UNIVERSITY OF CALGARY

Title of Thesis

Second Thesis Title Line

by

Student's name

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE

DEGREE OF NAME OF DEGREE IN FULL

DEPARTMENT OF NAME OF DEPARTMENT

CALGARY, ALBERTA

monthname, year

Abstract

Table of Contents

Chapter 1

Introduction

1.1 Related Work

(Under work...)

1.2 Statement of Results

Chapter 2

Model of Computation and Definitions

2.1 Asynchronous Shared Memory Model

In this chapter, we will describe our model of computation and give the definitions, which are based on Herlihy and Wing's [3] and Golab, Hadzilacos and Woelfel's [2].

The computational model we consider is the standard asynchronous shared memory model with a set $\mathcal{P}$ of n processes, denoted as $\{p_0, p_2, ..., p_{n-1}\}$, where up to $n-1$ processes may fail by crashing.

Type and Object. A *type* τ is defined as an automaton as follows [2],

$$\tau = (\mathcal{S}, s_{init}, \mathcal{O}, \mathcal{R}, \delta)$$

where $\mathcal{S}$ is a set of states, $s_{init} \in \mathcal{S}$ is the initial state, $\mathcal{O}$ is a set of operation types, $\mathcal{R}$ is the set of responses, and $\delta : \mathcal{S} \times \mathcal{O} \rightarrow \mathcal{S} \times \mathcal{R}$ is a state transition mapping.

An *object* is an implementation of a type. For each type τ, the transition mapping δ captures the behaviour of objects of type τ, in the absence of concurreny, as follows: if a process applies an operation of type *opt* to an object of type τ which is in state s, the object may return to the process a response *rsp* and change its states to s' if and only if $(s', rsp) \in \delta(s, opt)$.

History A *history* H, obtained by processes executing operations on objects, is a sequence of invocation and response events.

An invocation event is a 5-tuple,

$$\text{INV} = (invocation, p, obj, opt, t)$$

where *invocation* is the event type, p is the process executing the operation, *obj* is the object on which the operation is executed, *opt* is the operation type and t is the *time* when INV happens which is defined as the position of event INV in history H.

A response event is also a 5-tuple,

$$\text{RSP} = (response, p, obj, rsp, t)$$

where *response* is the event type, p is the process receiving response *rsp* from an oeration on object *obj* and t is the time when RSP happens which is defined as the position of event RSP in history H.

For an event with respect to process p, we also say this event is *applied* by process p.

In the following discussion, we suppose in a history H, the situation that an invocation event $(invocation, p_i, obj_p, opt_0, t_0)$ is followed immediately by another invocation event $(invocation, p_j, obj_q, opt_1, t_1)$ where $i = j$ and $p = q$ will not happen.

Response event $(response, p_j, obj_q, rsp, t_1)$ *matches* invocation event $(invocation, p_i, obj_p, opt, t_0)$ in history H, if the two events are applied by the same process to the same object, i.e, $i = j$ and $p = q$. In this case, the response event is also called the *matching* response of the invocation event.

An *operation execution* in H is a pair $oe =$ (INV, RSP) consisting of an invocation event INV and its matching response event RSP, or just an invocation event INV with no matching response event, denoted as $oe =$ (INV, *null*). In the latter case, we say the operation execution is *pending*. In the former case, we say the operation execution is *complete*. A history H is *complete* if all operation executions in H are *complete*, otherwise, it is *incomplete*. If events INV and RSP are applied by process p, then we say operation execution $oe =$ (INV, RSP) is *performed* by process p. Thus, two operation executions performed by the same process on the same project will not interleave in a history H.

History H' is an extension of history H if H is a prefix of H'. History H' is a *completion* of history H if H' contains all the events in H and H' is an extension of H, and each operation execution in H' is complete.

$H|obj$ of history H is the subsequence of all invocation and response events in H on object obj. If all invocation and response events in a history H have the same object name obj, then the $H|obj = H$.

Let H be a complete history. We associate a time interval $I_{oe} = [t_0, t_1]$ with each operation execution $oe =$ (INV, RSP) in H, where t_0 and t_1 are the points in time when INV and RSP happen. Similarly, for an incomplete history, we denote the time interval I_{oe} with respect to a pending operation execution $oe =$ (INV, *null*) by $I_{oe} = [t_0, \infty]$.

Operation execution oe_0 *precedes* operation execution oe_1 in H if the response event of oe_0 happens before the invocation event of oe_1 in H. We say that oe_0 and oe_1 are *concurrent* in H if neither precedes the other.

A history is *sequential* if its first event is an invocation event, and each invocation event, except possibly the last one, is immediately followed by a matching response event.

A *sequential specification* of an object is the set of all possible sequential histories for that object.

A sequential history S is *valid*, if for each object obj, $S|obj$ is in the sequential specification of obj.

Linearization. A history H *linearizes* to a sequential history S, if and only if S satisfies the following conditions: (1) S and any completion of H have the same operation executions, (2) sequential history S is valid, and (3) there is a mapping from each time interval I_{oe} to a time point $t_{oe} \in I_{oe}$, such that the sequential history S is obtained by sorting the operations in H based on their t_{oe} values.

A history is *linearizable* if and only if H linearizes to some sequential history S. In this case, S is called the *linearization* of H. For each operation *opt* in history H, we call time point t_{oe}, which is defined as above, the *linearization point* of *opt*. An object *obj* is linearizable if every history H on *obj* is linearizable.

2.2 Primitive Objects

In this section, we describe the two primitive objects which will be used in our following discussion, namely *read-write register* and *compare-and-swap* (CAS) objects. Most implementations of more sophisticated objects use these objects as the base objects in their implementations, since most modern architectures support either read-write registers and CAS objects [**?**] [**?**].

Read-Write Register. An object that supports only `read()` and `write(x)` operations is called a read-write register (or just *register*). Operation `read()` returns the current state of register and leaves the state unchanged. Operation `write(x)` changes the state of the register to x and returns nothing. If the set of states that can be stored in the register is unbounded then we say the register is *unbounded register*; otherwise the register is *bounded register*.

CAS Object. An object that supports `read()` and `CAS(x,y)` operations is called compare-and-swap (CAS) object. Operation `read()` is the same as defined above. Operation `CAS(x,y)` changes the state of the object if and only if the current state is equal to x and then operation `CAS(x,y)` succeeds, and the state is changed to y and *true* is returned. Otherwise, operation `CAS(x,y)` fails, the current state remains unchanged and $false$ is returned.

2.3 Adversary Models for Randomized Algorithms

Randomness. A randomized algorithm is an algorithm where processes are allowed to make random decisions for the next step. This is modelled by giving each process a special operation called *coin-flip operation*. We say a process can flip a coin when it calls this operation. When a process flips a coin, it receives a random value c from some arbitrary set Ω which is calle the *coin-flip domain*. The process can then use this random value c in its program for future decisions.

A vector $\overrightarrow{c} = (c_0, c_1, c_2, ...) \in \Omega^\infty$ is called a *coin-flip vector*. A history H is said to *observe* the coin-flip vector $\overrightarrow{c}$ if for an arbitrary integer $i \in [0, \infty)$, the i-th coin-flip operation in H returns value c_i.

In the following discussion, we use method `random(s)`, which is assumed to be linearizable, to return a value which is distributed uniformly at random over domain $\{0, 1, 2, ..., s-1\}$.

Adversary. In the standard shared memory model, each process executes its program by applying shared memory operations (`read()`, `write(x)`, `CAS(x,y)`, etc) on objects, as determined by their program. Operation executions of concurrent processes can be interleaved arbitrarily.

A *schedule* with *length* k is represented by a sequence of process ids

$$p = (p_0, p_1, p_2, ..., p_{k-1})$$

where $k \in [0, \infty)$ and for each $i \in [0, k-1]$, $p_i \in \mathcal{P}$.

Let a schedule $p = (p_0, p_1, p_2, ..., p_{k-1})$. A history H is said to *observe* schedule p if the number of events in H is k, and for each integer $i \in [0, k-1]$, the i-th event is applied by process p_i.

In randomized algorithm, the random choices processes make can influence the schedule. To model the worst-case possible way that the system can be influenced by the random choices, schedules are assumed to be generated by an adversarial scheduler, called the *adversary*.

Mathematically, an adversary is defined as a mapping:

$$\mathcal{A} : \Omega^\infty \to \mathcal{P}^\infty$$

Given an algorithm $\mathcal{M}$, an adversary $\mathcal{A}$, and a coin-flip vector $\overrightarrow{\mathrm{c}} \in \Omega^\infty$, a history $H_{\mathcal{M},\mathcal{A},\overrightarrow{\mathrm{c}}}$ is generated, such that all processes apply events as dictated by algorithm $\mathcal{M}$, and history $H_{\mathcal{M},\mathcal{A},\overrightarrow{\mathrm{c}}}$ observes the schedule $\mathcal{A}(\overrightarrow{\mathrm{c}})$.

There are several adversary models with different strengths [1]. In our thesis, we only conside the *adaptive adversary*.

Informally, the adaptive adversary makes scheduling decisions as follows: At any point, it can see the entire history up to that point. This includes all coin-flip operations and their return values up to that point. Depending on this, the adversary decides which process takes the next step.

For a history H that contains k coin-flip operations, we use $H[k]$ to denote the subsequence of H that contains all events up to the k-th invocation event of a coin-flip operation.

Adversary $\mathcal{A}$ is *adaptive* for algorithm $\mathcal{M}$ if, for any two coin-flip vectors $\overrightarrow{\mathrm{c}} \in \Omega^\infty$ and $\overrightarrow{\mathrm{d}} \in \Omega^\infty$ that have a common prefix of length k (i.e, the first k elements of them are the same), then we have

$$H_{\mathcal{M},\mathcal{A},\overrightarrow{\mathrm{c}}}[k+1] = H_{\mathcal{M},\mathcal{A},\overrightarrow{\mathrm{d}}}[k+1]$$

In this case, we also say Adversary $\mathcal{A}$ is an *adaptive adversary*.

From the above definition, we can see an adaptive adversary cannot use future coin flips to make current scheduling decisions.

2.4 The Dynamic Task Alloction Problem

We specify the dynamic task alloction problem in terms of a type DTA, and the properties that an implementation of type DTA must satisfy.

The dynamic task allocation (DTA) type supports two operation types, `DoTask()` and `InsertTask(`ℓ`)`, where ℓ is the identifier that is unique for each task (*define what is task, give the domain*).

We assume that there exists an atomic operation `PutTask(`M, ℓ`)`, and a process associates (*what is associate?*) task ℓ with memory location (*what is memory location?*) M by calling `PutTask(`M, ℓ`)`. It returns *success* if task ℓ is associated with location M, and returns $failure$ if location M was already associated with another task.

Similarly, we assume there exists an atomic operation `TryTask(`M`)`, and task ℓ associated with memory location M could be performed atomically by calling `TryTask(`M`)`. Out of several processes calling `TryTask(`M`)`, one receives *success* and the index ℓ of that task, while all the others receive $failure$. (*state transition for all initial states, what happens next if there is no...*)

A task is *done* or $performed$ if its index has been returned by a process after calling `DoTask()`. A task is *available* at location M if it has been inserted to M and *success* is returned by a process after calling `InsertTask(`ℓ`)`, but is not done yet. A task is *available*, if it is *available* at some memory location.

The aim of operation `DoTask()` is to perform an available task on location M by calling `TryTask(`M`)`. Every `DoTask()` may perform several `TryTask(`M`)` operations. However, only one of them will succeed. Once one `TryTask(`M`)` succeeds, then there is no available task on M and the task index ℓ will be returned by `DoTask()`. Additionally, if there is no available task, then operation `DoTask()` returns $\perp$.

The goal of `InsertTask(`ℓ`)` operation is to find a free memory location M and insert task ℓ atomically by calling `PutTask(`M, ℓ`)`. `PutTask(`M, ℓ`)` fails if location M has been associated with another task, so each `InsertTask(`ℓ`)` operation may perform several `PutTask(`M, ℓ`)` operations, but only one of them will succeed. Once one `TryTask(`M`)` succeeds, then task ℓ is available on location M and *success* notification is returned by `InsertTask(`ℓ`)` operation.

Type DTA is required to satisfy: (*Validity*) If a `DoTask()` operation returns ℓ, then before the `DoTask()` operation, an `InsertTask(`ℓ`)` was executed and returned *success*. (*Uniqueness*) Each task is performed at most once, i.e, for each task ℓ, at most one `DoTask()` operation returns ℓ.

In addition, the property that every inserted task is eventually done is also a desired progress property of the implementation of type DTA.

Chapter 3

Dynamic Task Allocation Object

Implementation of DTA

(under work...)

Method 1: `DoTask()`

```
while true do
    v ← root;
    if v.surplus() ≤ 0 then
        return ⊥;
    end
    /* Descent */;
    while v is not a leaf do
        (x_L, y_L) ← v.left.read();
        (x_R, y_R) ← v.right.read();
        s_L ← min(x_L − y_L, 2^{height(v)});
        s_R ← min(x_R − y_R, 2^{height(v)});
        r ← random(0, 1);
        if (s_L + s_R) = 0 then
            Mark-up(v);
        else if r < s_L/(s_L + s_R) then
            v ← v.left;
        else
            v ← v.rght;
        end
    end
    /* v is a leaf */;
    (x, y) ← v.read();
    (flag, l) ← v.TryTask(task[y + 1]);
    /* Update Insertion Count */;
    v.CAS((x, y), (x, y + 1));
    v ← v.parent;
    Mark-up(v);
    if flag = success then
        return ℓ
    end
end
```

Method 2: `InsertTask(ℓ)`

```
while true do
    v ← root;
    /* Descent */;
    while v is not a leaf do
        (x_L, y_L) ← v.left.read();
        (x_R, y_R) ← v.right.read();
        s_L ← 2^{height(v)} − min(x_L − y_L, 2^{height(v)});
        s_R ← 2^{height(v)} − min(x_R − y_R, 2^{height(v)});
        r ← random(0, 1);
        if (s_L + s_R) = 0 then
            Mark-up(v);
        else if r < s_L/(s_L + s_R) then
            v ← v.left;
        else
            v ← v.rght;
        end
    end
    /* v is a leaf */;
    (x, y) ← v.read();
    flag ← v.PutTask(task[x + 1]);
    /* Update Insertion Count */;
    v.CAS((x, y), (x + 1, y));
    v ← v.parent;
    Mark-up(v);
    if flag = success then
        return success
    end
end
```

Method 3: `Mark-up(v)`

```
if v is not null then
    for (i = 0; i < 2; i + +) do
        (x, y) ← v.read();
        (x_L, y_L) ← v.left.read();
        (x_R, y_R) ← v.right.read();
        v.CAS((x, y), (max(x, x_L + x_R), max(y, y_L + y_R)));
    end
end
```

Chapter 4

Correctness Proof

4.1 Correctness

The standard correctness condition for shared memory algorithms is linearizability, which was introduced by Herlihy and Wing in 1990 [3]. The intuition of linearizability is that real-time behavior of method calls must be preserved, i.e, if one method call precedes another, then the earlier call must have taken effect before the later one. By contrast, if two method calls overlap, we are free to order them in any convenient way since the order is ambiguous. Informally, a concurrent object is linearizable if each method call appears to take effect instantaneously at some moment between its invocation and response.

4.1.1 Analysis and Proofs

By the definitions in Subsection 3.1.1, one way to show an object obj is linearizable is to prove every history H of obj is linearizable. Thus, we need to identify for each `DoTask` and `InsertTask` operation op (i.e, interval $I_H(op)$) in H a linearization point $t_H(op)$, and prove that the sequential history S obtained by sorting these operations according to their $t_H(op)$ satisfies the sequential specification S_{obj} of obj.

We notice that each complete `DoTask` or `InsertTask` operation can be associated with a unique task array slot based on the task it removed or inserted. Additionally, the removal and insertion count are both monotonically increasing. Thus, we could associate the node counts with operations which have been propagated to that node.

Now we define "an operation is counted at a node" recursively to formalize the operation propagation.

A `DoTask` operation is counted at leaf v when the removal count of v is updated with the index of the task array slot where the performed task is located. Symmetrically, an `InsertTask` operation is counted at v when the insertion count of v is updated with the index of the task array slot where the inserted task is located.

Now we only define `DoTask` operation is counted at an inner node v because counting an `InsertTask` operation is symmetric as well.

Recall that the removal count of v is updated though CAS operation (line 6, method 3). Actually there could be more than one operations updating the count with the same value. We linearize all such CAS operations, which update the removal count of v with the same value y. We say for all these operations, only the first one in the linearization order counts the corresponding `DoTask` operation. In another word, a `DoTask` operation is counted at an inner node v as soon as the CAS updating operation that counts the `DoTask` is linearized. Based this definition, no operation will be counted twice at a node.

Please note that, the CAS operation counting the `DoTask` at node v is not necessary performed by the `DoTask` operation itself, i.e, suppose process p executes a `DoTask` operation and has successfully performed task ℓ at certain leaf. Then the CAS operation counting this $p.DoTask()$ at node v could be a different process q as long as q updates the removal count first in the linearization order.

Given the above concepts and properties, we could prove the following result: (under work...almost done)

4.2 Performance

4.2.1 DoTask Analysis

4.2.2 InsertTask Analysis

Bibliography

[1] James Aspnes. Randomized protocols for asynchronous consensus. *CoRR*, cs.DS/0209014, 2002.

[2] Wojciech Golab, Vassos Hadzilacos, Danny Hendler, and Philipp Woelfel. Constant-rmr implementations of cas and other synchronization primitives using read and write operations. In *Proceedings of the 26th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 3–12, 2007.

[3] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990.