Centrum voor Wiskunde en Informatica
REPORT*RAPPORT*

Waitfree Distributed Memory Management by Create, and Read Until Deletion (CRUD)

W.H. Hesselink, J.F. Groote

Software Engineering (SEN)

SEN-R9811 July 31, 1998

Report SEN-R9811
ISSN 1386-369X

CWI
P.O. Box 94079
1090 GB Amsterdam
The Netherlands

Waitfree Distributed Memory Management by Create, and Read Until Deletion (CRUD)

Wim H. Hesselink

Dept. of Math. and Computing Science, University of Groningen

P.O. Box 800, 9700 AV Groningen, The Netherlands

wim@cs.rug.nl

Jan Friso Groote

CWI

P.O. Box 94079, 1090 GB Amsterdam, The Netherlands

Department of Mathematics and Computing Science, Eindhoven University of Technology

P.O. Box 513, 5600 MB Eindhoven, The Netherlands

jfg@cwi.nl

Abstract

The acronym CRUD represents an interface specification and an algorithm for the management of memory shared by concurrent processes. The memory cells form a directed acyclic graph. This graph is only modified by adding a new node with a list of reachable children, and by removing unreachable nodes. If memory is not full, the algorithm ensures waitfree redistribution of free nodes. It uses atomic counters for reference counting and consensus variables to ensure exclusive access. Performance is enhanced by using nondeterminacy guided by insecure knowledge. Experiments indicate that the algorithm is very suitable for multiprocessing.

1991 Mathematics Subject Classification: 68Q20, 68Q22
1991 Computing Reviews Classification System: D.m, D.2.4, E.1
Keywords and Phrases: Distributed Garbage Collection, Reference Counting, Shared Memory, Waitfree, Consensus, Terms

1 Introduction

The setting of this note is a system of concurrent sequential processes that operate on a common database of terms [BKV96]. Terms must be understood in their normal mathematical sense, i.e. typical terms are $2 + x$, $f(a, x, y, g(a, b))$. The use of terms is convenient as they are very well known objects for which many theories and tools are available, e.g. rewriting, unification and automated reasoning. The software coordination architecture described in [BeK98] uses terms as the primary objects to be exchanged between tools.

From an implementation point of view, terms are simply directed acyclic graphs where each node is labelled with a function name. It turns out that terms can be efficiently used when only a limited number of operations are available on terms. A term can be created once and inspected as long as

needed. Terms that are not in use any longer can be deleted by a garbage collector. The creating and deleting of terms is primarily a memory management problem.

The restriction to these operations on terms has two advantages. First, it allows sharing of subterms, reducing memory requirements substantially. Second, it allows parallel access to terms, as terms are basically static objects in memory.

There are also disadvantages. As terms cannot be changed, common operations such as substitutions must be carried out by copying the whole term.

Some implementations of CRUD algorithms exist [BKV96, BeK98]. These all assume sequential creation, access and deletion of terms. However, it is explicitly intended that terms can be accessed concurrently within shared memory environments. As our experiments in section 8 confirm, the use of synchronisation primitives to guarantee exclusive access is relatively slow, and does not scale up to more processors. Therefore, there is reason to look for a waitfree solution in which a process that needs a new node, gets one within bounded delay, independently of actions of other processes, cf. [Her91]. Since several processes may be contending for the same node, consensus is needed to decide which process succeeds. Consensus can be forced by delegating redistribution to a central garbage collector. We prefer not to create this bottleneck and therefore also distribute the recycling of nodes.

Thus, in comparison with e.g. [Jon92], we extend the concept of garbage collection to include waitfree redistribution. On the other hand, we simplify matters by the assumption that accessible terms are not modified, and by an extension of the repertoire of atomic instructions. Indeed, it is known that consensus needs more than atomic read-write variables, and therefore waitfree redistribution requires the strength of consensus variables.

It seemed to us that a mark and sweep garbage collection algorithm as in [Jon92] is not likely to yield sufficient performance. We therefore decided to aim at garbage collection by means of reference counting, cf. [JoL96]. By our simplifying assumptions, the usual objection that reference counting does not discern cyclic structures, does not apply.

Algorithms in which concurrent processes manipulate a shared pointer structure are error prone. We therefore provide a proof of the algorithm by means of invariants. Since the verification of invariants when processes concurrently execute array modifications is rather tricky, we have verified the invariants mechanically with the theorem prover Nqthm of Boyer and Moore, cf. [BoM88]. In this paper we give no details of the mechanical aspects of this proof (it is somewhat simpler than the proof in [Hes98a]). The mechanical proof is available at the Web site [Hes@].

We did some experiments to test the performance of our algorithm, and to compare it with standard memory management techniques (mainly in sequential settings). The experiments indicate a quite satisfactory behaviour, but conclusions must be tentative, since there are many parameters that can vary wildly.

Overview

In Section 2, we describe the data structure and we specify the interface procedures by means of preconditions and postconditions. In Section 3, we extend these specifications by safety properties and progress properties. In fact, the interface procedures can be called concurrently by different processes. Therefore, safety properties of the atomic steps are needed. In this Section we also prove that the properties imply that the graph remains acyclic.

In Section 4 we describe the available repertoire of atomic actions, and we make a start with the construction of the interface procedures. Section 5 deals with the aspects of garbage collection for the implementation. In Section 6, we construct the remaining interface procedures by combining various procedures constructed before. Section 7 discusses the verification of the properties promised in Section 3. In Section 8 we describe experiments, which indicate that the algorithm is very suitable for multiprocessing. Finally, Section 9 contains some conclusions.

2 The interface

In this section we describe the memory management interface, as offered to the application programmers. It consists of a shared data structure, and a number of procedures that can be used in the application processes. The application programmers are responsible that a procedure offered is called only when its precondition holds. It is therefore a proof obligation of the system, that the precondition of any interface procedure for a process p be stable under the actions of all processes $\neq p$.

The database is organized as a modifiable directed graph. It is convenient for arguing about correctness to regard the attributes of cells as arrays indexed by the unstructured type *Node*. Therefore, if n is of the type *Node*, the data of n is denoted by $data\,[n]$ (instead of $n.data$ as it would be if n was a record with a field *data*).

So, we have a set *Node* of (numbers of) nodes. We use $0 \notin Node$ to indicate the absence of a node, and define $Node0 = Node \cup \{0\}$. We assume that all nodes n are equivalent, i.e., have the same maximal degree. We number the children of a node by means of some type *Index*. Therefore, they form a Sequence, and the directed graph is given by a variable *sons*, according to the declaration

type *Sequence* = **array** *Index* **of** *Node0* ;
var *sons* : **array** *Node* **of** *Sequence* .

Thus, $sons\,[n]$ is the sequence of children of node n and $sons\,[n, i]$ is the ith child of node n.

Each application process maintains a private variable *roots* that holds the nodes the process has direct read access to. We write $roots.p$ for the value of *roots* of process p. We use a predicate $R(p, n)$ to express that process p is allowed to read the data of node n. We shall ensure that predicate $R(p, n)$ can only be invalidated by process p itself. We define

$$R(p, n) \quad \equiv \quad (\exists\, m \in roots.p ::\ m \xrightarrow{*} n)\ ,$$

where relation $\xrightarrow{*}$ is the reflexive transitive closure of relation $\rightarrow$ on *Node* defined by

$$m \rightarrow n \quad \equiv \quad (\exists\, i \in Index ::\ sons\,[m, i] = n)\ .$$

For access and modification of the database we provide the application processes with a number of commands, each consisting of a number of atomic instructions. In the presentation the keyword **privar** stands for a private variable of a process (cf. [Dij76]). The following procedure serves to extend the graph with a new node.

procedure *Make* (x : *Data*, y : *Sequence*, **privar** v : *Node*)
{ **pre** $R^*(p, y) \quad \wedge \quad roots.p = X$ **;**
post $roots.p = X \cup \{v\} \quad \wedge \quad data\,[v] = x \quad \wedge \quad sons\,[v] = y$ } ,

where p stands for the calling process and X is a specification constant to express the initial value of *roots*. The precondition $R^*(p, y)$ expresses that all children for the new node must be accessible to the caller p. Here, accessibility of a sequence y is defined by

$$R^*(p, y) \quad \equiv \quad (\forall\, i \in Index ::\ y\,[i] = 0 \quad \vee \quad R(p, y\,[i]))\ .$$

The requirement that procedure *Make* does not change the accessible part of the graph, will be expressed in Section 3 below.

Under the precondition $R(p, v)$, process p may inspect the contents of node v by calling

procedure *Read* (v : *Node* , **privar** x : *Data* , **privar** y : *Sequence*)
{ **pre** $R(p, v)$ **; post** $x = data\,[v] \quad \wedge \quad y = sons\,[v]$ } .

Accessibility to nodes can be transferred between processes. Assume that $v \notin roots.p$ holds, and that process q satisfies $R(q, v)$ and *has agreed to preserve that* until p has acknowledged reception of node v. Then process p may claim (direct) access to node v by calling

procedure *Accept* $(v : Node)$
$\{$ **pre** $v \notin roots.p = X$ **; guaranteed** $R(q, v)$ **;**
post $roots.p = X \cup \{v\}$ $\}$.

The application programmer who uses procedure *Accept* has to supply some coordination protocol such that process q does not release node v before process p has completed *Accept*. Note that p and q may be equal.

If it has $v \in roots.p$, process p can relinquish its rights on a node v by

procedure *Delete* $(v : Node)$
$\{$ **pre** $v \in roots.p = X$ **; post** $roots.p = X \setminus \{v\}$ $\}$.

Finally, we provide two procedures for memory management that, at the interface level, are equivalent to *skip*:

procedure *Serve* () ;
procedure *Browse* () .

Procedure *Serve* can be called by an application process that has time to do some garbage collecting. Procedure *Browse* is for a dedicated garbage collecting process. Both procedures are superfluous and only serve for smoother performance. Both are waitfree.

3 System properties

We need safety properties and progress properties. In fact, we want to express that the accessible part of the graph is never modified (safety), and that procedure calls terminate (progress). First some notation to express such properties formally.

We write $p : P \ \triangleright \ Q$ to express that, if precondition P holds and process p performs an atomic action, this action has postcondition Q. We write $P \ \triangleright \ Q$ to express that $q : P \ \triangleright \ Q$ holds for all processes q. We write p **in** Pd to express that process p is executing procedure Pd. We write $p : P \ o\!\!\rightarrow \ Q$ to express the existence of a constant k such that every execution that starts in a state where P holds and that contains at least k atomic steps of process p, contains a state that satisfies Q.

We characterize the reachable nodes of the graph by

$$ER(n) \quad \equiv \quad (\exists\, q \in Process :: R(q, n)) \ .$$

The main safety properties are that $data\,[n]$ and $sons\,[n]$ of a reachable node n are not modified, and that $roots.q$ is modified only when process q itself executes *Make*, *Accept*, or *Delete*. This is formalized in the requirements

(Sq0) $\quad ER(n) \ \wedge \ data\,[n] = X \quad \triangleright \quad data\,[n] = X$;
(Sq1) $\quad ER(n) \ \wedge \ sons\,[n, i] = X \quad \triangleright \quad sons\,[n, i] = X$;
(Sq2) $\quad p : \ p \neq q \ \wedge \ roots.q = X \quad \triangleright \quad roots.q = X$;
(Sq3) $\quad p : \ \neg\,(p$ **in** $Delete) \ \wedge \ X \in roots.p \quad \triangleright \quad X \in roots.p$;
(Sq4) $\quad p : \ p$ **in** $Delete(v) \ \wedge \ v \neq X \in roots.p \quad \triangleright \quad X \in roots.p$.

As before, X is a specification constant (logical variable) to express that the value of a modifiable field is not changed in the step, or that a protected node remains protected.

Waitfree termination of five of the six interface procedures is expressed in

(Sq5) $\quad p : \ p$ **in** $Pd \quad o\!\!\rightarrow \quad \neg\,(p$ **in** $Pd)$
for $Pd \in \{Read, Accept, Delete, Serve, Browse\}$.

Procedure *Make* can only be guaranteed to terminate if there are free nodes to be found. Therefore, in the progress assertion for *Make*, we need an alternative *Full* in the following way:

(Sq6) $\quad p:\ p \textbf{ in } \mathit{Make} \quad o\!\!\rightarrow \quad \neg\,(p \textbf{ in } \mathit{Make}) \quad \vee \quad \mathit{Full}\ .$

We require that, if *Full* holds, all nodes are in use or there exists a process q that will negate *Full* within a bounded number of steps of q.

Clearly, the alternative *Full* violates waitfreedom, but this is unavoidable, since processes are allowed to claim as much memory as needed. The problem is also slightly complicated by the possibility that a process stops functioning when it is about to make nodes free for reuse. We come back to predicate *Full* in Section 7.

We now show that reachability $R(p,n)$ can only be falsified by process p itself, and only in procedure *Delete*. In fact, for processes p, q, and node n, we claim

(Hq0) $\quad p:\ p \neq q \quad \wedge \quad R(q,n) \quad \rhd \quad R(q,n)\ ;$
(Hq1) $\quad p:\ \neg\,(p \textbf{ in } \mathit{Delete}) \quad \wedge \quad R(p,n) \quad \rhd \quad R(p,n)\ .$

Both assertions follow from the definition of $R(q,n)$, via (Sq1), (Sq2), and (Sq3), by induction in the length of the path to node n in the precondition.

It follows from (Sq2) and (Hq0) that, indeed, the precondition of any interface procedure for a process p be stable under the actions of all processes $\neq p$.

We turn to the point that the graph should remain acyclic. For this purpose, we postulate that an unreachable node does not become a new child:

(Sq7) $\quad \neg\, ER(n) \quad \wedge \quad \mathit{sons}\,[m,i] \neq n \quad \rhd \quad \mathit{sons}\,[m,i] \neq n\ .$

It follows from (Sq7) and (Sq1) that we have

(Hq2) $\quad \neg\, ER(n) \quad \wedge \quad \neg\,(m \rightarrow n) \quad \rhd \quad \neg\,(m \rightarrow n)\ ,$
$\qquad\quad ER(m) \quad \wedge \quad \neg\,(m \rightarrow n) \quad \rhd \quad \neg\,(m \rightarrow n)\ .$

Now assume that an atomic action has in its postcondition a cycle of nodes $v_i \rightarrow v_{i+1}$ for $0 \leq i < k$, where $k \geq 1$ and $v_k = v_0$. Then the precondition of this atomic action satisfies, for all i with $0 \leq i < k$,

$$\begin{array}{l} \neg\,(v_i \rightarrow v_{i+1}) \quad \Rightarrow \quad ER(v_{i+1})\ , \\ ER(v_i) \quad \Rightarrow \quad v_i \rightarrow v_{i+1}\ , \\ ER(v_i) \quad \wedge \quad v_i \rightarrow v_{i+1} \quad \Rightarrow \quad ER(v_{i+1})\ , \end{array}$$

by the formulas (Hq2) and the definition of *ER*. It follows that the cycle also existed in the precondition of the atomic action. For, in the precondition, the absence of an edge of the cycle implies that some v_j is reachable, and if some v_j is reachable then all v_j are reachable and all edges are present.

We now assume that, initially, the graph $(\mathit{Node}, \rightarrow)$ has no cycles. Then it follows that the graph invariantly has no cycles.

4 The implementation

We turn to a proposal for implementing the system in shared memory.

We use the following repertoire of elementary instructions. Every elementary instruction refers to at most one shared variable, cf. [OwG76], preferably at most once. We have two types of shared variables `t` that can occur more than once in an atomic instruction: counters and consensus variables. Apart from reading and writing, such a variable `t` has one of the special instructions

$$\begin{array}{l} \texttt{t} := \texttt{t} \pm 1\ , \text{ or } \texttt{t}{+}{+} \textbf{ and } \texttt{t}{-}{-} \quad \{\text{counter}\}\ ; \\ \textbf{if } \texttt{t} = 0 \textbf{ then } \texttt{t} := w \textbf{ fi} \quad \{\text{consensus}\}\ ; \end{array}$$

where w is a private variable, and $\pm$ stands for either $+$ or $-$. We assume that modifications of private variables can be combined atomically with an operation on a shared variable. Moreover, we assume that the conditional setting of a consensus variable is combined with the setting of a boolean flag, so that the **then** branch and the virtual **else** branch can be combined with private actions. This is called strong consensus in [Hes98a].

In our experiments (see Section 8) we had to implement the atomic counter modification by means of

```
repeat  tmp := t ;
    ⟨ b := (tmp = t) ; if b then t := tmp ± 1 fi ⟩
until  b .
```

The brackets $\langle\ \rangle$ are used to enclose an atomic region, which is a strong compare&swap operation here. The loop is not waitfree, but turns out to work satisfactorily.

We now turn to the implementation of the CRUD interface. It is trivial to implement

```
procedure Read (v : Node , privar x : Data , privar y : Sequence ) =
{ pre R(p, v) ; post x = data[v]  ∧  y = sons[v] }
    x := data[v] ;   y := sons[v]
end .
```

We use the notation $v.q$ to refer to the value of a private variable v of process q. For an efficient implementation of *Make*, we give each process a private variable *res* of type **set** of *Node* to hold free nodes reserved for private use. Now one of the problems is to guarantee that, for every process q, if needed, $res.q$ becomes nonempty within bounded delay. Experience seems to show that this must be made a shared responsibility for all processes together. We therefore provide every process with a consensus variable $waiting[q]$ to receive free nodes, according to the declaration

```
waiting :  array Process of Node0 .
```

By convention, $waiting[p] = 0$ means that process p is waiting for a new node, while $waiting[p] = n$ with $n \neq 0$ means that p can use the new node n by means of

```
procedure receive () =
{ pre  waiting[p] ≠ 0 }
25      v := waiting[p] ;
26      res := res ∪ {v} ;  waiting[p] := 0
end .
```

Here each numbered instruction is one atomic command; we give each process q a corresponding instruction pointer $pc.q$. The bigger atomic instruction 26 is allowed since *res* and v are private variables. We use numbered instructions and (below) **goto**s since the concurrency forces us in the invariants to be very precise where which property holds. Moreover, the use of structured programming with **if** and **while** tends to obscure which instructions are regarded as atomic.

We assume that processes share their wealth in a fair way. For the purpose of redistribution of nodes, we give every process a private variable *fav* of type *Process* (for current favourite). We say that a function *next* traverses a set X (cf. [Hes98b]) iff, for every pair x, $y \in X$, there is a number k with $next^k(x) = y$. It follows that $next^k(x) = x$ for $k = \#X$. We give every process a private function *nextp* that traverses *Process*, the set of process numbers, to choose the next favourite. A process may try to share its wealth by executing

```
procedure share (v : Node) =
{ pre  v ∈ res }
29      if  waiting[fav] = 0  then
```

$waiting\,[fav] := v\ ;\ res := res \setminus \{v\}$ **fi** ;
30 $fav := nextp(fav)$
end .

Here we use that *waiting* is an array of consensus variables and that actions on the private variable *res* may be combined atomically. Note that the value of *res* is retained if the test fails.

For the purpose of garbage collecting, we introduce reference counting by means of a shared array

cnt : **array** *Node* **of** *Integer*

We assume available the atomic increment and decrement operations $cnt\,[n]{+}{+}$ and $cnt\,[n]{-}{-}$. The idea is that $cnt\,[n]$ estimates the number of edges directed towards n plus the number of processes that have direct access to n. More precisely, we postulate for all nodes n:

(Jq0) $cnt\,[n] = (\#(m,i) \in Edge :: sons\,[m,i] = n) + (\#q \in Process :: n \in roots.q) + (\#q \in Process :: waiting\,[q] = n) + (\#q$ **at** $(*) :: n = w.q)$.

Here *Edge* is the set of pairs (m,i) where m is a node, and i is an index. We use the notation q **at** $(*)$ to indicate that the next action of process q is marked with $(*)$, and we assume that every process q **at** $(*)$ has a private variable w, which will be used if the *cnt* of some child must be modified.

In order to prove that (Jq0) is preserved when a process executes instruction 26 of *receive*, we postulate the invariants

(Jq1) $pc.q = 26 \;\Rightarrow\; waiting\,[q] = v.q$;
(Jq2) $n \in res.q \;\Rightarrow\; cnt\,[n] = 1$.

Note that (Jq0) and (Jq2) together with the typing restriction $res.r \subseteq Node$ imply

(Hq3) $waiting\,[q] \notin res.r$.

To preserve (Jq1) and (Jq2) in *receive*, we postulate

(Jq3) $24 < pc.q \leq 26 \;\Rightarrow\; waiting\,[q] \neq 0$;
(Jq4) $waiting\,[q] \neq 0 \;\Rightarrow\; cnt\,[waiting\,[q]] = 1$.

At this point the reader is invited to verify that (Jq0), (Jq1), (Jq2), (Jq3), (Jq4) are preserved by all atomic actions in the procedures *Read*, *receive*, and *share*. Note that the preconditions of *receive* and *share* are used in these verifications.

We can now easily implement *Accept*.

procedure *Accept* $(v : Node) =$
{ **pre** $v \notin roots.p = X$ **; guaranteed** $R(q,v)$ **;**
post $roots.p = X \cup \{v\}$ }
33 $cnt\,[v]{+}{+}\ ;\quad roots := roots \cup \{v\}$
end .

To prove that *Accept* preserves (Jq2) and (Jq4), we use the guarantee $R(q,v)$ together with the predicates

(Hq4) $n \in res.q \;\Rightarrow\; \neg\, ER(n)$,
(Hq5) $n = waiting\,[q] \;\Rightarrow\; \neg\, ER(n)$,

which follow from (Jq0) and (Jq2).

We introduce a shared variable *clean* of type **array** *Node* **of** *Boolean* with the invariant

(Jq5) $clean\,[n] \;\Rightarrow\; sons\,[n,j] = 0$.

We now turn to a partial implementation of *Make*.

Since we want a fine grain of atomicity for all instructions concerning shared memory, and no more determinacy than necessary, we encode the loop over the indices by means of three jumps and a shrinking set of indices F, which is a private variable.

```
          procedure branch (x : Data, y : Sequence, v : Node) =
          { pre R*(p, y)  ∧  roots.p = X  ∧  v ∈ res.p ;
            post roots.p = X ∪ {v}  ∧  data[v] = x  ∧  sons[v] = y }
41          data[v] := x ;
42          clean[v] := false ;   F := Index ;
43          if F = ∅ then goto 47 fi ;
44              choose i ∈ F ;   F := F \ {i} ;
                w := y[i] ;   if w = 0 then goto 43 fi ;
45              cnt[w] + + ;
46        (*)   sons[v, i] := w ;   goto 43 ;
47          roots := roots ∪ {v} ;   res := res \ {v} ;
48        end .
```

To prove preservation of (Jq0) in 46, of (Jq2) at 45, and of (Jq5) at 46, we postulate the invariants

(Jq6) $44 < pc.q \leq 46 \;\Rightarrow\; sons[v.q, i.q] = 0$;
(Jq7) $40 < pc.q \leq 47 \;\wedge\; y.q[j] \neq 0 \;\Rightarrow\; R(q, y.q[j])$;
(Jq8) $42 < pc.q \leq 47 \;\Rightarrow\; \neg clean[q]$.

For preservation of (Jq6) in 44, we postulate

(Jq9) $42 < pc.q \leq 46 \;\wedge\; j \in F.q \;\Rightarrow\; sons[v.q, j] = 0$.

Preservation of (Jq6) when another process executes 46 will follow from (Jq0) and (Jq2).

Preservation of (Jq9) in 42 follows from (Jq5) and the new postulate

(Jq10) $v.q \in res.q \;\wedge\; \neg\, clean[v.q] \;\Rightarrow\; 42 < pc.q \leq 47$.

Here it may be mentioned that, in the mechanical proof, we treat the parameter v of process q as a private variable $v.q$ that is nondeterministically modified at every procedure call. Preservation of (Jq10) is proved by means of the new postulates

(Jq11) $waiting[q] \neq 0 \;\Rightarrow\; clean[waiting[q]]$;
(Jq12) $n \in res.q \;\wedge\; \neg\, clean[n] \;\Rightarrow\; n = v.q$.

At this point the above invariants (Jq...) can all be proved. In these proofs we also use the following obvious invariants, which are only concerned with the private variables of a single process:

(Pq0) $pc.q = 29 \;\Rightarrow\; v.q \in res.q$;
(Pq1) $pc.q = 33 \;\Rightarrow\; v.q \notin roots.q$;
(Pq2) $40 < pc.q \leq 47 \;\Rightarrow\; v.q \in res.q$;
(Pq3) $44 < pc.q \leq 46 \;\Rightarrow\; w.q = y.q[i.q] \neq 0$;
(Pq4) $44 < pc.q \leq 46 \;\Rightarrow\; i.q \notin F.q$.

Of course, we also need the application guarantee of *Accept*:

(AG) $pc.q = 33 \;\Rightarrow\; ER(v)$.

Finally, for the specification of *branch*, we observe that $pc.q = 48$ implies $sons[v.q, j] = y.q[j]$, as follows from the invariants

(Jq13) $42 < pc.q \leq 48 \;\Rightarrow\; sons\,[v.q, j] = y.q\,[j] \;\vee\; j \in F.q \;\vee\; (j = i.q \wedge 44 < pc.q \leq 46)$;
(Pq5) $46 < pc.q \leq 48 \;\Rightarrow\; F.q = \emptyset$.

Note that proofs of invariance may be circular: the assumption is that all invariants hold in the precondition of every atomic instruction, and for each invariant one then proves that it holds in the postcondition of every instruction.

5 Garbage collection

We now come to the point where nodes are made free again. If $n \in roots.p$ holds, process p may relinquish its rights on n (and the dependent nodes) by removing n from $roots.p$ and decrementing $cnt\,[n]$.

It is attractive to combine this with garbage collection, if $cnt\,[n] = 1$ holds in the precondition. This idea is not sufficient for garbage collection since $cnt\,[n] > 1$ is not stable in the precondition: two processes may observe that $cnt\,[n] = 2$ and both decide to decrement $cnt\,[n]$ without garbage collection. We therefore decide to do garbage collection for nodes with $cnt\,[n] = 0$. Indeed, when decrementing $cnt\,[n]$ establishes $cnt\,[n] = 0$, the acting process can observe this. Unfortunately, there may be more than one process that observes it and wants to use this knowledge. We therefore treat $cnt\,[n] = 0$ as *insecure* knowledge that is not needed for correctness but only used for improving performance.

We give every process a private variable *list* which holds a bounded list of nodes n with a fair probability of $cnt\,[n] = 0$. The variable *list* is important for the performance of the system, but is formally superfluous: it does not occur in the invariants. It is used as follows. Whenever a process decrements the counter of a node v, it subsequently calls

procedure *collect* $(v : Node) =$
  if $cnt\,[v] = 0$ **then** $list := truncate\ (v : list)$ **fi**
end .

Here v is placed at the head of the list *list* and, if in this way *list* becomes too long, the last element of *list* is removed.

In particular, if a process relinquishes a root, it should do so by means of

procedure *Delete* $(v : Node)$
{ **pre** $v \in roots.p = X$ **; post** $roots.p = X \setminus \{v\}$ }
65 $cnt\,[v] - -$; $roots := roots \setminus \{v\}$;
66 *collect* (v)
end .

To prove preservation of (Jq0), we use the invariant

(Pq6) $pc.q = 65 \;\Rightarrow\; v.q \in roots.q$.

It follows from (Jq0) that $cnt\,[n] = 0$ implies that node n is free. A free node can be claimed by any process that needs new nodes. Therefore, if two or more processes want to claim the same free node they need consensus to decide which claimant succeeds. We therefore introduce locking of nodes, by means of shared consensus variables

$lock$: **array** *Node* **of** *Boolean* .

We now introduce the garbage collecting procedure *untarget* that tries to obtain a node v for *res*, after resetting its targets if necessary.

```
        procedure untarget (v : Node) =
51          if ¬lock[v] then lock[v] := true else return fi ;
52          if cnt[v] ≠ 0 then goto 61 fi ;
53          if clean[v] then goto 60 else F := Index fi ;
54          if F = ∅ then goto 59 fi ;
55              choose i ∈ F ;   F := F \ {i} ;
                w := sons[v, i] ;   if w = 0 then goto 54 fi ;
56              sons[v, i] := 0 ;
57      (*)     cnt[w] − − ;
58              collect (w) ;   goto 54 ;
59          clean[v] := true ;
60          cnt[v] := 1 ; res := res ∪ {v} ;
61          lock[v] := false
        end .
```

Note that *lock* is an array of "strong" consensus variables, see the atomic instruction 51.

To prove preservation of (Jq0) in 56, 57, 60, we postulate

(Kq0) $pc.q = 56 \;\Rightarrow\; w.q = sons[v.q, i.q]$;
(Kq1) $52 < pc.q \leq 60 \;\Rightarrow\; cnt[v.q] = 0$.

Preservation of (Jq5) follows from

(Kq2) $pc.q = 59 \;\Rightarrow\; sons[v.q, j] = 0$.

Preservation of (Jq10) at 60 follows from

(Kq3) $pc.q = 60 \;\Rightarrow\; clean[v.q]$.

Preservation of (Kq1) under *Accept* and instruction 45 follows from

(Hq6) $cnt[n] = 0 \;\Rightarrow\; \neg\, ER(n)$,

which follows from (Jq0). Preservation of (Kq2) in 54 follows from

(Kq4) $53 < pc.q \leq 58 \;\wedge\; sons[v.q, j] \neq 0 \;\wedge\; j \notin F.q$
$\Rightarrow\; j = i.q \;\wedge\; pc.q = 56$.

We now have to prove that the above invariants, especially (Kq0) and (Kq1), are not violated by another process that executes *untarget*. So we want to have interference freedom, as expressed by

(Kq5) $51 < pc.q \leq 61 \;\wedge\; 51 < pc.r \leq 61 \;\wedge\; v.q = v.r \;\Rightarrow\; q = r$.

This is accomplished by locking. Preservation of (Kq5) easily follows from the invariant

(Kq6) $51 < pc.q \leq 61 \;\Rightarrow\; lock[v.q]$.

Preservation of (Kq6) is proved by means of (Kq5).

6 A strategy for redistribution

The elements of *list* are good candidates for procedure *untarget*. If *list* is empty, however, the process can choose an arbitrary node, if it does so in a fair way. We therefore give every process a private variable *nod* and a private function *nextn* that traverses the set *Node*.

The next procedure serves to make *res* nonempty.

```
procedure get () =
    while res = ∅ do
        if waiting[p] ≠ 0 then receive () else search () fi
    od
end .
```

where

```
procedure search () =
    if list = ∅ then v := nod ; nod := nextn(nod)
    else v := head (list) ; list := tail (list) fi ;
    untarget (v) ;
    if v ∈ res then share (v) fi ;
end .
```

The call of *share* in *search* is needed to guarantee waitfreedom: each process is served within bounded delay (although the bound depends on the number of processes and the size of the memory).

In Section 7, we'll show that procedure *get* is waitfree provided there are always enough free nodes to be found.

Now, finally, we construct the procedure to call when a new node must be made:

```
procedure Make (x : Data, y : Sequence, privar v : Node) =
{ pre R*(p, y) ∧ roots.p = X ;
  post roots.p = X ∪ {v} ∧ data[v] = x ∧ sons[v] = y }
    get () ;
    choose v ∈ res ;
    branch (x, y, v)
end .
```

We turn to memory management activities that are invisible at the interface level. The application processes are allowed to accumulate nodes in their sets *res*, provided they try to share and *res* does not become too large. We therefore provide each process with a private constant $maxres \geq 1$, with the invariant

(Pq7) $\quad \#res.q \leq maxres.q$.

It is easy to see that (Pq7) is only threatened by *untarget*, and that it is preserved when *untarget* is called with precondition $\#res.p < maxres.p$.

Therefore, whenever process q has time to do some garbage collecting, it may call

```
procedure Serve () =
    if #res < maxres then search ()
    else choose v ∈ res ; share (v) fi
end .
```

For the sake of efficiency it may be preferable to have one additional garbage collecting process *gc* with $maxres.gc = 1$. Process *gc* only frequently calls

```
procedure Browse () =
    search () ;
    if res ≠ ∅ then
        choose v ∈ res ;
        ⟨ res := ∅ ; cnt[v] := 0 ⟩
    fi
end .
```

It preserves (Pq7) since it has $res.gc = \emptyset$ in the idle states.

Remark. The conditional jump in 53 of *untarget* is only useful if *Index* is large and the probability of $cnt\,[v] = 0 \ \wedge \ clean\,[v]$ is sufficiently high. In particular, the jump is useless if we have the invariant

$$clean\,[n] \quad \Rightarrow \quad (\exists\, q :: n = waiting\,[q] \ \vee \ n \in res.q \ \vee \ (n = v.q \ \wedge \ pc.q = 60)) \ .$$

This predicate is preserved by all procedures except for *Browse*. So, in a system that does not use *Browse* or in which *Index* is very small, we had better remove the jump and replace instruction 53 by

53' $\quad F := Index$;

In that case, variable *clean* becomes a ghost variable and can therefore be removed from the algorithm. □

7 Verification of properties

It remains to verify the global properties (Sq0) through (Sq7). Since *data* and *sons* are modified only in *branch* and *untarget*, the properties (Sq0), (Sq1), (Sq7) follow from (Hq4) and (Pq2), and (Hq6) and (Kq1), and the specification of *branch*. Since *roots* is a private variable, the validity of (Sq2), (Sq3), and (Sq4) is easily verified. The loops in *branch* and *untarget* are bounded by the size of *Index*. Therefore, the only unbounded loop occurs in *get*. Since *get* is only used in the interface procedure *Make*, this implies that the other interface procedures are waitfree, i.e., (Sq5).

In order to prove (Sq6), we define predicate *Full* by

$$Full \quad \equiv \quad (\forall\, n :: cnt\,[n] > 0 \quad \vee \quad lock\,[n]) \ .$$

Now, informally, property (Sq6) is shown as follows. If *Full* is false during an execution sequence in which process p executes *get*, there are always unlocked nodes n with $cnt\,[n] = 0$. After having exhausted its list *list*, process p traverses the set *Node* and eventually finds unlocked nodes with $cnt\,[n] = 0$. It executes *untarget* on every such node, and then calls *share*; this advances $fav.p$. Therefore, if process p does not terminate early enough, a state is reached with $fav.p = p$. Then process p serves itself, and the call of *get* terminates. The argument can be made formal by means of the techniques developed in [Hes98b].

We finally show that, if *Full* holds, then all nodes are in use, or there is a process that will negate *Full* within a bounded number of steps. For this purpose, we formalize "node n being in use" by predicate $Used(n)$ defined by

$$Used(n) \quad \equiv \quad ER(n) \quad \vee \quad (\exists\, q :: waiting\,[q] = n \quad \vee \quad n \in res.q) \ .$$

The definition of *Used* together with the invariants (Jq5), (Jq6), (Jq9), (Jq10), (Jq11) implies

(Hq7) $\quad Used(m) \quad \wedge \quad m \to n \quad \Rightarrow \quad Used(n)$.

We now define

$$LL(n) \quad \equiv \quad (cnt\,[n] = 0 \quad \wedge \quad lock\,[n]) \quad \vee \quad (\exists\, q :: pc.q = 57 \quad \wedge \quad n = w.q) \ ,$$

and claim

Lemma. Let n be a node with $\neg\, Used(n)$ and $(\forall\, m : m \to n : Used.m)$. Then we have
(a) $\quad cnt\,[n] \quad = \quad (\#q :: \ pc.q = 57 \quad \wedge \quad n = w.q)$;
(b) $\quad Full \quad \Rightarrow \quad LL(n)$.

Proof. (a) Using (Hq7), we obtain $\neg\, (m \to n)$ for all nodes m. Using (Jq0) and $\neg\, Used(n)$, we then get $cnt\,[n] = (\#q$ **at** $(*) :: n = w.q)$. The marker (*) only occurs at 46 in *branch*, and at 57 in *untarget*. The assertion follows, since $w.q$ is *Used* at 46 because of (Pq3).

(b) If $cnt\,[n] > 0$, then $LL(n)$ follows from (a). If $cnt\,[n] = 0$, then $LL(n)$ follows from *Full*. □

Since the graph $(Node, \rightarrow)$ is acyclic, it follows that we have

Theorem. Assume that *Full* holds. Then every node n satisfies

$$Used(n) \quad \vee \quad (\exists\, m :: LL(m) \quad \wedge \quad m \stackrel{*}{\rightarrow} n)\ .$$

□

This theorem shows the absence of memory leakage. In fact, for every node n with $LL(n)$, there is a unique process q that will release n within a bounded number of steps. Therefore, if *Full* holds and not all nodes are *Used*, there exists at least one process that will negate *Full* within a bounded number of steps. Note that establishing $\neg$ *Full* is not waitfree, since the actions of a specific process may be required.

The efficiency of redistribution is hard to estimate. If every process claims new nodes more or less in the same rate as it relinquishes old ones, communication of the nodes plays no significant role. If these rates differ wildly, however, the sets *list* of some processes are often empty in which case these processes to some extent rely on the charity of other processes in procedure *share*, although they also get new nodes by inspection of arbitrary nodes. Then it is important that congestion of node inspections is avoided by taking the traversing functions *nextn* of the processes all different, see [Hes96], section 8.1. Similarly, to avoid congestion of charity, one should take the traversing functions *nextp* all different.

8 Performance

How does the algorithm proposed compare in performance to the more standard algorithms in use for the CRUD problem? To answer this question we implemented four different algorithms including the one proposed in the previous sections.

Efficiency of this kind of algorithms is hard to estimate. Since the algorithms for this problem generally use constant time for all their operations on nodes, we must actually compare constant times necessary per operation. Such constant times are heavily influenced by machine architecture, caching strategy, processor scheduling, the compiler that is being used, the details of the program, and the data being processed. The CRUD algorithm is very strongly influenced by the rate of occupation of the nodes array and the level of redistribution of nodes over different processes. On parallel architectures, it is not easy to keep these under control. There are many parameters that can vary and some of them can be tuned.

Therefore, we only provide these figures as an indication, and urge the reader to be careful to draw conclusions for different settings.

We have implemented the following four algorithms in C[1]. We use the compare and swap instruction (`uscas32`) offered by the IRIX operating system to implement atomic counters and consensus variables.

[**A**] The 'classical' algorithm where all free nodes are stored in a 'free list'. This algorithm is only suitable for one single process. When the process needs a new node, it uses normal pointer manipulation to get it from the free list. Every node has a counter to count the number of nodes that point to it. To *Accept* a node, this counter is incremented by one and to *Delete* a node, the counter is decreased. If this counter becomes zero, the node is put at the head of the free list. This algorithm is very efficient and it is considered here to understand how many nodes can maximally be recycled and accessed per second, when there is no overhead due to multiprocessing.

[1] An implementation suitable for Silicon Graphics computers running the IRIX operating system can be received by contacting the authors.

	#P	# *list*	**A**	**B**	**C**	**CRUD**
read	1	63	$1.6\ 10^6$	$1.6\ 10^6$	$1.7\ 10^6$	$1.6\ 10^6$
construct	1	63	$6.2\ 10^5$	$5.8\ 10^5$	$1.8\ 10^5$	$1.9\ 10^5$
read	1	0	-	-	-	$1.6\ 10^6$
construct	1	0	-	-	-	$9.4\ 10^4$
read	30	63	-	-	$8.8\ 10^5$	$1.6\ 10^6$
construct	30	63	-	-	$1.6\ 10^4$	$2.4\ 10^5$
read	30	0	-	-	-	$1.5\ 10^6$
construct	30	0	-	-	-	$1.1\ 10^5$

Table 1: Experimental results on an SGI O2 single processor machine

	#P	# *list*	#M	**A**	**B**	**C**	**CRUD**
read	1	63	-	$3.3\ 10^6$	$3.3\ 10^6$	$3.3\ 10^6$	$3.0\ 10^6$
construct	1	63	-	$3.8\ 10^5$	$3.9\ 10^5$	$4.1\ 10^5$	$1.5\ 10^5$
read	1	0	-	-	-	-	$2.5\ 10^6$
construct	1	0	-	-	-	-	$1.4\ 10^5$
read	30	63	1	-	-	$1.9\ 10^6$	$3.5\ 10^6$
construct	30	63	1	-	-	$9.0\ 10^3$	$2.5\ 10^5$
read	30	63	2	-	-	$1.6\ 10^6$	$1.0\ 10^7$
construct	30	63	2	-	-	$8.2\ 10^3$	$1.5\ 10^5$
read	30	0	1	-	-	-	$3.0\ 10^6$
construct	30	0	1	-	-	-	$1.6\ 10^5$
read	30	0	2	-	-	-	$9.9\ 10^6$
construct	30	0	2	-	-	-	$5.7\ 10^4$

Table 2: Experimental results on an SGI Onyx2 multiprocessor machine

[**B**] The same algorithm as above, with the difference that incrementing and decrementing the counters are made atomic using compare and swap instructions. Such instructions are also used to "protect" addition to and removal from the free list, although this is insufficient for multiprocessing. This algorithm is used to detect the effect of using the compare and swap instruction on the overall performance of the system.

[**C**] In order to understand whether it would be fruitful to implement a CRUD algorithm with explicit synchronisation, we extend algorithm **A** with a semaphore and let the interface functions *Make*, *Accept* and *Delete* from section 2 start by grabbing the semaphore and by releasing it just before exiting the function. The function *Read* does not need to be be protected in this way and *Serve* and *Browse* have not been implemented.

[**CRUD**] The fourth implemented algorithm is the one proposed in this article.

For the sake of the experiment, we wrote a program to construct binary trees with 63 nodes, to read each tree a number of times and to delete it afterwards. This is then typically repeated 1000 to 100000 times. A number of these processes can be put in parallel, all using the same repository of nodes, containing 2000 nodes.

This experiment was run on a single and a multi processor machine. The single processor machine is a Silicon Graphics O2 with a 180Mhz R5000 microprocessor, a secondary cache of 512Kb and sufficient main memory. We used the standard `cc` compiler on this machine and compiled using the -O optimisation flag. The results for this machine are listed in Table 1.

The multiprocessor machine is a Silicon Graphics Onyx2 with two 195Mhz R10000 processors and 4MB of secondary cache for each processor and sufficient main memory. We used the MIPSpro Compiler: Version 7.2.1 and compiled the program using the -O optimisation flag. Results for this machine are summarized in Table 2.

In the tables we list how many nodes can be constructed and how many can be accessed per second. Constructing includes the time for disposing a node, and reading means on average accessing the node three times for reading the contents, the number of sons and on average one pointer to a son. The times have been obtained by measuring the 'real' running time of the total program using the `time` function in unix. The figures behind *construct* in the tables have been obtained by running the program in such a way that the test trees are only constructed but never read. The number of constructed nodes is then divided by the total running time of the program. The figures behind *read* have been obtained by reading the test trees a number of times after construction. Using the "construct only" experiment it is estimated how much running time is used to read nodes. This time is divided by the number of times a node is accessed.

The experiments have been carried out with either 1 or 30 threads constructing and reading trees in parallel (see column headed with #P). In order to assess the effect of the length of *list* the program has been run with the length of *list* being set to 0 (non optimal) and equal to 63 (very optimal). On the multiprocessor machine the program has been run on either one or two processors (see column headed by #M). In the latter case, the reported time is the time for both processors together.

We see that the figures are generally quite close. For constructing nodes, the performance of the **CRUD** algorithm appears to be slightly lower than the other algorithms when only one program is running in parallel. The performance of algorithm **C** however, drops dramatically with 30 parallel threads. For the **CRUD** algorithm performance increases under these circumstances. A remarkable and unexpected phenomenon is the performance degradation that occurs when using more processors. For algorithm **C** the use of two processors seems to lead to a slight performance degradation. This can be explained by the fact that – apparently due to synchronization – there is only enough work for one processor, forcing the other to be idle. This behaviour is also displayed on 4 processor machines where only one processor can be kept employed. For the **CRUD** algorithm, we see that additional processors work at a 100% load. However, in case of two processors, this leads to a performance degradation of almost 50%. We explain this by the need for cache synchronisation. Actually, inspection of the number

of executed instructions when one or two processors are working show little difference, suggesting that two processors can finish the work in half the time, but the time to execute an instruction per processor is almost a factor four higher in the latter case.

The time to read a node seems rather constant, except that for algorithm **C** it takes twice as much time to inspect nodes, when there is a lot of synchronization between processes. This even is worsening slightly, when more processors are added. For the **CRUD** algorithm, we see that an unexpected 3 fold performance increase occurs when adding a single processor, for which we do not have a proper explanation.

In the experiments, almost all running time is used for constructing and reading nodes, whereas in realistic circumstances processes will also perform local computations. We therefore expect that the performance of realistic applications on multiprocessing systems is even better.

So, we can conclude that the **CRUD** algorithm appears to be very suitable when there are many parallel processes, clearly beating alternatives with explicit synchronization. A weak point of the algorithm is that its performance may degrade when it has to search for free nodes.

9 Conclusions

The CRUD interface is a useful and viable abstraction for a graph-like data structure shared by concurrent processes. If atomic counters and consensus variables are available, the CRUD algorithm seems to be a very suitable implementation.

This supports the claim that multiprocessor shared memory machines should provide atomic counters and consensus variables (or even compare&swap variables).

Acknowledgements.

We are grateful for comments and suggestions of Jan Jongejan, Pieter Olivier and John Tromp.

References

[BeK98] Bergstra, J.A., and Klint, P.: The discrete time ToolBus – a software coordination architecture. Science of Computer Programming **31** (1998) 205–229.

[BKV96] Brand, M.G.J. van den, Klint, P., Verhoef, C.: Core Technologies for System Renovation. In: Jeffery, K.G., Král, J., Bartošek, M. (eds.): SOFSEM'96: Theory and Practice of Informatics. Springer-Verlag (1996) (LNCS 1175), pp. 235–255.

[BoM88] Boyer, R.S., Moore, J S.: A Computational Logic Handbook. Academic Press, Boston etc., 1988.

[BGea95] Buhrman, H., Garay, J.A., Hoepman, J.H., Moir, M.: Long-lived renaming made fast. In *14th Ann. Symp. on Principles of Distributed Computing* (Ottawa, Ont., Canada) ACM Press 1995, pp. 194–203

[Dij76] Dijkstra, E.W.: A discipline of programming. Prentice–Hall 1976.

[Her91] Herlihy, M.P.: Wait–free synchronization. ACM Trans. on Program. Languages and Systems **13** (1991) 124–149.

[Hes96] Hesselink, W.H.: Bounded Delay for a Free Address. Acta Informatica **33** (1996) 233–254.

[Hes98a] Hesselink, W.H.: The design of a linearization of a concurrent data object. In: D. Gries, W.-P. de Roever (eds.): Programming Concepts and Methods, Proceedings Procomet '98, Chapman & Hall, IFIP 1998, pp. 205–224.

[Hes98b] Hesselink, W.H.: A formal proof of fault-tolerant progress. Submitted.

[Hes@] Hesselink, W.H.: `http://www.cs.rug.nl/~wim/crud`

[JoL96] Jones, R., Lins, R.: Garbage Collection, algorithms for automatic dynamic memory management. Wiley, 1996.

[Jon92] Jonker, J.E.: On-the-fly garbage collection for several mutators. Distr. Comput. **5** (1992) 187–199.

[OwG76] Owicki, S., Gries, D.: An axiomatic proof technique for parallel programs. Acta Informatica **6** (1976) 319–340.