Science of Computer Programming 76 (2011) 542–554

Contents lists available at ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

Queue based mutual exclusion with linearly bounded overtaking

Wim H. Hesselink [a,*], Alex A. Aravind [b]

[a] *Department of Computing Science, University of Groningen, P.O. Box 407, 9700 AK Groningen, The Netherlands*

[b] *Computer Science Program, University of Northern British Columbia, Prince George, BC, Canada, V2N4Z9*

ARTICLE INFO

Article history:
Received 11 June 2009
Received in revised form 6 April 2010
Accepted 7 November 2010
Available online 26 November 2010

Keywords:
Mutual exclusion
Refinement
Shared memory
Bounded overtaking
Verification

ABSTRACT

The queue based mutual exclusion protocol establishes mutual exclusion for $N > 1$ threads by means of not necessarily atomic variables. In order to enter the critical section, a competing thread needs to traverse as many levels as there are currently competing threads. Competing threads can be overtaken by other competing threads. It is proved here, however, that every competing thread is overtaken less than N times, and that the overtaking threads were competing when the first one of them exits.

1. Introduction

Resolving access conflicts to shared resources by concurrent threads is a fundamental problem in distributed computing that goes back to [7]. Many solutions to this problem have been proposed. Surveys can be found in [3,15,16]. In particular, the solutions by Lamport [10] and Peterson [14] have inspired several variations [2,5,9,16].

Here, we treat the solution we proposed in [4], which can be regarded as a variation of the protocol of Block and Woo [5]. In [4], we proved that our protocol guarantees mutual exclusion, as well as progress, in the sense that, whenever some threads are competing to enter the critical section, eventually some thread will enter the critical section. We were not able to prove the absence of individual starvation, i.e., that every competing thread eventually enters the critical section. This is remedied here.

In the protocol of [5], every competing thread can be overtaken, roughly speaking, $\frac{1}{2}N^2$ times. We here prove that, in our algorithm, every competing thread is overtaken at most once by any other thread, and that threads overtaking some thread p were competing at the moment thread p was overtaken for the first time. This implies the absence of individual starvation.

In comparison with [4], we need an additional atomicity condition, one which is taken for granted in [5]. The proof of bounded overtaking is closely tied to the proof of mutual exclusion. The mutual exclusion proof is simplified in comparison with [4] because of the additional atomicity condition. We thus provide a complete proof of both properties. The proof was designed and verified [8] with the proof assistant PVS [13].

When we were completing this manuscript, Uri Abraham obtained an independent proof of mutual exclusion and bounded overtaking for our protocol (private communication). His proof is completely different from ours, and is based on "Tarskian system executions".

* Corresponding author. Tel.: +31 503633933; fax: +31 503633800.
E-mail addresses: w.h.hesselink@rug.nl (W.H. Hesselink), csalex@unbc.ca (A.A. Aravind).

0167-6423/$ – see front matter © 2010 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2010.11.002

The setting is traditionally modelled as follows. There are $N > 1$ threads that communicate via shared variables and that repeatedly may compete for access to a shared resource. The threads are thus of the form:

```
**thread** $member(p : Thread) =$
    **loop**
        $NCS$ ; $Intro$ ; $CS$ ; $Exit$
    **endloop** .
```

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 they terminate and that the number of threads in *CS* is guaranteed to remain ≤ 1 (mutual exclusion).

The solution we present here also satisfies bounded overtaking: while any thread q is in *Intro*, the number of overtaking threads is bounded by $N - 1$, where “overtaking” means to execute *Intro*, *CS*, and *Exit* before q enters *CS*.

Both mutual exclusion and bounded overtaking are safety properties. Indeed, recall that, intuitively, a safety property is one asserting that nothing bad happens [1, Section 2.2]. Given a bound K, it would be “something bad” when some competing thread is overtaken more than K times. We deal with these safety properties by means of (mechanically verified) invariants and variant functions. The other relevant properties are the absence of deadlock and livelock. These are dealt with informally.

1.1. The queue based protocol

In [4], we introduced the queue based protocol for mutual exclusion for N threads by heuristic arguments. Here we only give a straightforward description. The algorithm is based on the shared variables:

```
act : **array** [$Thread$] **of** $Boolean := (\lambda\, q : false)$ ,
turn : **array** $[1 .. N - 1]$ **of** $Thread$ .
```

Thread p indicates its interest in the critical section by setting the flag act[p] at the start of *Intro*. It then chooses a sufficiently high *level*. It will enter *CS* when its level is 0. It sets turn[k] $:= p$ when it needs permission from some other thread to proceed to a level below k.

Every thread has private variables:

```
$level : Integer$,
$est$ : **set of** $Thread$ .
```

If v is a private variable, we write $v.p$ for the value of v of thread p outside the code for p, unless the thread is clear from the context.

The value of *level.p* is the current level of thread p as introduced above. Thread p uses its private variable *est* to repeatedly estimate its set of competitors by means of the command

```
$inspect$ :
    **for all** $q \in est$ **do**
        **if** $\neg$ act[$q$] **then** $remove\ q\ from\ est$ **endif** ;
    **endfor** .
```

The protocol is encoded in Fig. 1. When a thread, say p, needs to enter the critical section, it sets a boolean flag act[p], sets *est* to the set of all other threads, executes *inspect*, and sets $level := \#est$, the number of elements of *est*. It then enters the while loop at line 22 that terminates when its $level = 0$, where the critical section is. A thread that leaves the critical section, clears its flag.

In the body of the loop of 22, the thread first enters turn at its own *level*. The thread repeatedly executes *inspect* in loop 24 until either it is *pushed* from turn[*level*] or its estimate of the number of competitors *#est* is less than the *level*. It then decreases its *level* in 26.

Mutual exclusion is expressed by the condition

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

where *crit* is the set of threads q that are in *CS*, i.e., at line 30.

The line numbers in Fig. 1 have no formal meaning yet. We use them again in Section 2.1, when we formalize Fig. 1 to a transition system with numbered atomic transitions that each modify at most one shared variable.

Remarks. Fig. 1 differs from [4]. It looks more like the first versions proposed by one of us (Aravind) several years ago, and is also inspired by the version of Uri Abraham. Line 21 follows [4]. Abraham's version has line 21 replaced by $level := N - 1$. This is also correct. We prefer the above version, because it usually avoids a superfluous assignment to turn[$N - 1$].

It is also correct to move the assignment $est := Thread \setminus \{p\}$ from line 23 to line 25 before *inspect*. If one does this, the set *est* can be replaced by an integer variable *cest* for its cardinality *#est*. Then line 25 can be replaced by

```
$cest := 0$ ; **for all** $q$ **with** act[$q$] **do** $cest$++ **endfor** .
```

```
thread member(p : Thread) =
   loop
10    NCS ;
20    act[p] := true ;  est := Thread \ {p} ;
21    inspect ;
      level := #est ;
22    while  level > 0  do
23       turn[level] := p ; est := Thread \ {p} ;
24       repeat
25          inspect ;
         until  #est < level ∨ turn[level] ≠ p ;
26       level := min(level − 1, #est) ;
      endwhile ;
30    CS ;
40    act[p] := false
   endloop .
```

Fig. 1. The concrete protocol.

In the version proposed, we minimize the number of accesses of the shared variables act[q] at the cost of some private memory. The version of [4] minimizes the number of accesses of turn, at the cost of inspections of act. Our correctness proof applies to all versions of the algorithm, but for the version with *cest* one needs the variables *est* as history variables.

In [4], we allowed the elements of the arrays act and turn to be safe and write-safe, respectively, and proved mutual exclusion under this assumption. We noted, however, that this implies that a thread that takes time trying to write to turn can be passed arbitrary often. We therefore assume here that access to the elements of turn is atomic. For simplicity of exposition, we also assume that access to the elements of act is atomic. The PVS proof, however, allows flickering assignments to act just as in [4].

1.2. Command inspect is not atomic

Command *inspect* is a loop that is not executed atomically. As we need to consider intermediate states of the loop, we introduce an additional private variable *lis* to hold the thread identifiers that have yet to be treated in the loop. Command *inspect* is thus interpreted as

```
a     lis := est ;
b     while  nonempty(lis)  do
c        extract some  q from lis ;
d        if  ¬ act[q]  then remove q from est  endif ;
      endwhile .
```

The body of this loop can be regarded as atomic because it contains only one inspection of a shared variable (act[q]).

1.3. Scenarios

The protocol of Section 1.1 guarantees mutual exclusion and bounded overtaking. In order to see how it does so, it may help to consider some Scenarios. In each of these, initially all threads are idle (at line 10). We number the threads as $q_0, \ldots, q_{N-1}$. We introduce some terminology for a convenient description:

A thread *enters* when it executes line 20.

A thread *moves* when it assigns #*est* to its *level* in line 21 or 26.

A thread *is pushed* at line 26 when it lowers its *level* below #*est* that was just computed in line 25.

A thread *pushes at level k* when it executes line 23 with *level* $= k$.

A thread *exit*s by executing lines 30 and 40.

Scenario A (no congestion). Repeatedly, a thread enters, computes #*est* $= 0$ and sets *level* to 0 in line 21, enters *CS*, and exits.

Scenario B (burst congestion). We let k threads enter, each of them finds #*est* $= k - 1$ in line 21, and pushes at level $k - 1$ in line 23. Then all but one of them are pushed to lower levels, where they again push. This repeats until one of them is pushed to level 0, enters *CS*, and exits. If no other thread has entered in the mean time, the thread turn[$k - 1$] can move, and push at level $k - 2$, etc.

Scenario C (maximal overtaking). We consider $k + 1 \leq N$ threads that repeatedly want access to the critical section. The Scenario shows that one competing period of a thread (q_0) can contain $2k$ exits of other threads. We need to interpret *inspect* as done in Section 1.2. First, thread q_k enters and proceeds to line 22 with *level* $= 0$. Then each of the threads q_i for $i = k - 1$ down to $i = 0$ enters and proceeds to line 21, and executes loop 21b (see 1.2) until *lis* $=$ *est* $= \{q_j \mid j > i\}$. Thread q_0, the final process to enter in this scenario, proceeds to line 21 and gets $est = \{q_j \mid j > 0\}$. Then q_k enters *CS* and exits. Then, one after the other, each of the threads q_i for $i = k - 1$ down to $i = 1$ completes its loop 21b with $est = \emptyset$, enters *CS*, and exits. After this, the competing period of q_0 contains k exits.

Then each of the threads q_i for $i = 1$ up to $i = k$ enters again, moves to level i at line 21, and puts `turn`$[i] := q_i$ in line 23. Then q_0 completes loop 21b with $est = \{q_i \mid 1 \le i \le k\}$ and pushes at level k. In this way, q_0 pushes the whole train of threads one step forward towards *CS*. This is repeated k times and results in that each of the threads q_i for $i = 1$ up to $i = k$ enters *CS*, and exits. At this point, the competing period of q_0 contains $2k$ exits, and q_0 is overtaken k times. Finally q_0 itself enters *CS* and exits.

Scenario D (a move, after being pushed). We use $k + 2 \le N$ threads that want access to the critical section. Thread q_{k+1} enters first and proceeds to line 22 with $level = 0$. Then each of the threads q_i for $i = k$ down to $i = 2$ enters and proceeds to line 21, and executes loop 21b (see 1.2) until $lis = est = \{q_j \mid j > i\}$. Then q_0 and q_1 enter together (or one after the other), then they inspect and find $\#est = k + 1$. Then q_0 followed by q_1 both push at level $k + 1$. Thread q_0 remains at level k.

Then q_{k+1} enters *CS* and exits. Then, one after the other, each of the threads q_i for $i = k$ down to $i = 2$ completes its loop 21b with $est = \emptyset$, enters *CS*, and exits. Then thread q_1 inspects, finds $\#est = 1$, and moves to level 1.

At this point, only threads q_0 and q_1 want access to *CS*. Thread q_1 must wait. By weak fairness, thread q_0 will eventually inspect, find $\#est = 1$, and move to level 1, where it pushes q_1. Then q_1 can enter *CS* and exit.

This scenario shows that one must not disable *inspect* in line 25 for threads (like q_0) that have been pushed at line 26.

1.4. Overview

In the remainder of the paper, we prove that the protocol guarantees mutual exclusion (0), as well as bounded overtaking in the sense that Scenario C of 1.3 describes the worst-case behaviour.

In Section 2, we first transform the pseudocode of Fig. 1 into a transition system *QmxC*. We then develop a number of refinement steps:

$$QmxC \to QmxA \mathrel{-\!\!\triangleright} QmxH \to QmxI\ ,$$

from the *concrete* system *QmxC* via an *abstract* system *QmxA* and a *history* system *QmxH* towards an *ideal* transition system *QmxI*. While *QmxC* contains a nested loop and other complicated statements, system *QmxI* has only four types of atomic steps. The arrows $\to$ represent refinement functions, the arrow $\mathrel{-\!\!\triangleright}$ is an extension with history variables [1]. Every execution of the concrete system *QmxC* corresponds to an execution of *QmxI*. The refinements preserve the observables: whether a thread is idle, competing (in *Intro*), or critical (in *CS*). For the proof of safety, it therefore suffices to prove that *QmxI* guarantees mutual exclusion and bounded overtaking.

Section 3 contains the analysis of the transition system *QmxI*. The proof of mutual exclusion is somewhat easier than in [4] because here array `turn` is modified atomically. The proof of bounded overtaking gives new insights in the protocol. The key result is that, when a competing thread q is overtaken by another thread r, a train of threads is formed, starting with r and containing q, that are forced to exit one after the other. In order to formalize "being overtaken", we extend *QmxI* with sequence numbers for competing threads.

In Section 4, we argue informally that individual starvation would lead to global deadlock or livelock, and that this is impossible in the concrete system. Section 5 contains concluding remarks.

2. Refinement steps

We formalize the pseudocode of Fig. 1 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 then perform a sequence of refinements of the protocol. This sequence starts in the same way as in [4], but subtly deviates, and ends in a much more abstract system.

2.1. The concrete transition system

We formalize the pseudocode of Fig. 1 as a transition system *QmxC* where the N threads can change the global state by atomic steps. The state space *XC* of this system is spanned by the shared variables `turn` and `act`, and the private variables *level*, *est*, *lis*, *bb*, and *pc* for all threads.

For the ease of the analysis, we have extended the pseudocode with two assignments to *level*. We set *level* to $N - 1$ in line 20 upon entry of the competing phase, and to -1 in 30, when exiting the critical section. This is completely harmless because *level* is a private variable.

The resulting transition system *QmxC* is given in Fig. 2. This transition system is the starting point of the PVS verification in [8]. The nondeterministic choice in 10 expresses that *NCS* need not terminate. The line numbers correspond to those of Fig. 1, but now have a formal meaning: a line with number k represents a guarded command with the guard $pc.p = k$, where pc is the program counter of thread p. The calls of *inspect* are expanded according to Section 1.2.

The reader should verify that at every line number the command inspects or modifies at most one shared variable. The most complicated case is line 25, where thread p reads either $act[q]$ or $turn[level.p]$.

```
$cMember(p) =$
10:      $NCS(p)$ ; **goto** 10 **or** 20 .
20:      $act[p] := true$ ; $est := Thread \setminus \{p\}$ ;
         $level := N - 1$ ; $lis := est$ ; **goto** 21 .
21:      **if** $nonempty(lis)$ **then**
             *extract some q from lis* ;
             **if** $\neg act[q]$ **then** *remove q from est* **endif** ;
             **goto** 21
         **else** $level := \#est$ ; **goto** 22 **endif** .
22:      **if** $level > 0$ **then goto** 23 **else goto** 30 **endif** .
23:      $turn[level] := p$ ; $est := Thread \setminus \{p\}$ ; **goto** 24 .
24:      $lis := est$ ; **goto** 25 .
25:      **if** $nonempty(lis)$ **then**
             *extract some q from lis* ;
             **if** $\neg act[q]$ **then** *remove q from est* **endif** ;
             **goto** 25
         **elsif** $\#est < level \lor turn[level] \neq p$ **then goto** 26
         **else goto** 24 **endif** .
26:      $level := \min(level - 1, \#est)$ ; **goto** 22 .
30:      $CS(p)$ ; $level := -1$ ; **goto** 40 .
40:      $act[p] := false$ ; **goto** 10 .
```

Fig. 2. The concrete transition system *QmxC*.

2.2. The abstract protocol

For the ease of the analysis, it is useful to eliminate the program counters and the sets *lis*. We write *XA* to denote the state space of this system. This is the concrete state space *XC* from which the private variables *lis* and *pc* have been removed.

The resulting algorithm is much more nondeterministic. It may be regarded as a UNITY program, see [6]. We abstract the program counters $pc.q$ into the private boolean variables $bb.q$ with the meaning $24 \leq pc.q < 30$.

The resulting abstract algorithm is the parallel composition:

$$QmxA \quad = \quad ||_p \; aMember(p) ,$$

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

```
$aMember(p) =$
    ( $entry(p)$ [] $discard(p)$ [] $move(p)$ [] $push(p)$ []
      $wait(p)$ [] $relax(p)$ [] $exit(p)$ [] **skip** )$^\infty$
```

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 $crit = \{q \mid level.q = 0\}$.

Command 20 is matched by:

```
$entry(p) =$
    $level < 0 \quad \rightarrow$
        $act[p] := true$ ; $est := Thread \setminus \{p\}$ ; $level := N - 1$ .
```

The removal of q from $est.p$ in commands 21 and 25 is matched by:

```
$discard(p) =$
    *extract if possible some q from est with* $\neg act[q]$ .
```

Notice that *discard* does not need a guard other than the "if possible". This is one point where the abstract protocol is much more nondeterministic than the concrete protocol.

The instruction at line 23 is matched by:

```
$push(p) =$
    $level > 0 \land \neg bb \quad \rightarrow$
        $turn[level] := p$ ; $est := Thread \setminus \{p\}$ ; $bb := true$ .
```

The assignment $level := \#est$ in 21 is matched by:

```
$move(p) =$
    $\#est < level \quad \rightarrow \quad level := \#est$ ; $bb := false$ .
```

This command also matches the first alternative in the **elsif** branch of command 25 together with command 26.

The second alternative in the **elsif** branch of command 25 together with command 26 are matched by:

```
wait(p) =
   $bb \wedge \texttt{turn}[level] \neq p \quad \rightarrow \quad level\texttt{--}\ ;\ bb := false$ .
```

Command 30, leaving the critical section, is matched by:

```
exit(p) =
   $level = 0 \quad \rightarrow \quad level := -1$ .
```

We can use the present section without any modification to prove the first variation mentioned in the remark in Section 1.1. For the proof of the second variation we additionally match the assignment to *est* in line 25 by the relaxation command

```
relax(p) =
   $choose\ \ U \supseteq est\ ;\ \ est := U$ .
```

This command is represented in the relational semantics that we use in PVS by

```
relax(p, x, y): bool =
  EXISTS (u: setof[Thread]): subset?(x‘est(p), u) AND
    y = x WITH [ ‘est(p) := u ]
```

where x is the current state and y is the next state.

The PVS verification contains the proof that the obvious projection function $fca : XC \rightarrow XA$ that removes the variables *lis* and *pc*, is a refinement function from *QmxC* to *QmxA*. This proof is analogous to the corresponding proof in [4, Section 3.3]. System *QmxA* suffers from livelock when **skip** (or *relax*) are applied too often. This is not a problem, however, because we use these refinements only for the proof of safety.

2.3. Extension with history variables

In this subsection we prepare the elimination of the sets *est.p* by introducing lower bounds lwbset[*p*] for them. We also introduce a lower bound lwb[*p*] for #lwbset[*p*]. We define the set of competing threads as $Cp = \{q \mid level.q \geq 0\}$, and for convenience, we introduce a shared counter cact for #*Cp*. We thus introduce shared history variables

```
lwbset : **array** [Thread] **of set of** Thread := $(\lambda\, q : \emptyset)$ ,
lwb : **array** [Thread] **of** Integer := $(\lambda\, q : 0)$ ,
cact : Integer := 0 .
```

More precisely, in order to prove that the system *QmxA* of the previous section satisfies mutual exclusion and bounded overtaking, we extend it to a system *QmxH* by adding the variables just declared as history variables [1] (or auxiliary variables [12]). Such variables only serve in the correctness proof, not in the implementation. It is therefore allowed that they are inspected or modified together with a single actual shared variable in an atomic command. Formally, the extension serves its purpose because there is a forward simulation from *QmxA* to *QmxH*: every behaviour of *QmxA* can be mimicked by *QmxH*.

In *QmxH*, we intend to preserve the following invariants:

$K0$: $\texttt{cact} = \#Cp$,
$K1$: $\texttt{lbw}[q] \leq \#\texttt{lwbset}[q]$,
$K2$: $level.q > 0 \;\Rightarrow\; \texttt{lwbset}[q] \subseteq est.q$.

Because the actions *move* and *wait* of thread *p* only influence private variables of *p*, we intend to fuse them with the next action of *p*. For this purpose, we introduce a private variable *lev* that in a next refinement will take the role of *level*, and that should be bigger than *level* when *push* can be executed. The new state space *XH* is the state space *XA* extended with lwbset, lwb, and cact, and *lev*.

The new variables are modified in *entry*, *exit*, and *push* in the following ways:

```
entry(p) =
   $level < 0 \quad \rightarrow$
      lwbset[p] := Cp ;  lwb[p] := cact ;  cact++ ;  lev := N ;
      act[p] := true ;  est := Thread \ {p} ;  level := N − 1 .

exit(p) =
   $level = 0 \quad \rightarrow$
      **for all** q **do**
         lwbset[q] := lwbset[q] \ {p} ;  lwb[q] := max(lwb[q] − 1, 0)
      **enddo** ;
      level := −1 ;  lev := −1 ;  cact-- .
```

```
$push(p) =$
    $level > 0 \ \wedge \ \neg\, bb \quad \rightarrow$
        $\mathtt{turn}[level] := p$ ; $est := Thread \setminus \{p\}$ ; $bb := true$ ;
        $\mathtt{lwbset}[p] := Cp \setminus \{p\}$ ; $\mathtt{lwb}[p] := \mathtt{cact} - 1$ ; $lev := level$ .
```

Recall our aim to fuse the actions *move* and *wait* of thread *p* with the next action of *p*. In the special case that these actions decrease the *level* to 0, we let them also execute a kind of *push*. For this purpose, we introduce a shared ghost variable `tu0`, which will later be replaced by `turn`[0]. We first introduce the action

```
$moveF(p, k) =$
    $level := k$ ; $bb := false$ ;
    **if** $k = 0$ **then**
        $lev := 0$ ; $\mathtt{tu0} := p$ ; $\mathtt{lwbset}[p] := Cp \setminus \{p\}$ ; $\mathtt{lwb}[p] := \mathtt{cact} - 1$
    **endif** .
```

Now the actions *wait* and *move* are defined by

```
$wait(p) =$
    $bb \ \wedge \ \mathtt{turn}[level] \neq p \quad \rightarrow \quad moveF(p, level - 1)$ .

$move(p) =$
    $\#est < level \quad \rightarrow \quad moveF(p, \#est)$ .
```

The actions *discard* and *relax* are lifted to the new state space without modification.

With respect to atomicity, the reader should note that the new shared variables `lwbset`, `lwb`, `cact` are history variables, ghost variables useful for the analysis of the algorithm, but not implemented. There is therefore no problem of interference: we may still treat the complicated guarded commands as atomic instructions. The same holds for the private variables *level.q* that now also occur combined in the shared state function *Cp*.

This gives a new transition system *QmxH* with an obvious forward simulation $QmxA \twoheadrightarrow QmxH$: every step of *QmxA* can be matched by a corresponding step of *QmxH*.

2.4. Removing implementation variables

We have added the history variables `lwb`, `lwbset`, `tu0`, *lev* as a preparation to eliminate several of the implementation variables. In other words, the aim is to transfer the behaviour of *QmxH* to a next transition system *QmxI* with a restricted state space. This requires that the actions are "translated" in terms of the variables that are retained. The translation is based on a number of invariants for the system *QmxH*.

For the translation of *entry* and *exit*, we postulate the invariant

$L0: \quad lev.q = level.q \quad \vee \quad 0 < level.q < lev.q\,.$

For the translation of *move* and *wait*, we postulate the invariants

$L1: \quad bb.q \quad \Rightarrow \quad level.q = lev.q\,,$
$L2: \quad level.q > 0 \quad \Rightarrow \quad \mathtt{lwb}[q] \leq \#est.q\,.$

For the translation of *push*, we postulate the invariants

$L3: \quad \neg\, bb.q \ \wedge \ 0 < level.q \quad \Rightarrow \quad level.q < lev.q\,,$
$L4: \quad \neg\, bb.q \ \wedge \ 0 < level.q < \mathtt{lwb}[q]$
$\quad\quad \Rightarrow \quad lev.q = level.q + 1 \ \wedge \ \mathtt{turn}[level.q] \neq q\,.$

The proofs of these invariants L^* are fairly standard. We give the details for the interested reader. Some auxiliary invariants (K^*) are needed. Preservation of $L0$ under *entry* needs the postulate $N > 1$. Its preservation under *wait* is proved with the auxiliary invariant

$K3: \quad bb.q \quad \Rightarrow \quad level.q > 0\,.$

Preservation of $K3$ is easy. Preservation of $L1$ also follows, in the case of *entry*, from $K3$.

Predicate $L2$ follows from the invariants $K1$ and $K2$ postulated above. Predicate $K1$ is invariant because of $K0$ and $K3$. Preservation of $K0$ is easy, but also needs $K3$. In the proof of invariance of $K2$ under the actions *discard*, we need the additional invariants:

$K4: \quad \mathtt{lwbset}[q] \subseteq Cp\,,$
$K5: \quad level.q \geq 0 \ \Rightarrow \ \mathtt{act}[q]\,.$

Preservation of $K4$ follows from $K3$ in the case of *wait* with $q \neq p$ because $K3$ ensures that *p* does not leave the set *Cp*. Preservation of $K5$ is trivial.

Predicate $L3$ is threatened only by *move* and *wait*. It is preserved because of $L0$. Predicate $L4$ is threatened only by *entry*, *move*, and *wait*. It is preserved by *entry* because $K0$ implies that $entry(p)$ has the postcondition $\mathtt{lwb}[p] \le N - 1 = level.p$. The action $move(p)$ preserves $L4$ because it establishes $\mathtt{lwb}[p] \le level.p$ by $L2$. The action *wait* preserves $L4$ because of $L1$. This concludes the proof of the invariants L^*.

We are now going to transform system *QmxH* into a more abstract system *QmxI*. We first argue informally to see how to do this. The proof comes when all ingredients of the transformation are collected. As in [4], we observe that the sets $est.q$ are superfluous. Indeed, the invariant $L2$ enables us to replace *move* for the moment by the nondeterministic version

```
$moveND(p) =$
    $\mathtt{lwb}[p] < level \quad \to$
        $choose\ some\ \ m\ \ with\ \mathtt{lwb}[p] \le m < level$ ;
        $moveF(p, m)$ .
```

Action $move(p)$ corresponds to $moveND(p)$ with $m = \#est.p$. Now the private variables *est* are no longer inspected, and can therefore be removed. The same holds for the shared variables `lwbset`. Then the modifications of *est* and `lwbset` in *entry*, *discard*, *relax*, and *push* can be removed. Therefore the actions *discard* and *relax* can be replaced by **skip**. Consequently, the shared variables `act` can be removed. This would leave us with the actions *entry*, *exit*, *push*, *wait*, *move*, and the variables `cact`, `turn`, `lwb`, *level*, *aa*, *bb*, *lev*. Similar steps were also taken in [4].

The variable *lev*, however, was introduced above to replace *level* and *bb*, and to enable us to fuse the actions *move* and *wait* with a subsequent *push*. We therefore now introduce a system *QmxI* with the state space *XI* spanned by the shared variables `cact`, `turn`, `lwb`, and the private variables *level*. We propose the projection function $fhi : XH \to XI$, given by

```
$fhi(x) =$
    $(\#\ \ \mathtt{cact} := x.\mathtt{cact}\ ,\ \mathtt{lwb} := x.\mathtt{lwb}\ ,\ level := x.lev$
         $\mathtt{turn} := (\lambda\, k :\ (k = 0\ ?\ x.\mathtt{tu0} : x.\mathtt{turn}[k]))\ \ \#)$ .
```

The brackets (# and #) are record constructors, as used in PVS. In words, the variables `cact` and `lwb` are retained. The variable *lev* of *QmxH* becomes *level* again in *QmxI*. The variable `turn` is extended to index 0 to capture the history variable `tu0`. In system *QmxI*, we propose the operations:

```
$entry(p) =$
    $level < 0 \quad \to \quad \mathtt{lwb}[p] := \mathtt{cact}\ ;\ \mathtt{cact}{+}{+}\ ;\ level := N$ .

$move(p) =$
    $\mathtt{lwb}[p] < level \quad \to$
        $choose\ some\ m\ with\ \mathtt{lwb}[p] \le m < level$ ;
        $level := m\ ;\ \mathtt{turn}[m] := p\ ;\ \mathtt{lwb}[p] := \mathtt{cact} - 1$ .

$wait(p) =$
    $1 \le level \le \mathtt{lwb}[p]\ \wedge\ \mathtt{turn}[level] \ne p \quad \to$
        $level{-}{-}\ ;\ \mathtt{turn}[level] := p\ ;\ \mathtt{lwb}[p] := \mathtt{cact} - 1$ .

$exit(p) =$
    $level = 0 \quad \to$
        for all $q$ do  $\mathtt{lwb}[q] := \max(\mathtt{lwb}[q] - 1, 0)$  enddo ;
        $\mathtt{cact}{-}{-}\ ;\ level := -1$ .
```

Action *move* of *QmxI* is the atomic contraction of *moveND* and *push* of *QmxH*. Similarly, action *wait* of *QmxI* is the atomic contraction of *wait* and *push* of *QmxH*. Moreover, we have strengthened the precondition of *wait* in such a way that there is no overlap with the precondition of *move*. This is justified by the fact that when the preconditions overlap the actions are the same.

More precisely, however, we justify the complete transformation from *QmxH* to *QmxI* by proving that function *fhi* is a refinement function. The actions *discard* and *relax* of *QmxH* correspond to **skip** in *QmxI*. The same holds for *wait* and *move* when the target level is nonzero. When the target level of *wait* and *move* is zero, they correspond to *wait* and *move* of *QmxI* because of the invariants $L1$ and $L2$. The actions *entry* and *exit* of *QmxH* correspond to the same actions in *QmxI* because of the invariant $L0$. The action $push(p)$ of *QmxH* corresponds to $move(p)$ of *QmxI* when $\mathtt{lwb}[p] \le level.p$, because of $L3$. Otherwise it corresponds to $wait(p)$ because of $L3$ and $L4$.

3. Analysis of system QmxI

In this section we prove that system *QmxI* guarantees mutual exclusion and bounded overtaking. The proof of bounded overtaking relies on details of the proof of mutual exclusion. We have therefore to prove mutual exclusion here, even though that was also done in [4], in a more complicated setting.

We first note that system *QmxI* (again) satisfies the easy invariants

$M0$: $\mathtt{cact} = \#Cp$,
$M1$: $\mathtt{lwb}[q] \le \max(\mathtt{cact} - 1, 0)$.

3.1. A proof of mutual exclusion

Mutual exclusion is expressed in the invariant

MX: $\#\{q \mid level.q = 0\} \leq 1\,.$

How to prove this? The scenarios of Section 1.3 give the impression that the competing threads, approximately, form a queue with *level* as queue number. This suggests that the number of threads with $0 \leq level < k$ should be bounded by k. This idea must be strengthened, however, because threads with higher level but $\mathtt{lwb} < k$ can *move* autonomously to a *level* below k. We therefore define $A(k)$ as the set of competing threads that have *level* below k or can *move* there, and postulate bounds on $\#A(k)$. We thus define for all $k \geq 1$ the sets:

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

and postulate the invariants

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

Clearly, predicate MX follows from $J0(1)$ because $\{q \mid level.q = 0\} \subseteq A(1)$.

Initially, the set Cp of the competing threads is empty. Therefore, all sets $A(k)$ are empty and all predicates $J0(k)$ hold.

Using invariant $M0$, it is easy to see that $J0(k)$ is preserved by *entry*. It is preserved by *move* because of $M0$ and $M1$ (the latter invariant is needed when $\mathtt{cact} \leq k$). Predicate $J0(k)$ holds after *exit* provided *exit* has the precondition $J0(k+1)$. Using $M1$, we see that $wait(p)$ preserves $J0(k)$ when $level.p \neq k$.

For the case of $wait(p)$ when $level.p = k$, we analyse the precondition of $wait(p)$. Indeed, it is only executed when some other thread has removed p from $\mathtt{turn}[k]$. In other words, there is some other thread at $\mathtt{turn}[k]$ with level k and $\mathtt{lwb} \geq k$. More precisely, we postulate and prove with PVS the additional invariant:

$M2:$ $0 < level.q \leq \mathtt{lwb}[q]$
$\quad\Rightarrow\quad level.q = level.\mathtt{turn}[level.q] \ \wedge\ \mathtt{lwb}[q] \leq \mathtt{lwb}[\mathtt{turn}[level.q]]\,.$

Predicate $M2$ is preserved by $entry(p)$ because it establishes $\mathtt{lwb}[p] = \mathtt{cact} - 1 < N = level.p$ by $M0$. Predicate $M2$ for thread q is preserved by $move(p)$ and $wait(p)$ for $p \neq q$ because of $M1$ for q. Preservation of $M2$ in the case of *exit* is trivial.

Now, indeed, $wait(p)$ with $level.p = k$ has the postcondition $J0(k)$ when it has the precondition $J0(k+1) \wedge M2$. This proves that all predicates $J0(k)$ are invariant, and hence mutual exclusion.

3.2. Bounded overtaking

The remainder of this section is devoted to the proof of bounded overtaking in the system $QmxI$. We present the proof in a top-down way. In this subsection we describe the global approach, which uses a variant function and an exit reservation predicate.

We prove bounded overtaking by proving that the number of *exit*s during any thread's competing period is bounded by $2N - 2$. For each thread q, we construct an integer valued *variant function* $vf(q) \geq 0$ that, during q's competing period, never increases, and that decreases with every *exit*. More formally, for every number V and threads q and p, any step of the algorithm will satisfy the Hoare triples

$$\begin{aligned}&\{q \in Cp \ \wedge\ vf(q) = V\}\ \ step\ \{vf(q) \leq V\}\ \ ,\\ &\{p \in Cp \ \wedge\ q \in Cp \ \wedge\ vf(q) = V\}\ \ step\ \{p \in Cp \ \vee\ vf(q) < V\}\ \ .\end{aligned} \tag{1}$$

The first triple says that $vf(q)$ never increases while q is competing. The second triple says that it decreases with every exit from Cp. The upper bound $2N - 2$ then follows from $vf(q) < 2N$, which will be immediate from the construction of vf.

The construction of vf is based on the observation that, when a competing thread is delayed and is overtaken by some other thread, a "phase transition" occurs in the execution of the algorithm: the nondeterminacy is greatly reduced. Compare Scenario C in Section 1.3.

We first concentrate on this second phase with the reduced nondeterminacy. In this phase, there is a set S of competing threads that will *exit* before all other threads. In order to formalize this, we form a *exit reservation predicate* $Q(S)$ such that, for every set S of threads and every thread p, every step satisfies the Hoare triples

$$\begin{aligned}&\{Q(S) \ \wedge\ p \notin S \ \wedge\ p \in Cp\}\ \ step\ \{p \in Cp\}\ \ ,\\ &\{Q(S)\}\ \ step\ \{Q(S \cap Cp) \ \vee\ S \cap Cp = \emptyset\}\ \ .\end{aligned} \tag{2}$$

The first Hoare triple says that threads outside S cannot exit. The second one says that exit reservation is preserved until all threads in S have exited. In order to avoid that $Q(S)$ prohibits all future exits, we also require that $Q(S)$ implies $S \neq \emptyset$ and $S \subseteq Cp$. In Section 3.3 below, we construct our exit reservation predicate $Q(S)$ and prove formula (2).

In Section 3.4, we treat the *approach towards* exit reservation. In order to prove that the second phase is reached, we formalize overtaking by giving every thread a new sequence number when it starts competing. We use this to construct an invariant such that an exiting overtaking thread p has a postcondition $Q(S)$ where S contains threads overtaken by p. In other words, the phase transition has happened when an overtaking thread exits. This result is then used to construct a variant function vf that satisfies formula (1) above.

3.3. Exiting trains of threads

The idea of exit reservation emerged with the observation that, e.g., when there are $k+1$ threads $p_0,\ldots,p_k$, with $level.p_i = i$ and $p_i = \texttt{turn}[i]$ and $k \le \texttt{lwb}[p_i]$ for all $i \le k$, then these threads will exit one after the other, and no other threads can overtake them anymore. In other words, we have an exit reservation for the set $S = \{p_0,\ldots,p_k\}$. We call this an exiting train of threads.

Before proving this, we note that the lwb inequality cannot be weakened to $i \le \texttt{lwb}[p_i]$, because (e.g.) the first *exit* then may lower $\texttt{lwb}[p_1]$ to 0, so that p_1 can move to $level := 0$ without being pushed by the remainder of the train. In this way, gaps may appear in the train, and these gaps can be filled by competing threads from behind the train.

We now need to find such an exit reservation predicate $Q(S)$. Firstly, as announced above, $Q(S)$ should imply that S is nonempty and contained in Cp. Next, $Q(S)$ should imply that S consists of the first $\#S$ competing threads that shall exit. Because the set $A(\#S)$ of Section 3.1 consists of at most $\#S$ threads that are likely to exit first, it is natural to guess as a first approximation:

$Q0(S)$: $\quad S \ne \emptyset \ \wedge\ A(\#S) \subseteq S \subseteq Cp\,.$

Every exiting thread belongs to $A(1)$. Therefore $Q0(S)$ implies that every exiting thread belongs to S. So, if $Q(S) \Rightarrow Q0(S)$, this settles the first Hoare triple of (2). In order to prove the second one, we split it into two parts:

$$\{Q(S)\}\ \ nexit\ \{Q(S)\}\ ,$$
$$\{Q(S)\ \wedge\ \#S > 1\}\ \ exit(p)\ \{Q(S \setminus \{p\})\}\ , \tag{3}$$

where *nexit* stands for an arbitrary non-*exit* step. The first triple means that the predicates we are constructing should be invariant under non-*exit* steps.

In order to preserve $Q0(S)$ under *wait*, we need the additional condition that all competing threads are in S or are, with respect to $\#S$, at a higher level or at the critical turn:

$Q1(S)$: $\quad \forall\, q : q \in Cp \ \Rightarrow\ q \in S \ \vee\ \#S < level.q \ \vee\ q = \texttt{turn}[\#S].$

In order to preserve $Q1(S)$ under *move* and *wait*, we also need:

$Q2(S)$: $\quad \forall\, q : q \in S \ \Rightarrow\ level.q \le \#S\,.$

Let us prove preservation of $Q1(S)$ under *wait*. The step $wait(p)$ threatens $Q1(S)$ only by decrementing $level.p$ or the assignment to turn. If decrementing $level.p$ invalidates the second disjunct of the consequent of $Q1(S)$, it makes the third disjunct true. We therefore only need to treat the case that the assignment to turn invalidates the third disjunct. More precisely, the critical case is that p executes $\texttt{turn}[k] := p$ with $k = \#S$, and with the precondition $level.q = k$ and $\texttt{turn}[k] = q$ and $level.p = k+1$. In this situation, we need to prove that $q \in S$. In the precondition of $wait(p)$, we have $p \ne \texttt{turn}[k+1]$. We put $r = \texttt{turn}[k+1]$. Then $p \ne r$. We also have $level.r = k+1$ because of $M2$. Therefore, p, q, r are all different. On the other hand, $S \cup \{p,q,r\} \subseteq A(k+2)$ because of $Q0(S)$ and $Q2(S)$. Then $J0(k+2)$ implies $\#(S \cup \{p,q,r\}) \le k+2$. Now p and r are not in S because of $Q2(S)$. Therefore $q \in S$. This shows that $Q1(S)$ is preserved under *wait*.

Preservation of $Q1(S)$ under *move* is somewhat easier. The step $move(p)$ threatens $Q1(S)$ only by decrementing $level.p$ or the assignment to turn. If decrementing $level.p$ invalidates the second disjunct of the consequent of $Q1(S)$, it makes the third disjunct true because $p \notin S$ implies $\#\texttt{lwb}[p] \ge \#S$ by $Q0(S)$. Again, we only need to treat the case that the assignment to turn invalidates the third disjunct. The critical case is that p executes $\texttt{turn}[k] := p$ with $k = \#S$, and with the precondition $level.q = k$ and $\texttt{turn}[k] = q$ and $\texttt{lwb}[p] = k$. In this situation, we use $Q2(S)$ to prove that $S \cup \{p,q\} \subseteq A(k+1)$ and hence $q \in S$ by $J0(k+1)$. This shows that $Q1(S)$ is preserved under *move*.

Preservation of $Q1(S)$ under *entry* follows from $Q0(S)$. In this way, it is proved that the conjunction $Q012(S)\ :\ Q0(S) \wedge Q1(S) \wedge Q2(S)$ of these three predicates is preserved by all non-*exit* steps:

$$\{Q012(S)\}\ \ nexit\ \{Q012(S)\}\ . \tag{4}$$

We need other predicates to ensure that *exit* has a useful postcondition as in (3). These predicates express that competitors can only reach the lower levels by performing *wait*, i.e. by being pushed:

$Q3(k)$: $\quad Q4(k)\ \vee\ (\exists\, m : 0 \le m < k\ \wedge\ Q4(m)\ \wedge\ Q5(m,k))\,,$

where

$Q4(m)$: $\quad \forall\, i : 1 \le i \le m\ \Rightarrow\ \#A(i) < i\,,$

$Q5(m,k)$: $\quad \forall\, i : m \le i \le k$
$\quad\quad \Rightarrow\ \ level.\texttt{turn}[i] = i\ \wedge\ \min(i+1,k) \le \texttt{lwb}[\texttt{turn}[i]]\,.$

Here, $Q4$ expresses that the lower levels are not as full as $J0$ allows, and $Q5$ expresses that a train of threads is being formed. We are only interested in $Q3(k)$ for $k = \#S - 1$, which is less than cact because of $Q0(S)$ and $M0$.

Using $M0$ and $M1$, it is easy to see that $Q4(m)$ is preserved by *entry* and *move* when $m \le \texttt{cact}$. By the invariant $M2$, the action *wait* preserves $Q4(m)$ provided the precondition also satisfies $Q4(m+1)$. This means that the $Q4$-stretch can shrink at the higher end under action *wait*. At the same time, however, the $Q5$-stretch extends. This is the train of threads to be formed. In this way, we obtain the Hoare triple:

$$\{k \le \texttt{cact}\ \wedge\ Q3(k)\}\ \ nexit\ \{Q3(k)\}\ . \tag{5}$$

We now define predicate $Q(S)$ as the conjunction

$$Q(S):\ Q012(S)\ \wedge\ Q3(\#S-1)\,.$$

It follows from formulas (4) and (5) that $Q(S)$ satisfies the first Hoare triple of requirement (3).

In order to prove the second Hoare triple of (3), we assume that some thread p exits while $Q(S)$ holds with $\#S>1$. We have $Q3(k)$ for $k=\#S-1\geq 1$. Condition $Q4(m)$ with $m>0$ implies that $A(1)$ is empty, so that the *exit* step is precluded. This implies that $Q5(0,k)$ holds.

Let us define $\mathit{train}(j,m) = \{\mathtt{turn}[i] \mid j\leq i\leq m\}$. Predicate $Q5(0,k)$ implies that $\#\mathit{train}(0,k) = k+1$ and $\mathit{train}(0,k)\subseteq A(k+1)$ and hence $\mathit{train}(0,k)=A(k+1)=S$ by *J0*. It even follows that $\mathit{train}(0,i)=A(i+1)$ for all $i\leq k$. Moreover the exiting thread p satisfies $p\in A(1)=\mathit{train}(0,0)$ and hence $p=\mathtt{turn}[0]$. This means that we have the situation completely under control. Without using any invariants, we then see that $\mathit{exit}(p)$ has the postcondition $Q012(S')\wedge Q4(k-1)$ for $S'=S\setminus\{p\}=\mathit{train}(1,k)$. This postcondition implies $Q(S')$. Therefore, the second Hoare triple of (3) holds. This concludes the proof that $Q(S)$ satisfies the formulas (2).

Example. The following Scenario establishes $Q(S)$. Assume that all threads are idle. Let S be a set of k threads. Assume all threads from S enter, and then some thread $p\notin S$ enters. Then all threads from S move to level k, and then p moves to level k. After this, they all satisfy $\mathtt{lwb}[q]=k$, and $\mathtt{turn}[k]=p$. Then $A(k)$ is empty. One can easily verify $Q(S')$. Therefore, the threads from S will exit one after the other, and before p, but the order of their exits is still completely undecided.

Remark. We only prove that condition $Q(S)$ is sufficient to guarantee that the threads in S are the first to terminate. It seems, however, that it is also necessary. □

3.4. Progress towards an exiting train

In order to know whether a thread is being overtaken by other threads, we extend the system with history variables that serve as sequence numbers for competing threads. For this purpose we introduce a shared variable `eCnt` that is incremented at every *entry*, and we give the threads private variables *nr*. The variables are modified only in *entry*, and then according to:

```
$\mathit{entry}(p)=$
   $\mathit{level}<0\ \to$
      $\mathit{nr}:=\mathtt{eCnt}\ ;\ \mathtt{eCnt}\ {+}{+}\ ;$
      $\mathtt{lwb}[p]:=\mathtt{cact}\ ;\mathtt{cact}\ {+}{+}\ ;\mathit{level}:=N\ .$
```

The other operations are lifted to the extended state space without modification.

Formally speaking, this amounts to a new forward simulation. For reasons of efficiency for the mechanical proof, we include the variables `eCnt` and *nr* in the systems *QmxI* as introduced in the Sections 2.3 and 2.4. We give these variables the initial values $\mathtt{eCnt}=N$, and $\mathit{nr}.q=q$ for all threads q, where we assume that the thread identifiers are the numbers from 0 to $N-1$. System *QmxI* now additionally has the easy invariants:

$M3:\quad \mathit{nr}.q<\mathtt{eCnt}\,,$
$M4:\quad \mathit{nr}.q=\mathit{nr}.r\ \Rightarrow\ q=r\,.$

For competing threads p and q, we define p to be a *predecessor* of q iff $\mathit{nr}.p<\mathit{nr}.q$. We thus define the set of predecessors:

$$\mathit{pred}(q)=\{p\in Cp\mid \mathit{nr}.p<\mathit{nr}.q\}\,.$$

The aim is to show that when thread q is overtaking some of its predecessors, it is forming a train of threads that will exit together. For the analysis of such a thread q, it is convenient to assume that q itself has not been overtaken by another thread. We therefore define a thread q to be *fresh* when it is competing and none of its successors have exited yet:

$$\mathit{fresh}=\{q\in Cp\mid \forall\, i:\mathit{nr}.q\leq i<\mathtt{eCnt}\ \Rightarrow\ \exists\, r\in Cp:\mathit{nr}.r=i\}\,.$$

For $q\in Cp$, its predecessors were competing when q updated $\mathtt{lwb}[q]$ in *move* or *wait* for the last time. If q is still in *fresh*, none of its successors have exited. Therefore $\#\mathit{pred}(q)$ is a lower bound of $\mathtt{lwb}[q]$. We thus have the invariant:

$M5:\quad q\in\mathit{fresh}\ \Rightarrow\ \#\mathit{pred}(q)\leq\mathtt{lwb}[q]\,.$

In the proof of the invariance of $M5$, exits ask for special attention. An exit of thread p (usually) decrements $\mathtt{lwb}[q]$, but, if p is a predecessor of q, it also decrements $\#\mathit{pred}(q)$, and otherwise it invalidates $q\in\mathit{fresh}$.

Invariant $M5$ implies that thread q cannot go to a level below $\#\mathit{pred}(q)$ by the action *move*. Therefore, in order to overtake predecessors, thread q must be pushed. It turns out that in this way a train is built according to the following invariant:

$J1:\quad q\in\mathit{fresh}\ \wedge\ \mathit{level}.q=m\ \wedge\ m+|\mathtt{turn}[m]=q|\ \leq\ \#\mathit{pred}(q)$
$\qquad\Rightarrow\quad Q6(m,\#\mathit{pred}(q))\,,$

where $|b|=(b\ ?\ 1:0)$ for boolean b and

$Q6(m,k):\quad \forall\, i:m\leq i\leq k\ \Rightarrow\ \mathit{level}.\mathtt{turn}[i]=i\ \wedge\ k\leq\mathtt{lwb}[\mathtt{turn}[i]].$

Note that $Q6$ looks like $Q5$ of Section 3.3, but that the inequality for `lwb` is stronger. We need this stronger inequality to guarantee that $J1$ is preserved under *exit*s. In Section 3.3, this is not needed because $Q5(m, k)$ is accompanied by $Q4(m)$ that precludes *exit*s unless $m = 0$.

We turn to the proof that $J1$ is an invariant. First observe that the actions *move* and *wait* do not change the sets $pred(q)$ and *fresh*. We next prove that $Q6(m, k)$ are preserved by *entry*, *move*, and *wait* because of $M0$ and $M1$. Predicate $J1$ is preserved by $entry(p)$ because, if $p = q$, the antecedent of $J1$ remains false, and otherwise nothing changes. It is preserved by $move(p)$ with $p = q$ because in the postcondition the antecedent of $J1$ is false by $M5$. In the case of $move(p)$ with $p \neq q$, we use that $\mathtt{lwb}[p]$ becomes $\mathtt{cact} - 1 \geq \#pred(q)$ by $M0$. The case of *wait* is more or less similar. The most interesting case is $exit(p)$. If $p \notin pred(q)$, then $fresh(q)$ becomes false. Otherwise $\#pred(q)$ decreases with 1 and all values $\mathtt{lwb}[r] > 0$ decrease with 1, and $J1$ is also preserved.

Using $J1$, we prove

Theorem 1. *If thread $q \in fresh$ has $level.q = 0$ and $pred(q) \neq \emptyset$, there is a number $k > 0$ with $Q5(0, k)$ and $train(0, k) = \{r\} \cup pred(r)$ for some thread r.*

This means that, when a thread q has reached level 0 and is about to exit and overtake some other threads, it has a train behind it consisting of all predecessors of some thread r, which equals q or is a successor of q. In particular, all predecessors of q are in this train.

Theorem 1 is the core of the progress argument. It is proved as follows. By $J1$ applied to thread q, we have $Q6(0, k_0)$ for $k_0 = \#pred(q) > 0$. This implies $Q5(0, k_0)$. If $Q5(0, k)$ holds for some k, $train(0, k)$ is a set of $k+1$ different threads and hence $k < N$. Let k be the maximal number for which $Q5(0, k)$ holds. Then $k_0 \leq k < N$. Let $r \in train(0, k)$ be a thread for which $nr.r$ is maximal among those of $train(0, k)$. We have $nr.q \leq nr.r$ because mutual exclusion implies $q = \mathtt{turn}[0] \in train(0, k)$. This implies $r \in fresh$. Put $m = level.r$. We have $m \leq k$ because of $r \in train(0, k)$ and $Q5(0, k)$. Put $j = \#pred(r)$. If $k < j$, the invariant $J1$ for $q := r$ implies $Q6(m, j)$. Together with $Q5(0, k)$ and $m \leq k$, this would imply $Q5(0, j)$, contradicting the maximality of k. This proves that $\#pred(r) \leq k$. On the other hand, maximality of $nr.r$ implies $train(0, k) \subseteq \{r\} \cup pred(r)$ and hence $k + 1 \leq 1 + \#pred(r)$. This implies $train(0, k) = \{r\} \cup pred(r)$. □

Because freshness of thread q is only invalidated by some exiting thread $p \in fresh$ with $level.p = 0$, Theorem 1 together with Hoare triples of (2) imply that we have the invariant

$$J2: \qquad q \in Cp \;\Rightarrow\; q \in fresh \;\vee\; (\exists\, S : q \in S \;\wedge\; \#S < N \;\wedge\; Q(S))\,,$$

which expresses that every competing thread is fresh, or is contained in a set of threads that will exit one after the other.

The termination argument now goes as follows. When a thread q enters, it becomes fresh. It remains fresh when predecessors of q exit. Its number of predecessors, however, is bounded by $N - 1$. When a successor of q exits, thread q ceases to be fresh. Therefore, $J2$ implies that q becomes a member of a set S of threads that will exit one after the other. It follows that the number of exits of other threads during one competing period of q is bounded by $2N - 2$.

This termination argument is formalized by defining the function

$$\begin{aligned} vf(q) = \;& (q \in fresh \;?\; N + \#pred(q) \\ & : q \in Cp \;?\; \min\{\#S \mid q \in S \;\wedge\; Q(S)\} \\ & : 0\;)\,. \end{aligned}$$

In the second case, the minimum exists because of invariant $J2$.

This definition implies that $0 \leq vf(q) \leq 2N - 1$ always holds. Furthermore, $vf(q) > 0$ iff $q \in Cp$. Invariant $J2$ implies that $vf(q) \geq N$ iff $q \in fresh$. The main result is:

Theorem 2. *While thread $q \in Cp$ holds, $vf(q)$ never increases, and it decreases with every exit. In other words, it satisfies the Hoare triples (1).*

This implies that a competing period of q contains not more than $2N - 2$ exits of other threads.

One can be slightly more precise. As soon as a competing thread q is overtaken, it is a member of a set S of competing threads that will exit together and $\#S \leq N - 1$. Therefore, q is overtaken by at most $N - 1$ threads and these are competing when the first of them exits. In any case, Scenario C of 1.3 is a worst case scenario.

4. Liveness

In this section, we argue informally that every competing period of every thread terminates, i.e, that individual starvation does not occur.

Assume that in some execution of the protocol of Section 1.1 some thread q remains competing forever. This execution of system *QmxC* induces an execution of *QmxI* in which q remains competing forever. The number of *exit*s after q's entrance is bounded by $2N - 2$. Therefore, from some point onward, no *exit*s occur anymore. We thus have global deadlock or livelock, and every thread is either idle ($level < 0$) or competing ($level > 0$).

In the concrete system, therefore, eventually array `act` is constant. Let k be the number of threads that are competing. Then all competing threads find $\#est = k - 1$ in line 25. Every thread q therefore moves or has moved to a $level \leq k - 1$, and as it cannot proceed further it occupies (equals) $\mathtt{turn}[level.q]$. This gives a contradiction because there are not more than $k - 1$ levels between 1 and $k - 1$. This concludes the proof that every competing period terminates. Note that this proof also applies to the other versions mentioned in the remark in Section 1.1.

5. Concluding remarks

The protocol offers strong fairness guarantees. In every competing period, a thread is overtaken by at most $K - 1$ other threads where K is the number of competing threads when the first of them exits.

The result of the present paper is easily extended to the case that the boolean flags $\mathtt{act}[q]$ are not atomic but only safe. If the variables $\mathtt{turn}[k]$ are not taken to be atomic but only "write-safe", mutual exclusion is still guaranteed, but unbounded overtaking may occur during flickering periods of $\mathtt{turn}$. In [4], we conjectured that, when the variables $\mathtt{turn}[k]$ are atomic, every competing period of any thread contains at most one competing period of any other thread. This follows from the present proof because all overtaking threads are competing when the first of them exits.

The proof of this was difficult to find. For us the idea that, once a thread is overtaken, a train of threads is formed that will exit one after the other, was new. According to one referee, however, it is very similar to that used in the implementation of starvation-free algorithms with weak semaphores by Morris [11] and Udding [17]. It is likely that this idea can be used elsewhere as well, but we have no candidate algorithms.

The resulting proof can be verified by hand, though not conveniently. During the development of the proof, the proof assistant PVS [13] was indispensable, for instance because it gives confidence in intermediate results, even when the goal is not yet in view. The PVS proof script is available at [8].

The refinement steps of Section 2 serve as abstraction steps. Strictly speaking, they are not essential for the proof. Yet, if they had not been taken, the proof would have been unmanageable and incomprehensible.

The question remains how much of the above can be retained when the elements of $\mathtt{turn}$ are only write-safe. One may guess, e.g., that, when a thread q always writes $\mathtt{turn}$ atomically, while $\mathtt{turn}$ is write-safe for other threads, thread q is never overtaken more than $N - 1$ times. This is left, however, to future research.

References

[1] M. Abadi, L. Lamport, The existence of refinement mappings, Theor. Comput. Sci. 82 (1991) 253–284.
[2] K. Alagarsamy, A mutual exclusion algorithm with optimally bounded bypasses, Inform. Process. Lett. 96 (2005) 36–40.
[3] J.H. Anderson, Y.J. Kim, T. Herman, Shared-memory mutual exclusion: major research trends since 1986, Distrib. Comput. 16 (2003) 75–110.
[4] A.A. Aravind, W.H. Hesselink, A queue based mutual exclusion algorithm, Acta Inf. 46 (2009) 73–86.
[5] K. Block, T.-K. Woo, A more efficient generalization of Peterson's mutual exclusion algorithm, Inform. Process. Lett. 35 (1990) 219–222.
[6] K.M. Chandy, J. Misra, Parallel Program Design, A Foundation, Addison-Wesley, 1988.
[7] E.W. Dijkstra, Solution of a problem in concurrent programming control, Commun. ACM 8 (1965) 569.
[8] W.H. Hesselink, PVS proof scripts of "queue based mutual exclusion". Available at: www.cs.rug.nl/~wim/mechver/queueMX/index.html, 2009.
[9] Y. Igarashi, Y. Nishitani, Speedup of the *n*-process mutual exclusion algorithm, Parallel Process. Lett. 9 (1999) 475–485.
[10] L. Lamport, A new solution of Dijkstra's concurrent programming problem, Commun. ACM 17 (1974) 453–455.
[11] J.M. Morris, A starvation-free solution to the mutual exclusion problem, Inform. Process. Lett. 8 (1979) 76–80.
[12] S. Owicki, D. Gries, An axiomatic proof technique for parallel programs, Acta Inf. 6 (1976) 319–340.
[13] S. Owre, N. Shankar, J.M. Rushby, D.W.J. Stringer-Calvert, PVS Version 2.4, System Guide, Prover Guide, PVS Language Reference, 2001. http://pvs.csl.sri.com.
[14] G.L. Peterson, Myths about the mutual exclusion problem, Inform. Process. Lett. 12 (1981) 115–116.
[15] M. Raynal, Algorithms for Mutual Exclusion, MIT Press, 1986.
[16] G. Taubenfeld, Synchronization Algorithms and Concurrent Programming, Pearson Education/Prentice Hall, 2006.
[17] J.T. Udding, Absence of individual starvation using weak semaphores, Inform. Process. Lett. 23 (1986) 159–162.