Priority Mutual Exclusion: Specification and Algorithm

Chien-Chung Huang[1] and Prasad Jayanti[2(✉)]

[1] Chalmers University of Technology, Gothenburg, Sweden
villars@gmail.com
[2] Dartmouth College, Hanover, USA
prasad@cs.dartmouth.edu

Abstract. Mutual exclusion is a fundamental problem in distributed computing. In one well known variant of this problem, which we call *priority mutual exclusion*, processes have priorities and the requirement is that, whenever the critical section becomes vacant, the next occupant should be the process that has the highest priority among the waiting processes. Instead of first capturing this vague, but intuitively appealing requirement by a rigorously specified condition, earlier research rushed into proposing algorithms. Consequently, as we explain in the paper, none of the existing algorithms meet the reasonable expectations we would have of an algorithm that claims to respect process priorities. This paper fixes this situation by rigorously specifying the priority mutual exclusion problem and designing an efficient algorithm for it. Our algorithm supports an arbitrary number of processes and, when configured to support m priority levels (m can be any natural number), the algorithm has $O(m)$ RMR complexity on both DSM and CC machines.

1 Introduction

Mutual Exclusion [5] is a fundamental problem in distributed computing. This problem and its variants—such as readers-writers exclusion [3], group mutual exclusion [8], and abortable mutual exclusion [12]—are extensively researched. One such well-known variant, known as *priority* mutual exclusion, is the subject of this paper.

In the *priority mutual exclusion problem*, each process, as it moves from the Remainder section to the Try section, picks a number, called its priority. Informally, the requirement is that processes enter the CS by the priority order: when selecting which process enters the CS next, the algorithm must pick the process whose priority is the highest among the waiting processes. This statement has an intuitive appeal, but its meaning is vague. Unfortunately, instead of first capturing this intuitive requirement of "enter by priority" by a rigorously specified condition, prior works on priority mutual exclusion that we are aware of [4,6,7,10] rushed to propose algorithms.

In this paper, we make two contributions. First, we give a rigorous specification for the priority mutual exclusion problem and explain how existing

C. Gavoille and D. Ilcinkas (Eds.): DISC 2016, LNCS 9888, pp. 385–398, 2016.
DOI: 10.1007/978-3-662-53426-7_28

algorithms do not meet this specification; thus, they do not meet some reasonable expectations we would have for an algorithm that claims to respect priority. Our second contribution is a novel algorithm that fills this gap. The algorithm supports a finite number m of priority levels, but can handle an arbitrary and unknown number of processes. It has $O(m)$ RMR complexity on both CC and DSM machines. The algorithm uses the *swap* operation, where $swap(X, v)$ writes the value v in the shared variable X and returns X's previous value.

2 Specification of Priority Mutual Exclusion

We consider a system that consists of asynchronous processes that communicate by applying atomic operations on shared variables. The program of each process is a loop that consists of two sections of code—Try section and Exit section. We say a process is in the Remainder section if its program counter points to the first statement of the Try section; and that it is in the Critical section (CS) if its program counter points to the first statement of the Exit section. The Try section, in turn, consists of two code fragments—a *doorway*, followed by a *waiting room*—with the requirement that the doorway is a bounded "straight line" code [9]. Intuitively, a process "registers" its request for the CS by executing the doorway, and then busywaits in the waiting room until it has the "permission" to enter the CS. Initially, all processes are in their Remainder section. Each time a process p executes the first step of the Try section, it selects a number, which is p's *priority* until it returns to the Remainder section.

A *run* is a (finite or infinite) sequence of steps, where each step is executed by some process at some time. An *attempt* in a run is $(p, [t, t'])$, where p is a process, t is a time when p enters the Try section, and t' is the earliest time after t when p returns to the Remainder section. Notice that a process may have many attempts in a run. If $\alpha = (p, [t, t'])$ and $\beta = (q, [\tau, \tau'])$ are any two attempts in a run, we say:

- α is an attempt by p.
- priority(α) is p's priority during the attempt α.
- α is *active* at time s if $t \leq s \leq t'$.
- α is in the Waiting Room (respectively, Try, CS, or Exit) at time s if p is in the Waiting Room (respectively, Try, CS, or Exit) at s and $t \leq s \leq t'$.
- α *doorway precedes* β if the time when p completes the doorway during the attempt α is before the time when q enters the doorway during the attempt β.

The *Priority Mutual Exclusion Problem* is to design the code for the Try and Exit sections so that five properties—the three stated below and the two defined in Sects. 2.1 and 2.2—hold in all runs.

- *Mutual Exclusion:* At most one process is in the CS at any time.
- *Bounded Exit:* There is an integer b such that every process completes the Exit section in at most b of its steps.

– *Livelock Freedom:* On the assumption that no process permanently stops taking steps in the Try and Exit sections and no process stays in the CS forever, if a process is in the Try section at a point in time, then some process is in the CS at a later point in time.

The remaining two properties capture priorities. Since these properties are new, we have motivated and carefully defined them in the next two subsections.

2.1 Priority Entry

Intuitively, the purpose of the bounded doorway is to make it possible for a process to "register" its priority and its interest in the CS, without being hindered by other processes. Accordingly, if a high priority process p completes the doorway before a lower priority process q even enters the doorway, it makes sense to require that q does not enter the CS before p.

What else must we ensure? Consider a scenario where, while p is in the CS and q is in the waiting room, a process r of higher priority than q enters the Try section, completes the doorway, and enters the waiting room. Later, when p leaves the CS, we would want r, and not q, to enter the CS, even though q has been waiting much longer than r.

To capture the above expectations, we define a "*dominates*" binary relation, denoted $\succ$, on the set of attempts in a run as follows:

Definition 1. For any two attempts α and α' in a run, we say α *dominates* α', written $\alpha \succ \alpha'$, if

priority(α) > priority(α') and α doorway precedes α', or
priority(α) > priority(α') and there is a time when some process is in the CS, α is in the waiting room, and α' is in the Try section.

The next property states what it really means for processes to enter the CS by priority order.

– *Priority Entry*: If α and α' are any two attempts in a run such that $\alpha \succ \alpha'$, then α' does not enter the CS before α.

2.2 Wait-Free Progress for Dominator

The Priority Entry property stated above only ensures that a low priority process does not race ahead of a higher priority process into the CS, but it makes no guarantee that the higher priority process is not obstructed by a lower priority process. This observation motivates the need to formulate a property that guarantees that, under suitable conditions, a high priority process' progress to the CS is not hindered by a lower priority process.

To understand what this property should be fleshed out, consider a scenario where a process q is in the CS, a process p is in the waiting room, and a set S of additional processes are in the Try section. Assume that p has a higher priority

than every process in the set S. Then, p dominates every process in S. Suppose now that q leaves the CS, completes the Exit section, and goes back to the remainder section. Assuming that no new processes will enter the Try section with a priority higher than p's, process p finds itself in a favorable situation where no process is in the CS or the Exit section, and p dominates every other process in the Try section forever. Under these circumstances, it makes sense to require that p be able to enter the CS in a finite number of its own steps, regardless of whether other processes are slow, fast, or have crashed. Thus, we are led to the following property: If no process is in the CS or the Exit section and a process p in the waiting room dominates forever every other process in the Try section, then p enters the CS in a finite number of its own steps. More precisely:

– *Wait-free Progress for Dominator:* If an attempt α by a process p is in the waiting room at time τ, no process is in the CS or the Exit section at τ, and, for all $\tau' \geq \tau$, α dominates every attempt that is active at τ', then p enters the CS during the attempt α in a finite number of its steps, regardless of whether other processes take any steps or crash.

3 Stronger Version of Priority Mutual Exclusion

So far we have been silent on the order in which processes of the same priority may enter the CS, and on whether a process can obstruct another process of the same priority. In particular, the properties stated above allow a process p to be stuck in the waiting room while another process q of equal priority repeatedly enters and leaves the CS. We could prevent such a scenario by requiring that processes of equal priority enter the CS in the FCFS order: if p and q have the same priority and p doorway-precedes q, then q does not enter the CS before p.

Similarly, if p and q have the same priority and p doorway-precedes q, we might wish to strengthen the "Wait-free Progress for Dominator" property by requiring that q does not obstruct p's entry into the CS.

To capture these ideas, it is convenient to weaken the earlier defined "dominates" relation slightly by replacing ">" with "$\geq$" in the first condition, as follows:

Definition 2. For any two attempts α and α' in a run, we say α *weakly dominates* α', written $\alpha \succ_w \alpha'$, if

priority(α) $\geq$ priority(α') and α doorway precedes α', or
priority(α) $>$ priority(α') and there is a time when some process is in the CS, α is in the waiting room, and α' is in the Try section.

Notice that, for any two attempts α and α', $\alpha \succ \alpha'$ implies $\alpha \succ_w \alpha'$. Now, we can capture both Priority Entry and FCFS (among processes of equal priority) together as follows:

– *Priority Entry + FCFS*: If α and α' are any two attempts in a run such that $\alpha \succ_w \alpha'$, then α' does not enter the CS before α.

And we can strengthen the "Wait-free Progress for Dominator" property by replacing "$\succ$" with the weaker "$\succ_w$" in its definition:

– *Strong Wait-free Progress for Dominator*: If an attempt α by a process p is in the waiting room at time τ, no process is in the CS or the Exit section at τ, and, for all $\tau' \geq \tau$, α weakly dominates every attempt that is active at τ', then p enters the CS during the attempt α in a finite number of its steps, regardless of whether other processes take any steps or crash.

Finally, we define the *Strong Priority Mutual Exclusion Problem* as designing the code for the Try and Exit sections so that the following five properties hold in all runs: Mutual Exclusion, Livelock Freedom, Bounded Exit, Priority Entry + FCFS, and Strong Wait-free Progress for Dominator.

4 Discussion of Previous Research

The idea of dividing the Try section into a bounded doorway and waiting room is due to Lamport [9], who introduced this structure to capture FCFS—a fairness condition for Mutual Exclusion. Our specification of the Priority Mutual Exclusion Problem is inspired by Bhatt and Jayanti's formulations of the Readers-Writers Exclusion Problem [1]. To the best of our knowledge, there are four papers that proposed algorithms of bounded RMR complexity for the priority mutual exclusion problem [4,6,7,10], but no prior work has attempted a rigorous specification of the problem. Below, we briefly describe why the algorithms in [4,6,7,10] do not meet our specification.

4.1 Algorithms of Markatos [10] and Craig [4]

Markatos' and Craig's algorithms, which are adaptations of Mellor-Crummey and Scott's queue-based mutual exclusion algorithm [11], do not satisfy Priority Entry because they admit the following scenario:

– While a process p is in the CS, three processes q, r, and s enter the Try section and queue themselves up in that order (q, followed by r, followed by s). Assume that s has a higher priority than both q and r.
– q and s complete their doorways, but r stops in the doorway before setting the link at q to point to r.
– p leaves the CS and makes the "best effort" to traverse the queue of waiting processes to locate the highest priority process. It sees q, but will not be able to see the next element r in the queue (since r has not yet installed a link at q to point to r). Because p can't see r, it can't see s either. Thus, q is the only waiting process that p is aware of. So, p lets q into the CS.

In the above scenario, $s \succ q$, yet q enters the CS before s, thereby violating Priority Entry.

4.2 A Second Algorithm of Markatos [10]

Another algorithm in Markatos' paper adapts Burns' algorithm [2] for standard mutual exclusion. In Markatos' algorithm, when a process p leaves the CS, it scans all n processes to find out which ones are waiting and lets into the CS the highest priority process among them. To implement this strategy, p reads $want[1], want[2], \ldots, want[n]$ in that order, where $want[i]$ is a shared flag that process i sets to request the CS. This algorithm violates Priority Entry because it admits the following scenario:

- Process p is in the CS while all others are in the remainder section. Then, p leaves the CS and reads *false* in $want[1]$ (because process 1 is in the remainder section).
- Process 1 enters the Try section with a high priority, completes the doorway, and proceeds to busywait in the waiting room. (During the doorway, process 1 sets $want[1]$, but p does not notice it because it has already gone past 1.)
- Process 2 enters the Try section with a low priority, completes the doorway (setting $want[2]$ along the way), and proceeds to busywait in the waiting room.
- p reads *true* in $want[2]$ and *false* in $want[3], want[4], \ldots, want[n]$. Believing that process 2 is the only waiting process, p lets process 2 into the CS.

In the above scenario, process 1 dominates process 2 because it has a higher priority and doorway-precedes process 2. Therefore, process 2 entering the CS before process 1 is a violation of Priority Entry.

4.3 Johnson and Harathi's Algorithm [7]

Johnson and Harathi's algorithm maintains a list of waiting processes, ordered by their priority. For example, if the list is p, q, r, it means that three processes are waiting, p has the highest priority, q has the second highest priority, and r has the least priority. When a process enters the Try section, it traverses this list and attempts to insert itself at the appropriate position (so as to preserve the priority order of the list). When a process leaves the CS, if the list is not empty, it removes the first process in the list and lets it into the CS. The algorithm uses CAS to manipulate the queue.

Consider a scenario involving four processes a, b, c, x, where a, b, c have the same priority and x has a higher priority. Suppose that a is in the CS; the list has b, c; and x is in the Try section, trying to insert itself at the front of the list. Suppose that a then leaves the CS and acts on the list (to remove the front element) concurrently with x, which also acts on the list to insert itself at the front. Suppose that a succeeds and x fails (because a's CAS succeeds and x's CAS fails). Then, the situation is that b is in the CS, a is back in the remainder section, and x has still not inserted itself in the list. Suppose that a then comes back and inserts itself in the list (to make the list c, a). As b leaves the CS, as before suppose that b and x act on the list concurrently, b's CAS succeeds and x's CAS once again fails. We can repeat the above actions forever to ensure that

x will never be able to insert itself into the list, even though x never ceases to take steps.

The above scenario is damning: even though x has higher priority than all others and takes steps repeatedly forever, it is stuck in the Try section without being able to even get into the wait-queue! In particular, the algorithm violates Priority Entry.

4.4 Jayanti's Algorithm [6]

Jayanti's algorithm satisfies Priority Entry, but violates "Wait-free Progress for Dominator" by admitting the following scenario. While a process p is in the CS, a high priority process q inserts its name into the wait-queue, which is maintained as a priority queue (using LL/SC). Then, p leaves the CS, depositing a token in a central location (to indicate that the CS is vacant) and proceeding to inspect the priority queue. A low priority process r then enters the Try section, inserts its name in the priority queue, grabs the token from the central location, and goes back to the priority queue to hand the token to the highest priority waiting process. However, before r finds out that q is the highest priority waiting process, process p completes the Exit section and goes back to the remainder section, which is possible with that algorithm. At this point, there is no process in the CS or the Exit section, and q dominates r. Yet, q cannot enter the CS until r hands it the token, violating "Wait-free Progress for Dominator."

5 The Auxiliary *Lock* object

Our priority mutual exclusion algorithm, described in the next section, is designed using auxiliary objects that we call *lock* objects. In this section, we specify this object and state how it can be efficiently implemented.

5.1 Specification of a Lock Object

A *lock object* L is an abstraction that helps solve FCFS mutual exclusion, and is specified in Fig. 1. Its state is represented by (i) L.waitqueue, the sequence of processes waiting for the lock, and (ii) L.open, which is *true* if and only if the lock is available. A process p requests the lock by executing $L.\text{request}_p()$, which appends p to the wait queue. When the lock is not open, p can open it by executing $L.\text{release}_p()$. After p requests the lock, it can execute $L.\text{isGranted}_p()$ to attempt to own the lock. If the lock is open and p is at the front of the wait queue, the attempt succeeds—i.e., the lock is granted to p, p is removed from the queue, and the lock is no longer open. Finally, p can find out if one or more processes are waiting for the lock by executing L.areProcsWaiting(). We named this object a lock object because it is trivial to solve FCFS mutual exclusion using this object, as follows:

State

L.waitqueue: Sequence of processes, initially empty
L.open: Boolean, initially *true* if we want the lock to be open initially, and *false* otherwise

Operations

Precondition: $p \notin L$.waitqueue
$L.\text{request}_p()$
append p to L.waitqueue

Precondition: L.open is false
$L.\text{release}_p()$
$L.\text{open} = \mathit{true}$

Precondition: $p \in L$.waitqueue
$L.\text{isGranted}_p()$
if (L.open == *true*) $\wedge$ (p is at the front of L.waitqueue)
remove p from L.waitqueue
$L.\text{open} = \mathit{false}$
return *true*
else return *false*

$L.\text{areProcsWaiting}_p()$
return (L.waitqueue == *empty*)

Fig. 1. Specification of a lock object L

$L.\text{request}_p()$
repeat till $L.\text{isGranted}_p()$ returns *true*
CRITICAL SECTION
$L.\text{release}_p()$

5.2 Implementing the Lock Object

Mellor-Crummey and Scott [11] and Craig [4] designed constant RMR algorithms for FCFS mutual exclusion using shared variables that support the *swap* operation. With a straightforward adaptation of their algorithm, we get an implementation of the lock object, which we omit due to space constraints. The result that we achieve is summarized as follows:

Theorem 1. There is an algorithm that correctly implements a lock object L (using read, write, and swap operations) under the assumption that $L.\text{open} = \mathit{false}$ in every interval during which a process p executes $L.\text{areProcsWaiting}_p()$. On both DSM and CC machines, executing any of $L.\text{request}_p()$, $L.\text{release}_p()$, or $L.\text{areProcsWaiting}_p()$, or repeatedly executing $L.\text{isGranted}_p()$ until it returns *true* incurs only $O(1)$ RMRs.

6 The Algorithm

In this section, we present a novel priority mutual exclusion algorithm that supports priorities from a set $\{1, 2, \ldots, m\}$, can handle an arbitrary and unknown number of processes, and has $O(m)$ RMR complexity on both DSM and CC machines.

The algorithm employs m lock objects LOCK$[1 \cdot\cdot m]$. When a process p enters the Try section with a priority $\pi \in \{1, 2, \ldots, m\}$, it inserts itself in the wait-queue of LOCK$[\pi]$, the lock associated with priority π. When p leaves the CS, it checks if processes are waiting in any of the wait-queues. If there are, it releases the highest priority lock whose wait-queue is nonempty. On the other hand, if no processes are waiting, p leaves a token in a "depository" DEP so that a process q that enters the Try section in the future can grab the token and enter the CS. These ideas lead to our first attempt towards an algorithm, which we present in Fig. 2. Two lines (2 and 7) are currently left blank, which we will fill later. The code is described informally as follows.

Process p picks a priority π (Line 0) and inserts itself in the wait-queue of LOCK$[\pi]$, the lock associated with its priority (Line 1). Since it is possible that the depository contains the token, p attempts to grab the token from the depository (Line 3). If the depository contains the token, p grabs it and simultaneously erases the token from the depository by swapping the integer π (its priority). If p gets the token, it opens LOCK$[\pi]$ (Line 4) so that the process at the front of its wait-queue is granted the lock, enabling that process to proceed to the CS (note that p knows that LOCK$[\pi]$'s wait-queue is nonempty because p itself is in that wait-queue). The process p then busywaits until it is granted the lock and then enters the CS (Line 5). When p leaves the CS, it goes through all locks, from 1 to m, to identify the highest priority nonempty wait-queue, if there is any (Lines 8 to 10). If $k \neq 0$ when the for-loop on Line 9 terminates, it means that some processes are waiting at LOCK$[k]$ and p found the wait-queues associated with locks $k+1, k+2, \ldots, n$ to be empty. In this case, p skips Line 11 and releases LOCK$[k]$ (Line 12), which enables the earliest process waiting on that lock to enter the CS. On the other hand, if $k = 0$ when the for-loop on Line 9 terminates, it means that p found all wait-queues to be empty. In this case, p puts a token in the depository DEP (Line 11) so that a process q that enters the Try section in the future can grab the token and proceed to the CS. The *swap* operation at Line 11 enables p to both deposit the token in DEP and simultaneously read into k what was in DEP. If $k = \perp$, p infers that since the time that p had cleared the depository at Line 6 no process executed Line 3; in this case, p skips Line 12 and exits to the remainder section, aware that the token it left behind in DEP will be picked up by a process that enters the Try section in the future. On the other hand, if the swap operation at Line 11 swaps into k a positive integer, p infers that since the time it had cleared DEP at Line 6, some process q of priority k executed the swap operation at Line 3. The value that this swap operation returned to q could not have been "token" since p had not deposited the token in DEP by that time. So, q must have skipped Line 4 and be busywaiting at Line 5 for grant of access to LOCK$[k]$. If p takes no action and

Shared variables
DEP $\in \{token, \perp, 1, 2, \ldots, m\}$, initialized to *token*; supports *swap* operation
LOCK$[1 \ldots m]$: array of m lock objects; initially all are closed,
i.e., $\forall i$, LOCK$[i]$.open = *false*

```
0.  select a priority π ∈ {1, 2, ..., m}
1.  LOCK[π].request_p()
2.
3.      if swap(DEP, π) == token
4.          LOCK[π].release_p()
5.  repeat till LOCK[π].isGranted_p() returns true
    Critical Section
6.  DEP = ⊥
7.
8.  k = 0
9.  for i = 1 to m
10.     if LOCK[i].areProcsWaiting_p() then k = i
11. if k = 0 then k = swap(DEP, token)
12. if k ∈ {1, 2, ..., m} then LOCK[k].release_p()
```

Fig. 2. First attempt towards a priority mutual exclusion algorithm: code shown is for process p

moves on to the remainder section, q will busywait forever at Line 5, leading to livelock. To prevent this situation, p releases LOCK$[k]$ (Line 12), which enables the process r at the front of that lock's wait-queue (which is possibly, but not necessarily, q) to enter the CS. There is however a nasty race condition here: the depository DEP currently contains the token, which means that a process s that might now enter the Try section with a new priority π executes Lines 0 through 3, grabs the token at Line 3, releases its lock at Line 4, so finds the lock granted at Line 5, and enters the CS that already contains r, violating mutual exclusion! We prevent this scenario by placing a "gate" to regulate access to the depository. Specifically, our final algorithm, presented in Fig. 3, is obtained by making the following three small additions to the code described so far:

- A new shared variable GATE, which can take on two values—*open* or *closed*. Initially, the gate is open, i.e., GATE = *open*.
- Line 2, which ensures that a process p attempts to grab the token at Line 3 only if it finds the gate open. The swap operation at Line 2 lets p close the gate and simultaneously learn if the gate was open just before the operation.
- Line 7, where an exiting process opens the gate.

With these changes, the algorithm prevents violation of mutual exclusion because, as we now explain, an exiting process either wakes up a waiting process or leaves the token behind for later pick up, but never does both. Consider a process p as it leaves the CS. When p's for-loop at Lines 9 and 10 terminates, either $k = 0$

Shared variables

GATE $\in \{open, closed\}$, initialized to *open*; supports *swap* operation
DEP $\in \{token, \perp, 1, 2, \ldots, m\}$, initialized to *token*; supports *swap* operation
LOCK$[1 \ldots m]$: array of m lock objects; initially all are closed, i.e.,
$\forall i$, LOCK$[i]$.open = *false*

```
0.  select a priority π ∈ {1, 2, ..., m}
1.  LOCK[π].request_p()
2.  if swap(GATE, closed) == open
3.      if swap(DEP, π) == token
4.          LOCK[π].release_p()
5.  repeat till LOCK[π].isGranted_p() returns true
    Critical Section
6.  DEP = ⊥
7.  GATE = open
8.  k = 0
9.  for i = 1 to m
10.     if LOCK[i].areProcsWaiting_p() then k = i
11. if k = 0 then k = swap(DEP, token)
12. if k ∈ {1, 2, ..., m} then LOCK[k].release_p()
```

Fig. 3. Priority mutual exclusion algorithm: code shown is for process p. Supports limited priorities from $\{1, 2, \ldots, m\}$, but an arbitrary and unknown number of processes.

or $k \in \{1, 2, \ldots, m\}$. If $k > 0$, p skips Line 11 and executes Line 12; thus, p opens a lock but does not leave the token in the depository, thereby giving no scope for any violation of mutual exclusion. For the remaining case, suppose that $k = 0$, which implies that p found every wait-queue to be empty when it executed the for-loop on Lines 9 and 10. It follows that all wait-queues were empty at the point when p had executed Line 6. Therefore, during the interval I spanning from when p had executed Line 6 to when p executes Line 11, at most one process could have gone past the gate at Line 2 and onto Line 3 (because the first process to execute Line 2 closes the gate). If no process executed Line 3 during the interval I, when p executes Line 11 to put the token in the depository, the swap operation returns $\perp$ (because p had put $\perp$ in DEP at Line 6), so p skips Line 12 and goes back to the remainder section without waking anyone from a wait-queue (so there is no scope for violating mutual exclusion). On the other hand, if exactly one process q of priority π executes Line 3 during the interval I, q would read $\perp$ from DEP at Line 3 (because p had put $\perp$ in DEP at Line 6) and busywait at Line 5. And when p executes Line 11 to put the token in the depository, the swap operation returns $\pi \in \{1, 2, \ldots, m\}$ (that q put into DEP). In this case, p executes Line 12 to open LOCK$[\pi]$, while still leaving the token in the depository. However, there is no danger of some other process picking up this token because the gate is closed at this point, so no process will be able to get to Line 3 to grab the token! Hence, mutual exclusion won't be violated. This is just the intuition, and the next section provides a rigorous proof of all properties.

6.1 Proof of Mutual Exclusion and Livelock Freedom

We present a rigorous proof of correctness, which we found to be as challenging as the algorithm. The crux lies in identifying the invariant, presented in Fig. 4. The proof that the algorithm satisfies the invariant is by induction, which is omitted due to space constraints.

Lemma 1 (Mutual Exclusion). *The algorithm satisfies Mutual Exclusion.*

Proof. Immediate from Part (4) of the invariant. ■

Lemma 2 (Bounded Exit). *The algorithm satisfies Bounded Exit.*

Proof. Obvious since the exit section involves no waiting and consists of at most $m + 4$ steps. ■

Lemma 3 (Livelock Freedom). *The algorithm satisfies Livelock Freedom.*

Proof. Let C be any configuration in which no process is in the CS or the exit section (i.e., $\forall p : PC_p \notin \{7, 8, 9, 10, 11, 12\}$) and at least one process is in the Try section (i.e., $PC_p \in \{2, 3, 4, 5\}$ for some p). To prove the lemma, we argue below that some process is guaranteed to eventually enter the CS. We begin by noting that Part (7a) of the invariant is false in C. Then, it follows from (7) that exactly one of (7b) or (7c) is true.

Case 1: Assume that (7c) is true and (7b) is false. Then, there are three subcases: (i) DEP = token and GATE = *open*, or (ii) DEP = token and $PC_p = 3$ for some p, or (iii) $PC_p = 4$ for some p.

In Subcase (i), it follows from Part (6) of the invariant that all processes in the Try section are at Lines 2. Whichever process executes Line 2 first, it finds that GATE = *open* and moves to Line 3, thereby bringing the configuration to Subcase (ii).

In Subcase (ii), it follows from Part (5) of the invariant that GATE = *closed* and no process other than p is at Lines 3 or 4. Thus, we have the gate closed, p at Line 3, all other processes at Lines 0, 1, 2, or 5, and none of the locks in an open state (since (7b) is false in the case under consideration). So, no process can go past Line 5 or enter Line 3 until p executes a step. When p executes Line 3, it finds the token in DEP and moves to Line 4, thereby bringing the configuration to Subcase (iii).

In Subcase (iii), p is at Line 4 and, by Part (1) of the invariant, the wait-queue of LOCK$[\pi_p]$ is nonempty. When p executes Line 4, it opens this lock, thereby bringing the configuration to Case (2), which we deal with below.

Case 2: Assume that (7b) is true, i.e., some lock ℓ is open. Then, by Part (2), the wait-queue associated with this lock is nonempty. Let q be the process at the front of this queue. By (1), q is at one of Lines 2, 3, 4, or 5. When q moves to Line 5 and executes the first iteration of the repeat-until loop at Line 5, q finds the lock granted to it (because lock ℓ is open), so moves to Line 6 (i.e., enters the CS).

We conclude from the above that livelock is not possible. ■

1. $$\forall p \forall \ell : (p \in \text{LOCK}[\ell].\text{wait-queue} \Leftrightarrow (PC_p \in \{2,3,4,5\} \wedge \pi_p = \ell))$$
2. $$\forall \ell : (\text{LOCK}[\ell].\text{open} \Rightarrow \text{LOCK}[\ell].\text{wait-queue} \neq \epsilon)$$
3. $$\forall \ell \forall \ell' \neq \ell : (\text{LOCK}[\ell].\text{open} \Rightarrow \neg \text{LOCK}[\ell'].\text{open})$$
4. $$\forall p \forall q \neq p : ((PC_p \in \{6,7,8,9,10,11,12\} \Rightarrow PC_q \notin \{6,7,8,9,10,11,12\}))$$
5. $$\forall p : ((PC_p = 4 \vee (PC_p = 3 \wedge \text{DEP} = \textit{token})) \Rightarrow (\text{GATE} = \textit{closed} \wedge \forall q \neq p : PC_q \notin \{3,4\}))$$
6. $$(\text{DEP} = \textit{token} \wedge \text{GATE} = \textit{open}) \Rightarrow \forall p : PC_p \in \{0,1,2\}$$
7. $$((a) \wedge \neg(b) \wedge \neg(c)) \vee (\neg(a) \wedge (b) \wedge \neg(c)) \vee (\neg(a) \wedge \neg(b) \wedge (c)), \text{where}$$
   (a) $\exists p : PC_p \in \{6,7,8,9,10,11,12\}$
   (b) $\exists \ell : \text{LOCK}[\ell].\text{open}$
   (c) $(\exists p : PC_p = 4) \vee ((\text{DEP} = \textit{token}) \wedge (\text{GATE} = \textit{open} \vee \exists p : PC_p = 3))$
8. $$\forall p : ((PC_p \in \{7,8,9,10,11\} \wedge \text{DEP} \neq \perp) \Rightarrow (\text{DEP} \in \{1,2,\ldots,m\} \wedge \text{LOCK}[\text{DEP}].\text{wait-queue} \neq \epsilon))$$
9. $$\forall p : ((PC_p \in \{10,11\} \wedge k_p > 0) \Rightarrow ((1 \leq k_p < i_p) \wedge \text{LOCK}[k_p].\text{wait-queue} \neq \epsilon))$$
10. $$\forall p : ((PC_p \in \{10,11\} \wedge k_p = 0 \wedge \forall j \in \{i_p, i_{p+1}, \ldots, m\} : \text{LOCK}[j].\text{wait-queue} = \epsilon) \Rightarrow ((a) \vee (b) \vee (c)))$$
    (a) $(\text{GATE} = \textit{open}) \wedge (\text{DEP} = \perp) \wedge (\forall q \neq p : PC_q \in \{0,1,2\})$
    (b) $(\text{GATE} = \textit{closed}) \wedge (\text{DEP} = \perp) \wedge (\exists q : PC_q = 3 \wedge (\forall r \neq q : PC_r \notin \{3,4\}))$
    (c) $(\text{GATE} = \textit{closed}) \wedge (\text{DEP} \in \{1,2,\ldots,i_p - 1\}) \wedge (\forall q : PC_q \notin \{3,4\}) \wedge \text{LOCK}[\text{DEP}].\text{wait-queue} \neq \epsilon$
11. $$\forall p : (PC_p = 12 \Rightarrow \text{LOCK}[k_p].\text{wait-queue} \neq \epsilon)$$

Fig. 4. Invariant satisfied by the algorithm

Due to space limitation, we omit the proof of the Priority Entry + FCFS and Strong Wait-free Progress for Dominator properties, and proceed to state the main result of this work.

Theorem 2. *The algorithm in* Fig. 3 *correctly solves the Strong Priority Mutual Exclusion Problem for an arbitrary and unknown number of processes, when priorities are drawn from* $\{1, 2, \ldots, m\}$. *The algorithm has* $O(m)$ *RMR complexity on both DSM and CC machines.*

References

1. Bhatt, V., Jayanti, P.: Constant RMR solutions to reader writer synchronization. In: PODC 2010: Proceedings of the Twenty-Ninth Annual Symposium on Principles of Distributed Computing, pp. 468–477 (2010)
2. Burns, J.E.: Mutual exclusion with linear waiting using binary shared variables. SIGACT News **10**(2), 42–47 (1978)
3. Courtois, P.J., Heymans, F., Parnas, D.L.: Concurrent control with "readers" and "writers". Commun. ACM **14**(10), 667–668 (1971)
4. Craig, T.: Queuing spin lock algorithms to support timing predictability. In: Proceedings of the 14th IEEE Real-time Systems Symposium, pp. 148–156. IEEE (1993)
5. Dijkstra, E.W.: Solution of a problem in concurrent programming control. Commun. ACM **8**(9), 569 (1965)
6. Jayanti, P.: Adaptive and efficient abortable mutual exclusion. In: PODC 2003: Proceedings of the Twenty-Second Annual Symposium on Principles of Distributed Computing, pp. 295–304. ACM, New York (2003)
7. Johnson, T., Harathi, K.: A prioritized multiprocessor spin lock. IEEE Trans. Parallel Distrib. Syst. **8**, 926–933 (1997)
8. Joung, Y.J.: Asynchronous group mutual exclusion (extended abstract). In: PODC 1998: Proceedings of the Seventeenth Annual ACM Symposium on Principles of Distributed Computing, pp. 51–60. ACM, New York (1998)
9. Lamport, L.: A new solution of Dijkstra's concurrent programming problem. Commun. ACM **17**(8), 453–455 (1974)
10. Markatos, E.: Multiprocessor synchronization primitives with priorities. In: Proceedings of the 1991 IFAC Workshop on Real-Time Programming, pp. 1–7 (1991)
11. Mellor-Crummey, J.M., Scott, M.L.: Algorithms for scalable synchronization on shared-memory multiprocessors. ACM Trans. Comput. Syst. **9**(1), 21–65 (1991)
12. Scott, M., Scherer III., W.: Scalable queue-based spin locks with timeout. In: Proceedings of the Eight Symposium on Principles and Practice of Parallel Programming, June 2001