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

In this chapter, we will introduce the task allocation problem as well as its dynamic version and survey the past work related to it.

Task allocation problem, also called do-all problem, is already a foundational topic in distributing computing. During the last two decades, significant research was dedicated to studying the task allocation problem in various models of computation, including message passing, shared memory, etc. under specific assumption about the asynchrony and failures.

In this thesis, we consider the dynamic version of the task allocation problem via randomized asynchronous shared memory. The dynamic version could be described as follows:

p processes cooperatively perform a set of tasks in the presence of adversity and the tasks are injected dynamically by the adversary during the execution.

1.1 Related Work

1.2 Statement of Results

Chapter 2

Model of Computation and Definitions

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

2.1 Computational Model

The computational model we consider is the standard asynchronous shared memory model with n processes, each process has a unique identifier $i \in 1, ..., n$. The set of processes is denoted $\{p_1, p_2, ..., p_n\}$ and up to $n-1$ processes may fail by crashing.

A *shared object* is defined as a data structure with a well-defined set of states and a set of *operation types* [1]. The state transition happens when an operation of an operation type is applied to shared object in a given state. Process interact with shared objects by applying operations on them. In the following sections, we consider atomic and non-atomic operations. Atomic operations occur instantaneously and are represented as *atomic steps*. Non-atomic operations are represented using separate *invocation* and *response steps*. A response step matches an invocation step if the two steps are applied by the same process to the same shared object.

Each process executes its program by taking *steps* until it *terminates* which means a special state is reached and the process takes no further action.

Each shared object has a *type* τ which can be defined 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 $\delta : \mathcal{S} \times \mathcal{O} \to \mathcal{S} \times \mathcal{R}$ is a state transition mapping.

In this thesis, we consider the system that supports atomic compare-and-swap (CAS) object. A CAS object v is a shared object which stores a value from some set and supports two atomic operations: v.`read()` and v.`CAS(x,y)`. Operation v.`read()` returns the current value of v and leaves the value unchanged. Operation v.`CAS(x,y)` writes new value y into it only if it matches the given value x, i.e, if current value of v equals x, then operation v.`CAS(x,y)` succeeds, and the value of v is changed to be y and *true* is returned. Otherwise, operation v.`CAS(x,y)` fails, the current value of v remains unchanged and $false$ is returned.

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 $\{0, 1, 2, ..., s-1\}$.

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. Based on this information, it decides which process takes the next step.

Here are more definitions we will use in our following discussions,

History. A *history* H, obtained by processes executing operations on shared objects, is a sequence of steps. $H|obj$ of history H is the subsequence of all invocation and response steps in H whose object names are obj. If all invocation and response steps 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 objet obj, the history H and $H|obj$ are the same.

We use $inv_H(op)$ to denote the position of the invocation of operation op in history H and use $rep_H(op)$ to denote the position of the corresponding response of op in H. Actually, for each operation op, it is not necessary to have a matching response. In this case, we call the operation op is *pending* and denote it as $rsp_H(op) = \infty$. Otherwise, we call the operation is *complete* in H. Obviously, all atomic operations are complete.

A history is *complete* if all its operations are complete. A completion of an incomplete history H is an extension H' of H, such that H' contains exactly the same operations as H but the responses are added for the pending operations in H.

Let H be a complete history. With each non-atomic operation op in H we could associate a time interval $I_H(op) = [inv(op), rsp(op)]$. Similarly, for an uncomplete history, we denote the interval with respect to the pending operation op by $I_H(op) = [inv(op), \infty]$.

A history is *sequential* if the first step in the history is an invocation (or an atomic step), and each invocation, except possibly the last one, is immediately followed by a corresponding response. It is obvious that a sequential history defines a total order over all operations.

Sequential Specification. The sequential specification S_{obj} for a shared object obj is a set of sequential histories on obj. A sequential history S satisfies the sequential specification S_{obj} of object obj, it means $S \in S_{obj}$. A sequential history S of object obj is valid (sometimes also called legal) if it satisfies 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 the 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 could be obtained by sorting the operations in H by their $t_H(op)$ values.

A history is linearizable if and only if there exists a sequential history S that linearizes H. In this case, S is called the linearization of H. Each linearization of H defines a point $t_H(op)$. For each operation op in history H, we call the point $t_H(op)$ linearization point of op. A shared object is linearizable if every history H of that object is linearizable.

Chapter 3

Dynamic Task Allocation Object

Our dynamic task allocation (DTA) type supports two operations, `DoTask()` and `InsertTask(`ℓ`)`, where ℓ 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, ℓ`)`, and a process associates task ℓ with 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. We say task ℓ is inserted if it is associated with a memory location M.

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`)`, only one receives *success* and the index ℓ of that task, while all the others receive $failure$.

A task is *done* 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.

Implementation of DTA

The main data structure we applied for the implementation of DTA is a binary tree.

Method 1: `DoTask()`

```
1  while true do
2  |  v ← root;
3  |  if v.surplus() ≤ 0 then
4  |  |  return ⊥;
5  |  end
6  |  /* Descent */;
7  |  while v is not a leaf do
8  |  |  (x_L, y_L) ← v.left.read();
9  |  |  (x_R, y_R) ← v.right.read();
10 |  |  s_L ← min(x_L − y_L, 2^height(v));
11 |  |  s_R ← min(x_R − y_R, 2^height(v));
12 |  |  r ← 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 ← v.left;
17 |  |  else
18 |  |  |  v ← v.rght;
19 |  |  end
20 |  end
21 |  /* v is a leaf */;
22 |  (x, y) ← v.read();
23 |  (flag, l) ← v.TryTask(task[y + 1]);
24 |  /* Update Insertion Count */;
25 |  v.CAS((x, y), (x, y + 1));
26 |  v ← v.parent;
27 |  Mark-up(v);
28 |  if flag = success then
29 |  |  return ℓ
30 |  end
31 end
```

Method 2: `InsertTask(`ℓ`)`

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

Lemma 1. *Let v be a tree node,*

(1) If (x, y) is the return value of $v.read()$, then there exists a set of x `InsertTask` operations and a set of y `DoTask` operation that have been counted at node v by the end of the execution of $v.read()$.

(2) If there are x `DoTask` operations that have been counted at v before the execution of $v.read()$, then the removal count value returned by $v.read()$ is not less than x.

Proof. TBD □

Lemma 2. *Consider a history H and an arbitrary operation op in H, let $t_H(op)$ be the point*

when op is counted at the root, then $t_H(op)$ is between $inv_H(op)$ and $rsp_H(op)$.

Proof. Without loss of generality, suppose process p executes DoTask operation op which has performed task ℓ successfully at the leaf. When it reaches the *root* and executes Mark-up(*root*), there will be two cases:

Case 1: It increments the removal count of *root* successfully via CAS (line 6, method 3) at point τ. If it is the first one in the linearization order of all CAS operation updating of the removal count with the same value, then $t_H(op) = \tau$, therefore $inv_H(op) < t_H(op) < rsp_H(op)$. Otherwise, we could let $t_H(op) = \tau'$, where τ' is the time when the first CAS operation in the linearization order updated the removal count. Thus $t_H(op) < \tau < rsp_H(op)$, Because only if the task has been performed then the removal count of root could be updated. so $t_H(op) > inv_H(op)$. Therefore, in this case, $inv_H(op) < t_H(op) < rsp_H(op)$ holds.

Case 2: It fails to increment the removal count. The CAS operation of p fails if and only if the value of the counts were updated by another process at a point before τ, suppose it is τ'. Please note that, we say the counts were updated, it means the removal count or the insertion count was updated because the two counts are stored in one memory location. Thus, there are two subcases.

Subcase 2.1: If it is the removal count that was incremented at τ', it means the DoTask operation has already been counted by another process. Thus, $t_H(op) \leq \tau' < rsp_H(op)$.

Subcase 2.2: If it is not the removal count but the insertion count that was updated at τ'. We notice that the CAS operation (line 6, method 3) will be repeated by p, during the second iteration, if the CAS of p succeed at τ'', then we could deduce that $t_H(op) \leq \tau''$, therefore $t_H(op) < rsp_H(op)$. If it fails again at point τ'' , then there must be another process updated the counts of root again. This time, the counts of the children will be noticed and the DoTask operation will propagate to the root. We could deduce $t_H(op) < \tau''$. Thus, $t_H(op) < rsp_H(op)$ as well. □

Under the above definitions and properties, we claim the point $t_H(op)$ when op is counted

at the root is the linearization point of operation *op*.

Lemma 3. *The dynamic task allocation object in the above figure is linearizable.*

Proof. Consider an arbitrary history H containing DoTask and InsertTask operations. We should prove for any execution of our algorithm, the total order given by the linearization point is verifies the uniqueness and validity. If H is not complete, then we let all processes that have not finished their operations continue to take steps in an arbitrary order until all operations are completed. Every operation will finally be done is ensured by our computation model and the randomness of our algorithm. This way we obtain a completion H' of H and it suffices to prove H' is linearizable. Thus, to prove this lemma, we should prove the total order obtained by sorting the operations by their $t_H(op)$ values is valid.

The uniqueness is obvious. When multiple processes are calling TryTask(ℓ) at the memory location, only one of them will receive success and the index ℓ of the task, all the other competitor processes will get failure. Task ℓ is performed successfully as long as the value of corresponding memory location is turned to 1. The following process will never repeatedly turn it be to 0 and turn 0 to 1 which is guaranteed by the our semantics of task insertion and removing.

Now we prove the validity, i.e. each task that is performed successfully must have been inserted before. We should prove the insertion operation of a task is always counted at the root before the removal operation. To prove this result holds for the root, we now prove it by induction from the leaf.

At the leaf, this holds because if and only if the insertion count of newly inserted task ℓ has been incremented (line 19, method 2) then the following removal operation could read that (line 20, method 1) to know the available task ℓ at the leaf and then try to perform it. In another word, suppose the task ℓ is inserted but the insertion count is not incremented (i.e. insertion has not been counted yet), then the following removal operation has no way to know the available task ℓ, perform it and increment the removal count. Thus, the removal

will not be counted.

For an arbitrary inner node v, we suppose, by the induction step and lemma 1, the result holds for children $v.left$ and $v.right$. We could notice that any process updates the insertion count and removal count as an atomic operation (line 6, method 3). If some remove operation has been counted at node v, then the corresponding insert operation for that task must have been counted at v simultaneously by the double compare-and-swap operation. Apply this to the root, then validity condition holds. □

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.