upon an execution of line 1. Furthermore, if the value in A is ≥ 0, then after executing line 1, a process will execute line 3 and effect a net decrease of the value in A. The highest value in A is obtained when all n processes execute line 1 with an input of $x = k-1$. After the nth process executes line 1, the value of A will be $n(k-1)$. Hence, the claim.

We claim that the lower bound for the value in A is $-nk$. The value in A will only decrease upon an execution of line 3. The lowest value in A is obtained when all n processes execute 1 with an input of $x = 0$ and then all n processes execute line 3. After the nth process executes line 3, the value in A will be $-nk$. Hence, the claim.

Combining the upper and lower bound, we see that A requires $n(k-1) + nk = n(2k-1)$ space. Since k is constant, $n(2k-1) = O(n)$ and we have that the algorithm requires an amount of space linear in n (and k). □

6 Conclusions

We have shown three algorithms: obstruction-free snapshot, obstruction-free consensus, and fetch-and-add mod k. We have also conjectured an elegant algorithm for obstruction-free consensus.

The following questions may be addressed by future research. Is it possible to use fetch-and-add mod k to eliminate unbounded counters in other algorithms? What is the lower bound on space for a fetch-and-add mod k object? Is the conjectured consensus algorithm correct for any number of processes? If it is correct, what is the lower bound on the size of the array used?

References

[1] Afek, Y., Attiya, H., Dolev, D., Gafni, E., Merritt, M., and Shavit, N. 1990. Atomic snapshots of shared memory. In *Proceedings of the 9th Annual Symposium on Principles of Distributed Computing*, pages 114.

[2] Anderson, J. 1990. Composite registers. In *Proceedings of the 9th Annual Symposium on Principles of Distributed Computing*, pages 1529.

[3] Anderson, T. 1990. The performance of spin lock alternatives for shared memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6-16.

[4] Attiya, H., Guerraoui, R., Hendler, D., and Kuznetsov, P. 2009. The complexity of obstruction-free implementations. *J. ACM* 56, 4, Article 24 (June 2009), 33 pages.

[5] Fischer, M. J., Lynch, N. A., and Patterson, M. S. 1985. Impossibility of distributed consensus with one faulty process. *J. ACM* 32, 3, 374382.

[6] Guerraoui, R., Ruppert, E. 2007. Anonymous and fault-tolerant shared memory computing. *Distrib. Comput.* 20(3), 165177.

[7] Herlihy, M. 1991. Wait-free synchronization, *ACM Trans. on Progr. Lang. and Systems 11*, no. 1, 124-149.

[8] Herlihy, M., Luchangco, V., and Moir, M. 2003. Obstruction-free synchronization: Double-ended queues as an example. In *Proceedings of the 23rd IEEE International Conference on Distributed Computing Systems (ICDCS).* IEEE Computer Society Press, Los Alamitos, 522529.

[9] Herlihy, M., and Wing, J. M. 1990. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst. 12*, 3 (June), 463492.

[10] Jayanti, P. 2005. An optimal multi-writer snapshot algorithm. *Proceedings of the 37th Annual ACM Symposium on Theory of Computing*, Baltimore, Maryland, May 2005, 723-732.