

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

© Student's name year

# Abstract

# Table of Contents

<b>Abstract</b>	i
Table of Contents	ii
1 Introduction	1
1.1 Related Work	1
1.2 Statement of Results	1
2 Model of Computation and Definitions	2
3 Dynamic Task Allocation Object	5
4 Correctness Proof	9
4.1 Correctness	9
4.1.1 Analysis and Proofs	9
4.2 Performance	11
4.2.1 DoTask Analysis	11
4.2.2 InsertTask Analysis	11

# Chapter 1

## Introduction

### 1.1 Related Work

(Under work...)

### 1.2 Statement of Results

## Chapter 2

### Model of Computation and Definitions

In this chapter, we will describe our model of computation and give the definitions, which are based on Herlihy and Wing's [2] and Golab, Hadzilacos and Woelfel's [1].

The computational model we consider is the standard asynchronous shared memory model with a set of  $n$  processes, denoted as  $\{p_1, p_2, \dots, p_n\}$ , where up to  $n - 1$  processes may fail by crashing.

**Type and Object.** A *type*  $\tau$  could be defined as an automaton as follows [1],

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

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 one-to-many state transition mapping.

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

An object supports only **read()** and **write(x)** operations is called a *read/writeregister* (or just *register*). Operation **read()** returns the current state of register and leaves the state unchanged. Operation **write(x)** changes the state of register to be  $x$ .

An object supports **read()** and **CAS(x,y)** operations is called *compare – and – swap* (CAS) object. Operation **CAS(x,y)** changes the state of the obejct if and only if the current state is equal to the given state  $x$ , i.e, if current state equals  $x$ , then operation **CAS(x,y)** succeeds, and the state is changed to be  $y$  and *true* is returned. Otherwise, operation **CAS(x,y)** fails,

the current state remains unchanged and *false* is returned.

**History** (*add more explanation for history!*). A *history*  $H$ , obtained by processes executing operations on shared objects, is a sequence of method *invocation* and *response* events (*what are invocation and response events based on type definition.*).  $H|obj$  of history  $H$  is the subsequence of all invocation and response events in  $H$  whose object names are  $obj$ . If all invocation and response events in a history  $H$  have the same object name  $obj$ , then the  $H|obj = H$ . Thus, in the following discussion, when we discuss the concurrency behavior of a specific object  $obj$ , the history  $H$  and  $H|obj$  are the same.

We use  $inv_H(op)$  to denote the invocation of operation  $op$  in history  $H$  and use  $rsp_H(op)$  to denote the matching response of  $op$  in  $H$ . An invocation event  $inv_H(op)$ , it is not necessarily to have a matching response. In this case, we say the operation  $op$  is *pending*. Otherwise, we call operation  $op$  is *complete*.

A history  $H$  is *complete* if all operations in  $H$  are *complete*. A history  $H'$  is an extension of a history  $H$  if  $H$  is the prefix of  $H'$ . A history  $H'$  is called the completion of an incomplete history  $H$  if  $H'$ , containing the same events as  $H$ , is an extension of  $H$  and each operation in  $H'$  is complete.

We say an invocation or response event happens at time  $T$  if it is the  $T$ th event in the history. Let  $H$  be a complete history, we could associate a time interval  $I_H(op) = [T_{inv(op)}, T_{rsp(op)}]$  with each operation  $op$  in  $H$ , where  $T_{inv(op)}$  and  $T_{rsp(op)}$  is the time when event  $inv(op)$  and  $rsp(op)$  happens respectively. Similarly, for an uncomplete history, we denote the interval with respect to the pending operation  $op$  by  $I_H(op) = [T_{inv(op)}, \infty]$ .

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

**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 operations, (2) sequential

history  $S$  is valid, and (3) there is a mapping from each time interval  $I_H(op)$  to a time point  $t_H(op) \in I_H(op)$ , such that the sequential history  $S$  is obtained by sorting the operations in  $H$  by their  $t_H(op)$  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  $op$  in history  $H$ , we call the point  $t_H(op)$  the linearization point of  $op$ . A shared object is linearizable if every history  $H$  on that object is linearizable.

**Randomness.** A process can execute local coin flip operation that returns an integer value distributed uniformly at random from an arbitrary finite set of integers. In the following discussion, we use method `random(s)` to return a value which is distributed uniformly at random from set  $\{0, 1, 2, \dots, s - 1\}$ .

**Adversary.** We analyze our algorithm under the assumption of a strong adaptive adversary. At any point of time, it can see the entire past history and know the states of all processes.

*(What a process do to execute a program? What is an adversary? What is the functionality of an adversary? (to schedule processes )? How it schedules processes?)*

## Chapter 3

### Dynamic Task Allocation Object

Our dynamic task allocation (DTA) type supports two operations, `DoTask()` and `InsertTask( $\ell$ )`, where  $\ell$  is the task identifier that is unique for each task.

Now we formalize the notion of type DTA by specifying the above two operations. We assume that there exists an atomic operation `PutTask( $M, \ell$ )`, and a process associates task  $\ell$  with memory location  $M$  by calling `PutTask( $M, \ell$ )`. It returns *success* if task  $\ell$  is associated with location  $M$ , and returns *failure* if location  $M$  was already associated with another task. We say task  $\ell$  is inserted if it is associated with a memory location  $M$ .

Similarly, we assume there exists an atomic operation `TryTask( $M$ )`, and task  $\ell$  associated with memory location  $M$  could be performed atomically by calling `TryTask( $M$ )`. Out of several processes calling `TryTask( $M$ )`, only one receives *success* and the index  $\ell$  of that task, while all the others receive *failure*.

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( $\ell$ )`, 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  $\ell$  will be returned by `DoTask()`. Additionally, if there is no available task, then operation `DoTask()` returns  $\perp$ .



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

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

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

## Implementation of DTA

(under work...)

---

**Method 1: DoTask()**

---

```
1 while true do
2    $v \leftarrow \text{root};$ 
3   if  $v.\text{surplus}() \leq 0$  then
4     return  $\perp$ ;
5   end
6   /* Descent */;
7   while  $v$  is not a leaf do
8      $(x_L, y_L) \leftarrow v.\text{left}.\text{read}();$ 
9      $(x_R, y_R) \leftarrow v.\text{right}.\text{read}();$ 
10     $s_L \leftarrow \min(x_L - y_L, 2^{\text{height}(v)});$ 
11     $s_R \leftarrow \min(x_R - y_R, 2^{\text{height}(v)});$ 
12     $r \leftarrow \text{random}(0, 1);$ 
13    if  $(s_L + s_R) = 0$  then
14      Mark-up( $v$ );
15    else if  $r < s_L / (s_L + s_R)$  then
16       $v \leftarrow v.\text{left};$ 
17    else
18       $v \leftarrow v.\text{right};$ 
19    end
20  end
21  /* v is a leaf */;
22   $(x, y) \leftarrow v.\text{read}();$ 
23   $(\text{flag}, \ell) \leftarrow v.\text{TryTask}(\text{task}[y + 1]);$ 
24  /* Update Insertion Count */;
25   $v.\text{CAS}((x, y), (x, y + 1));$ 
26   $v \leftarrow v.\text{parent};$ 
27  Mark-up( $v$ );
28  if  $\text{flag} = \text{success}$  then
29    return  $\ell$ 
30  end
31 end
```

---

---

**Method 2: InsertTask( $\ell$ )**

---

```
32 while true do
33    $v \leftarrow \text{root};$ 
34   /* Descent */;
35   while v is not a leaf do
36      $(x_L, y_L) \leftarrow v.\text{left.read}();$ 
37      $(x_R, y_R) \leftarrow v.\text{right.read}();$ 
38      $s_L \leftarrow 2^{\text{height}(v)} - \min(x_L - y_L, 2^{\text{height}(v)});$ 
39      $s_R \leftarrow 2^{\text{height}(v)} - \min(x_R - y_R, 2^{\text{height}(v)});$ 
40      $r \leftarrow \text{random}(0, 1);$ 
41     if  $(s_L + s_R) = 0$  then
42       |  $\text{Mark-up}(v);$ 
43     else if  $r < s_L / (s_L + s_R)$  then
44       |  $v \leftarrow v.\text{left};$ 
45     else
46       |  $v \leftarrow v.\text{right};$ 
47     end
48   end
49   /* v is a leaf */;
50    $(x, y) \leftarrow v.\text{read}();$ 
51    $\text{flag} \leftarrow v.\text{PutTask}(\text{task}[x + 1]);$ 
52   /* Update Insertion Count */;
53    $v.\text{CAS}((x, y), (x + 1, y));$ 
54    $v \leftarrow v.\text{parent};$ 
55    $\text{Mark-up}(v);$ 
56   if  $\text{flag} = \text{success}$  then
57     | return success
58   end
59 end
```

---

---

**Method 3: Mark-up( $v$ )**

---

```
60 if v is not null then
61   for  $(i = 0; i < 2; i++)$  do
62     |  $(x, y) \leftarrow v.\text{read}();$ 
63     |  $(x_L, y_L) \leftarrow v.\text{left.read}();$ 
64     |  $(x_R, y_R) \leftarrow v.\text{right.read}();$ 
65     |  $v.\text{CAS}((x, y), (\max(x, x_L + x_R), \max(y, y_L + y_R)));$ 
66   end
67 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 [2]. 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 through 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 necessarily performed by the **DoTask** operation itself, i.e, suppose process  $p$  executes a **DoTask** operation and has successfully performed task  $\ell$  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] 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.
- [2] 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.