A study of mutual exclusion algorithms using a long-lived splitter

Abstract
In this essay, a new fast-path mutual exclusion algorithm is presented, and is compared to two previously known fast-path mutual exclusion algorithm (including a discussion on a problematic aspect of one of the above algorithms). A more time efficient variant of the new algorithm (with black box calls to an adaptive collect object) is also presented.

1 Introduction and preliminaries
In the mutual exclusion (mutex) problem, a set of n processes repeatedly and concurrently try to access a specific section in the code, called the critical section, which should be accessed by at most one process at a time. The critical section is protected by an entry and an exit section, which are executed by a process before entering and after exiting the critical section, correspondingly. As the system is asynchronous, processes can't "wait" for each other, and the only assumptions are that a process doesn't remain in the critical or exit sections forever. Any code fragment which is not the critical section, or the entry or exit sections, is called the remainder section.

Several liveness properties exist in the context of mutual exclusion. One of the strongest liveness notions is that of no starvation. A mutex algorithm is said to insure no starvation (or to be starvation free), if it guarantees that if at some configuration a process is in its entry section, then at some later configuration, the same process is in its critical section.

The first mutual exclusion algorithms were designed with atomic read and write operations only, but several later algorithms use stronger primitive (e.g. read-modify-write). The circular-queue algorithm presented in class is an example of such an algorithm, which insures mutual exclusion and bounded waiting (the only liveness property stronger than no starvation, it guarantees there is always a bound on the amount of time a process waits from the moment it reaches its entry section, and until it's admitted into its critical section). Designing an "efficient" mutual exclusion algorithm which uses only atomic reads and writes remains an interesting problem to this day. Particularly, the performance of such algorithms usually goes down as n (the number of processes participating in the algorithm) increases, and (usually) a process interested in entering the critical section must take many steps, even in the absence of contention. Several mutex algorithms presented in class suffer from these disadvantages (e.g. the Bakery and the tournament algorithms). The aforementioned problems motivated a study of "fast path" algorithms (e.g. [1,2]), i.e. mutual exclusion algorithms with only \(O(1)\) operations needed to enter the critical section, for a process running with no contention. This is referred to later in this essay as the fast path property. Most of the fast-path algorithms

---

1 The assumption that a process always leaves the exit section, is known as unobstructed exit, and algorithms designed to solve the mutual exclusion problem should ensure this property is maintained.
use the notion of a splitter (first defined in [5], and here in fig. 4), used to split all competing processes into up to three groups: the processes deflected right, the processes deflected left, and the winner. The splitter guarantees the splitter property, namely that at most one process wins the splitter.

Fast path algorithms (also known as long-lived splitter algorithms) presented in previous work [1,2], vary according to their space complexity (e.g. one of the algorithms in [1] uses unbounded memory), time complexity in the presence of contention, and "algorithm complexity" (i.e. how intuitive is the algorithm and its correctness proof). In this essay, a new long-lived splitter is suggested, using linear memory, $O(1)$ operations for a process running with no contention, and at most $O(n)$ operations per process in the presence of contention (accumulating to potentially $O(n^2)$ operations over all the processes).

The rest of the essay is structured as follows: The new algorithm (alg. 1) is presented in section 2. Section 3 contains a discussion of the advantages and disadvantages of the new algorithm in comparison to two other long lived splitter algorithms. A formal correctness proof for the new algorithm is presented in section 4. Section 5 contains an example of a problematic execution for the long-lived splitter algorithm presented in [3]. A lower bound on the space complexity of mutual exclusion algorithms using the splitter notion is discussed in section 6. An improved algorithm (based on alg. 1, and trading time complexity for space complexity) is presented in section 7. Concluding remarks and open questions are given in section 8.
A long-lived splitter algorithm

shared variables:
\[ X : 0, \ldots, n-1 \]

\[ Y : boolean, \text{ initially } false \]

\[ Checking : boolean, \text{ initially } false \]

\[ InCount : \text{ array } [0, \ldots, n-1] \text{ of } boolean, \text{ initially } false \]

private variable
\[ i : 0, \ldots, n \]

process \( p \):
while (true) do
\[ 0 : \text{ non-critical section} \]
\[ 1 : X := p \]
\[ 2 : InCount[p] := true \]
\[ 3 : \text{ if } (Y = true) \text{ then } \text{Fail & Reset ( )} \]
\[ \text{else} \]
\[ 4 : \text{ if } (Checking) \text{ then } \text{Fail & Reset ( )} \]
\[ 5 : Y := true \]
\[ 6 : \text{ if } (X \neq p) \text{ then } \text{Fail & Reset ( )} \]
\[ \text{else} \]
\[ 7 : \text{ Entry}_2(0) \]
\[ 8 : \text{ critical section} \]
\[ 9 : Y := false \]
\[ 10 : InCount[p] := false \]
\[ 11 : \text{ Exit}_2(0) \]

procedure \text{ Fail & Reset ( )}:
\[ 12 : \text{ Entry}_n(p) \]
\[ 13 : \text{ Entry}_2(1) \]
\[ 14 : \text{ critical section} \]
\[ 15 : Checking := true \]
\[ 16 : InCount[p] := false \]
\[ 17 : \text{ for } (i = 0; i < n; i++) \]
\[ \text{ if } (InCount[i]) \text{ then break} \]
\[ 18 : \text{ if } (i = n) \text{ then } \]
\[ \quad Y := false \]
\[ 19 : Checking := false \]
\[ 20 : \text{ Exit}_2(1) \]
\[ 21 : \text{ Exit}_n(p) \]

Figure 1: A long-lived splitter algorithm (Algorithm 1)

Discussion

Alg. 1 is based on Anderson\&Kim’s unbounded algorithm presented first in [1] (and here in fig. 2), but shares many of its properties with Anderson\&Yang’s algorithm, suggested in [2] (and here in fig. 3). A formal correctness proof for the proposed algorithm is given in section 4, but first the differences between the algorithms are discussed. The main disadvantage of Anderson\&Kim’s algorithm, is the fact that \( Y.indx \) is unbounded, and therefore each time the fast path is re-opened, a new set of indices is used. Consequently, the \NameTaken array uses unbounded memory. Anderson\&Kim suggest an algorithm with bounded variables (fig. 4 in [1], not presented here), using mod \( n \) calculations, which does solve the problem of
unbounded memory, but still uses \( n \) splitter "generations" and is fairly complicated. The algorithm presented in fig. 1 reflects a different approach – there is only one splitter code fragment (only one pair of \( X,Y \) variables without splitter "generations"), which is re-opened by the process in the fast path (if such a process exists) or by the "last" process that leaves \( \text{Fail} \& \text{Reset}(\ ) \). The proposed algorithm uses less variables than Anderson&Kim's algorithm with bounded variables (only one \( n \)-bit array, instead of 2) and simpler data structures (no unique types used), and the correctness proof also seems simpler. Nonetheless, the proposed algorithm has one main drawback – a process re-opening the splitter from \( \text{Fail} \& \text{Reset}(\ ) \) will have to read the entire \( \text{InCount} \) array \( (n \text{ steps}) \) before it can reset the splitter. This means the number of operations performed in \( \text{Fail} \& \text{Reset}(\ ) \) is potentially \( O(n) \) even if the underlying \( n \)-process mutex algorithm requires only \( O(\log n) \) operations, e.g. the tournament algorithm (notice that the additional checks appear after the critical section, a "pay when you leave" policy). In Anderson&Kim's algorithms these extra checks aren't needed, as a process may reset the splitter even if other processes are still in \( \text{Slow}1(\ ) \) or \( \text{Slow}2(\ ) \), since new processes reaching line 1 after the splitter has been closed, will compete in a different "generation" of the splitter.

As mentioned above, the algorithm in fig. 1 shares much of the structure of the algorithm in fig. 2. However, there are several crucial differences which are now explained. As there are no splitter "generations", no \( \text{indx} \) field is used and no \( \text{Name-Taken} \) array is necessary (these were used in alg. 2 to mark which splitter "generations" have already been used, and what the current generation is). Instead, the \( \text{InCount} \) array is used to (roughly speaking) "count" the number of processes in the splitter or in the fast or slow paths. \( \text{InCount}[p] \) is set by process \( p \) after it has influenced the splitter (specifically, after it set \( X \) ) and reset only after exiting the critical section in the fast or slow paths. By checking the entries of \( \text{InCount} \), the last process to leave \( \text{Fail} \& \text{Reset}(\ ) \) can notice it is the last to execute \( \text{Exit}_n \) (in this contention period), and reset the splitter. The \( \text{Checking} \) variable, similar to the \( \text{Choosing} \) variable in the Bakery algorithm\(^2\), is used to "protect" the scanning of \( \text{InCount} \) array, which is updated outside the critical section. To see why this precaution is necessary, consider the following scenario: assume \( n \geq 4 \) and 2 processes \( p_1, p_2 \) compete in the splitter, \( p_1 \) wins the fast path and \( p_2 \) is deflected to \( \text{Fail} \& \text{Reset}(\ ) \) (there are several scenarios in which this is possible, for example if \( p_2 \) executes lines 1-2, then \( p_1 \) executes lines 1-6, followed by \( q \) that executes line 3). \( p_1 \) enters the critical section, opens the splitter (line 9) and leaves the fast path (resetting \( \text{InCount}[p] \) at line 10). Then, \( p_2 \) enters the critical section, and scans \( \text{InCount}[1],... \text{InCount}[4] \), finding them all false. Another process \( p_3 \) executes line 1-6 (remember line 4 is missing since \( \text{Checking} \) isn’t used).

\(^2\) In the Bakery algorithm, the \( \text{Choosing} \) variable is used to insure that when a process \( p \) is choosing its number, no other process is checking \( p \) ’s number.
type Ytype = record free : boolean; index: 0,..,\infty end

shared variables:
X : 0,.., n-1
Y, Reset : Ytype, initially (true, 0)
Name_Taken : array[0,.., \infty] of boolean, initially false
InFast : boolean, initially false

private variables
y : Ytype
i : 0,.., n

process p :
while (true) do
0: non-critical section
1: X := p
2: y := Y
   if (-y.free) then SLOW1( )
   else
3: Y := (false, y.idx)
4: if (X \neq p)
5: \quad \lor InFast) then SLOW2( )
   else
6: Name_Taken[y, idx] := true
7: if Reset \neq y then
8: \quad Name_Taken[y, idx] := false
   \quad SLOW2( )
   else
9: InFast := true
10: Entry_2(0)
11: \quad \text{critical section}
12: \quad Reset := (true, y.idx +1)
13: \quad Y := (true, y.idx +1)
14: Exit_2(0)
15: InFast := false

procedure SLOW1( )
16: Entry_n(p)
17: Entry_2(1)
18: \text{critical section}
19: Exit_2(1)
20: Exit_n(p)

procedure SLOW2( )
21: Entry_n(p)
22: Entry_2(1)
23: \text{critical section}
24: y := Reset
25: Reset := (false, y.idx)
26: if (-Name_Taken[y.idx]) then
27: \quad Reset := (true, y.idx +1)
28: \quad Y := (true, y.idx +1)
29: Exit_2(1)
30: Exit_n(p)

Figure 2: Anderson&Kim's long-lived splitter algorithm with unbounded variables (Algorithm 2)
As \( Y = false \) ( \( p_1 \) reset it), \( p_3 \) will enter the fast path. Now, \( p_2 \) runs again, finishes scanning \( \text{InCount} \), and resets the splitter. Then a forth process \( p_4 \) runs and sets \( X = p_4 \) at line 1. As \( Y = false \), \( p_4 \) isn't deflected to the slow path at lines 4 or 6, and enters the fast path, resulting in the existence of two processes in the fast path at the same time! When using \( \text{Checking} \), this scenario is impossible, as is proven in section 4.

The algorithm in fig. 1 doesn't use a \( \text{Reset} \) variable, which in Anderson&Kim's algorithm points at the splitter generation currently in use, and is also used to deflect processes, using an "old" splitter, to the slow path. As the proposed algorithm doesn't use splitter "generations", \( \text{Reset} \) becomes obsolete. Similarly, alg. 1 doesn't use an \( \text{InFast} \) variable, since there is a way of identifying the location of a process (outside the remainder), by reading its \( \text{InCount} \) entry, and the additional checks in the splitter (line 4). The fact that in alg. 1 the splitter is open when \( Y = false \) (in alg. 2 and in other traditional splitter algorithms the splitter is open when \( Y = true \)) has no profound reason, but it seems more appropriate, since the splitter is said to be reset when it is re-opened.

It is important to note, that the proposed algorithm still possesses the property of "splitting the losers", but the algorithm in fig. 1 doesn't take advantage of this trait. It may seem that a process \( p \) deflected to \( \text{Fail} \& \text{Reset}(\ ) \) at line 3 can safely return to the remainder without trying to re-open the splitter, since some other process \( q \) must have reached line 4 before \( p \) executed line 3, and \( q \) would therefore attempt to re-open the splitter. It is an interesting question whether there is a long-lived splitter algorithm that doesn't use splitter "generations", in which processes deflected from the splitter at line 3 don't try to re-open the splitter. However, splitting the losers into \( \text{Slow1}(\ ) \) and \( \text{Slow2}(\ ) \), and having processes in \( \text{Slow1}(\ ) \) make no attempt to reset the splitter (as is the case in alg. 2) may leave the splitter closed when a contention period ends. To see why, suppose at line 3 of alg. 1 a process would be deflected to some procedure \( \text{Fail}(\ ) \) in which it doesn't try to reset the splitter, and consider the next execution: \( p \) executes lines 1-5, followed by \( q \) that executes lines 1-2. Then \( p \) executes line 6 and is deflected to \( \text{Fail} \& \text{Reset}(\ ) \), reads \( \text{InCount}[q]=true \) (line 17) and therefore leaves \( \text{Fail} \& \text{Reset}(\ ) \) without re-opening the splitter. Then \( q \) executes line 3 and is deflected to \( \text{Fail}(\ ) \), from which \( q \) will return to the remainder without re-opening the splitter.

As mentioned above, alg. 1 shares many of the properties of Yang&Anderson's algorithm. The \( \text{Checking} \) variable is equivalent to the \( Z \) variable of alg. 3, and \( \text{InCount} \) and \( B \) also serve a similar purpose, which is to mark (when all array entries are false) that a slow process may (and should) re-open the splitter. However, \( \text{InCount}[p] \) is set after \( p \) has influenced the splitter (set \( X \)), while \( B[p] \) is set only if \( p \) changed \( Y \)'s value uninterruptedly (i.e. no other process has set \( X \) in the meantime). This seems like a crucial difference, as setting \( \text{InCount} \) right after \( X \) is updated, prevents the following improvement of alg. 1, in which processes deflected to the slow path at line 3 don't attempt to re-open the splitter. However, a more thorough examination of alg. 3 shows it is susceptible to similar executions, that may leave the splitter closed at the end of a contention period. This topic is addressed in more detail in section 5.

As already noted, in alg. 3 a slow process attempts to re-open the splitter (and particularly scans \( B \)) only if \( X = p \), while in alg. 1 all slow processes scan \( \text{InCount} \). Therefore, when there is high contention, only one
slow process will scan $B$. However, alg. 1 allows a slow process to re-open the splitter even when other processes are competing within it, which means the splitter (and consequently the fast path) may be opened more often. This may have positive effects on the performance when $n$ is asymptotically small: if $n \approx \log n$, it is better to have the splitter open more often, resulting in more processes having an opportunity to enter the fast path, while the extra checks in the slow path don't greatly influence the total step complexity of a process competing in the n-process mutex algorithm.

shared variables:
$B$: array $[0,..,n]$ of boolean, initially false
$X,Y: -1,0,..,n-1,Y$ initially $-1$
$Z$: boolean, initially false

private variables
$flag$: boolean
$i: 0,..,n$

```
process p:
while (true) do
0: non-critical section
1: $X := p$
2: if ($Y \neq -1$) then SLOW ( )
3: $Y := p$
4: if ($X \neq p$) then SLOW ( )
5: $B[p] := true$
6: if ($Z = true$) then SLOW ( )
7: if ($Y \neq i$) then SLOW ( )
8: $Entry_2(0)$
9: critical section
10: $Exit_2(0)$
11: $Y := -1$
12: $B[p] := false$
13: goto 0
14: $Entry_n(p)$
15: $Entry_2(1)$
16: critical section
17: $B[p] := false$
18: if ($X = p$) then
19: $Z := true$
20: $flag := true$
21: $i := 0$
22: while ($i < n$) do
23: if ($B[i] = true$) then $flag := false$
24: $i := i + 1$
25: if ($flag = true$) then $Y := -1$
26: $Z := false$
27: $Exit_2(1)$
28: $Exit_n(p)$
```

Figure 3: Anderson\&Yang’s long-lived splitter algorithm (Algorithm 3)

Another notable difference, is the fact that alg. 1 presents the splitter code as presented in [2], while alg. 3 uses a somewhat different splitter code, which resembles the splitter algorithm suggested in [1]. Particularly, in alg. 3 $Y$ is not binary, which is crucial for the correctness of the algorithm. If $Y$ would have been binary
(i.e. $-1$ corresponds to the value "true" and all other values correspond to the value "false"), then the next (faulty) execution would have been possible for $n \geq 4$ processes: $p'$ executes lines 1-2, followed by $p$ that executes lines 1-4. Then $q'$ executes lines 1-2, is deflected to Slow() (since $p$ set $Y$ at line 3. Remember all values other than $-1$ represent the value "false"), enters and exits the critical section, and particularly re-opens the splitter (by setting $Y$, which is binary by assumption) before returning to the remainder (since $B[i] = false$ for every process $i$, as $p$ didn't execute line 5 yet). Some forth process $q$ executes lines 1-7 and enters the fast path (since it finds $Y = true$ at line 2), followed by $p$ that executes lines 5-7 and also enters the fast path (since it reads $Y = false$ at line 7, as $Y$ is binary). Therefore, at some configuration $p,q$ are both in the fast path at the same time, i.e. the splitter property is violated. This execution is impossible in alg. 1, since InCount is updated before reading $Y$. The early updating of InCount allows the use of a binary $Y$ (among other things), and as a consequence, the omission of the condition at line 7 (in alg. 3). In alg. 3, line 7 is necessary in order to prevent the next execution: $p$ executes lines 1-2, followed by $q$ that executes lines 1-4. Then a third process $p'$ executes lines 1-2, is deflected to Slow(), enters and exits the critical section, and particularly re-opens the splitter (since $B[i] = false$ for every process $i$). Then $q$ executes lines 5-6 and enters the fast path (remember line 7 is missing), followed by a forth process $q'$ that executes lines 1-6 and enters the fast path (since it is the last process that wrote to $X$, and $Y$ was reset to $-1$ by $p'$), therefore $q,q'$ are both in the fast path in the same configuration, in violation to the splitter property.

4 Correctness proof for Alg. 1

This section contains a formal correctness proof for alg. 1. Specifically, it is proven that the mutual exclusion property holds, the algorithm insures no starvation, and a process running with no contention enters the critical section in $O(1)$ steps.

A process is said to be in the fast path or simply fast if it is between lines 7-11. It is in the slow path, or slow, if it is between lines 12-21. The splitter is said to be reset when some process resets $Y$ at lines 9 or 18. The splitter is open iff $Y = false$.

Lemma 4.1: If the underlying n-process mutex algorithm insures mutual exclusion, then when some process $p$ is between lines 16-18, Checking =true.

Proof: When $p$ is between lines 16-18, it is after it set Checking at line 15 and before it reset Checking at line 19. Therefore, if no other process writes to Checking in between, then Checking =true. Checking is written to only in Fail & Reset(), after Entry$_a(p)$ is invoked and before Exit$_a(p)$ is invoked. Therefore, the mutual exclusion property of the underlying n-process mutex algorithm guarantees no other process will write to Checking before $p$ executes line 19. ∎
Note 4.2: If the underlying n-process mutex algorithm insures mutual exclusion, then at every configuration, at most one process is between lines 13-20. Particularly, \( \text{Entry}_2(1) \), \( \text{Exit}_2(1) \) are never simultaneously invoked, and each of these procedures is never simultaneously invoked by different processes.

Note 4.3: Assume at some configuration \( C \), two processes \( p, q \) are in the fast path. Assume WLOG that \( q \) was the last of the two to execute line 6. Then at some configuration \( C' \) before \( C \), \( p \) was in the fast path, and \( q \) was between lines 4-6 (e.g. when \( q \) executed line 6).

Lemma 4.4: If the underlying 2- and n-process mutex algorithms insure mutual exclusion, and at some configuration \( C \), two processes \( p, q \) are between lines 4-6, or one of them is in the fast path and the other is between lines 4-6, then at least one of them will be deflected to \( \text{Fail & Reset} \).

Proof: By induction on \( k \), the number of times the splitter has been reset before the last of \( p, q \) executed line 3.

**Basis:** \( k=0 \), the splitter has never been reset. If at least one of \( p, q \) doesn't reach line 6, then that process was deflected to \( \text{Fail & Reset} \) (at lines 3 or 4) and won't enter the fast path. Otherwise, assume WLOG that \( p \) executes line 6 first. If either \( p \) or \( q \) reads \( X \not= p \) or \( X \not= q \) (respectively) when it executes line 6, then it would be deflected to \( \text{Fail & Reset} \) as claimed. Otherwise, \( p \) read \( X = p \) and \( q \) reads \( X = q \) at line 6. \( p, q \) are the only processes that write the values \( p, q \) to \( X \) (respectively), and only at line 1. Since \( p \) executed line 6 first, this means \( q \) executed line 1 after \( p \) executed line 6, and particularly \( q \) read \( Y \)'s value at line 3 after \( p \) set it at line 5. As the splitter has never been reset before \( q \) executed line 3, this means \( q \) must have read \( Y = \text{true} \) and was deflected to \( \text{Fail & Reset} \) (particularly, \( q \) didn't enter the fast path).

**Step:** assume the lemma is correct for the first \( k \) times the splitter was reset. Particularly, if at some configuration \( C' \) before \( C \) the splitter has been reset at most \( k \) times, then note 4.3 and the induction hypothesis imply that at \( C' \), at most one process was in the fast path (i.e. \( \text{Entry}_2(0) \), \( \text{Exit}_2(0) \) weren't invoked simultaneously by different processes).

Assume that the splitter has been reset \( k+1 \) times before the last of \( p, q \) executed line 3 for the last time before \( C \) was reached. Assume in negation both \( p, q \) entered the fast path. Particularly, both executed line 6, and let \( p \) be the process that executed it first. As explained in the basis case, if both entered the fast path then the fact that \( p \) executed line 6 first implies that \( p \) also set \( Y \) at line 5 before \( q \) executed line 3. Since \( q \) wasn't deflected to \( \text{Fail & Reset} \) at line 3, \( q \) must have read \( Y = \text{false} \). Therefore, some process must have reset \( Y \) after \( p \) set it at line 5 and before \( q \) executed line 3, and let \( p' \) be the last process to do so. The splitter is reset only at lines 9 or 18, but the first case is impossible: when \( p' \) reset the splitter, the splitter had already been reset at most \( k \) times, and \( p \) was either in the fast path or at line 6 (since \( p \) already executed line 5 and hasn't left the fast path yet). Therefore, if \( p' \) reset the splitter in configuration \( C' \) by executing line 9, then at \( C' \), either \( p' \) was in the fast path and \( p \) at lines 4-6, or both were in the fast path, and before \( C' \) the splitter has been reset at most \( k \) times. Consequently, when the last one of \( p, p' \)
executed line 6, the other was already in the fast path, a contradiction to the induction hypothesis since both processes entered the fast path.

In the second case, \( p' \) must have reset the splitter in \( \text{Fail \& Reset}(\ ) \), which means \( p' \) read \( \text{InCount}[p] = \text{false} \) at line 17 (otherwise it would break from the loop, and wouldn't have reset the splitter at line 18 since \( i \neq n \)). Therefore, either \( p' \) read \( \text{InCount}[p] \) at line 17 before \( p \) executed line 2; or \( p' \) read \( \text{InCount}[p] \) after \( p \) reset it at line 10 (these are the only possible cases since \( p \) is the only process that writes to \( \text{InCount}[p] \)). In the former case, \( p' \) executed line 15 before \( p \) executed line 2, and as noted above, \( p' \) reset \( Y \) at line 18 only after \( p \) executed line 5, which means when \( p \) read \( \text{Checking} \) at line 4, it was true (according to lemma 4.1, since \( p' \) was between lines 16-18) and therefore \( p \) was deflected to \( \text{Fail \& Reset}(\ ) \) at line 4 (a contradiction to the assumption that \( p \) entered the fast path). In the latter case, at any configuration \( C'' \) before \( C' \), at most one of \( \text{Entry}_{2}(0), \text{Exit}_{2}(0) \) was invoked, and by at most one process (as noted above). The same is true for \( \text{Entry}_{2}(1), \text{Exit}_{2}(1) \), as follows from note 4.2. Therefore, the mutual exclusion property of the underlying 2-process mutex algorithm guarantees that at every configuration, at most one process can be between lines 8-10 or 14-19. As \( p' \) executed line 17 only after \( p \) executed line 10, it follows that \( p \) left the fast path (i.e. executed line 11) before \( p' \) executed line 18 and particularly before \( q \) executed line 3 (\( q \) executed line 3 after \( p' \) executed line 18), a contradiction to the choice of \( p,q \).

Claim 4.5 (mutual exclusion): The algorithm insures mutual exclusion, if the underlying 2- and \( n \)-process mutex algorithms insure mutual exclusion.

Proof: It is enough to prove that at any configuration, at most one process is in the fast path (i.e. the splitter property holds), because in such a case the mutual exclusion property of the underlying 2- and \( n \)-process mutex algorithms guarantees mutual exclusion.

Assume in negation the algorithm doesn’t ensure the splitter property, and let \( C \) be the first configuration in which two processes \( p,q \) are both in the fast path. Particularly, both processes executed line 6 before \( C \) was reached, and hadn't executed line 11 yet. Let \( q \) be the last of them to execute line 6. Then when \( q \) executed line 6 for the last time before \( C \) was reached, \( p \) was already in the fast path. By lemma 4.4, \( q \) would be deflected to the slow path, a contradiction to the choice of \( p,q \).

Claim 4.6 (no starvation): The algorithm insures no starvation, assuming the underlying 2- and \( n \)-process mutex algorithms insure no starvation.

Proof: Any process \( p \) leaving the remainder section, will either reach \( \text{Fail \& Reset}(\ ) \) (if it is deflected from the fast path at lines 3, 4, or 6) or will reach the fast path. In the former case, \( p \) will compete in the \( n \)-process mutex algorithm, and the no starvation property of the \( n \)-process mutex algorithm ensures at some
later configuration \( p \) will execute line 13 and compete in the 2-process mutex algorithm. By no starvation of the underlying 2-process mutex, at some later configuration \( p \) will enter the critical section. In the latter case, \( p \) will reach line 7 and compete in the 2-process mutex algorithm, which guarantees (by no-starvation of the 2-process mutex) at some later configuration \( p \) will enter the fast path.

**Lemma 4.7:** If at some configuration \( C \) process \( p \) is in the remainder (line 0), then \( InCount[p] = false \) (actually, \( InCount[p] = false \) even if \( p \) is at Exit\(_2\) or Exit\(_n\)). Moreover, If some configuration \( C \) is quiet, then at \( C \) \( InCount[p] = false \) for all processes \( p \).

**Proof:** If \( p \) is in the remainder (or at Exit\(_2\) or Exit\(_n\)), then either it hasn't left the reminder (line 0), or it was in the critical section and returned to the remainder after executing Exit\(_2\). In the former case, since only \( p \) writes to \( InCount[p] \) (and doesn't write to it in the remainder) and it is initialized to false, \( InCount[p] = false \). In the latter case, before reaching Exit\(_2\), \( p \) executed line 10 (if it was fast) or line 16 (if it was slow) and reset \( InCount[p] \). If \( C \) is a quiet configuration, then in \( C \) all processes are in the remainder section, and therefore, as has already been established, \( InCount[p] = false \) for all processes \( p \).

**Lemma 4.8:** If \( C \) is a quiet configuration, then in \( C \) the splitter is open and Checking = false.

**Proof:** Let \( q \) to be the last process that enters the critical section before \( C \) is reached. In order to show that in \( C \) the splitter is open, it is enough to prove that before returning to the remainder, \( q \) resets the splitter. This will suffice because the splitter is set only at line 5, and the definition of \( q \) guarantees that when \( q \) executes Exit\(_2\), all other processes outside the remainder have already executed Exit\(_2\), and particularly won't set \( Y \).

\( q \) can be either fast of slow. Assume first \( q \) is slow. When \( q \) executes line 17, all other processes are either in the remainder, or have already executed Exit\(_2\). Therefore according to lemma 4.7, \( InCount[p] = false \) for all processes \( p \) (including \( q \), as it reset \( InCount[q] \) at line 16 and \( q \) is the only process that writes to \( InCount[q] \)). Consequently, \( q \) won't break from the loop at line 17, the condition at line 18 will hold and \( q \) will reset the splitter.

If \( q \) is fast, then it will reach line 9 and reset \( Y \) (since no process remains in the critical section forever). As no other process writes to \( Y \) after \( q \) (at least until \( C \) is reached), in \( C \), \( Y = false \).

**Checking** is initialized to false, and is written to only at lines 15 and 19. Consequently, if no process is deflected to **Fail & Reset** before \( C \) is reached, then no process writes to **Checking** and in \( C \)
Checking = false. Otherwise, at least one process was slow before \( C \) was reached. Every slow process must execute line 19 before it returns to the remainder, and let \( q \) be the last process that executes line 19. Then \( q \) resets Checking, and no other process sets Checking after that (at least until \( C \) is reached), because if some process \( p \) sets Checking at line 15 after \( q \) resets it, then \( p \) will execute line 19 before \( C \) is reached (as \( C \) is quiet, in \( C \) \( p \) is in the remainder) and after \( q \) executed line 19, a contradiction to the choice of \( q \).

Claim 4.9 (fast path): In the absence of contention, a process takes \( O(1) \) steps before entering the critical section, and \( O(1) \) steps when exiting it (Assuming it takes only \( O(1) \) steps to enter or exit the critical section in the underlying 2-process mutex algorithm).

Proof: Let \( p \) be a process running with no contention. According to lemma 4.8, when \( p \) executes lines 1-4, the splitter is open and Checking = false (this is true before \( p \) reached line 1, and \( p \) didn’t close the splitter yet). Particularly, \( p \) executes lines 3 and 4 without being deflected to Fail & Reset\( (\quad) \). \( p \) reads \( X = p \) at line 6 (\( p \) updated \( X \) at line 1 and no other process wrote to \( X \) after that, since \( X \) isn’t written to in the remainder). Therefore, \( p \) isn’t deflected to Fail & Reset\( (\quad) \) at line 6 and reaches line 7. All this is done in \( O(1) \) operations, since there are no loops in lines 1-6. Once \( p \) reaches line 7, there are only 2 non-loop operations it performs on top of Entry\(_2\), Exit\(_2\), so \( p \) takes only \( O(1) \) steps before entering the critical section, and only \( O(1) \) steps when leaving it.

Note 4.10: Time complexity in the presence of contention. In the presence of contention, a process may be deflected to Fail & Reset\( (\quad) \), and when exiting the critical section will try to reset the splitter. The splitter is reset after the entire InCount array is scanned and all entries are found to be false. Therefore, a slow process may potentially take \( O(n) \) additional steps when exiting the fast path, and if all \( n \) processes are deflected to the slow path (which is possible and is discussed in section 6), the total number of additional steps taken is potentially \( O(n^2) \). However, it seems likely that at least when there is high contention, most slow processes will break from the loop at line 17 (since when there is high contention, most InCount entries are true). Furthermore, it is possible to add a check whether the splitter is closed (i.e. \( Y = false \)) before a slow process scans the InCount array, which may slightly improve the asymptotic time complexity, but makes the correctness proof trickier. Additional time complexity improvements for alg. 1 are discussed in section 7.
5 A counter example for Anderson & Yang's algorithm
As noted in section 3, Anderson&Yang's algorithm (alg. 3) doesn't always ensure a process running with no contention will enter the fast path in $O(1)$ steps. In this section, the formal proof of this fact is given.

Claim 5.1: If $n \geq 3$ then Anderson&Yang's algorithm doesn't guarantee $O(1)$ operations before entering the fast path, for a process running with no contention.

Proof: It is enough to show that a contention period might end (i.e. some quiet configuration $C$ is reached) with a closed splitter ($Y \neq -1$), because if at configuration $C$ some process is scheduled to run with no contention, it will be deflected to the slow path at line 2, and then there is no guarantee that it'll take only $O(1)$ steps before entering the critical section (it depends on the underlying n-process mutex algorithm).

Consider an execution, in which some process $p$ executes lines 1-11 ($p$ is the first to run, and runs with no contention, therefore it'll enter the fast path (lines 8-10)) and particularly resets the splitter ($Y \leftarrow -1$). Then some other process $q$ executes lines 1-3 and sets $Y = q$, followed by $q'$ that executes lines 1-2 and is deflected to slow (since $Y = q \neq -1$). Then $q$ executes line 4 and is deflected to Slow($X$) (since $X = q' \neq q$).

Next, $q$ is scheduled to run until it enters and exits the critical section, and returns to the remainder ($q$ doesn't attempt to reset the splitter, since when it executes line 18, $X \neq q$). Then $q'$ is scheduled to run until it enters the critical section, and reaches line 18. Since $X = q'$, $q'$ tries to reset the splitter (by executing lines 19-26), but when it reads $B[p] = true$ at line 23 (remember $p$ set it at line 5 and hasn't executed line 12 yet. As $p$ is the only process that writes to $B[p]$, it is still true), it resets the flag and consequently skips line 25 and doesn't reset the splitter. $q'$ returns to the remainder, and then $p$ resets $B[p]$ and returns to the remainder, which means a quiet configuration has been reached, and the splitter is closed. □

Note: The proof of claim 5.1 uses one main fact, which is that a fast process resets the splitter (and it's array entry in $B$) after leaving the "safe zone" created by the critical section (i.e. after executing $Exit_2$). Therefore, claim 5.1 is true regardless of identity of the underlying 2- and n-process mutex algorithms. In other known long-lived splitter algorithms (e.g. alg. 2) the splitter is always reset "near" the critical section. Particularly, in alg. 1 a fast process opens the splitter before it executes $Exit_2$, which makes alg. 1 "immune" to such executions as presented in the proof of claim 5.1.

6 A lower bound for long-lived splitter algorithms
As discussed in section 3, every slow process in Alg. 1 checks whether it should reset the splitter, by scanning the $InCount$ array. Therefore, every slow process may potentially take $O(n)$ steps when exiting the critical section. It is well known that slow processes must be able to reset the splitter. Otherwise, an execution in which all processes are slow (e.g. an execution in which a process $p$ executes lines 1-3 of the alg. in fig. 4, followed by a second process $q$ that executes lines 1-2, and then $p$ executes line 4. In this execution both
process are slow) may leave the splitter closed, and if some process $p'$ is scheduled to run with no contention (after all the slow processes have returned to the remainder section), then $p'$ will also be deflected to the slow path (since the splitter is closed), and particularly $p'$ may take more than $O(1)$ steps before it reaches the critical section, in violation to the fast path property. However, it is not known whether the extra $O(n)$ operations in $Fail & Reset( )$ are necessary. Particularly, it is possible that in a solution to the mutual exclusion problem with a fast path, only a fraction of the slow processes (preferably only one of them) attempts to reset the splitter; or a process may reset the splitter without reading $O(n)$ registers first.

As is established in this section, under certain assumptions $O(n)$ registers are needed. This may serve to explain the intuition concerning the use of an $InCount$ array in Alg. 1.

This section contains the formulation of the lower bound on the number of registers. It is left open whether a slow process must read all these registers before resetting the splitter, and if the $O(n)$ lower bound may be improved if some of the assumptions are removed.

Define a splitter fragment to be the fraction of code which appears in Fig. 4, using one $X$ and one $Y$ parameter. Kim&Anderson's unbounded long-lived splitter algorithm (Fig. 2), potentially uses infinitely many splitter fragments, and the bounded version as appears in [1] uses $n$ splitter fragments.

\begin{verbatim}
(Y is initially true)
1: X := p
2: if (¬Y) then FailRight( )
   else
3: Y := false
4: if (X ≠ p) the FailLeft( )

(in Fast path)

Figure 4: Splitter code
\end{verbatim}

Claim 6.1 (lower bound): Let $\mathcal{A}$ be a long-lived splitter algorithm insuring no starvation for $n ≥ 4$ processes. Assume $\mathcal{A}$ contains only one Splitter fragment, in addition to multi-reader-single-writer registers and Boolean multi-reader-multi-writer registers. Assume also that if some variable other than $X,Y$ is update within the splitter code, its value isn't read inside the splitter code. Under these assumptions, in $\mathcal{A}$ each process $p$ should have its own "update" variable.

Note 6.2: It is informally assumed that processes don't use the information they've obtained from reading "update" variables, or from $X,Y$, in order to "spread the knowledge" between other processes. Particularly, if some process $p$ doesn't have its own "update" variable, and some other process $q$ can detect that $p$ is

\footnote{Notice that the splitter code here is as appears in [1], i.e. the splitter is open when $Y = true$. Claim 6.1 holds also for algorithms in which the splitter code is reversed (and the splitter is open when $Y = false$).

\footnote{I.e an operation that updates some new variable is added between lines 1-4 of fig. 4.}

\footnote{Intuitively, $p$'s variable will hold information that will indicate $p$'s location in the code execution.}
not in the remainder section, \( q \) won’t update any variable in any way that may "inform" other processes about \( p \)’s location in the code execution.

**Proof outline:** As there is no assumption on the structure of the algorithm (except that it contains all the commands in the splitter code as appears in fig. 4, in their original order), there may be additional commands between the original splitter commands. As these new commands can’t be referred to, whenever line numbers are mentioned they refer to the original splitter code as appears in fig. 4. Terms such as "\( q \) runs until it reaches line \( x \)" means \( q \) is allowed to run and will execute all commands (including new ones) up to (not including) the command at line \( x \). Particularly, notice that if \( q \) has not yet executed line \( x \), then when \( q \) is scheduled to run again, the first command it executes is the one at line \( x \). If \( q \) is at line \( x \), then "\( q \) executes line \( x+1 \)" means \( q \) executes all the new commands between lines \( x \) and \( x+1 \) (if such commands exist), and also executes line \( x+1 \). As processes are using only atomic reads and writes, these terms will allow the construction of executions - in which 3 processes run - appear to a couple of these processes as if the third process isn't taking part in the execution. The "cost" is somewhat more cumbersome executions used in the proof, but as explained above, this is necessary if further assumptions on the code structure are to be avoided.

**Proof:** Assume in negation \( n-1 \) variables suffice. As these variables are single-writer, some process \( p \) doesn't write to an "update" variable. Therefore all the variables that \( p \) may write to are Boolean. Consider the next executions: In execution \( A \), \( q \) leaves the remainder and executes line 1, then some other process \( q' \) runs until it reaches line 1, followed by a third process \( p \), that runs until it reaches line 1. Next, \( p \) executes line 1, \( q \) runs until it reaches line 2, \( p \) executes line 2, \( q \) reaches line 3, and \( p \) executes line 3. Then, \( q \) reaches line 4, \( p \) executes line 4 (and enters the fast path), \( q' \) executes line 1, followed by \( q \), that executes line 4 and is deflected to \( \text{FailLeft}(\text{x}) \) since it reads \( X = q' \) (\( q' \) was the last to write to \( X \) at line 1). Finally, \( q' \) executes line 2 and is deflected to \( \text{FailRight}(\text{x}) \).

In execution \( B \), \( q \) runs until it reaches line 4, followed by \( q' \), that executes line 1. Then \( q \) executes line 4 and is deflected to \( \text{FailLeft}(\text{x}) \), and \( q' \) executes line 2 and is deflected to \( \text{FailRight}(\text{x}) \). Notice that in both executions, both \( q,q' \) have no idea that \( p \) has left the remainder, as the cautious way in which both executions were built insures that whenever \( p \) might have updated some variable, it is after the other processes read it’s value, and right after \( p \) writes to it, the value is overwritten by some other process (remember that there are only Boolean variables (aside for \( X \)), and any added variable that is updated within the splitter fragment isn’t read in the splitter fragment).

Let \( A' \) be some execution following \( A \), in which only \( q,q' \) are scheduled to run until both return to the remainder section. \( q' \) can't reset the splitter in \( A' \), otherwise if some other process \( p' \) is scheduled to run without interruption (after \( q,q' \) returned to the remainder), it will pass line 2 (since \( q' \) reset the splitter, \( Y = \text{true} \)) and line 4 (since \( p' \) was the last to execute line 1). Therefore, both \( p \) and \( p' \) will be in the fast path at the same time, in contradiction to the splitter property. As \( q' \) can’t distinguish between executions

---

\(^{6}\) E.g. if \( p \) resets \( X \) before exiting the fast or slow path, and \( q \) finds that \( X = p \), then \( q \) knows that \( p \) executed line 1 and has not left the fast\slow path yet.
A, B, q’ won’t reset the splitter in any execution following execution B, in which only q, q’ are scheduled to run. Consequently, q must reset the splitter in all such executions that follow execution B. Otherwise, consider an execution which starts with execution B, followed by q, q’ running sequentially until both return to the remainder (without resetting the splitter). Then some process q” runs with no contention. As q” finds Y = false at line 2, it will be deflected to FailRight( ) and therefore is not guaranteed to take only O(1) steps before reaching the critical section, i.e. the fast path property is violated. As q can’t distinguish between executions A, B, q will reset the splitter in any execution following A.

Finally, consider an execution which starts with execution A, then q’, q are scheduled to run sequentially until both reach the remainder. As noted above, q will reset the splitter. Now, some forth process p’ is scheduled to run until it executes line 4. Since q reset the splitter (and no other process was scheduled to run after that, so Y has not been reset again), p’ finds Y = true at line 2 and X = p’ at line 4. Consequently, p’ will enter the fast path and both p, p’ will be in the fast path at the same time, a contradiction to the splitter property.

7 Trading time for memory
As noted in section 3, the main drawback of alg. 1 is the fact that regardless of the identity of the underlying n-process mutex algorithm used, a slow process will perform O(n) additional steps when leaving the slow path (the additional steps are needed to check if the slow process should (and can) reset the splitter). The question whether this can be improved by a complete transformation of the algorithm (i.e. not using an InCount array, or splitting the losers) is briefly discussed in section 8. However, the time complexity of alg. 1 can be improved without drastic changes to the algorithm structure (at the cost of additional space complexity). These changes are discussed in this section. As there are several related topics mentioned in this context (i.e. collect objects, renaming algorithms, adaptive algorithms, etc.), each subject is briefly reviewed, and formal definitions are omitted.

By making use of an adaptive collect object, alg. 1 can be transformed to an algorithm with a more time efficient slow path. The new algorithm (alg. 4) is presented in fig. 5, and is obtained from alg. 1 by replacing the updating and scanning of InCount array with black-box accesses to an adaptive collect object.

A collect object is designed to solves the following problem. n processes, each with its own single-writer-multi-reader register, wish to update the value of their register, or read the values of all the registers. A collect object implements two high-level operations: Update(val) called by process p writes the value val to p’s associated register; and Collect( ) collects the values of all n registers. An algorithm is said to be adaptive, if its efficiency (in time and memory) is a bounded function of the number of processes concurrently participating in the algorithm (as opposed to a function of n, the total number of processes). Thus, a collect object is adaptive, if its Update, Collect operations use f(k) memory and require g(k) time (“time” units are measured as the number of steps taken by a process. A step is an access to a remote memory reference).
Classic adaptive collect objects (e.g. [3]) make use of renaming algorithms as building blocks, which in turn use splitter algorithms. Therefore, it seems alg. 4 creates a circular dependency between long-lived splitter algorithms and long-lived adaptive collect algorithms. However, as stressed above, alg. 4 makes only a black box use of adaptive collect objects, and the reference to a specific implementation of such an object (i.e. the long-lived adaptive collect object of Afek, Stupp & Touitou in [3]) is needed only for the time and space complexity analysis of alg. 4.

Due to space limitations, the correctness proof for alg. 4 is omitted, and the main ideas of the proof are briefly sketched. The key to showing that the alg. is correct, is to show that if all processes are slow, then at least one of them will reset the splitter. This may not be the case only if the \textit{Collect} procedure returns an old value of $\text{InCount}[p]$ for some process $p$ (namely, if $\text{InCount}[p]$ is read as true, although $p$ has already left the fast\(\text{\#}\)slow paths). The above scenario is impossible, since values are only collected in the "safe zone" in the vicinity of the critical section, and due to the properties of the collect object (i.e. the \textit{Collect} procedure returns the value of the last \textit{Update} call that ended before it, or the value of some concurrent \textit{Update}; and for every process $p$ and every two sequential \textit{Collect} calls $C_1, C_2$, the value $C_1[p]$ was written not before $C_2[p]$). These guarantee that if a slow process $q$ doesn’t open the splitter, then either some other process is between lines 1-21 (and that process will later try to reset the splitter), or a process has left the fast path after $q$’s call to \textit{Collect} (in which case that fast process has already reset the splitter).

Once the fact that when a contentention period ends the splitter is open is established, it follows that the validity of the other properties is preserved. The mutual exclusion property holds just as in alg. 1, since the same protection measures are used to protect the splitter, and particularly \textit{Checking} is used in alg. 4 in the same way as in alg. 1 (namely to deflect a process to the slow path, if it updated its \textit{InCount} entry after some other process had already begun collecting the values in \textit{InCount}).

The no starvation property holds since the code leading to \textit{Entry}_2 is the same as in alg. 1, except for the call to $\text{InCount}.\text{Update}$ at line 2 (replacing a direct assignment of values in alg. 1). As this call is adaptive in the number of concurrent processes, the time complexity of the operation is bounded, and therefore starvation freedom is preserved. Moreover, the fast path property is kept, since in the absence of contention the \textit{Update} operation takes only $O(1)$ steps. Furthermore, the added time complexity of a slow process is adaptive in the number of concurrent processes. Excluding the operations at lines 16-17, a slow process takes only $O(1)$ additional steps in $\text{Fail} & \text{Reset}()$ (the loop at line 19 is local). Lines 16-17 contain a single \textit{Update} call and a single \textit{Collect} call, which are both adaptive (guaranteed by the underlying adaptive collect object). In fact, the adaptive collect object of [3] requires $O(n^2)$ space and the \textit{Update,Collect} procedure calls may take at most $O(k^4)$ time (where $k$ is the number of concurrent processes). The time

\[7\] A renaming algorithm solves the m-renaming problem, for some parameter $m$. in the m-renaming problem, each one of $k$ processes ($k \leq m < n$) is required to choose a distinct name in $\{0, 1, \ldots, m-1\}$.

\[8\] i.e. additional to \textit{Entry}_2, \textit{Exit}_2, \textit{Entry}_n, \textit{Exit}_n, whose time complexity depends on the time complexity of the underlying 2\(\text{\#}\)n-process mutex algorithms.
complexity can be improved to \( O\left(k^2\right) \) if Attiya’s adaptive long-lived Collect object [4] is used, instead of the collect object of [3].

type \textit{Item} = \text{record pid : 0,..,n−1; val : boolean;}

\hspace{1cm} \text{timestamp : 0,..,}\infty \text{ end}

\text{shared variables:}

\text{X : 0,..,n−1}

\text{Y : boolean, initially false}

\text{Checking : boolean, initially false}

\text{InCount : adaptive collect object holding type Item}

\text{private variable}

\text{i : 0,..,n}

\text{inCount : array[0,..,n−1] of Item, initially inCount\left[i\right] = (i, false, 0)}

\text{process p :}

\text{while (true) do}

0: \text{non-critical section}

1: \text{X := p}

2: \text{InCount.Update\left(true\right)}

3: \text{if (Y = true) then Fail & Reset\left(\right)}

\hspace{1cm} \text{else}

4: \text{if (Checking) then Fail & Reset\left(\right)}

5: \text{Y := true}

6: \text{if (X \neq p) then Fail & Reset\left(\right)}

\hspace{1cm} \text{else}

7: \text{Entry\left(2\right)}

8: \text{critical section}

9: \text{Y := false}

10: \text{InCount.Update\left(false\right)}

11: \text{Exit\left(2\right)}

\text{procedure Fail & Reset\left(\right)}

12: \text{Entry\left(n\right)}\left(p\right)

13: \text{Entry\left(1\right)}

14: \text{critical section}

15: \text{Checking := true}

16: \text{InCount.Update\left(false\right)}

17: \text{inCount\left[0,..,n−1\right] := InCount.Collect\left(\right)}

18: \text{for (i = 0; i < n; i++)}

\hspace{1cm} \text{if (inCount\left[i\right].val) then break}

\hspace{1cm} \text{if (i = n) then}

\hspace{2cm} \text{Y := false}

19: \text{Checking := false}

20: \text{Exit\left(1\right)}

21: \text{Exit\left(n\right)}\left(p\right)

\text{Figure 5: A long-lived splitter algorithm, adaptive in the slow path (Algorithm 4)}
8 conclusion and "open questions"

This essay studies a possible solution for the mutual exclusion property, using the long-lived splitter notion and insuring $O(1)$ operations for a process running with no contention. A new algorithm (alg. 1) refraining from using splitter "generations" was suggested, and compared to two previously known algorithms: Anderson&Kim's long-lived splitter algorithm (alg. 2) with splitter "generations" (using unbounded memory and ensuring $O(\log n)$ operations in the slow path); and Anderson&Yang's long-lived splitter algorithm (alg. 3) without splitter "generations" but with $O(n)$ operations in the slow path, and a possibly problematic property that leads to a possibly closed splitter at the end of a contention period.

As explained in sections 3 and 4, although alg. 1 solves the mutual exclusion problem with a fast path in a relatively simple way, it still has several disadvantages, mainly the added time complexity in the slow path. The more efficient variant of alg. 4 has a different drawback – the added polynomial space complexity. The lower bound given in section 6 shows that under specific assumptions $n$ registers must be used in the algorithm, but doesn't prove all these registers should be read by every slow process. This gives rise to the next questions:

- Can we close the gap between the number of registers that should be written to, and the number of registers that should be read before a slow process resets the splitter?
- Do all slow processes have to try and reset the splitter?
- Furthermore, can the slow processes be split in some way in advance (e.g. into "right" and "left" processes when deflected from the splitter), such that only one type of slow processes will need to reset the splitter?
- Does the lower bound still hold when multi-reader-multi-writer registers are used?
- Can more efficient long-lived splitter algorithms be implemented, making further use of renaming or collect objects?

References


