VU VRIJE UNIVERSITEIT AMSTERDAM

VU Research Portal

A queue based mutual exclusion algorithm

Aravind, A.; Hesselink, W.H.A.

published in
Acta Informatica
2009

DOI (link to publisher)
10.1007/s00236-008-0086-z

document version
Publisher's PDF, also known as Version of record

Link to publication in VU Research Portal

citation for published version (APA)
Aravind, A., & Hesselink, W. H. A. (2009). A queue based mutual exclusion algorithm. *Acta Informatica*, *46*(1), 73-86. https://doi.org/10.1007/s00236-008-0086-z

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

E-mail address:
vuresearchportal.ub@vu.nl

Download date: 13. Nov. 2024

Acta Informatica (2009) 46:73–86
DOI 10.1007/s00236-008-0086-z

ORIGINAL ARTICLE

A queue based mutual exclusion algorithm

Alex A. Aravind · Wim H. Hesselink

Received: 5 November 2007 / Revised: 5 November 2008 / Accepted: 11 November 2008 /
Published online: 2 December 2008
© Springer-Verlag 2008

Abstract A new elegant and simple algorithm for mutual exclusion of N processes is proposed. It only requires shared variables in a memory model where shared variables need not be accessed atomically. We prove mutual exclusion by reformulating the algorithm as a transition system (automaton), and applying simulation of automata. The proof has been verified with the higher-order interactive theorem prover PVS. Under an additional atomicity assumption, the algorithm is starvation free, and we conjecture that no competing process is passed by any other process more than once. This conjecture was verified by model checking for systems with at most five processes.

1 Introduction

Resolving conflicting access to shared memory by concurrent processes is a fundamental problem in distributed computing that goes back to [5]. Numerous solutions have been proposed with various characteristics to solve this problem. Surveys of these solutions can be found in [2,18,21]. Among them, Peterson's algorithm [17] and Lamport's Bakery algorithm [12] are popular for their simplicity and other remarkable properties. As a result, many variations have appeared in the literature for these two algorithms [1,3,11,21].

This paper proposes a simple and elegant algorithm for mutual exclusion for N processes. Our algorithm may be considered as a variation of Peterson's algorithm [17]. This algorithm and its variations given in [1,11] use $N-1$ stages for the processes to pass before accessing the shared resource. Our algorithm has the property that the number of stages to be passed by a

A. A. Aravind
Computer Science Program, University of Northern British Columbia,
Prince George, BC V2N4Z9, Canada
e-mail: csalex@unbc.ca

W. H. Hesselink (✉)
Department of Computing Science, University of Groningen,
P.O. Box 407, 9700 AK Groningen, The Netherlands
e-mail: w.h.hesselink@rug.nl
URL: http://www.cs.rug.nl/∼wim

process does not depend on N but only on the number of concurrently competing processes. The algorithm is approximately FIFO. We concentrate on the proof of mutual exclusion, which is verified mechanically. Progress is proved informally.

The setting is modeled as follows. There are $N \geq 2$ processes that communicate via shared variables and that repeatedly compete for a shared resource. The processes are thus of the form:

process *member*(p : *Process*) =
  loop
    NCS; *Intro*; *CS*; *Exit*
  end.

Here, *NCS* and *CS* are given program fragments that stand for the noncritical section and the critical section, respectively. *NCS* need not terminate, *CS* is guaranteed to terminate. The aim is to implement *Intro* and *Exit* in such a way that the number of processes in *CS* is guaranteed to remain ≤ 1 (*mutual exclusion*). The progress requirement is that, when some processes have entered *Intro*, eventually some process will enter *CS*. Freedom from starvation is the condition that, if some process, say p, has entered *Intro*, then p will eventually enter *CS*.

In the following sketch of the algorithm, the processes are children and the critical section is the eating of cake.

Sketch of the protocol as a children's party. Several (N) children are invited to a party. The party is held in a room with N corners, numbered from 0 upward. When a child enters the room, it goes to corner $N - 1$. A child in corner r that sees that there are not more than k children in the room other than itself, may move to corner k if $k < r$. Corner 0 has a table where the children eat a piece of cake. When a child has had its cake, it leaves the room to go to the garden from which it may return to the room to get more cake.

Each corner with a positive number has a chair one child can sit on. Any child in this corner can decide to climb onto the chair. If the chair is occupied and a child decides to climb onto it, it first pushes the present occupant from the chair. A child pushed from the chair in corner k goes to corner $k - 1$. The children in the room are greedy and want to reach the cake in corner 0. The organizers claim that there is always at most one child eating cake, and that every child in the room will eventually get cake.

Counting children takes time, but a child need not count children that entered after it started counting, and it may discount children that leave the room before it finished counting. When a child has pushed another from a chair, it must climb the chair, although another child may succeed earlier. After climbing a chair, a child that wants to count children needs to start from scratch.

Results and overview. We prove mutual exclusion of the algorithm by means of history variables and two abstraction steps with refinement functions. The proof has been verified with the proof assistant PVS [16]. Progress is fairly easy to prove. This is done informally. The above holds under weak atomicity conditions for the shared variables. Freedom from starvation does not hold under these conditions, but can be proved to hold under stronger atomicity conditions. Under these conditions, we even conjecture that every competing process is not passed more than once by any other process. We have verified this for $N \leq 5$ by means of the model checker Spin [10].

The rest of the paper is organized as follows. We present the algorithm in Sect. 2. In Sect. 3, we give a formal proof of mutual exclusion. In Sect. 4, we give an informal proof of progress and discuss our model checking results on individual starvation. Section 5 describes related works and concludes the paper.

2 A heuristic approach to the algorithm

In this section, we present the algorithm by approximating it by simpler versions. We start from scratch. So, the reader is asked to forget the children's party for the moment. The first version is a correct algorithm with a too coarse grain of atomicity. The second version has adequate atomicity, but gives deadlock. The third version is the correct one.

The starting point is the idea that every competing process should give priority to all processes that were competing at the moment this process started competing. This requires two shared variables: `act` to record the set of currently competing processes and an array with, for every process p, a set `prio`$[p]$ for the set of processes process p gives priority to:

`act` : **set of** *Process* := ∅;
`prio` : **array** [*Process*] **of set of** *Process*.

The commands *Intro* and *Exit* are given by

Intro0(p) =
  ⟨ `prio`$[p]$:= `act` ; *add* p *to* `act` ⟩;
  await `prio`$[p] = \emptyset$.

Here the brackets ⟨ ⟩ are used to indicate that the composition needs to be executed atomically.

Exit0(p) =
  for all q **do** *remove* p *from* `prio`$[q]$ **enddo**;
  remove p *from* `act`.

This algorithm is correct: it guarantees mutual exclusion and every competing process will enter the criticial section eventually. It is even FIFO: the processes enter the critical section in the order they start competing. Unfortunately, with current architectures, this algorithm is unimplementable because the big atomic action at the start of *Intro* is not offered by current hardware, even if the set `act` can be implemented as a single machine word.

We therefore decide to regard `act` as a boolean array indexed by process numbers, and to estimate `act` by private variables *est*, as declared in

est : **set of** *Process*.

Notice that we write private variables slanted whereas shared variables are in type writer font. If v is a private variable (for all processes), we write $v.p$ for the value of v of process p outside of the code for p.

For any process p, the value $est.p$ is the set of the processes q for which $\neg$ `act`$[q]$ has not yet been observed by p. The **await** command is implemented by busy waiting.

Intro1(p) =
  `act`$[p]$:= *true*; *est* := *Process* \ $\{p\}$;
  while *nonempty*(*est*) **do**
    for all $q \in est$ **do**
      if $\neg$ `act`$[q]$ **then** *remove* q *from est* **endif**
    enddo
  enddo.

Now *Exit0* can be simplified to

Exit1(p) = (`act`$[p]$:= *false*).

Unfortunately, this version leads to deadlock when processes enter one after the other, and the first one has not yet observed that the second one was not competing.

We therefore introduce a private integer variable *level* as a counter to see whether the inner loop has decreased the number of elements of *est*. When deadlock would occur, several processes would have the same *level*. In order to enforce progress at that point, some of these should be allowed to decrease their *level*.

```
Intro2(p) =
    act[p] := true; level := N − 1;
    est := Process \ {p};
    while level > 0 do
        for all q ∈ est do
            if ¬ act[q] then remove q from est endif
        enddo ;
        ReconsiderLevel
    enddo.
```

We introduce a shared variable `turn` so that `turn`[k] serves to enable one of the processes to decrease its *level* below k. A process that cannot decrease its *level* beyond k may try to enter `turn`[k]. We thus declare:

`turn` : **array** [1 . . $N - 1$] **of** *Process*.

We give every process a private boolean variable *bb*, initially false, to indicate that it has entered `turn`[*level*]. When *bb* holds and the process has been removed from `turn`[*level*], it can decrease its *level*. We thus get:

```
ReconsiderLevel :
    if #est < level then
        level := #est ; bb := false
    elsif ¬ bb then
        turn[level] := p ; bb := true;
        est := Process \ {p};
    elsif turn[level] ≠ p then
        level− − ; bb := false
    else skip endif.
```

ReconsiderLevel tries to decrease *level*. We use #S to denote the number of elements of a set S. In the first alternative, *level* becomes the number of observed competitors. In the second alternative, the process enters the `turn` of its *level*, sets the flag *bb* to remember this, and resets *est*. In the third alternative, when it has set flag *bb* and yet differs from `turn`, it is allowed to decrement *level*. Variable *bb* can be eliminated at the cost of code duplication.

The second alternative, called *push*, allows the process that was equal to `turn`[*level*] to decrement its *level* in the third alternative. Resetting of *est* is needed to avoid too many processes with low *level*s. Note however that a process that has set *bb* to true, does recalculate *est*, and will decrease *level* when it finds #*est* < *level*.

We take *Exit2* = *Exit1*. This algorithm turns out to be correct. Looking back to the children's party of the Introduction, the reader may recognize *level* as the number of the child's corner and `turn`[k] as the chair in corner k.

Mutual exclusion is expressed by the required invariant *MX* defined by

$$MX \quad \equiv \quad \#crit \leq 1, \tag{0}$$

where *crit* is the set of processes that are at command *CS*. The idea of the proof is to generalize *MX* to an invariant

$$J(k): \quad \#A_0(k) \leq k,$$

for all natural numbers k, where $A_0(k)$ is the set of processes q that are in *Intro* or *CS* and have $level.q < k$. Unfortunately, this invariant is difficult to prove. The problem is the nonatomic counting of competitors. We shall solve this by introducing a history variable that holds a lower bound for the set of observed competitors. In Sect. 3.5, we define containing sets $A(k)$, for which the analog of $J(k)$ can be proved.

3 Nonatomic variables and correctness

The presentation of the algorithm implicitly suggests that the shared variables `act`$[p]$ and `turn`$[k]$ can be written and read atomically. The algorithm allows us, however, to weaken this assumption. This is important because concurrent hardware often guarantees atomicity of variables only under specific assumptions. The algorithm remains correct when the variables `act`$[p]$ are *safe* in the sense of [13]. It even remains correct when the variable `turn`$[k]$ are *write-safe*. We shall introduce and formalize these concepts in Sect. 3.1.

The remainder of the section is devoted to the proof of mutual exclusion. In Sect. 3.2, we reformulate the algorithm as a transition system or automaton *QmxC*. We treat the variables `act`$[p]$ as safe, and the variables `turn`$[k]$ as write-safe. In Sect. 3.3, we reduce the *concrete* automaton *QmxC* to an *abstract* automaton *QmxA*, by a refinement function from *QmxC* to *QmxA*. In Sect. 3.4, automaton *QmxA* is extended with a *history variable* to *QmxH*, by means of a forward simulation. Then we use a refinement function from *QmxH* to an *idealized* automaton *QmxI* to remove the variables `act` and *est* and the associated actions.

$$QmxC \rightarrow QmxA \rightarrowtriangle QmxH \rightarrow QmxI.$$

Mutual exclusion in *QmxI* is proved in Sect. 3.5. In Sect. 3.6, we use the composed simulation from *QmxC* to *QmxI* to conclude that *QmxC* also satisfies mutual exclusion, and we briefly describe how the proof assistant PVS is used to verify this proof.

3.1 Safe and write-safe shared variables

Let us call a shared variable an *output variable* (of process p) if it is only written by a single process (p). Recall from [13] that an output variable is called *safe* if every read action that does not overlap with any write action returns the most recently written value and every read action that overlaps with a write action returns some (arbitrary) value of the correct type.

In our algorithm, the variables `act`$[p]$ are output variables, and we shall prove that the algorithm is correct under the assumption that these variables are safe. The shared variables `turn`$[k]$ are not output variables: different processes may concurrently write them. Safety of these variables is therefore not applicable, but there is a relevant related assumption.

Let a shared variable `x` be called *write-safe*, if concurrent writing to `x` is allowed and has the effect that `x` gets a value that was being written by (at least) one of the writing processes, a read action not concurrent with any write actions gets the value written latest, and a read action concurrent with one or more write actions yields an arbitrary value of the correct type.

We formalize safe output variables and write-safe shared variables in the same way, as follows. A read action of a shared variable `x` to a private variable v is denoted $v :=$ `x`. It can

be regarded as atomic, since it does not influence the shared state. We only need to reckon with the possibility that it overlaps with one or more write actions to x.

A write action of a private expression E to a safe output variable or write-safe shared variable x is denoted

$$\text{(flickering) } \mathrm{x} := E. \tag{1}$$

We model this in relational semantics, such as TLA [14], by specifying that command (1) has the relational meaning

$$pc^+ = pc \;\; \vee \;\; (pc^+ = pc + 1 \;\wedge\; \mathrm{x}^+ = E). \tag{2}$$

Here the superscript $+$ is used for the values of the variables after the step, *pc* stands for the location pointer, and by convention all shared or private variables apart from x or *pc* are unchanged. In other words, command (1) is modeled as a repetition of arbitrary assignments to x that ends with the actual assignment of E to x. The value of x during the repetition is indeterminate. Liveness conditions are used to ensure that the repetition terminates, cf. [8].

We often combine an action on a shared variable atomically with some action S on private variables. In the case of a flickering write action, we need to formalize that command S is done precisely once. In terms of goto commands, therefore,

$$\ell_0 : \quad \text{(flickering)} \mathrm{x} := E \;;\; S \;;\; \textbf{goto } \ell_1 \tag{3}$$

is modeled as the nondeterministic choice

$$\begin{aligned} \ell_0 : \quad & \mathrm{x} := \text{arbitrary} \;;\; \textbf{goto } \ell_0 \\ [\!] \quad & \mathrm{x} := E \;;\; S \;;\; \textbf{goto } \ell_1. \end{aligned} \tag{4}$$

3.2 Formalization as a transition system

To prove mutual exclusion in our algorithm, we first formalize it as a transition system, i.e., we reformulate the algorithm into a **goto** program with numbered atomic statements that each refer to at most one shared variable. We assume that the output variables act[p] are safe and that the variables turn[k] are write-safe.

We do not want to enforce an order of treating the processes in the inner **for** loop of *Intro2*, but only that all elements of *est* are treated once. We therefore introduce, for each process, a private variable

lis : **set of** *Process*

to hold the set of processes not yet treated in the inner **for** loop. We thus get:

cmember(p) =
10: *NCS*(p) ; **goto** 10 **or** 20.
20: (flickering) act[p] := *true*;
level := $N - 1$; *est* := *Process* \ {p} ; **goto** 21.
21: **if** *level* > 0 **then** *lis* := *est* ; **goto** 22
else goto 30 **endif**.
22: **if** *nonempty*(*lis*) **then**
extract some q from lis;
if ¬ act[q] **then** *remove q from est* **endif**;
goto 22
else goto 23 **endif**.

23: **if** #*est* < *level* **then** *level* := #*est* ; *bb* := *false*
elsif ¬ *bb* **then goto** 24
elsif `turn`[*level*] ≠ *p* **then** *level*− − ; *bb* := *false*
else skip endif ;
goto 21.
24: (flickering) `turn`[*level*] := *p*;
bb := *true* ; *est* := *Process* \ {*p*} ; **goto** 21.
30: *CS*(*p*) ; **goto** 40.
40: (flickering) `act`[*p*] := *false* ; **goto** 10.

The nondeterministic choice in 10 expresses that *NCS* need not terminate. *Intro2* is modeled by commands 20 up to 24. The inner **for** loop is at 22. *ReconsiderLevel* is captured by 23 and 24. We need a separate location 24 for the process to return to while it is doing incomplete assignments to `turn`. The initial state is characterized by

$$\forall\, q : pc.q = 10 \ \wedge\ \neg\, \texttt{act}[q] \ \wedge\ \neg\, bb.q,$$

while the values of `turn`, *level*, *est*, and *lis* can be arbitrary.

The algorithm is formalized in the parallel composition

$$QmxC \ = \ ||_p\, cmember(p).$$

The state space *XC* of algorithm *QmxC* is spanned by the shared variables `act` and `turn`, and the private variables *level*, *lis*, *est*, *bb*, and *pc*, where *pc*.*q* is the location counter of process *q*.

Mutual exclusion for *QmxC* is expressed by *MX* as defined in (0) where *crit* is the set of processes *q* with *pc*.*q* = 30. We postpone the proof of *MX*, but first establish the auxiliary invariants:

I0: $22 \le pc.q < 30 \ \Rightarrow\ level.q > 0,$
I1: $pc.q = 30 \ \Rightarrow\ level.q = 0,$
I2: $pc.q > 20 \ \Rightarrow\ level.q \ge 0.$

The proof of invariance of *I0* is trivial. The proof of invariance of *I1* needs *I2* at step 21. The proof of invariance of *I2* needs *I0* at the third alternative of step 23.

Remark on atomicity As one of the referees noted, some of the instructions are encoded as rather complicated conditional statements that are to be evaluated atomically. In concurrent algorithms, separating a test of a condition from an assignment can give rise to critical race conditions. Here this is not the case, because in every instruction there is at most one reference (read or write) to a shared variable, viz. to `turn`[*level*] in 23 and 24, and to a field of `act` in 20, 22, and 40. It is well-known that inspection or modification of private variables can be included in such a command. One way to justify this is by introducing ghost variables as aliases of the private variables. These ghost variables are then inspected or modified in the atomic command, while the inspection or modification of the actual private variables is done in a completely private command just before or after the atomic command. One then introduces and proves additional invariants asserting that the private variables are equal to their ghost aliases at all relevant locations.

The same argument applies to the guarded commands in Sect. 3.3. In Sect. 3.4, we introduce shared history variables `lwb` that are inspected or modified concurrently with `act` or `turn`. This is allowed because `lwb` is a ghost variable that only serves in the proof, not in the algorithm.

3.3 The abstract algorithm

For the ease of the proof of *MX*, we eliminate the locations and the variable *lis*. The resulting algorithm is much more nondeterministic. It may be regarded as a UNITY program, see [4]. To indicate process p's status as noncompeting, we make $level.p$ negative when process p is in *Exit* or *NCS*. The resulting abstract algorithm is

$$QmxA \;=\; ||_p \; amember(p),$$

where $amember(p)$ is defined as the repeated nondeterministic choice:

$$\begin{array}{l} amember(p) = \\ \quad (\; entry(p) \;[\!]\; flickerAct(p) \;[\!]\; discard(p) \;[\!] \\ \quad\; move(p) \;[\!]\; toPush(p) \;[\!]\; flickerTurn(p) \;[\!] \\ \quad\; push(p) \;[\!]\; wait(p) \;[\!]\; exit(p) \;[\!]\; \mathbf{skip}\;)^{\infty}, \end{array}$$

where the atomic commands *entry* up to *exit* are given below. We remove *NCS* and *CS* as irrelevant and express mutual exclusion $MX{:}\ \#crit \leq 1$ as in (0) with $crit = \{q \mid level.q = 0\}$.

We remove the program counters but introduce private variables *cc* (climbing chair) to indicate that the process is at 24. The state space *XA* of algorithm *QmxA* is spanned by the shared variables `act` and `turn`, and the private variables *level*, *est*, *bb*, and *cc*. The initial state is characterized by:

$$\forall\, q : level.q < 0 \;\wedge\; \neg\, \mathtt{act}[q] \;\wedge\; \neg\, bb.q \;\wedge\; \neg\, cc.q\,.$$

The atomic commands *entry* up to *exit* are defined as follows. The main part of command 20 is matched by:

$$\begin{array}{l} entry(p) = \\ \quad level < 0 \;\rightarrow \\ \qquad \mathtt{act}[p] := true \;;\; level := N - 1; \\ \qquad est := Process \setminus \{p\}. \end{array}$$

The flickering assignments to $\mathtt{act}[p]$ in 20 and 40 are matched by:

$$\begin{array}{l} flickerAct(p) = \\ \quad level < 0 \;\rightarrow\; \mathtt{act}[p] := \text{arbitrary}. \end{array}$$

Here, we allow $\mathtt{act}[p]$ to remain flickering during the noncritical section of p because we concentrate on safety of the algorithm. For progress, we need $\mathtt{act}[p]$ to become stably false during the noncritical section.

The removal of q from $est.p$ in line 22 is matched by:

$$\begin{array}{l} discard(p) = \\ \quad level > 0 \;\rightarrow \\ \qquad extract\ if\ possible\ some\ q\ from\ est\ with\ \neg\, \mathtt{act}[q]. \end{array}$$

The first alternative of 23 is matched by:

$$\begin{array}{l} move(p) = \\ \quad \#est < level \;\wedge\; \neg cc \;\rightarrow \\ \qquad level := \#est \;;\; bb := false. \end{array}$$

The second alternative of 23 is matched by:

$$\begin{array}{l} toPush(p) = \\ \quad level > 0 \;\wedge\; \neg\, bb \;\wedge\; \neg\, cc \;\rightarrow \\ \qquad cc := true. \end{array}$$

The third alternative of 23 is matched by:

$$\begin{array}{l} \mathit{wait}(p) = \\ \quad \mathit{level} > 0 \ \wedge \ \mathit{bb} \ \wedge \ \mathtt{turn}[\mathit{level}] \neq p \ \ \rightarrow \\ \quad\quad \mathit{level}{-}\ {-}\ ; \ \mathit{bb} := \mathit{false}. \end{array}$$

The main action at line 24 is matched by:

$$\begin{array}{l} \mathit{push}(p) = \\ \quad \mathit{level} > 0 \ \wedge \ \mathit{cc} \ \ \rightarrow \\ \quad\quad \mathtt{turn}[\mathit{level}] := p \ ; \ \mathit{bb} := \mathit{true}; \\ \quad\quad \mathit{cc} := \mathit{false} \ ; \ \mathit{est} := \mathit{Process} \setminus \{p\}. \end{array}$$

The flickering at 24 is matched by:

$$\begin{array}{l} \mathit{flickerTurn}(p) = \\ \quad \mathit{level} > 0 \ \wedge \ \mathit{cc} \ \ \rightarrow \\ \quad\quad \mathtt{turn}[\mathit{level}] := \mathit{arbitrary}. \end{array}$$

Command 30 is matched by:

$$\begin{array}{l} \mathit{exit}(p) = \\ \quad \mathit{level} = 0 \ \ \rightarrow \ \ \mathit{level} := -1. \end{array}$$

In order to prove that the goto program implements this abstract algorithm, we form a refinement function [9], i.e., a function $f : XC \rightarrow XA$ such that every initial state of *QmxC* is mapped to an initial state of *QmxA* and that, for every step (x, y) of algorithm *QmxC*, the pair $(f(x), f(y))$ is a step of *QmxA*. Function f removes the private variables *lis* and *pc*. It gives $\mathit{cc}.p$ the meaning that p is at 24, and it makes $\mathit{level}.p < 0$ when $\mathit{pc}.p \notin \{21 \ldots 30\}$:

$$\begin{array}{l} f(x) = (\# \\ \quad \mathtt{act} := x.\mathtt{act}, \mathtt{turn} := x.\mathtt{turn}, \\ \quad \mathit{level} := (\lambda\, q : \textbf{if } 21 \leq x.\mathit{pc}.q \leq 30 \\ \qquad\qquad\qquad \textbf{then } \ x.\mathit{level}.q \ \ \textbf{else } \ -\,1 \ \textbf{endif}), \\ \quad \mathit{est} := x.\mathit{est}, \mathit{bb} := x.\mathit{bb}, \\ \quad \mathit{cc} := (\lambda\, q : x.\mathit{pc}.q = 24) \ \ \#). \end{array}$$

The brackets (# and #) are record constructors, as used in PVS. We use $x.\mathtt{act}$ to refer to the field `act` of state $x \in XC$, and $x.\mathit{level}.q$ for the value of *level* of process q in state x (etc.). The identifiers *level* and *cc* are private variables and can therefore be treated as functions.

We next show that every step of the concrete algorithm *QmxC* is matched by a step of the abstract algorithm *QmxA*.

Step 10 of *QmxC* corresponds to a **skip** step of *QmxA*. Step 20 of *QmxC* corresponds to *entry* or *flickerAct* of *QmxA*. Step 21 of *QmxC* corresponds to **skip**. Step 22 of *QmxC* corresponds to **skip** or *discard*. The first, second, and third alternative of step 23 correspond to steps *move*, *toPush*, and *wait*, respectively. In the cases of *discard*, *move*, *toPush*, and *wait*, we use invariant *I0* to ascertain $\mathit{level}.p > 0$. The fourth alternative of step 23 corresponds to **skip**. The steps at 24 are matched by *flickerTurn* and *push* because of *I0*. Step 30 of *QmxC* corresponds to step *exit* because of *I1*. Step 40 of *QmxC* corresponds to step *flickerAct* or to **skip**.

3.4 Extending with a history variable

In order to prove mutual exclusion, we extend algorithm *QmxA* with a shared history variable `lwb` that, for each process q, records the competing processes active since the latest execution of *entry*(q) or *push*(q):

$$\mathtt{lwb} : \textbf{array } [\mathit{Process}] \textbf{ of set of } \mathit{Process} := (\lambda\, q : \emptyset).$$

Such a history variable does not influence the computation, but is used only in the proof of correctness. It is therefore allowed to include inspection and modification of `lwb` in the atomic commands of the previous section. The reason for the name `lwb` is that the set `lwb`$[q]$ will serve as a lower bound for the private variable $est.q$. Array `lwb` will play the role of array `prio` in version 0 of the algorithm in Sect. 2.

We define the set of competitors by $Cp = \{q \mid level.q \geq 0\}$. This set is used to update `lwb`$[p]$ in *entry* and *push*. When process p exits, p is removed from `lwb`$[q]$ for all q. This can be done in the atomic step *exit* because `lwb` is only a history variable. We thus get:

$$
\begin{array}{l}
entry(p) = \\
\quad level < 0 \;\; \rightarrow \\
\qquad \texttt{lwb}[p] := Cp; \; \texttt{act}[p] := true; \\
\qquad level := N - 1; \; est := Process \setminus \{p\}.
\end{array}
$$

$$
\begin{array}{l}
push(p) = \\
\quad level > 0 \;\wedge\; cc \;\; \rightarrow \\
\qquad \texttt{turn}[level] := p \;;\; bb := true \;;\; cc := false; \\
\qquad est := Process \setminus \{p\}; \; \texttt{lwb}[p] := Cp \setminus \{p\}.
\end{array}
$$

$$
\begin{array}{l}
exit(p) = \\
\quad level = 0 \;\; \rightarrow \\
\qquad \textbf{for all } q \textbf{ do } remove \; p \; from \; \texttt{lwb}[q] \; \textbf{enddo}; \\
\qquad level := -1.
\end{array}
$$

The other actions are lifted to the new state space without modification. The new state space *XH* is the state space *XA* extended with `lwb`. This concludes the description of the extended algorithm *QmxH*.

The relevance of `lwb` is expressed by the invariants:

K0: $\texttt{lwb}[q] \subseteq est.q$,
K1: $\texttt{lwb}[q] \subseteq Cp \setminus \{q\}$.

In order to prove these invariants, we also note the invariant:

K2: $level.q \geq 0 \;\Rightarrow\; \texttt{act}[q]$.

The proof of *K0* uses *K1* and *K2* in *discard*. The proofs of *K1* and *K2* are straightforward.

The next step is to realize that, at this point, the sets *est* are essentially superfluous. The invariant *K0* allows us to replace *move* by the nondeterministic version

$$
\begin{array}{l}
moveND(p) = \\
\quad \#\texttt{lwb}[p] < level \;\wedge\; \neg\, cc \;\; \rightarrow \\
\qquad choose \; some \; m \; with \; \#\texttt{lwb}[p] \leq m < level; \\
\qquad level := m \;;\; bb := false.
\end{array}
$$

Note that $move(p)$ corresponds to $moveND(p)$ with $m = \#est.p$.

Now the private variables *est* and the modifications of them in *entry* and *push* can be removed. Therefore, the actions *discard* can be replaced by **skip**. Consequently, the shared variable `act` can be removed and *flickerAct* can be replaced by **skip**.

Let *XI* be the state space spanned by the shared variables `turn` and `lwb`, and the private variables *level*, *bb*, and *cc*. Let *QmxI* be the (idealized) algorithm determined by the actions *entry*, *moveND*, *toPush*, *wait*, *push*, *flickerTurn*, and *exit*. Then the function $g : XH \rightarrow XI$ defined by removing *est* and `act` is a refinement function from *QmxH* to *QmxI* because of invariant *K0* used for *move*.

3.5 Proof of mutual exclusion in *QmxI*

We now prove mutual exclusion for the algorithm *QmxI*, as expressed by MX: $\#crit \leq 1$ with $crit = \{q \mid level.q = 0\}$.

As announced in Sect. 2, the starting point is the idea that the set $A_0(k) = \{q \in Cp \mid level.q < k\}$ should always have at most k elements. This property, however, can easily be falsified by $moveND(p)$ when $cc.p$ is false. We therefore adapt the definition to reckon with this possibility.

For every natural number k, we define the set of processes

$$A(k) \;=\; \{q \in Cp \mid level.q < k \;\vee\; (\neg\, cc.q \;\wedge\; \#\mathtt{lwb}[q] < k)\}.$$

Since $A(1)$ contains the set $crit$, mutual exclusion MX is implied by the invariant $J0(1)$ where for any $k \in \mathbb{N}$ we define the invariant

$$J0(k): \quad \#A(k) \leq k.$$

This invariant is threatened by the action *wait* that decrements *level*. A process p that executes *wait*, has executed *push*, and then some other process, say q, has established $\mathtt{turn}[level.p] \neq p$ by executing *push* or *flickerTurn*. This process q has the same *level* as p and satisfies $cc.q \vee bb.q$. In order to account for such processes, we form the slightly bigger set:

$$\begin{aligned} B(k) \;=\; & \{q \in Cp \mid level.q < k + |cc.q \vee bb.q| \\ & \vee\; (\neg\, cc.q \;\wedge\; \#\mathtt{lwb}[q] < k)\}, \end{aligned}$$

where, for any boolean value b, we define $|b| = 1$ or 0 when b is true or false, respectively. We now propose the additional invariants

$$J1(k): \quad \#B(k) \leq k + |T(k)|,$$

where $T(k)$ is defined by

$$\begin{aligned} T(k) \;\equiv\; & (\exists\, q : level.q = k \\ & \wedge\; (cc.q \;\vee\; (\mathtt{turn}[k] = q \wedge bb.q \wedge k \leq \#\mathtt{lwb}[q]))). \end{aligned}$$

$T(k)$ expresses that there are processes with $level = k$ that have executed *toPush* at that level and that cannot escape from this state by *moveND*.

We clearly have $A(k) \subseteq B(k)$. If q is a witness of $T(k)$ then $q \in B(k) \setminus A(k)$. Therefore, $J1(k)$ implies $J0(k)$.

In order to prove the invariants $J1(k)$, we first note the easy invariants:

$$\begin{aligned} &L0: && \forall\, q :\; \neg\, bb.q \;\vee\; \neg\, cc.q, \\ &L1: && \forall\, q :\; bb.q \vee cc.q \;\Rightarrow\; level.q > 0. \end{aligned}$$

We now prove that the universal quantification $(\forall\, k \in \mathbb{N} : J1(k))$ is preserved by every step. For any state expression E, let E^+ represent the value after the step and E represent the value before the step, just as in Sect. 3.1. We treat the different steps of *QmxI* one by one.

Firstly, for *moveND* and *flickerTurn*, it is not very difficult to see that $B(k)^+ = B(k)$ and that $T(k)$ implies $T(k)^+$. The same result applies to *wait* where we need the invariant $L0$. This clearly implies preservation of $J1(k)$ for *moveND*, *flickerTurn*, and *wait*.

For *push* and *entry*, we distinguish the cases $\#Cp^+ \leq k$ and $k < \#Cp^+$. In the first case, the postcondition $\#B(k)^+ \leq k$ is obvious. In the second case, we have that $B(k)^+ = B(k)$ and that $T(k)$ implies $T(k)^+$ as before. Whence preservation of $J1(k)$. The invariant $L1$ is needed for *entry*.

For *exit*(p), we have $B(k)^+ \subseteq A(k+1) \setminus \{p\}$. Therefore $J1(k)^+$ follows from $J0(k+1)$, and hence from $J1(k+1)$.

For *toPush*(p) with $k \neq level.p$, we have that $B(k)^+ \subseteq B(k)$ and that $T(k)$ implies $T(k)^+$ as before. Whence preservation of $J1(k)$. For *toPush*(p) with $k = level.p$, we have that $B(k)^+ \subseteq A(k+1)$ and p itself is a witness of $T(k)^+$. Therefore $J1(k)^+$ follows from $J0(k+1)$, and hence from $J1(k+1)$.

All this together implies that the universal quantification $(\forall\ k \in \mathbb{N} : J1(k))$ is preserved by every step. Because it holds initially, this concludes the proof that $(\forall\ k \in \mathbb{N} : J1(k))$ is an invariant of *QmxI*. It follows that mutual exclusion (*MX*) is also an invariant of *QmxI*.

3.6 Theorem proving and formal conclusion

We have used the proof assistant PVS of [16] to mechanically verify the steps from algorithm *QmxC* of Sect. 3.2 to mutual exclusion in *QmxI* of Sect. 3.5. In this mechanical proof, we indeed construct refinement functions from *QmxC* to *QmxA* and from *QmxH* to *QmxI*, cf. [9], and we show that the extension of *QmxA* with the history variable `lwb` to *QmxH* is a so-called forward simulation, cf. [7,9,15]. We therefore have a composition which is a simulation from *QmxC* to *QmxI*. All this also requires the verification of the invariants *I** and *K**.

In the automaton of *QmxI*, we verify the invariants *L0* and *L1*, and then the more complicated invariants *J0* and *J1*, as described in Sect. 3.5. We thus prove that *QmxI* satisfies *MX*. Because *QmxI* satisfies the invariant *MX*, this invariant can be traced backward to *QmxC* to yield that there is always at most one process q with $pc.q = 30$ and $level.q = 0$. Finally, the invariant *I1* therefore implies that there is always at most one process q with $pc.q = 30$. This proves mutual exclusion for *QmxC*. The PVS proof script is available at http://www.cs.rug.nl/~wim/mechver/queueMX.

4 Progress and bounded overtaking

The proof of progress is relatively straightforward. That is, whenever some processes have entered *Intro*, eventually some processes will enter *CS*. Suppose that this is not the case. Then we have an infinite execution of the system, in which eventually $k > 0$ processes remain in *Intro* and no process enters *CS* anymore. Since *CS* and *Exit* are terminating commands, we have eventually all other $(N - k)$ processes in *NCS*.

The k processes q in *Intro* repeatedly compute $est.q$ (or `lwb`$[q]$) and eventually always find $\#est.q < k$ (or $\#$`lwb`$[q] < k$). Therefore, eventually, they all get $level.q < k$. They keep $level.q > 0$ because they remain in *Intro*. They all try and set `turn`$[level.q] := q$ and set $bb.q := true$. Since there are only $k-1$ elements `turn`$[i]$ with $0 < i < k$, at least one of them will eventually be enabled to set $level.q := 0$ and exit *Intro*. This proves the progress property.

4.1 Bounded overtaking

In an execution of the system, let us call a *competing period* of process p a period that starts with *entry*(p) and ends with *exit*(p). *Bounded overtaking* is the property that there is some number k such that, in every execution, for any pair of processes p and q, any competing period of p contains not more than k competing periods of q.

Under the assumption that the variables `turn` are only write-safe, bounded overtaking is not valid. This is shown by the following scenario, found by model checking. Let p and q be two processes. Process p enters first. Then q enters, sets $level.q := 1$, and executes

toPush. Then process p enters *CS*, and exits. At this point, an infinite cycle starts: process p enters and executes *push* with $level.p = 1$; then q executes *flickerTurn* so that p can set $level.p := 0$; consequently, p can enter *CS* and then exit. This is the end of the cycle. In this infinite loop p passes q infinitely often. Of course, the write-safeness of `turn` implies that process q executes *flickerTurn* only finitely often before it executes *push*, but no upper bound is specified. We can therefore not find an upper bound for the number of times p passes q.

This scenario depends on the fact that writing periods for different processes on a write-safe variable can overlap. If it is somehow guaranteed that such writing periods are always disjoint (in particular, when the variables `turn`$[k]$ are atomic), we conjecture that bounded overtaking with $k = 1$ holds, i.e., that every competing period of any process contains at most one competing period of any other process.

The reason for this conjecture is as follows. When p is competing and process q enters, q can indeed pass p when p executes *push* and therewith refreshes `lwb`$[p]$. For q to reach *CS*, however, it seems that all j processes passed by q need to line up in `turn`$[1]$ up to `turn`$[j]$ in such a way that they cannot be passed again by newcoming processes.

The conjecture is justified by model checking it with $N \leq 5$ processes. Indeed, we model checked algorithm *QmxI* of Sect. 3.4 in which the actions *toPush* and *push* are combined in a single atomic action in accordance with atomic writing of `turn`. We used the model checker Spin of [10], which created a model with 0.92×10^8 states and 7.96×10^8 transitions, using 10.5 GByte memory. The case of nonatomic writing of `turn` with mutual exclusion of the writers leads to bigger models, which could only be checked for $N \leq 4$ processes.

The conjecture clearly implies freedom from starvation. For, once p starts competing, there cannot be more than $2N - 2$ processes that exit before p itself exits (at most $N - 1$ of them entered after p).

The conjecture can be formalized by introducing history variables $\mathtt{a}[q, r]$ for the number of times process q entered while process $r \in Cp$. In $entry(p)$, $\mathtt{a}[p, q]$ is incremented for all $q \in Cp$. In $exit(p)$, $\mathtt{a}[q, p]$ is reset to 0 for all q. Now the conjecture amounts to invariant validity of $\mathtt{a}[q, r] \leq 2$, for all q and r. We are unable to prove this invariant.

Recently, we found a proof of freedom from starvation in the case that the variables `turn`$[k]$ are atomic. Specifically, we can prove that, once some process p starts competing, there cannot be more than N^2 processes that exit before p itself exits. We leave this to future work because the proof is not appealing and the bound is still unsatisfactory.

5 Related work and conclusion

Our algorithm may be considered as a variant of Peterson's algorithm [17]. Another variant presented by Block and Woo has some similarity in the shared space usage and the number of *level*s crossed by the competing processes. In Block-Woo's algorithm, the number of *level*s to be crossed increases whenever a new process starts its competition for CS. In our algorithm, it is determined in the beginning and does not change due to future contention for CS. Also, the number of bypasses over a process in accessing CS is high in Peterson's algorithm and its variations (unbounded for the algorithms given in [11,17] and $(N(N-1)/2)$ for the algorithm given in [3]). These algorithms require the shared variable `turn` to be atomic.

The Bakery algorithm [12] has the attractive property that it works with safe variables, but it requires unbounded shared space. There are many attempts to bound the token numbers [21], but they all compromise the nonatomicity property [21]. Our algorithm assures fairness similar to the Bakery algorithm. Peterson's algorithm and the Bakery algorithm are widely touted for their simplicity and elegance. The algorithm presented in this paper is quite simple

and elegant. Also, our algorithm uses bounded shared registers and seems to assure high fairness. We have to leave the conjecture of Sect. 4.1 to future work or as a challenge to the reader.

Multi-port memories allow concurrent accesses to memory through multiple ports. Such weaker memories are increasingly used in smart-phones, multi-mode handsets, multiprocessor systems, network processors, graphics chips, and other high performance electronic devices [6,19,20]. The applicability of our algorithm for weaker memory with safe and write-safe variables increases its practical value for systems with multi-port memories.

References

1. Alagarsamy, K.: A mutual exclusion algorithm with optimally bounded bypasses. Inf. Process. Lett. **96**, 36–40 (2005)
2. Anderson, J.H., Kim, Y.J., Herman, T.: Shared-memory mutual exclusion: major research trends since 1986. Distr. Comput. **16**, 75–110 (2003)
3. Block, K., Woo, T.-K.: A more efficient generalization of Peterson's mutual exclusion algorithm. Inf. Process. Lett. **35**, 219–222 (1990)
4. Chandy, K.M., Misra, J.: Parallel Program Design, A Foundation. Addison-Wesley, Reading (1988)
5. Dijkstra, E.W.: Solution of a problem in concurrent programming control. Commun. ACM **8**, 569 (1965)
6. Frenzel, L.E.: Dual-port SRAM accelerates smart-phone development. Electron. Des. (2004)
7. He, J., Hoare, C.A.R., Sanders, J.W.: Data refinement refined. In: Robinet, B., Wilhelm, R. (eds.) ESOP 86. Lecture notes in computer science, vol. 213, pp. 187–196. Springer, New York (1986)
8. Hesselink, W.H.: An assertional proof for a construction of an atomic variable. Formal Asp. Comput. **16**, 387–393 (2004)
9. Hesselink, W.H.: A criterion for atomicity revisited. Acta Inf. **44**, 123–151 (2007)
10. Holzmann, G.J.: The SPIN Model Checker, Primer and Reference Manual. Addison-Wesley, Reading (2004)
11. Igarashi, Y., Nishitani, Y.: Speedup of the n-process mutual exclusion algorithm. Parallel Process. Lett. **9**, 475–485 (1999)
12. Lamport, L.: A new solution of Dijkstra's concurrent programming problem. Commun. ACM **17**, 453–455 (1974)
13. Lamport, L.: On interprocess communication. Parts I and II. Distr. Comput. **1**, 77–101 (1986)
14. Lamport, L.: The temporal logic of actions. ACM Trans. Program. Lang. Syst. **16**, 872–923 (1994)
15. Milner, R.: An algebraic definition of simulation between programs. In: Proc. 2nd Int. Joint Conf. on Artificial Intelligence, pp. 481–489. British Computer Society, UK (1971)
16. Owre, S., Shankar, N., Rushby, J.M., Stringer-Calvert, D.W.J.: PVS Version 2.4, System Guide, Prover Guide, PVS Language Reference (2001). http://pvs.csl.sri.com
17. Peterson, G.L.: Myths about the mutual exclusion problem. Inf. Process. Lett. **12**, 115–116 (1981)
18. Raynal, M.: Algorithms for Mutual Exclusion. MIT Press, Cambridge (1986)
19. Shiue, W.-T., Chakrabarti, C.: Multi-module multi-port memory design for low power embedded systems. Des. Autom. Embed. Syst. **9**, 235–261 (2004)
20. Springer, C.: Enabling multimode handsets. EE Times, Oct. 2004
21. Taubenfeld, G.: Synchronization Algorithms and Concurrent Programming. Pearson Education/Prentice Hall, Englewood Cliffs (2006)