WAIT-FREE LINEARIZATION WITH AN ASSERTIONAL PROOF

Hesselink, Wim H.

Published in:
Distributed computing

Publication date:
1994

Distrib Comput (1994) 8: 65–80

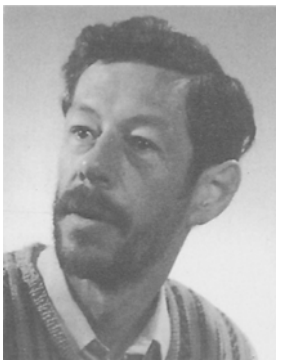
DISTRIBUTED COMPUTING

Wait-free linearization with an assertional proof

Wim H. Hesselink

Rijksuniversiteit Groningen, Department of Computing Science, P.O. Box 800, NL-9700 AV Groningen, The Netherlands

Received: December 1991/Accepted: April 1994



Wim H. Hesselink received his Ph.D. in mathematics from the University of Utrecht in 1975. After ten years of research in algebraic groups he turned to computer science. Since 1985 he has been an associate professor with the Department of Computing Science at the University of Groningen. In 1986/1987 he was on sabbatical leave with the Department of Computer Sciences of the University of Texas at Austin. His research interests include aspects and modalities of nondeterminacy, predicate transformation semantics, distributed programming, design and correctness of algorithms, and mechanical theorem proving.

Summary Given a sequential implementation of an arbitrary data object, a wait-free, linearizable concurrent implementation is constructed with space complexity quadratic in the number of processes. If processes do not concurrently invoke, the amortized time complexity of the invocations is independent of the number of processes. The worst case time complexity is linear in the number of processes. The construction is based on a compare&swap register. The correctness is proved by means of invariants and stability properties. Since it concerns memory reallocation by concurrent processes in a fault-tolerant setting, this proof is highly nontrivial.

Key words: Linearizable – Concurrent data object – Consensus – Wait-free – Memory management – Correctness – Invariant – Stability – Space complexity – Time complexity

Introduction

Background

A concurrent data object is a data structure shared by concurrent processes. The object must behave as if the invocations are processed in some sequential order. This requirement is formalized in the concept of linearizability (see [7], we come back to this in Sect. 2). Traditionally, linearizability is achieved by means of operations that temporarily block the progress of some processes. The disadvantage of such operations is that if a process is delayed (or stopped) other processes are delayed as well.

Therefore, recently, the concept of wait-free implementations has been proposed. A wait-free implementation of the concurrent data object is one in which every process completes its invocation in a bounded number of atomic actions, regardless of the actions and the execution speeds of the other processes, see [4] and [5].

Of course, the concept of wait-free implementation only makes sense if there are no atomic actions that can be blocked (e.g. semaphores and synchronous communications). We therefore assume that the processes do not contain actions that can be blocked. The processes are not assumed to make progress. A wait-free implementation is fault-tolerant in the sense that, if some process stops executing, the invocations of other processes are not affected.

One of the simplest concurrent data objects is the atomic read-write register: a shared variable, say $\mathtt{x}$, with the only atomic actions $\mathtt{x} := u$ and $z := \mathtt{x}$, for some private variables u and z. It has been shown by Herlihy [5] that atomic read-write registers are not sufficient to construct wait-free implementations of interesting data objects. Notice that we use the convention that shared program variables are in typewriter font. Constants, private program variables, parameters and mathematical variables are in math-italic.

For variables of arbitrary types we assume safety, in the sense that any read operation not concurrent with a modifying write operation obtains the most recently written value, that concurrent non-modifying write operations do not interfere, and that concurrent modifying write operations of the same value do not interfere with each other (see Sect. 1 for more details). We assume that read and write operations of integers and booleans are atomic.

The read-write register can be compared with the consensus object (cf. [3]). This is a shared variable, say $\mathtt{x}$, with an atomic read action $u := \mathtt{x}$ and an atomic setting action

(0) $\langle \textbf{if } \mathtt{x} = 0 \textbf{ then } \mathtt{x} := u \textbf{ fi} \rangle$

where 0 is some constant of the same type as $\mathtt{x}$, and where u is a private variable as before. If $\mathtt{x} \neq 0$, command (0) is equivalent to skip. There is also an atomic reset action $\mathtt{x} := 0$. Consensus objects are also called logical variables

or permanents, see e.g. [12] Sect. 1.4. A slightly more powerful object is the compare&swap register. This is a shared variable x with atomic read actions $z := x$ and an atomic setting action

(1) $\langle$**if** x = u **then** x := v **fi**$\rangle$

where z, u and v are private variables, cf. [5] p. 135.

Herlihy [4] and Plotkin [14] have shown that wait-free implementations of arbitrary data objects can be constructed by means of atomic read-write registers–together with consensus objects. Let n be the number of processes. The implementation of [14] requires memory of order n^2 and has a worst-case time complexity of order n^3. The implementation of [5] requires memory of order n^3, has a worst-case time complexity of order n^2 and, moreover, requires unbounded integers. The latter requirement is a serious drawback, since unbounded integers are seldom available in hardware.

In [6], Herlihy presents a construction based on a compare&swap register with memory of order n^2 and worst-case time complexity of order n^2. For all three implementations, the outline provided is rather sketchy. This makes it hard to prove or to refute their correctness.

Contributions

In this paper, we present a wait-free implementation of an arbitrary data object that requires memory of order n^2 and that has a worst-case time complexity of order n. The implementation requires a compare&swap register, just as in [6].

Since a compare&swap register is a stronger primitive than a consensus object, the complexity of our algorithm can only be compared with the algorithm of [6]. Our algorithm has the same space complexity (n^2) and a better time complexity (order n). If there is only one active process, the amortized time complexity is constant, i.e., independent of the number of processes.

For us, however, the main interest of these algorithms is not increased efficiency, but to learn how to develop such an algorithm together with a complete proof of correctness. This is especially important since the combination of concurrency with memory reallocation is very delicate, cf. [9]. In fact, even though a previous version of our algorithm had a careful operational "proof", the search for an assertional proof uncovered a delicate gap in the argument (see Sect. 7).

Technically, our algorithm is a modification of the algorithm of [5]. Each process transverses the linked list of invocations in a more greedy way than in [5]. This implies that a delayed process needs only constant space to perform its actions without disturbing the data structure. The data structure to support delayed processes can therefore be smaller than in [5]. For reasons of simplicity, we assume that the data object is deterministic. Herlihy [5] avoids this assumption by means of a second family of consensus objects.

Plan of the paper

In Sect. 1, we present our programming notations and discuss the granularity of the commands. In Sect. 2, we introduce data objects and concurrent linearizable implementations of data objects. As a stepping stone for the algorithm, we present in Sect. 3 an easier algorithm that uses an atomic action with a larger grain of atomicity than compare&swap registers.

Sections 4, 5 and 6 form the heart of the paper. Here the program is developed in a top down fashion. So, in Sect. 4, we give the actions on the main shared variables and we prove linearizability under assumption of six invariants. In Sect. 5, the abstract algorithm is extended with private computations and with communication by means of other shared variables. The resulting program is shown to satisfy a set of roughly twenty invariants, which imply the six invariants mentioned above. In Sect. 6, the program is made wait-free. For this purpose an additional shared variable is introduced with two new invariants.

The complexity of the proof is cumbersome but, in our view, not out of proportion. In fact, the program uses a pointer structure in bounded memory, and even proofs of sequential programs with pointers are usually complicated. In a sequential program, the invariants would occur as preconditions of specific commands. Here they are discussed globally since other processes are concurrently active.

In Sect. 7, we discuss points of the program where a seemingly innocent modification would be disastrous. Section 8 contains a comparison of our program with Herlihy's program in [5]. We give an explicit interpretation of the memory management suggested in [5]. In Sect. 9, we briefly discuss the complexity of our algorithm.

Our presentation is deliberately rather technical and formal. We refer to [5] and [6] for the intuition and motivation underlying the algorithms and also for extensive overviews of related work.

1 Programming notations and concepts

We use the infix operator "." for subscription of arrays. For example, nx.g is the g'th element of array nx. The other programming notations used are variations of well-known constructs of Pascal:

if . . . **then** . . . **fi**
while . . . **do** . . . **od**
for each . . . **do** . . . **od**.

In particular, the **if then fi** construct means *skip* if the guard is false.

We assume that all shared variables are *safe* in the following sense. Any read operation not concurrent with a modifying write operation obtains the most recently written value. Concurrent non-modifying write operations do not interfere. Concurrent modifying write operations of the same value do not interfere with each other. We assume that write operations of integers and booleans are atomic. For other types, a modifying write operation is non-atomic in the sense that any concurrent operation (apart from a write of the same value) leads to unpredictable results. A command is made atomic by enclosing it in angled brackets $\langle$and$\rangle$, cf. [1] Chapter 6.

We use an assertional method to prove the correctness of our concurrent programs. Most assertions are

predicates on the state of the shared variables. We also need predicates concerning the flow of control of the processes, cf. [10] and [11]. We write P **in** C to indicate that process P is executing the possibly composite command C. More precisely, it means that P has completed the last command before C and has not yet completed C. In particular, for a composition $(C; D)$ we always have

$$P \textbf{ in } C; D \equiv P \textbf{ in } C \vee P \textbf{ in } D.$$

Another relevant example is a composition like

if B **then** C **fi**; D.

Here, evaluation of guard B by process P establishes the disjunction

$$(B \wedge (P \textbf{ in } C)) \vee (\neg B \wedge (P \textbf{ in } D)).$$

Unconditional entering or exiting of blocks is not regarded as a separate action, but (for example) the last action of a **while**-loop is the test that yields the negation of the guard. Location predicates like P **in** C can be eliminated by introducing ghost variables or auxiliary variables, cf. [1]. Our reasons for using location predicates are discussed at the end of Sect. 5, see also [11] p. 290.

We say that a predicate I is invariant if I holds initially and remains valid during every execution sequence of the system of processes. For predicates X and Y, we say that "X is stable while Y" if there is an invariant I such that every atomic action C of each of the processes satisfies the Hoare triple $\{X \wedge I \wedge Y\}\, C\, \{X\}$. For an expression E, we say that "E is constant while Y" if $E = v$ is stable while Y for every value v. Notice that, for a predicate X, the assertion "X is constant while Y" is stronger than "X is stable while Y", for it also implies that "$\neg X$ is stable while Y".

We say that a predicate X is not threatened by a command C, if it is easy to see that $\{X\}\, C\, \{X\}$ holds. In many cases the reason is that C is an assignment to a variable that does not occur in X.

2 Data objects and concurrency

A data object is a tuple $\langle X, U, Z, x_0, R\rangle$ where X is the state space of the object, $x_0 \in X$ is the initial state, U is the input space (the set of invocations), Z is the output space (the set of result values) and $R \subseteq U \times X \times X \times Z$ is the transition relation. If the object is invoked with invocation u in state x it may go into state y and return the output z if and only if $\langle u, x, y, z\rangle \in R$.

In this paper, every object is supposed to be *total* and *deterministic*, in the sense that in every state every invocation allows precisely one new state and precisely one result: for every pair $\langle x, u\rangle$ with $x \in X$ and $u \in U$, there is precisely one pair $\langle y, z\rangle$ with $\langle u, x, y, z\rangle \in R$. The requirement of totality (the existence of a resulting pair $\langle y, z\rangle$ for every pair $\langle x, u\rangle$) formalizes the assumption that no operation can be blocked. Determinacy is postulated for the sake of simplicity of the algorithm. This assumption is essential for the present algorithm, but a variation of the algorithm that avoids this assumption is in preparation. Herlihy [5] avoids the assumption of determinacy by means of a second family of consensus objects.

Example. The compare&swap register introduced in Sect. 0 can be formalized as follows. Let X be the type of variable $\mathbf{x}$, i.e., its set of values. The action to read $\mathbf{x}$ is denoted *read*; it does not modify $\mathbf{x}$ and yields the value $z = \mathbf{x}$. An invocation of instruction (1):

$\langle$**if** $\mathbf{x} = u$ **then** $\mathbf{x} := v$ **fi**$\rangle$

with values u and v is denoted as a pair $\langle u, v\rangle$; this invocation may or may not modify $\mathbf{x}$; the instruction always yields the value *ok*. We therefore take $Z = X + \{ok\}$, the disjoint union of the set X with the singleton set $\{ok\}$, and $U = \{read\} + X^2$ where X^2 is the set of the pairs $\langle u, v\rangle$ with $u, v \in X$. Relation R is defined by

$$\langle read, x, y, z\rangle \in R \equiv y = x \wedge z = x,$$

$$\langle\langle u, v\rangle, x, y, z\rangle \in R \equiv$$

$$z = ok \wedge ((x = u \wedge y = v) \vee (x \neq u \wedge y = x)).$$

It is easy to verify that indeed relation R is total and deterministic. □

We assume that there are n processes, represented by

type *process* $= 0 \,..\, n - 1$.

A concurrent implementation of a data object $\langle X, U, Z, x_0, R\rangle$ is a procedure that, conceptually, acts on some global program variable $\mathbf{x}$ of type X and that could be specified by

proc *apply* (**in** P:*process*, u:U; **out** z:Z)
{**pre** $\mathbf{x} = w$, **post** $\langle u, w, \mathbf{x}, z\rangle \in R$}.

Here, w is a logical variable that stands for the value of $\mathbf{x}$ in the precondition. Process P calls procedure *apply* in the form *apply* (P, u, z) for the treatment of invocation u with result z. So P and u are input parameters and z is a result parameter.

All processes may call *apply* concurrently and repeatedly. The data object itself is passive; the subcommands of *apply* are executed by the invoking process. Yet the implementation is required to be *linearizable*, in the sense that each call of *apply* appears to take effect instantaneously at some point between the invocation and the response. Linearizability implies that processes appear to be interleaved at the granularity of complete operations and that the order of non-overlapping operations is preserved. See [7] for a detailed exposition.

The implementation (i.e., procedure *apply*) is called *wait-free* if it does not contain operations that can be blocked and if there is a number N such that every call *apply*(P, u, z) terminates after at most N atomic actions of process P, independently of concurrent calls of *apply* by other processes.

In order to formulate a concrete proof obligation that implies linearizability we proceed as follows. Since the object is deterministic, the current state and the output are functions of the sequence of invocations that have been treated. Let U^* be the set of sequences of invocation values. Let ε be the empty sequence. For $\alpha \in U^*$ and $u \in U$, let $(\alpha; u)$ be the sequence obtained by postfixing α with the singleton u. The current state and the output are

determined by functions $e: U^* \to X$ and $f: (U^* \setminus \{\varepsilon\}) \to Z$. These functions satisfy

(2) $e(\varepsilon) = x_0$ (the initial state),

$\langle u, e(\alpha), e(\alpha; u), f(\alpha; u) \rangle \in R$

for all $\alpha \in U^*$ and $u \in U$.

We need to distinguish invocations of different processes. We do this by tagging and therefore introduce the cartesian product $V = U \times process$, with as elements tagged invocations $\langle u, P \rangle$ with $u \in U$ and $P \in process$. Let V^* be the set of sequences of tagged invocations. For a sequence $\sigma \in V^*$, let $dt(\sigma) \in U^*$ be obtained by deletion of the tags. For a process P, let $\sigma \| P \in V^*$ be the subsequence of σ of the invocations tagged with P. Since the tags P of $\sigma \| P$ are irrelevant, we define $\sigma | P = dt(\sigma \| P) \in U^*$.

We define $\sigma * P \in V^*$ as the shortest prefix of σ that contains $\sigma \| P$. We define $\sigma_P = dt(\sigma * P) \in U^*$. So, if $\sigma | P \neq \varepsilon$, then σ_P is the shortest prefix of $dt(\sigma)$ that contains the last invocation of P. In particular, $\sigma | P$ is a subsequence of σ_P. After these preparations we can formulate a concrete proof obligation.

The implementation of the data object is *linearizable* if one can construct a ghost variable $\sigma: V^*$, initially $\sigma = \varepsilon$, that satisfies

(CL0) whenever process P is not invoking, $\sigma | P$ is the sequence of invocation values of P in the order of submission;

(CL1) whenever P is not invoking and $\sigma | P \neq \varepsilon$, then $f(\sigma_P)$ is the latest result obtained by P.

In fact, these conditions imply that σ is some linearization of the operations that refines the partial order of the nonoverlapping operations. If $\sigma | P \neq \varepsilon$ then σ_P is the sequence of invocations up to and including the latest invocation of P; therefore that invocation must have yielded result $f(\sigma_P)$. Notice that we have no conditions on the state since the state is not observable.

3 Implementing an arbitrary object

We now turn to the development of a wait-free concurrent implementation of an arbitrary data object $\langle X, U, Z, x_0, R \rangle$. This concurrent implementation is based on a given local implementation

locapply(**in** $u: U, x: X$; **out** $y: X, z: Z$)

which given u and x establishes $\langle u, x, y, z \rangle \in R$. The need for the two additional state parameters will become clear below.

The implementation of *apply* must be such that the object behaves linearizable, but the calls of *locapply* are not supposed to be atomic actions. These calls are regarded as read actions of the input arguments, followed by write actions into the output arguments. In the implementation in Sect. 4, we shall use that all arguments are safe registers. Determinacy of *locapply* will imply that the possibly concurrent write actions write the same value.

Our first implementations of the data object are based on atomic commands with a larger grain of atomicity than compare&swap registers. These implementations serve to give the reader a feeling for the problems associated with delayed processes, which by their delayed actions may interfere with other processes and disturb the data structure. The main issue is to separate the local computation of *locapply* from the interaction between the processes. The second point is that we only use bounded memory, linear in the number of processes. We begin using a global declaration

var x: X {initially x = x_0}.

It is of course correct but not satisfactory to encapsulate *locapply* atomically:

proc *apply* (**in** P: *process*, $u: U$; **out** $z: Z$);
  |[⟨*locapply* (u, x, x, z)
    {; $\sigma := (\sigma; \langle u, P \rangle)$}⟩]|.

Here, the action on ghost variable σ is given between curly brackets.

The next step is to introduce local variables for the results of *locapply* and a test to ensure that the executing process has not been delayed. For this purpose we introduce a sequence counter equal to the length of σ. We thus need the additional global declaration

var sq: *integer* {initially sq = 0}.

We first consider the tentative implementation, in which the guard h = sq serves to ensure that σ has not been modified in the mean time:

proc *apply* (**in** P: *process*, $u: U$; **out** $z: Z$);
  var h: *integer*; *wait*: *boolean*
  ; $w, y: X$; $t: Z$;
  |[*wait* := *true*
  ; **while** *wait* **do**
      h := sq; ⟨w := x⟩
    ; *locapply* (u, w, y, t)
    ; ⟨**if** h = sq **then**
        x := y; sq := sq + 1
      {; $\sigma := (\sigma; \langle u, P \rangle)$}
      ; $z := t$; *wait* := *false* **fi**⟩
    od]|.

Here it is not enough that x is a safe register, since the read action of x can be disturbed by a write action of some other process. Therefore, the read action w := x has been made atomic.

In this implementation, process P can be overtaken indefinitely while executing infinitely many commands in its repetition. So this implementation is not wait-free. Since other processes must not be blocked, the only remedy is that other processes are forced to execute P's invocation. For this purpose a shared data structure is introduced that can contain the invocation and result values of all processes, as well as booleans to indicate which processes are waiting. We therefore extend the global declarations with

var invo: **array** *process* **of** U
; resu: **array** *process* **of** Z
; wait: **array** *process* **of** *boolean*;
initially ($\forall P ::$ ¬wait.P).

The most recent invocation value of process P is located in invo.P. The corresponding result is delivered in resu.P. The condition wait.P indicates that the invocation of

process P is waiting to be treated. In order to get bounded delay, we ensure that the invocation of process Q is treated by every process when $(Q = \mathrm{sq}\ \mathbf{mod}\ n) \wedge \mathrm{wait}.Q$ holds. Since this condition depends on more than one shared variable, it requires careful programming. A correct solution is given in program (3).

```
(3) proc apply (in P: process, u: U; out z: Z);
    var h: integer; Q: process
    ; v: U; w, y: X; t: Z;
    |[  ⟨invo.P := u⟩; wait.P := true
    ; h := sq
    ; while wait.P do
      B0: Q := h mod n
      ; B1: if ¬wait.Q then Q := P fi
      ; B2: ⟨v := invo.Q⟩
      ; B3: ⟨w := x⟩
      ; B4: locapply (v, w, y, t)
      ; B5: ⟨if h = sq then
                 x := y; sq := sq + 1
                {; σ := (σ; ⟨v, Q⟩)}
                ; resu.Q := t; wait.Q := false fi⟩
      ; B6: h := sq
      od
    ; z := resu.P ]|.
```

Linearizability of (3) can be proved by showing that σ satisfies the conditions (CL0) and (CL1) of Sect. 2. This is done as follows. Let $C0$ be the composite command on the first line of the body of (3). Let $C1$ denote the composition of $h := \mathrm{sq}$ with the subsequent **while**-loop. We write $u.P$ to denote parameter u of process P. It is easy to see that we have the invariant:

(H0) $\mathrm{wait}.P \Rightarrow P\ \mathbf{in}\ C1 \wedge \mathrm{invo}.P = u.P.$

In (3), the tests $\mathrm{wait}.P$ and $\mathrm{wait}.Q$ are placed after the assignment $h := \mathrm{sq}$, so that $h = \mathrm{sq}$ implies that process Q has not been treated in the mean time. More precisely, if we write $Bi \sim j$ to denote the composition $Bi; \ldots; Bj$, the program satisfies the local invariants

$P\ \mathbf{in}\ B0 \sim 5 \wedge h = \mathrm{sq} \Rightarrow \mathrm{wait}.P,$
$P\ \mathbf{in}\ B2 \sim 5 \wedge h = \mathrm{sq} \Rightarrow \mathrm{wait}.Q,$
$P\ \mathbf{in}\ B3 \sim 5 \wedge h = \mathrm{sq} \Rightarrow v = \mathrm{invo}.Q,$
$P\ \mathbf{in}\ B4 \sim 5 \wedge h = \mathrm{sq} \Rightarrow w = \mathrm{x},$
$P\ \mathbf{in}\ B5 \Rightarrow \langle v, w, y, t\rangle \in R.$

Let $\beta.P$ be the sequence of invocations of process P in the order of submission. Using (H0), formula (2) and the local invariants, one can prove the invariants:

(H1) $\neg(P\ \mathbf{in}\ C0 \vee \mathrm{wait}.P) \Rightarrow \beta.P = \sigma|P,$
(H2) $P\ \mathbf{in}\ C0 \vee \mathrm{wait}.P \Rightarrow \beta.P = ((\sigma|P); u.P),$
(H3) $\mathrm{x} = e(dt(\sigma)),$
(H4) $\sigma|P = \varepsilon \vee P\ \mathbf{in}\ C0 \vee \mathrm{wait}.P \vee \mathrm{resu}.P = f(\sigma_P).$

Condition (CL0) of Sect. 2 requires that $\beta.P = \sigma|P$ when P is not invoking. Therefore, (CL0) follows from (H0) and (H1). Condition (CL1) follows from (H0) and (H4). The invariants (H2) and (H3) are used in the proofs of (H1) and (H4). The details are left to the reader.

As a preparation for the next program, we give an indication why program (3) is wait-free. During every execution of the body of the repetition, variable sq is incremented at least once. If $\mathrm{wait}.P$ holds when $\mathrm{sq} \bmod n$ becomes P, then $\mathrm{wait}.P$ becomes *false* during the first execution of that body with h equal to the new value of sq. Therefore, the loop of *apply* of process P terminates after at most $n + 1$ executions of its body by process P.

In this implementation safe registers are not enough. Therefore, reading and writing of x and elements of invo have been made atomic. Another disadvantage of this program is that it needs unbounded integers. This can be eliminated by taking sq and h modulo m for some integer $m > n$. Then the guard of $B5$ has to be replaced by $h = \mathrm{sq} \wedge \mathrm{wait}.P$. The additional conjunct $\mathrm{wait}.P$ ensures that sq does not make a full circle. The main disadvantage of (3), however, is the big atomic command $B5$. It involves four shared variables (x, sq, $\mathrm{resu}.Q$ and $\mathrm{wait}.Q$) and three private ones (h, y and t). The solution to be presented in the next section will remedy all these points.

4 The abstract implementation

In this section we begin the development of our main algorithm, which is a variation of the solution of Herlihy [5]. The crucial atomic actions are compare&swap actions on pointers to memory cells.

Since we want to present the algorithm together with a complete proof of correctness, the algorithm is developed in steps. We use five levels of invariants. The top level consists of the proof obligations (CL0), (CL1) of linearizability, cf. Sect. 2. The next level consists of invariants (J0), (J1) and (J2) of a first abstract implementation. These invariants imply (CL0) and (CL1). The predicates (J0), (J1) and (J2) are proved to be invariant under assumption of invariants (P0) through (P5).

The predicates (P) are reformulated and strengthened to get more convenient predicates (Q0) through (Q8). Two additional predicates (Q9) and (Q10) are introduced to guarantee safety of the non-atomic shared variables. The steps from level (Q) via levels (P) and (J) upto level (CL) are treated in this section. The separation of levels is also useful to eliminate ghost variable σ. In fact, σ only occurs in the predicates (CL) and (J).

In Sect. 5, the abstract implementation is refined to a more concrete implementation, which has invariants (L0) through (L16). An easy verification will show that the invariants (L) imply the invariants (Q). The proof of invariance of (L0) through (L16) is cumbersome. The result of this step is a linearizable implementation that is not yet wait-free. It may be mentioned that in Sects. 4 and 5, we also use a family of "easy" invariants (K0) through (K5).

The implementation of Sect. 5 still has some nondeterminacy. In Sect. 6, part of this nondeterminacy is resolved in such a way that the implementation becomes wait-free. For this purpose, the data structure is extended slightly. The proof that the implementation is wait-free requires two additional invariants and some stability results.

We now turn to the abstract implementation. We first discuss the design of the data structure. The first remarks to be made apply equally well to the implementation of Sect. 3.

We need a shared data structure that contains a current state and that can contain invocation and result

values for all n processes. All processes are concurrently active to compute the next state and the corresponding result. Processes can be delayed and the activity of a delayed process must not damage the shared data structure. So we have to choose between the alternatives:

(a) when a process modifies shared data, it does this in an atomic action that contains a test to ensure that it is not delayed,
(b) the shared data structure admits actions of delayed processes.

Implementation (3) is an elaboration of choice (a). It seems, however, that choice (a) is not adequate if a smaller grain of atomicity is required. So we turn to choice (b).

If a delayed process is allowed to compute the next state and result for some outdated state and invocation, the program may need as many as $2 \cdot n$ state variables and every process may need n invocation variables and n result variables. Since it is not known which processes are delayed and for how long, the state variables, the invocation variables and the result variables cannot be ordered in cyclic buffers.

Following Herlihy [5], we therefore group together invocation, next state and corresponding result in one "cell". The cell of the current state is equipped with a pointer that may point to the next cell. In other words, the consecutive states of the object are represented by the state components of a linked list of cells. Since a process must be able to test whether its invocation has been treated, every cell gets a boolean variable to indicate that the invocation is still waiting to be treated.

Since memory usage must be kept explicit we use ordinary arrays instead of the pointer facilities of Pascal or C. Now an array of records is the same as a system of arrays. The latter description is preferred here, since it gives cleaner code and cleaner formulae. So the linked list is represented by

type *address* = 0 . . . *upb* {*upb* to be chosen later};
var nx: **array** *address* **of** *address*
; st: **array** *address* **of** *X*
; inv: **array** *address* **of** *U*
; res: **array** *address* **of** *Z*
; wa: **array** *address* **of** *boolean*;
initially $(\forall k \in address :: \neg \mathrm{wa}.k)$.

The type *address* is a subrange of the integers. An element $k \in address$ is called a (memory) address. For given address k, the tuple

$\langle \mathrm{nx}.k, \mathrm{st}.k, \mathrm{inv}.k, \mathrm{res}.k, \mathrm{wa}.k \rangle$

is regarded as the cell at address k. The value $\mathrm{nx}.k$ is the next address, i.e., points to the next cell. The value $0 \in address$ is used as a nil pointer. The boolean $\mathrm{wa}.k$ indicates whether some invocation is waiting at address k.

To announce its invocations, each process P is provided with a variable $\mathrm{a}.P$ of type *address* ($\mathrm{a}.P$ corresponds to *announce*[P] of reference [5]). We thus have the additional declaration

var a : **array** *process* **of** *address*.

Once instantiated, the element $\mathrm{inv}.(\mathrm{a}.P)$ holds the latest invocation value of process P. The value of $\mathrm{a}.P$ can be read by all processes, but written only by process P.

For the purpose of linearization we introduce a shared variable g as the address of the current state of the object, according to the declaration

var g : *address*;
initially $\mathrm{st}.g = x_0 \wedge \mathrm{nx}.g = 0$.

Just as in program (3), the interaction between the processes and the common data structure requires a prelude in which the process places its invocation value in the common data structure, a working phase in which the process participates in the treatment of the current invocations and a postlude in which the process obtains the result of its invocation. We thus assume that procedure *apply* has the form

(4) **proc** *apply* (**in** P : *process*, u : U; **out** z : Z);
|[$C0$; $C1$
; $z := \mathrm{res}.(\mathrm{a}.P)$]|,

where $C0$ is the prelude and $C1$ is the working phase.

As a first precaution against interference, we give each process P its own pool of addresses, cf. [5]. We fix a natural number m to be determined later and let $addr.P$ be the subrange of *address* given by

(5) $k \in addr.P \equiv m \times P < k \leqq m \times (P + 1)$.

Since the processes have numbers from 0 through $n - 1$, it follows that we can take $upb = m \times n$. Notice that $0 \notin addr.P$ for all P. We assume that $\mathrm{a}.P \in addr.P$ holds initially.

In command $C0$, process P places its invocation value u at some address $\mathrm{a}.P$ in its own pool and announces the invocation by setting $\mathrm{wa}.(\mathrm{a}.P)$ to *true*. Since, as soon as $\mathrm{wa}.(\mathrm{a}.P)$ holds, other processes may use the cell at $\mathrm{a}.P$, the initialisation $\mathrm{nx}.(\mathrm{a}.P) := 0$ had better be placed before setting $\mathrm{wa}.(\mathrm{a}.P)$. We therefore refine command $C0$ in (4) by

(6) $C0$: |[**choose** $\mathrm{a}.P \in addr.P$
; E: |[$\mathrm{inv}.(\mathrm{a}.P) := u$; $\mathrm{nx}.(\mathrm{a}.P) := 0$
; $\mathrm{wa}.(\mathrm{a}.P) := true$]|]|.

The inner brackets |[and]| serve to indicate the extent of command E. No atomicity is intended.

In the design of $C1$ we introduce several subcommands with names that are chosen in such a way that they can be kept in later refinements. For the moment we only assume that command $C1$ is refined by

(7) $C1$: **while** $\mathrm{wa}.(\mathrm{a}.P)$ **do** DD **od**,

where command DD does not modify the arrays a and inv. Moreover, we assume that the only modifications of wa in DD are

$D5$: $\mathrm{wa}.i := false$,

where i is a private variable of P of type *address* and $i \neq 0$.

We introduce the convention that the parameter u and private variables h, i, . . . of process Q are denoted by $u.Q$, $h.Q$, $i.Q$, . . . , whenever convenient. If no additional process name is provided, a private variable belongs to

process P. In all invariants and postulates to be presented, we quantify universally over addresses k and processes P, Q and T. Now we can prove

Lemma 0. *(a)* $\mathtt{a}.Q$ *is constant while* $\neg(Q$ **in** $C0)$.
(b) The program has the invariants
(K0) $\mathtt{a}.Q \in addr.Q$,
(K1) $k \in addr.Q \wedge \mathtt{wa}.k$
$\Rightarrow k = \mathtt{a}.Q \wedge Q$ **in** $C1 \wedge \mathtt{inv}.k = u.Q$.

Proof. (a) The choice of $\mathtt{a}.Q$ in $C0$ of process Q is the only modification of $\mathtt{a}.Q$.

(b) Condition (K0) is invariant since, by assumption, it holds initially and it is preserved by each choice of $\mathtt{a}.Q$ in $C0$ of process Q. Condition (K1) is invariant, since $\mathtt{wa}.k$ is *false* initially and $\mathtt{wa}.k$ can only be made *true* by process Q in E with $k = \mathtt{a}.Q$. By this action, process Q enters $C1$. Process Q has set $\mathtt{inv}.k = u.Q$ in E. Since $k \notin addr.P$ for all other processes P, by (K0), $\mathtt{inv}.k$ is not modified by other processes. Process Q cannot leave $C1$ while $\mathtt{wa}.k$ holds. Consequently, $\mathtt{a}.Q$ and $\mathtt{inv}.k$ are not modified while $\mathtt{wa}.k$ remains valid. □

Remark. Often, we only need the following consequence of (K0) and (K1):

(K1a) $\mathtt{wa}.(\mathtt{a}.Q) \Rightarrow Q$ **in** $C1$.

We now proceed by constructing the ghost variable σ in such a way that condition (CL0) holds. We first introduce, for every process Q, a ghost variable $\beta.Q$ of type U^* to stand for the sequence of invocation values of process Q in the order of submission. So, initially $\beta.Q = \varepsilon$. Whenever process Q enters $C0$, variable $\beta.Q$ is modified by $\beta.Q := (\beta.Q; u.Q)$.

We strengthen condition (CL0) to

(J0) **if** Q **in** $C0 \vee \mathtt{wa}.(\mathtt{a}.Q)$
then $\beta.Q = ((\sigma|Q); u.Q)$
else $\beta.Q = \sigma|Q$ **fi**.

Indeed, (CL0) follows from (J0) since, if Q is not invoking, Q is not in $C0$ and $\neg\mathtt{wa}.(\mathtt{a}.Q)$ because of (K1a). Also notice that the **then**-part only applies when Q is in *apply* so that $u.Q$ is well-defined.

Predicate (J0) holds initially since initially the guard of (J0) is false and both $\beta.Q$ and σ are initially empty. When Q enters $C0$, predicate (J0) is preserved since at that point $\beta.Q$ is extended with $u.Q$. When Q leaves $C0$, predicate (J0) is preserved since at that point $\mathtt{wa}.(\mathtt{a}.Q)$ is made *true*. Predicate (J0) is not yet preserved when some process P executes $D5$ with $i.P = \mathtt{a}.Q$. We therefore extend command $D5$ with a modification of the ghost variable σ, i.e., we replace $D5$ by

$D5'$: $\langle$**if** $\mathtt{wa}.i$ **then** $\sigma := (\sigma; \langle \mathtt{inv}.i, pown.i\rangle)$ **fi**
; $\mathtt{wa}.i := false\rangle$,

where $pown.i$ is the unique process T with $i \in addr.T$. Process T exists since $0 < i < upb$. It is unique because of the disjunctivity of the sets $addr.T$. Now Lemma 0(b) implies that $D5'$ replaces $\sigma|T$ by $(\sigma|T; u.T)$ and keeps $\sigma|Q$ constant for $Q \neq T$. Since, moreover, $i = \mathtt{a}.T$ by (K1) and $i \neq \mathtt{a}.Q$ for $Q \neq T$, by (K0), it follows that $D5'$ preserves (J0). In view of Lemma 0(a), this shows that (J0) is invariant and, hence, (CL0) is invariant.

The variables $\mathtt{st}.k$ and $\mathtt{res}.k$ are intended to hold a new object state and a new result corresponding to a new invocation at $\mathtt{inv}.k$. Since the invocation at $\mathtt{inv}.k$ is only new, i.e., waiting for (completion of) treatment, if $\mathtt{wa}.k$ holds, we postulate

(8) $\mathtt{st}.k$ and $\mathtt{res}.k$ are constant while $\neg\mathtt{wa}.k$.

We now strengthen condition (CL1) to the invariant

(J1) $\sigma|Q = \varepsilon \vee Q$ **in** $C0 \vee \mathtt{wa}.(\mathtt{a}.Q) \vee \mathtt{res}.(\mathtt{a}.Q) = f(\sigma_Q)$.

Indeed, (CL1) follows from (J1), (K1a) and (8): if Q is not invoking and $\sigma|Q \neq \varepsilon$ then (J1) and (K1a) imply that $\mathtt{res}.(\mathtt{a}.Q) = f(\sigma_Q)$. Moreover, Q has submitted some invocation and, hence, has obtained some result. Now (8) and (K1a) imply that $\mathtt{res}.(\mathtt{a}.Q)$ still holds the latest result obtained by Q.

In order to establish the invariance of (J1), we have to use formula (2) that defines function f by means of relation R and function e. The computation of $f(\sigma_Q)$ requires the value of the state of the object and must be accompanied by a computation of the next state of the object. As announced above, we use g as the address of the current state of the object and $\mathtt{nx}.g$ as the address of the next state. One might regard $e(dt(\sigma))$ as the current state of the object, but σ is modified in command $D5'$ which does not modify g. Therefore, instead of $\mathtt{st}.g = e(dt(\sigma))$, we propose the invariant

(J2) **if** $\mathtt{nx}.g = 0 \vee \mathtt{wa}.(\mathtt{nx}.g)$
then $\mathtt{st}.g = e(dt(\sigma))$
else $\mathtt{st}.(\mathtt{nx}.g) = e(dt(\sigma))$ **fi**.

In order to preserve (J1) and (J2), we postulate that $D5'$ of P only modifies σ under specific circumstances, viz.,

(P0) P **in** $D5' \wedge \mathtt{wa}.i$
$\Rightarrow \mathtt{nx}.g = i \neq 0 \wedge \langle \mathtt{inv}.i, \mathtt{st}.g, \mathtt{st}.i, \mathtt{res}.i\rangle \in R$.

It follows that, if execution of $D5'$ of P modifies $\mathtt{wa}.i$ and $T = pown.i$, the execution has the precondition $\mathtt{st}.g = e(dt(\sigma))$ by (J2). In view of (P0) and formula (2), this precondition also satisfies

$\mathtt{st}.i = e(dt(\sigma); \mathtt{inv}.i) \wedge \mathtt{res}.i = f(dt(\sigma); \mathtt{inv}.i)$.

Since $D5'$ then replaces $dt(\sigma)$ by $(dt(\sigma); \mathtt{inv}.i)$, this execution establishes

$\mathtt{st}.i = e(dt(\sigma)) \wedge \mathtt{res}.i = f(dt(\sigma))$.

Since $i = \mathtt{a}.T$ and the postcondition has $\sigma_T = dt(\sigma)$, this implies that $D5'$ preserves (J1) for $Q = T$. For $Q \neq T$, the value σ_Q is not modified and, by (K0), $\mathtt{wa}.(\mathtt{a}.Q)$ also remains constant; therefore (J1) is also preserved for $Q \neq T$. Since $\mathtt{nx}.g = i \neq 0$ and the postcondition has $\neg\mathtt{wa}.i$, it also follows that $D5'$ preserves (J2).

Condition (J1) holds initially because of $\sigma = \varepsilon$. Condition (J2) holds initially because of $\mathtt{nx}.g = 0$ and $\mathtt{st}.g = x_0 = e(\varepsilon)$. By Lemma 0 (a), modification of array $\mathtt{a}$ does not falsify (J1). Also, modification of $\mathtt{wa}$ in $C0$ does not falsify (J1). By (8), modification of $\mathtt{res}$ does not falsify (J1). Therefore (J1) is invariant.

With respect to (J2), we still have to take care of the modifications of $\mathtt{g}$, $\mathtt{nx}$, $\mathtt{wa}$ and $\mathtt{st}$. We first treat modification of $\mathtt{g}$. After $D5'$, one might expect the update $\mathtt{g} := \mathtt{nx}.g$.

Since both g and nx are shared variables, such an update could be done by all processes (even concurrently). Since it must be done precisely once, we give each process a private variable h of type *address* that serves as a possibly outdated copy of g and we let g be modified only by the compare&swap action

$D6$: $\langle$**if** $g = h$ **then** $g := i$ **fi**$\rangle$.

We postulate

(P1) P **in** $D6 \wedge g = h$
$\Rightarrow \mathrm{nx}.g = i \neq 0 \wedge \neg \mathrm{wa}.i \wedge \mathrm{nx}.i = 0.$

This postulate guarantees that $D6$ preserves (J2). For, it implies that g is only modified if the guard of (J2) is *false*, that g becomes nx.g and that the guard of (J2) becomes *true*.

By postulate (P0), effective execution of $D5'$ presupposes $\mathrm{nx}.g \neq 0$. We therefore introduce some assignment to nx.g. To avoid the usage of two shared variables in one command, we prefer an assignment to $\mathrm{nx}.h$. Since it might violate (J2) if $\mathrm{nx}.h \neq 0$, we propose the consensus action

$D2$: $\langle$**if** $\mathrm{nx}.h = 0$ **then** $\mathrm{nx}.h := i$ **fi**$\rangle$,

with the postulate

(P2) P **in** $D2 \wedge \mathrm{nx}.h = 0 \Rightarrow i \neq 0 \wedge \mathrm{wa}.i$.

In fact, it is easy to see that (P2) implies that $D2$ preserves (J2).

Remark. Although it is not necessary, it is natural also to postulate

(P2′) P **in** $D2 \wedge \mathrm{nx}.h = 0 \Rightarrow g = h$.

This predicate will indeed be preserved, see (L6) below. □

In order to guarantee that the assignments to nx and wa in region E preserve (J2), it is sufficient to postulate invariant validity of

(P3) P **in** $E \Rightarrow \mathrm{a}.P \notin \{g, \mathrm{nx}.g\}$.

We now come to the central computation. Since h is supposed to be a private copy of g, postulate (P0) suggests that command DD should contain command $D4$ given by

$D4$: *locapply* (inv.i, st.h, st.i, res.i).

Since this command threatens postulate (8) for $k = i$, we postulate

(P4) P **in** $D4 \wedge \neg \mathrm{wa}.i$
$\Rightarrow \langle \mathrm{inv}.i, \mathrm{st}.h, \mathrm{st}.i, \mathrm{res}.i \rangle \in R$.

Now, indeed, since *locapply* is specified by R and relation R is deterministic, postulate (P4) implies that $D4$ does not violate (8). Since st and res will not be modified by other commands, we may henceforth regard (8) as a corollary of (P4).

Command $D4$ also threatens the invariance of (J2) by modifying st.g or st.(nx.g). Therefore, we postulate invariant validity of

(P5) $\neg \mathrm{wa}.g$.

Using (8) and (P5), one can easily verify that $D4$ preserves (J2). This shows that (J2) is an invariant.

The results obtained until now can be summarized in

Theorem 0. *Assume that the only actions in DD on the shared variables* wa, g, nx, st, res, a, inv *are D2, D4, D5′ and D6. Assume invariant validity of the postulates (P0) through (P5). Then* (CL0) *and* (CL1) *are invariants.*

It remains to guarantee the postulates (P0) through (P5). Since they do not involve the ghost variables σ and $\beta.Q$, we may forget about these ghost variables and thus, henceforward, argue about command $D5$ instead of $D5'$.

In order to get indications for the treatment of the variables, we reformulate the postulates (P) and strengthen them slightly. We first compile a list of new postulates (Q) and then verify that its conjunction implies all postulates (P). In view of (P0), (P1) and (P2), we postulate

(Q0) P **in** $D2, D4, D5, D6 \Rightarrow i \neq 0$, and

(Q1) P **in** $D4, D5, D6 \Rightarrow i = \mathrm{nx}.h$.

In order to get (P2), we also require

(Q2) P **in** $D2 \wedge \mathrm{nx}.h = 0 \Rightarrow \mathrm{wa}.i$.

In view of command $D4$ and postulates (P0) and (P4), we postulate

(Q3) $(P$ **in** $D4 \wedge \neg \mathrm{wa}.i) \vee P$ **in** $D5$
$\Rightarrow \langle \mathrm{inv}.i, \mathrm{st}.h, \mathrm{st}.i, \mathrm{res}.i \rangle \in R$.

In order to get (P0) from (Q3), we also require

(Q4) P **in** $D4, D5 \wedge \mathrm{wa}..i \Rightarrow h = g$.

Here, the possibility P **in** $D4$ is added for usage in Theorem 1 below.

In view of command $D5$ and postulate (P1), we postulate

(Q5) P **in** $D6 \Rightarrow \neg \mathrm{wa}.i$.

In order to get $\mathrm{nx}.i = 0$ in (P1), we require

(Q6) $\mathrm{nx}.g = 0 \vee \mathrm{nx}.(\mathrm{nx}.g) = 0$.

Postulates (P5) and (P3) are repeated as

(Q7) $\neg \mathrm{wa}.g$,
(Q8) P **in** $E \Rightarrow \mathrm{a}.P \notin \{g, \mathrm{nx}.g\}$.

In fact, (P0) follows from (Q0), (Q1), (Q3) and (Q4). Postulate (P1) follows from (Q0), (Q1), (Q5) and (Q6). Postulate (P2) follows from (Q0) and (Q2). Postulate (P3) is (Q8). Postulate (P4) follows from (Q3). Postulate (P5) is (Q7).

Since the non-atomic shared variables inv, st, res are only assumed to be safe, we also postulate

(Q9) P **in** $D4 \wedge Q$ **in** $E \Rightarrow \mathrm{a}.Q \notin \{h.P, \mathrm{nx}.(h.P)\}$,
(Q10) P **in** $D4 \Rightarrow \neg \mathrm{wa}.h$.

Now we can prove

Theorem 1. *Assume that the only actions in DD on the shared variables* wa, g, nx, st, res, a, inv *are D2, D4, D5 and D6. Assume invariant validity of the postulates* (Q0) *up to* (Q10). *Then the implementation of the data object is linearizable. Moreover, the read and write operations of the safe variables* inv.k, st.k *and* res.k *do not destructively interfere.*

Proof. The first assertion follows from Theorem 0 and the fact that the predicates (Q0) through (Q10) imply the predicates (P0) through (P5).

With respect to interference, we have to consider command $D4$ and the write action of inv in E and the final read action of res. We begin with $D4$. If two processes concurrently execute $D4$ there are two possibilities of interference. It may be that one of the processes, say P, is writing into a variable that is being read by the other process, say Q. Then $i.P = h.Q$. By (Q10), we then have $\neg$wa.$(h.Q)$ and hence $\neg$wa.$(i.P)$. By (8), this implies that P's write operation into st.$(i.P)$ is nonmodifying. Since st.$(i.P)$ is a safe variable, this is a harmless interference.

The other possibility is that P and Q are both writing into the same variables. We then have $i.P = i.Q$. If $\neg$wa.$(i.P)$ then (8) implies that both write actions are nonmodifying and the interference is harmless. If wa.$(i.P)$ then predicate (Q4) implies that $h.P = g = h.Q$. Then the determinacy of *locapply* implies that P and Q are writing the same values in st.$(i.P)$ and res.$(i.P)$. Since these variables are safe, this is a harmless interference.

Interference between $D4$ and E is precluded by (Q9) and (Q1). Interference between different processes in E is precluded by invariant (K0) and the disjointness of the sets *addr.P*. Finally, interference could occur between some process P in $D4$ and Q's final read action $z :=$ res.(a.Q). In that case, $i.P =$ a.Q and (K1a) implies $\neg$wa.(a.Q). Therefore, command $D4$ of P does not modify res.(a.Q). Again the interference is harmless. □

5 A sound implementation in quadratic space

In this section we provide a concrete program that satisfies the postulates (Q). More precisely, we extend the program such that values are assigned to the local variables h and i, and that the nondeterminate choice of a.P in $C0$ is guided. This is done in such a way that the postulates (Q0) up to (Q10) can be proved and that in a later stage progress can be ascertained.

In this section, several design choices are made to preclude specific harmful scenarios. It is only after all choices have been made that we can provide a long list of invariants and prove the *absence* of harmful scenarios.

As announced earlier, we use h as a local copy of g, which may be outdated. We therefore begin command DD with $h :=$ g. Since process P enters DD with wa.(a.P), postulate (Q2) may suggest to let command $D2$ be preceded by an assignment $i :=$ a.P. In this way, we get the tentative refinement of DD by

DD: |[$h :=$ g; $i :=$ a.P
; $D2$: ⟨**if** nx.$h = 0$ **then** nx.$h := i$ **fi**⟩
; $D3$: $i :=$ nx.h
; $D4$: *locapply* (inv.i, st.h, st.i, res.i)
; $D5$: wa.$i :=$ *false*
; $D6$: ⟨**if** $g = h$ **then** $g := i$ **fi**⟩]|.

Command $D3$ is introduced to establish (Q1) in the case that $D2$ does not modify nx.h. The choice $i :=$ a.P in combination with $D2$, however, raises the possibility of individual starvation: there is no way to guarantee that P ever establishes nx.g = a.P. We therefore allow other processes to establish this predicate. For the moment, we only introduce nondeterminacy. In Sect. 6, we show how this nondeterminacy is used (and eliminated) to guarantee bounded delay. Thus, $i :=$ a.P is replaced by

$D0$: |[**choose** $T \in$ *process*; $i :=$ a.T]|.

Now the consequent wa.i of condition (Q2) is in danger. We therefore provide a second option by introducing

$D1$: **if** $\neg$wa.i **then** $i :=$ a.P **fi**.

This, however, does not yet guarantee (Q2), for two reasons. Firstly, some other process may have set wa.(a.P) to *false* after P entered its loop body DD. Secondly, even after execution of $D1$, some process may set wa.i to *false*. For the moment, we only treat the first objection by introducing an additional test on wa.(a.P). In this way we come to the (still tentative) refinement

DD: |[$h :=$ g; D]|.

where D is given by

(9) D: **if** wa.(a.P) **then**
$D0$: |[**choose** $T \in$ *process*; $i :=$ a.T]|
; $D1$: **if** $\neg$wa.i **then** $i :=$ a.P **fi**
; $D2$: ⟨**if** nx.$h = 0$ **then** nx.$h := i$ **fi**⟩
; $D3$: $i :=$ nx.h
; $D4$: *locapply* (inv.i, st.h, st.i, res.i)
; $D5$: wa.$i :=$ *false*
; $D6$: ⟨**if** $g = h$ **then** $g := i$ **fi**⟩
fi.

In order to guarantee (Q9), the processes need to communicate the values of h. For this purpose we introduce

var b: **array** *process* **of** *address*,

with the intention that b.$P = h$ whenever P is in D. The value of b.P can be read by all processes, but modified only by process P. We do not want an assignment b.$P :=$ g, since it would refer to two shared variables at the same time. The sequential separation $h :=$ g; b.$P := h$ introduces the danger that some process Q reads b.P, before process P sets b.P and subsequently enters D with a value $h \neq g$ (entering with $h = g$ is harmless if (Q8) holds). This danger is precluded by providing a test $h = g$ to guard command D. In this way we arrive at the refinement of DD given by

(10) DD: |[$h :=$ g; b.$P := h$
; **if** $h =$ g **then** D **fi**]|,

where D is given by (9) above. We shall show that $C1$ as refined by (7), (10) and (9) indeed satisfies the aims set for this section. As indicated above, the tests $h =$ g and wa.(a.P) are both necessary. In Sect. 7, we shall show that it is not even allowed to reverse the order of the two tests.

Analogously to the convention in Sect. 3, we write $Di \sim j$ to denote a composition Di; . . . ;Dj. From the program text together with invariant (K0) and formula (5) we easily get that the following predicates are invariant:

(K2) P **in** $D \Rightarrow h.P =$ b.P,
(K3) P **in** $D1 \sim 2 \Rightarrow i.P \neq 0$.

We still have to refine the choice of $\mathtt{a}.P$ in command $C0$ in such a way that (Q8) and (Q9) can be proved. For this purpose, each process P is equipped with a persistent private variable $s.P$ to stand for the set of addresses currently not available for $\mathtt{a}.P$, according to the declaration

var $s.P$: **set of** *address*;
initially $s.P = (address \backslash addr.P) \cup \{\mathtt{g}, \mathtt{a}.P\}$.

Command $C0$ is refined by

(11) $C0$: |[**if** $s.P = address$ **then** A **fi**
; **choose** $\mathtt{a}.P \in address \backslash s.P$
; E: |[$\mathtt{inv}.(\mathtt{a}.P) := u$
; $\mathtt{nx}.(\mathtt{a}.P) := 0$
; $\mathtt{wa}.(\mathtt{a}.P) := true$
;; $s.P := s.P \cup \{\mathtt{a}.P\}$]|]|,

where command A establishes the postcondition $s.P \neq address$. Then the choice of $\mathtt{a}.P$ in the complement $address \backslash s.P$ is certainly possible and is easily implemented, say by a linear search.

Since we want that $C0$ as given in (11) is a refinement of $C0$ of (6), we need the postulate

(Q11) $address \backslash s.P \subseteq addr.P$.

The postulates (Q8) and (Q9) may now suggest the postulates

(Q12) $\{\mathtt{g}, \mathtt{nx}.\mathtt{g}\} \subseteq s.P$,
(Q13) P **in** $D \wedge \neg(Q$ **in** $A)$
$\Rightarrow \{h.P, \mathtt{nx}.(h.P)\} \subseteq s.Q$.

The assignment to $\mathtt{s}.P$ in command E of (11) is introduced for the preservation of (Q12). In fact, as soon as $\mathtt{wa}.(\mathtt{a}.P)$ is made *true*, some process may establish $\mathtt{nx}.\mathtt{g} = \mathtt{a}.P$ by performing $D2$.

Command A in (11) is refined by

(12) A: |[$s.P := (address \backslash addr.P) \cup \{\mathtt{g}, \mathtt{a}.P\}$
; **for each** $T \in process$ **with** $T \neq P$ **do**
$i := \mathtt{b}.T$
; $s.P := s.P \cup \{i, \mathtt{nx}.i\}$ **od**]|.

So, after its renewed initialization in A, variable $s.P$ is only made larger. This implies that we have the invariant

(K4) $address \backslash addr.P \subseteq s.P$.

Invariant (K4) clearly implies postulate (Q11).

In the first command of A, the value of $\mathtt{g}$ is put into $s.P$ in view of postulate (Q12). The decision to put $\mathtt{a}.P$ into $s.P$, instead of $\mathtt{nx}.\mathtt{g}$, is motivated by the fact $\mathtt{a}.P$ is not modified by other processes, whereas $\mathtt{nx}.\mathtt{g}$ uses two highly vulnerable shared variables. The fact that $\mathtt{a}.P$ can be used here will follow from the invariant (L14) presented below.

The for-loop of A is intended to establish (Q13). In the postcondition of the for-loop, the set $s.P$ clearly contains at most $2 \times n$ elements of $addr.P$. It now follows from (5) that, if we take $m > 2 \times n$, command A establishes $s.P \neq address$, as required. We prefer a choice like $m = 4 \times n$, see Sect. 9. This implies $upb = 4 \times n^2$ and thus justifies the expression quadratic space in the title of this section.

The assignment to $s.P$ in command E of (11) is disjoint from the actions of the processes $Q \neq P$, in the sense of [1] Sect. 5.1. By the atomicity rule ([1] Theorem 6.26), we may therefore regard the last two statements of region E as a single atomic statement. This is indicated by means of the symbol ";;". It is not a restriction on the implementation, but only a convenience for the proof (otherwise a ghost variable $s'.P$ must be introduced to stand for $s.P \cup \{\mathtt{a}.P\}$ at the point of ";;". It is easy to see that, after these preparations, the program satisfies the invariant

(K5) P **in** $E \equiv \mathtt{a}.P \notin s.P$.

It is clear that (K5), (Q12) and (Q13) imply (Q8) and (Q9).

Remark. For the elegance of the program one might prefer to replace s by its complement, which is a subset of $addr.P$. We have not done so, since the present version is more convenient for the proof that is to be given. □

We now claim that program (4) with $C0$ refined by (11) and (12), and $C1$ refined by (7), (10) and (9) satisfies the invariants (Q0) through (Q13).

In the proof of this claim, we use the invariants (K0) through (K5) obtained above and a new list of invariants (L) to be presented now. Since we shall prove invariance under actions of P, the invariants (L) use Q and T as process names. Since we have to go through the list several times, the invariants are numbered consecutively. Each invariant is motivated either as being needed for some of the postulates (Q) or for the preservation of one of the other invariants. In either case, the motivation is not meant as a proof but merely as an announcement.

In view of invariant (K3) and the contents of command $D3$, the postulates (Q0) and (Q1) are replaced by

(L0) Q **in** $D3 \sim 6 \Rightarrow \mathtt{nx}.(h.Q) \neq 0$,
(L1) Q **in** $D4 \sim 6 \Rightarrow \mathtt{nx}.(h.Q) = i.Q$.

Postulate (Q2) is repeated as

(L2) Q **in** $D2 \wedge \mathtt{nx}.(h.Q) = 0 \Rightarrow \mathtt{wa}.(i.Q)$.

In order to preserve (L2) under $D1$, we also introduce

(L3) Q **in** $D0 \sim 1 \wedge \mathtt{nx}.(h.Q) = 0 \Rightarrow \mathtt{wa}.(\mathtt{a}.Q)$.

Since process Q enters region D only if $h.Q = \mathtt{g}$, some invariants mention region D. Since $i.Q$ is modified in D, these invariants do not use $i.Q$. The first case is invariant (L4), which in conjunction with (L0) and (L1) implies (Q3):

(L4) Q **in** $D \wedge \mathtt{nx}.(h.Q) = k \neq 0 \wedge (\neg \mathtt{wa}.k \vee Q$ **in** $D5)$
$\Rightarrow \langle \mathtt{inv}.k, \mathtt{st}.(h.Q), \mathtt{st}.k, \mathtt{res}.k \rangle \in R$.

In order to preserve (L4) when process Q enters region D, we postulate

(L5) $\mathtt{nx}.\mathtt{g} = k \neq 0 \wedge \neg \mathtt{wa}.k$
$\Rightarrow \langle \mathtt{inv}.k, \mathtt{st}.\mathtt{g}, \mathtt{st}.k, \mathtt{res}.k \rangle \in R$.

In view of (L0) and (L1), postulate (Q4) combined with the optional postulate (P2') is replaced by the contraposition:

(L6) Q **in** $D \wedge h.Q \neq \mathtt{g}$
$\Rightarrow \mathtt{nx}.(h.Q) \neq 0 \wedge \neg \mathtt{wa}.(\mathtt{nx}.(h.Q))$.

Postulate (Q10) is strengthened to

(L7) Q **in** $D \Rightarrow \neg \mathrm{wa}.(h.Q)$.

Postulates (Q5), (Q7) and (Q6) are repeated as

(L8) Q **in** $D6 \Rightarrow \neg \mathrm{wa}.(i.Q)$,
(L9) $\neg \mathrm{wa}.\mathrm{g}$,
(L10) $\mathrm{nx}.\mathrm{g} = 0 \vee \mathrm{nx}.(\mathrm{nx}.\mathrm{g}) = 0$.

In order to keep (L10) invariant under $D2$, and in view of (L2), we postulate

(L11) $\mathrm{wa}.k \Rightarrow \mathrm{nx}.k = 0$.

In order to preserve (L11), we need

(L12) Q **in** $E0 \Rightarrow \mathrm{nx}.(\mathrm{a}.Q) = 0$,

where $E0$ is the final subcommand of E in (11) that consists of the assignments to wa and s.

The last part of the list consists of the invariants of memory management. Postulate (Q12) is repeated as

(L13) $\{\mathrm{g}, \mathrm{nx}.\mathrm{g}\} \subseteq s.Q$.

In order to preserve (L13) in the first command of A, we need (as announced above):

(L14) $\mathrm{nx}.\mathrm{g} \notin addr.Q \vee \mathrm{nx}.\mathrm{g} = \mathrm{a}.Q \vee Q$ **in** $E \vee \mathrm{wa}.(\mathrm{a}.Q)$.

For treatment of (Q13), we first define a location predicate. If $T \neq Q$, we write Q **done** T to denote that process Q is not in command A of (12) or has treated process T in the for-loop of command A. Now (Q13) is strengthened to

(L15) Q **done** $T \wedge T$ **in** D
$\Rightarrow \{h.T, \mathrm{nx}.(h.T)\} \subseteq s.Q$.

In order to preserve (L15), we also need

(L16) Q **at** $T \wedge T$ **in** D
$\Rightarrow i.Q = h.T \vee \{h.T, \mathrm{nx}.(h.T)\} \subseteq s.Q$,

where Q **at** T is used to denote that Q is in A, has executed $i := b.T$ in its for-loop and has not yet completed the subsequent assignment of $s.Q$.

This concludes the list of invariants (L). Before proving that these predicates are invariants of the program, we show that they imply the predicates (Q).

Well, (Q0) follows from (K3), (L0) and (L1); (Q1) is (L1); (Q2) is (L2); (Q3) follows from (L0), (L1) and (L4); (Q4) follows from (L1) and (L6); (Q5) is (L8); (Q6) is (L10); (Q7) is (L9). As noticed above, (Q8) and (Q9) follow from (K5), (Q12) and (Q13), and (Q11) follows from (K4). Finally, (Q10) follows from (L7); (Q12) is (L13); (Q13) follows from (L15).

Since the processes are tightly coupled, the proof that the predicates (L0) through (L16) are invariants of the algorithm is a huge case analysis. In each of the cases, a small argument is sufficient. We first treat some of the commands separately.

Lemma 1. *Command D2 of process P preserves the predicates* (L0) *through* (L16).

Proof. If $\mathrm{nx}.(h.P) \neq 0$ in the precondition of $D2$, then $D2$ is equivalent to *skip*. Since P enters $D3$ this action only threatens predicate (L0) for $Q := P$; it preserves this predicate since the consequent holds.

Otherwise we have $\mathrm{nx}.(h.P) = 0$ in the precondition. By (L2) and (L6), it follows that the precondition satisfies $h.P = \mathrm{g}$ and $\mathrm{wa}.(i.P)$. Therefore $D2$ only sets $\mathrm{nx}.\mathrm{g}$ to $i.P$ while P enters $D3$. Predicate (L0) for $Q := P$ is preserved because of (K3). If Q is a process in $D4 \sim 6$ with $\mathrm{nx}.(h.Q) = i.Q$ then $i.Q \neq 0$ by (L0), so that $D2$ of P does not modify $\mathrm{nx}.(h.Q)$. Therefore, (L1) is preserved. The predicates (L2), (L3), (L6), (L7), (L8) and (L9) are not threatened. Because of $\mathrm{wa}.(i.P)$, the predicates (L4) and (L5) are not threatened either.

Since $\mathrm{wa}.(i.P)$ implies $\mathrm{nx}.(i.P) = 0$ by (L11), predicate (L10) is preserved. Since $\mathrm{nx}.\mathrm{g}$ is set, preservation of (L11) follows from (L9). By (K5) and (L13), $\mathrm{a}.Q \neq \mathrm{g}$ for all Q in $E0$; therefore $D2$ preserves (L12).

Because of (K4), command $D2$ threatens (L13), (L15) and (L16) only for process Q with $i.P \in addr.Q$. Then $\mathrm{wa}.(i.P)$ implies $i.P = \mathrm{a}.Q$ and Q **in** $C1$ by (K1), so that $i.P \in s.Q$ by (K5). Therefore (L13), (L15) and (L16) are preserved.

Since $D2$ only modifies $\mathrm{nx}.h$, it only threatens (L14) if $h.P = \mathrm{g}$. If $i.P \notin addr.Q$ it establishes the first disjunct of (L14). Otherwise, $\mathrm{wa}.(i.P)$ implies $i.P = \mathrm{a}.Q$ by (K1), so that the second disjunct of (L14) is made *true*. Therefore, (L14) is preserved. □

Lemma 2. *Commands D3, D4 and D5 of process P preserve the predicates* (L0) *through* (L16).

Proof. Command $D3$ clearly preserves (L1) for $Q := P$ and does not threaten other predicates of the list.

Command $D4$ can only modify $\mathrm{st}.i$ and $\mathrm{res}.i$. Therefore, only (L4) and (L5) are threatened. By determinacy of relation R as specification of *locapply*, it follows from (L0), (L1) and (L4) that $D4$ can only modify $\mathrm{st}.i$ and $\mathrm{res}.i$ if $\mathrm{wa}.i$ holds. So we may assume $\mathrm{wa}.i$. Because of (L7), predicate (L4) is threatened only for processes Q in D5 with $\mathrm{nx}.(h.Q) = i.P$. Then $h.Q = \mathrm{g} = h.P$ by (L6) and (L1). Therefore, $D4$ establishes the consequent of (L4) for such Q. This proves that $D4$ preserves (L4). Since the antecedent of (L5) yields $\neg \mathrm{wa}.k$, predicate (L9) implies that (L5) is preserved.

Command $D5$ only threatens (L2), (L3), (L4), (L5) and (L14) by setting $\neg \mathrm{wa}.i$, and (L8) for $Q := P$ by entering $D6$. We first notice that (L0), (L1) and (L6) imply

P **in** $D5 \wedge \mathrm{wa}.(i.P) \Rightarrow h.P = \mathrm{g} \wedge \mathrm{nx}.\mathrm{g} = i.P \neq 0$.

Command $D5$ threatens (L2) and (L3) only for processes Q with $\mathrm{nx}.(h.Q) = 0$. Then (L6) implies $h.Q = \mathrm{g}$, contradicting $\mathrm{nx}.\mathrm{g} \neq 0$. This proves that (L2) and (L3) are preserved. Since $D5$ only makes $\mathrm{wa}.(i.P)$ *false*, predicate (L4) is threatened only for processes Q in D with $\mathrm{nx}.(h.Q) = i.P$. From $\mathrm{wa}.(i.P)$ and (L6) follows $h.Q = \mathrm{g} = h.P$. Therefore, preservation of (L4) for Q follows from (L4) for $Q := P$. It follows from $h.P = \mathrm{g}$ and (L4) for $Q := P$ that (L5) is preserved. If the fourth disjunct of (L14) is falsified the second disjunct remains valid. So (L14) is preserved.

Predicate (L8) for $Q := P$ is preserved since $D5$ establishes the consequent. □

Lemma 3. *Command D6 of process P preserves the predicates* (L0) *through* (L16).

Proof. If $h.P \neq g$, command $D6$ is equivalent to *skip* and preserves all predicates of the list. So, we assume that $h.P = g$ in the precondition. By (L0), (L1), (L8) and (L10), the precondition then satisfies

$$H\colon\ h.P = g \wedge nx.g = i.P \neq 0$$
$$\wedge \neg wa.(i.P) \wedge nx.(i.P) = 0.$$

Then $D6$ sets g to $i.P$. Since $nx.(i.P) = 0 \notin addr.Q$, predicate (L14) is preserved. In view of H, the only other predicates threatened are (L6) and (L13). If $D6$ threatens (L6) for process Q, it is because $h.Q = g$ holds in the precondition and is made *false* by $D6$. Since predicate H together with $h.Q = g$ implies

$$nx.(h.Q) \neq 0 \wedge \neg wa.(nx.(h.Q)),$$

predicate (L6) is preserved for all Q.

If $h.P = g$, command $D6$ replaces g by nx.g and nx.g by 0. Since $0 \in s.Q$ by (K4), this implies that (L13) is preserved. □

Lemma 4. *All commands of C0 of process P preserve the predicates* (L0) *through* (L16).

Proof. It is clear that command A of process P only threatens (L13), (L15) and (L16) for $Q := P$. The first assignment to $s.P$ in A only threatens (L13). By (L14) and (K1), its precondition satisfies $nx.g \notin addr.P$ or $nx.g = a.P$. Therefore, (L13) is preserved.

The assignment to i when P treats process T in its for-loop only threatens (L16), and only if T is in D. By (K2), the assignment to i has postcondition $i.P = h.T$, so that (L16) is preserved. The subsequent assignment to $s.P$ does not make $s.P$ smaller; it therefore only threatens (L15) for $Q := P$; it preserves (L15) because of (L16). The choice of $a.P$ only threatens (L14) for $Q := P$. This predicate is preserved since P enters E.

For the discussion of the commands of P in E, we first observe that (K5), (L13) and (L15) imply

$$a.P \notin \{g, nx.g\}$$
$$\wedge\ (\forall T{:}T \textbf{ in } D\colon a.P \notin \{h.T, nx.(h.T)\}).$$

The assignment to inv.(a.P) only threatens (L4) and (L5). Preservation of (L4) follows from $a.P \neq nx.(h.T)$ when T is in D. Preservation of (L5) follows from $a.P \neq nx.g$.

The assignment $nx.(a.P) := 0$ sets an element of nx equal to 0 and coincides with entering $E0$. Since $a.P \neq g$ and $a.P \neq h.T$ for all T in D, the only predicate threatened is (L12) for $Q := P$. This predicate is preserved since the consequent is established.

Command $E0$ is the pair of assignments to wa.(a.P) and $s.P$. It coincides with leaving E and entering $C1$. It therefore only threatens (L6), (L7), (L8), (L9) and (L11) by modifying wa and (L13), (L15) and (L16) for $Q := P$ by modifying $s.P$.

Preservation of (L6) and (L7) follows from $a.P \neq nx.(h.Q)$ and $a.P \neq h.Q$ for Q **in** D. Preservation of (L8) follows from (L1) and $a.P \neq nx.(h.Q)$ for Q in D. Predicate (L9) is preserved since $a.P \neq g$. Preservation of (L11) follows from (L12). Since $s.P$ is only made bigger (L13), (L15) and (L16) for $Q := P$ are preserved. □

Theorem 2. *Program* (4) *with C1 and C0 refined according to* (7), (9), (10), (11) *and* (12) *has the invariants* (L0) *through* (L16). *It is a linearizable implementation of the data object. Moreover, the read and write operations of the safe variables* inv.k, st.k *and* res.k *do not destructively interfere.*

Proof. Since Theorem 1 applies, it remains to prove that (L0) through (L16) are invariants.

Initially, all elements of wa are *false* and $g \in s.Q$ and $nx.g = 0$ and $0 \notin addr.Q$ and $0 \in s.Q$. This proves the initial validity of (L5), (L9), (L10), (L11), (L13) and (L14). The other predicates hold initially since no process is active.

It remains to verify that all commands of the program preserve the predicates. Command $C0$ preserves the predicates because of Lemma 4. Passing the guard of $C1$, and the assignments to $h.P$ and $b.P$ do not threaten any of the predicates.

We come to the point where process P enters D by passing the test $h = g$. Then P **in** D becomes *true*. The only predicates that are threatened by this action, are (L4), (L6) and (L7) for $Q := P$, and (L15) and (L16) for $T := P$. Preservation of (L4) follows from $h.P = g$ and (L5). Since P enters D only if $h.P = g$, preservation of (L6) is clear. Preservation of (L7) follows from $h.P = g$ and (L9). Preservation of (L15) and (L16) for $T := P$ follows from $h..P = g$ and (L13).

We now consider command D. It is clear that passing the test wa.(a.P) preserves (L3) for $Q := P$ and does not threaten the other invariants. Command $D0$ does not threaten any of the predicates (L0) through (L16). By (L3) for $Q := P$, command $D1$ preserves (L2) for $Q := P$. It does not threaten the other invariants.

The commands $D2$, $D3$, $D4$, $D5$ and $D6$ are treated in Lemmas 1, 2 and 3. Finally, none of the predicates (L0) through (L16) are threatened by the action of exiting D or exiting the repetition, or by the final assignment to z. □

Remarks. In the predicates and the proofs, the location predicates like P **in** D and Q **at** P can be eliminated by introducing ghost variables. For example, one can introduce an integer ghost variable t private to process P such that always $0 \leq t < 9$ and

P **in** $D \equiv t \geq 1$,
P **in** $D0 \equiv t = 2$,
P **in** $D1 \equiv t = 3$, etc.

Then a positive outcome of the test $h = g$ guarding D must be accompanied by a simultaneous assignment $t := 1$. A positive outcome of the subsequent test wa.(a.P) must be accompanied by $t := 2$; a negative outcome by $t := 0$. This could be encoded by

$\langle$ **if** wa.(a.P) **then** $t := 2\rangle$; $D0$; . . ; $D6$
else $t := 0\rangle$ **fi**.

If the test in $D1$ finds $\neg$wa.i, the assignment $i := a.P$ is accompanied by $t := 4$. If it finds wa.i, the assignment $t := 4$ is executed instantaneously. So the important tests in the program must be accompanied by simultaneous assignments to t. Since the program gets cluttered with assignments to ghost variables, we prefer to use location predicates whenever possible.

In this section we use a variation of the classical method of Owicki and Gries [13] in the style of Lamport and Schneider ([10], [11]). Previous versions of the proof used several shortcuts. Indeed, the invariants (L3), (L8), (L12), (L16) are so local, that it is a pity that we have to mention them. The shortcuts have been eliminated, however, since they increased the complexity of the argument without decreasing the size.

6 The program is made wait-free

We first give a result that can be interpreted as progress of the system as a whole.

Lemma 5. *During every execution of the body of C1 by process P while* wa.(a.P) *remains true, variable* g *is modified at least once.*

Proof. The body of $C1$ begins with $h := g$. If wa.(a.P) remains *true* and g is not modified before $D6$, it follows from the text of (9) and (10) that $D6$ is executed with $h = g$. Then (L0), (L1) and (L10) imply that $i \neq g$ and hence that command $D6$ modifies g. □

We now have to guarantee some kind of fairness in the choice of T in $D0$, see (9). Since it is never known which process is executing, it seems likely that fairness must be guaranteed by means of shared data. Following Herlihy [5], we (conceptually) add a process number to the state of the object. At every state transition of the object the process number is incremented by one modulo n. The process number at the old state indicates the process T to be chosen in $D0$. Compare program (3). So we additionally declare

var seq: **array** *address* **of** *process*,

and we replace the commands $D0$ and $D4$ in D by

(13) $D0'$: $i :=$ a.(seq.h),
$D4'$: |[*locapply* (inv.i, st.h, st.i, res.i)
; seq.$i :=$ (seq.h + 1) mod n]|.

It is clear that this yields a refinement of command D of (9). The element seq.k can be regarded as an additional component of the state st.k. So we consider an extended state space $X' = X \times process$ and an extended relation $R' \subseteq U \times X' \times X' \times Z$ given by

$\langle u, \langle x, Q\rangle, \langle y, T\rangle, z\rangle \in R'$
$\equiv \langle u, x, y, z\rangle \in R \wedge T = (Q + 1) \bmod n.$

Relation R' is also total and deterministic. Since $D4'$ is the analogue of $D4$ for the extended state and relation, (L5) and (L4) give rise to the additional invariants

(L17) nx.g $= k \neq 0 \wedge \neg$wa.k
$\Rightarrow$ seq.k = (seq.g + 1) mod n,

(L18) Q **in** $D \wedge$ nx.($h.Q$) $= k \neq 0 \wedge (\neg$wa.$k \vee Q$ **in** $D5$)
$\Rightarrow$ seq.k = (seq.($h.Q$) + 1) mod n.

Lemma 6. *The following stability properties hold:*
(S0) seq.g *is constant while* g *is constant,*
(S1) *whenever* g *is modified,* seq.g *is incremented by one modulo n,*
(S2) $\neg$wa.(a.Q) *is stable while* $\neg$(Q **in** $C0$),
(S3) $h.Q \neq$ g *is stable while* (Q **in** D).

Proof. It follows from (L0), (L1) and (L18) that seq.k is modified only by process P when $k =$ nx.($h.P$) and wa.k holds. By (L6) and (L10) we then have $h.P =$ g and $k =$ nx.g $\neq$ g. This proves (S0). Property (S1) follows from (L0), (L1), (L8) and (L17).

Property (S2) is only threatened by the assignment wa.(a.P) := *true* in E, and only if a.P = a.Q. By invariant (K0) and the disjointness of the sets *addr*.Q, the equality a.P = a.Q implies $P = Q$ and hence Q **in** $C0$. This proves (S2).

It is clear that $h.Q$ is constant while Q **in** D. Therefore (S3) is only threatened by modification of g, say by command $D6$ of some process P. This command threatens (S3) only if g = $h.P$ and $i.P = h.Q \neq$ g in the precondition. By (L0), (L1) and (L6), it follows that nx.g = $i.P \neq 0$ and nx.(nx.g) = nx.($h.Q$) $\neq 0$, contradicting (L10). This proves (S3). □

The proof that the program is wait-free relies on the rôle of the sequence number in command $D0'$. In the following result, we need a new location predicate: we write P **in** $D1T$ to indicate that P is in the **then**-part of command $D1$. In other words, the test $\neg$wa.i has been executed and has yielded *true* and the subsequent assignment to i has not yet been executed (this exceptional region $D1T$ was found to be needed in our verification by means of the theorem prover NQTHM, see [8]).

Lemma 7. *For a fixed process P, let predicate X be given by*

X: seq.g $= P$
$\Rightarrow$ nx.g $\in \{0,$ a.$P\}$
$\wedge$ ($\forall Q$: Q **in** $D1 \sim 6 \wedge h.Q =$ g
: $i.Q =$ a.$P \wedge \neg$(Q **in** $D1T$)).

(a) *Every modification of* g *establishes X.*
(b) X *is stable while* wa.(a.P).

Proof. (a) By postulate (P1), every modification of g has postcondition nx.g = 0. It follows from (S3) that every modification of g has postconditions $h.Q \neq$ g for all Q in D. These predicates imply X.

(b) Now let $X \wedge$ wa.(a.P) hold in the precondition of some command. We have to show that the command does not invalidate X. By (K1a), process P is in $C1$, so that a.P is constant. By part (a), any modification of g does not invalidate X. So we assume that g is constant. By (S0), this implies that seq.g is constant.

If nx.g gets a new value $\neq 0$, this is done by some process Q in $D2$ with $h.Q =$ g and hence $i.Q =$ a.P by X. So, this does not invalidate X. If seq.g = P and some process Q with $h.Q =$ g enters $D1$ by executing $D0$, it sets $i.Q :=$ a.P and therefore does not invalidate X. If Q with $i.Q =$ a.P executes the test of $D1$ then wa.(a.P) implies that Q does not enter $D1T$ and hence does not invalidate X. Finally, if some process Q with $h.Q =$ g executes $D3$, then (L0) implies nx.g $\neq 0$, so that X implies nx.($h.Q$) = nx.g = a.P. Therefore, $D3$ does not invalidate X. □

We are ready to prove our main result.

Theorem 3. *Program* (4) *with C0 refined according to* (11) *and* (12) *and C1 refined according to* (7), (9) *and* (10) *with D0 and D4 replaced by D0′ and D4′ of* (13) *is linearizable and wait-free.*

Proof. Since we have refined the program of Sect. 5, Theorem 2 implies that the implementation is linearizable. It remains to prove that the implementation is wait-free: *apply* of process P terminates after a bounded number of steps of process P. In view of the program text, it suffices to prove that the loop $C1$ of process P terminates after a bounded number of steps of P. Since wa.(a.P) is the guard of this loop, it follows from (S2) that it suffices to prove that wa.(a.P) becomes *false* after a bounded number of steps of P.

It follows from Lemma 5 and (S0) and (S1) in Lemma 6, that, if wa.(a.P) remains *true*, after at most n executions of the body of $C1$ of P, variable g gets a value with seq.g $= P$.

Now assume that wa.(a.P) still holds and that g gets a new value with seq.g $= P$. By Lemma 7, the modification of g establishes predicate X. Since seq.g $= P$, it follows from X that the next modification of g will establish g $=$ a.P and then (L9) implies that $\neg$wa.(a.P) has been established. Then, by (S2), command $C1$ of P terminates within a bounded number of steps of P. In this way, we see that the loop $C1$ of P terminates after at most $n+1$ executions of its body. □

Remark. It follows from postulate (P4) that command $D4$ may be replaced by **if** wa.i **then** $D4$ **fi**. This can be advantageous for the performance of the system if calls of *locapply* require extensive data transfer or computation.

7 Discussion of the design

Some aspects of our solution are delicate. For instance, in a previous version of this paper we claimed that, after the assignment b.$P := h$, the order of the tests $h =$ g and wa.(a.P) was irrelevant. The following scenario shows that a reverse order of testing leads to incorrectness. So, here, process P first tests wa.(a.P) and then $h.P =$ g.

We assume $n \geqq 2$. We let P and Q be two different processes and assume that seq.g $= P - 2$ (modulo n) initially. The scenario begins with process Q calling *apply* and treating its own invocation. We then have g $= k_1 \in$ *addr*.Q and $k_1 \notin$ {b.P, nx.(b.P)} and seq.g $= P - 1$ (modulo n). Then P calls *apply*, chooses a.$P = k_3$, sets wa.k_3 and $h.P := k_1$. Then Q calls *apply* again and treats its own invocation. We then have g $= k_2 \neq k_1$ and seq.g $= P$. Then Q calls *apply* again. It executes A and chooses a.$Q = k_1$. This is possible since $k_1 \notin$ {b.P, nx.(b.P)}. Then P sets b.$P := h.P$ and verifies that wa.(a.P) holds. Then Q enters $C1$ and sets nx.g $:=$ a.P ($= k_3$) since seq.g $= P$. Process Q treats the invocation of P, sets $\neg$wa.(a.P) and sets g $:= k_3$. Again Q enters the loop. Since k_1 contains the only waiting invocation, Q sets nx.$k_3 := k_1$ and treats its own invocation at k_1 and sets g $:= k_1$. Then P verifies that $h.P =$ g holds. It finds no waiting invocation and sets $i.P :=$ a.P ($= k_3$). It then treats its own invocation for the second time. In this way, P's invocation is treated twice. Notice that condition (P2) is violated.

Another seemingly innocent modification is as follows. One might be tempted to replace command A by

A': |[$s :=$ *address* \ *addr*.P
; **for each** $T \in$ *process* **do**
$i :=$ b.T
; $s := s \cup \{i,$ nx.$i\}$ **od**]|.

This is incorrect, for it allows the following scenario that violates (P0). We assume $n \geqq 3$. Let P, Q and T be three different processes.

The scenario begins in a situation where g $= k_1$ and seq.$k_1 = Q$ while process Q has completed $C0$ with a.$Q = k_2 \in$ *addr*.Q. Process T in $C1$ sets nx.$k_1 := k_2$, treats the invocation of Q and sets wa.$k_2 :=$ *false*. Command $D6$ of T is delayed, so that g remains k_1. Process Q evaluates wa.(a.Q), skips its repetition, terminates the invocation, begins a new invocation and begins A'. It makes $s.Q$ the complement of *addr*.Q. In its for-loop of A', process Q treats process P before T. After it has treated P, process P enters $C1$, sets $h.P :=$ g ($= k_1$) and subsequently enters region D with $h.P = k_1$, to recalculate the contents of address k_2.

Now process T executes $D6$ so that g $= k_2$. Then T executes the body of $C1$ once more and treats some invocation, so that g $= k_3 \neq k_2$. Then T enters the body of $C1$ again and sets b.$T := k_3$. Then Q resumes its for-loop in A'. Then $s.Q$ need not contain k_2. Therefore Q can choose a.$Q := k_2$. It then sets wa.(a.Q). Then process P continues, executes $D5$ and sets $\neg$wa.(a.Q). In this way, the new invocation of Q is destroyed before treatment. The call *apply* (Q) terminates and, erroneously, yields the result of the previous call. Notice that process P violates postulate (P0).

8 Comparison with Herlihy's program

Program (4) is a variation of Fig. 14 of [5]. For convenience of the reader, we give an interpretation of that program in our notation.

The array seq now is an array of unbounded integers. Compare&swap register g and array wa are not used. The equation seq.$k = 0$ is used as an encoding of wa.k. The program uses additional arrays

var pre: **array** *address* **of** *address*
; cnt: **array** *address* **of** *integer*,

and an initialization with, for an arbitrary address $k_0 \neq 0$ and for all processes P and addresses k,

st.$k_0 = x_0 \wedge$ nx.$k_0 = 0$
$\wedge$ b.$P = k_0 \wedge$ seq.$k > 0$
$\wedge (k \neq k_0 \equiv$ cnt.$k = 0)$.

Our program variable g is encoded in the array b. The program is given in (14).

(14) **proc** *apply* (**in** P: *process*, u: U; **out** z: Z);
|[*free* (P, u)
; **for** $Q \in$ *process* **with** $Q \neq P$ **do**
if seq.(b.P) < seq.(b.Q)
then b.P := b.Q **fi** **od**
; **while** seq.(a.P) $= 0$ **do** *thread* (P) **od**

```
;  b.P := a.P
;  release (P)
;  z := res.(a.P) ]|,
```

where procedure *thread* is given by

```
proc thread (in P:process);
  var h, i: address;
  |[  h := b.P
  ;   i := a.(seq.h mod n)
  ;   if seq.i ≠ 0 then i := a.P fi
  ;   ⟨if nx.h = 0 then nx.h := i fi⟩
  ;   i := nx.h
  ;   locapply (inv.i, st.h, st.i, res.i)
  ;   pre.i := h
  ;   seq.i := seq.h + 1
  ;   b.P := i ]|.
```

We gather that one of the methods for memory management mentioned in [5] consists of procedures *free* and *release* as given by

```
proc free (in Q:process, u:U);
  var j: address;
  |[  choose j ∈ addr.Q with cnt.j = 0
  ;   nx.j := 0; inv.j := u
  ;   cnt.j := n + 1; seq.j := 0
  ;   a.Q := j ]|,
proc release (in Q:process);
  var j: address; i: 0 .. n + 1;
  |[  j := a.Q; i := n + 1
  ;   while i ≠ 0 do
        j := pre.j; i := i − 1
      ; ⟨cnt.j := cnt.j − 1⟩
      od ]|.
```

In view of procedure *release*, the initialization of the linked list requires some care. One solution is to start with $\mathrm{pre}.k_0 = k_0$ and $\mathrm{cnt}.k_0 = \frac{1}{2} \times (n+1) \times (n+2)$.

The main difference between our program and (14) is that in our program the assignment $h := g$ is inside of repetition $C1$, so that h jumps repeatedly to the estimate of the current address. The corresponding commands of (14) are the for-loop that updates $\mathrm{b}.P$ before the repetition and the updating $\mathrm{b}.P := i$ in *thread*. This has the effect that $\mathrm{b}.P$ starts at an estimate of the current address and subsequently traverses the list. Therefore, our program may be regarded as a greedy version of (14).

The assignment $h := g$ in (10) is simpler, but program (14) has a simpler repetition, since $\mathrm{b}.P$ simply traverses the linked list. Our program requires the compare&swap register g, see $D6$. Program (14) only requires the consensus objects $\mathrm{nx}.k$. It seems that the conditional updates of $\mathrm{b}.P$ in the for-loop of (14) need not be atomic and may be expressed as

```
|[  i := b.Q
;   if seq.(b.P) < seq.i then
      i := b.Q; b.P := i fi ]|.
```

In (14), the final update $\mathrm{b}.P := \mathrm{a}.P$ is a forward jump in the linked list. We regard this as a superfluous, but harmless optimization. In our program, one could introduce a corresponding command after $C1$, but that would have to be an atomic conditional update like

$\langle$**if** $\mathrm{nx}.(\mathrm{a}.P) = 0$ **then** $g := \mathrm{a}.P$ **fi**$\rangle$.

Another striking difference between our program and (14) is that the memory management of (14) uses a release loop backward through the linked list, after the main repetition, whereas in (11) and (12) the current state is sufficient to find a free address. In fact, in our program, array b is only used for the memory management, whereas in (14) it is only used to replace or implement our compare&swap register g.

In program (14), the sequence numbers must be unbounded integers, since they are used in the initial **for**-loop for updating $\mathrm{b}.P$. Finally, as argued in [5] Sect. 4.2, program (14) requires $upb = m \times n$ with $m = n^2$.

We do not yet have a proof of (14). We have the impression that (14) (if correct) is just as difficult to prove correct as our more greedy program.

9 Complexity

The measure of space complexity is the number of memory addresses needed for all processes together. The measure of time complexity is the maximal number of steps one process has to perform for one invocation to be treated. In both cases the parameter is n, the number of processes. For the space complexity, the local computation of *locapply* gives a linear contribution, since it is performed by each process. For the time complexity it is regarded as a constant.

It is easy to see that the memory space required is proportional to the size of *address*, which is $upb = m \times n$. Here we use that each set $s.P$ only requires m booleans because of (K4). As mentioned in the discussion of region A of (12), we need $m > 2 \times n$. In order to get a better time complexity, we choose m with $3 \times n \leqq m \leqq 6 \times n$. Then the space complexity is quadratic.

The time complexity of command A is proportional to n. The worst case time complexity of the choice of $\mathrm{a}.P$ in $C0$ is proportional to m, and hence to n as well. As proved in Sect. 6, the loop $C1$ terminates after at most $n+1$ executions of its body. Therefore, the worst case time complexity of *apply* is of order n.

Since $m \geqq 3 \times n$, command A establishes

$$\#(address \setminus s.P) \geqq m - 2 \times n \geqq n.$$

Therefore, command A is executed at most once in n calls of *apply*. This implies that the amortized contribution of command A to the time complexity is constant. The choice of $\mathrm{a}.P$ in $C0$ can be implemented by a linear search in $addr.P$. Then it need not require more than constant time in amortized sense. If process P is the only active process, it executes the body of loop $C1$ just once. This implies that, if process P is the only active process, the time complexity of *apply* is constant in amortized sense.

The space complexity of Herlihy's program (see (14)) is cubic, cf. [5], since it requires $upb = n^3$. The time complexity of (14) is quadratic since the search space for *free* (P) has size $m = n^2$. It must be mentioned, however, that this is not a fair comparison, since the program in [5] does not need

a compare&swap register. The program in [6] has quadratic time complexity and quadratic space complexity. We cannot give a detailed comparison with [6], since that paper is based on completely different ideas.

10 Concluding remarks

Inspired by Herlihy's program in [5], we designed a variation based on a compare&swap register, in which the memory management looked simpler and turned out at least to be cheaper. Correctness of our program could only be achieved by searching alternately for stable predicates and refuting scenarios. In an earlier version, we had an operational argument for the invariance of (L13) and (L15). When the referees were not convinced, a reexamination uncovered a bug in the program (see the first part of Sect. 7) and lead to an explosion of invariants. The understanding gained in this way enabled us to construct the abstract program of Section 4 and finally to provide a formal proof.

In the program, the processes are so tightly coupled and the invariants are so unwieldy that we do not use the standard separation between proof outlines for the processes and interference freedom, as exposed in [1]. Instead of this, we use global invariants with location predicates. Indeed, we found it to be more convenient to consider, for every separate action, the list of all invariants than to consider, for every separate invariant, the list of all actions. This approach is inspired by UNITY of [2].

Since there are more than twenty simple commands and more than twenty invariants, the proof requires more than 400 verifications. The present proof is sufficiently detailed that it can be verified by means of a proof checker. In fact, in [8], we report on a mechanical verification of a program with an even smaller grain of atomicity. Every implementation of course must be tested, but the scenarios of Sect. 7 are so unlikely that postitive results of testing must not increase our confidence. In fact, experience shows that in this area assertional methods are indispensable.

There are several directions open for future research. Firstly, it would be interesting to prove Herlihy's program, say in the version (14). Secondly, it may be possible to eliminate the compare&swap register g from our program, without introducing unbounded integers. Thirdly, the assumption that relation R and procedure *locapply* are deterministic should be removed, without introducing structured consensus variables. In fact, current research suggests that this can be done even without increasing the computational complexity. Finally, one could wish to bring the space complexity of the program down from quadratic to linear.

Acknowledgements. We profited from many discussions with J.E. Jonker, R.M. Dijkstra, R. Groenboom, J.H. Jongejan, P.G. Lucassen. We greatfully acknowledge the important contributions of one of the referees who gave detailed comments and insisted on a complete assertional proof. Among other things, he thus saved us from an embarrassing bug.

References

1. Apt KR, Olderog E-R: Verification of sequential and concurrent programs. Springer, New York, 1991
2. Chandy KM, Misra J: Parallel program design: a Foundation. Addison-Wesley, 1988
3. Fischer MJ, Lynch NA, Paterson MS: Impossibility of distributed consensus with one faulty process. J ACM 32: 374–382 (1985)
4. Herlihy MP: Impossibility and universality results for wait-free synchronization. In: Proc 7th Annual ACM Symposium on Principles of Distributed Computing, August 1988
5. Herlihy MP: Wait-free synchronization. ACM Trans Program Lang Syst 13: 124–149 (1991)
6. Herlihy MP: A methodology for implementing highly concurrent data structures. In: Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. SIGPLAN Notices 25(3) 197–206 (1990)
7. Herlihy MP, Wing J: Linearizability: a correctness condition for concurrent objects. ACM Trans Program Lang Syst 12: 563–492 (1990)
8. Hesselink WH: Wait-free linearization with a mechanical proof. Computing Science Notes Groningen CS 9306
9. Jonker JE: On-the-fly garbage collection for several mutators. Distrib Comput 5: 187–199 (1992)
10. Lamport L: The 'Hoare Logic' of concurrent programs. Acta Inf 14: 21–37 (1980)
11. Lamport L, Schneider F: The "Hoare Logic" of CSP, and all that. ACM Trans Program Lang Syst 6: 281–296 (1984)
12. Misra J: Loosely-coupled processes. In: Aarts EHL, Van Leeuwen J, Rem M (eds): Parallel architectures and languages Europe, vol 2. Lect Notes Comput Sci, vol 506 Springer, Berlin Heidelberg New York 1991, pp 1–26
13. Owicki S, Gries D: An axiomatic proof technique for parallel programs. Acta Inf 6: 319–340 (1976)
14. Plotkin SA: Sticky bits and universality of consensus. In: Proceedings of the 8th ACM Symposium on Principles of Distributed Computing 1989, pp 159–176