Shared-memory Mutual Exclusion: Major Research Trends Since 1986*

James H. Anderson and Yong-Jik Kim
Department of Computer Science
University of North Carolina at Chapel Hill

Ted Herman
Department of Computer Science
University of Iowa

June 2001, Revised May 2002, September 2002, and January 2003

Abstract

In 1986, Michel Raynal published a comprehensive survey of algorithms for mutual exclusion [70]. In this paper, we survey major research trends since 1986 in work on shared-memory mutual exclusion.

Keywords: Adaptive mutual exclusion, fast mutual exclusion, group mutual exclusion, k-exclusion, local spinning, nonatomic algorithms, shared-memory systems, time complexity, timing-based algorithms

1 Introduction

Mutual exclusion algorithms are used to resolve conflicting accesses to shared resources by concurrent processes. The problem of designing such an algorithm is widely regarded as one of the "classic" problems in concurrent programming. In the mutual exclusion problem, a process accesses the resource to be managed by executing a "critical section" of code. Activities not involving the resource occur within a corresponding "noncritical section." Before and after executing its critical section, a process executes two other code fragments, called "entry" and "exit" sections, respectively. A process may halt within its noncritical section but not within its critical section. Furthermore, no variables (other than program counters) accessed within a process's entry or exit section may be accessed within its critical or noncritical section. The objective (at a minimum) is to design the entry and exit sections so that the following requirements hold.[1]

- **Exclusion:** At most one process executes its critical section at any time.
- **Livelock-freedom:** If some process is in its entry section, then some process eventually executes its critical section.

Often, livelock-freedom is replaced by the following stronger property.

- **Starvation-freedom:** If some process is in its entry section, then *that* process eventually executes its critical section.

For either variant, livelock-freedom or starvation-freedom, we require the following: if a process is in its exit section, then it eventually enters its noncritical section. (This property holds trivially for most algorithms.)

In this paper, we survey recent research on mutual exclusion algorithms for shared-memory systems. For the most part, we will consider only "user-level" algorithms that do not rely on operating-system services.

*Work supported by NSF grants CCR 9732916, CCR 9972211, CCR 9988327, ITR 0082866, and CCR 0208289.

[1]Some authors use the term "deadlock-freedom" to denote the progress property termed livelock-freedom here. We prefer the latter term, because deadlock-freedom has been inconsistently used in the literature as both a safety property and a progress property. Also, some authors prefer "lockout-freedom" to starvation-freedom.

Such algorithms have been studied for many years. The first N-process algorithm was published by Dijkstra in 1965 [33]. Dijkstra's algorithm, which is based on an earlier two-process algorithm by Dekker, is livelock-free but not starvation-free. A related algorithm published by Knuth in 1966 was the first starvation-free solution [48]. In the years since the publication of Dijksta's and Knuth's algorithms, a great many other algorithms have been proposed. Many of these algorithms are described in a survey on the topic published by Michel Raynal in 1986 [70]. In this paper, we supplement Raynal's survey of shared-memory algorithms by considering research conducted since its publication. (Raynal's book also discusses message-passing algorithms, which are not considered here.) Taken together, these two surveys should provide readers with a comprehensive overview of the entire history to date of work on shared-memory mutual exclusion algorithms.

In assessing research on shared-memory mutual exclusion over the last seventeen years, several important trends are evident. Perhaps most important is work on "local-spin" algorithms. Most early shared-memory algorithms employ somewhat complicated busy-waiting loops in which many shared variables are read and written. (In fact, most of the shared-memory algorithms covered in Raynal's book are designed this way.) Under contention, such busy-waiting loops generate excessive traffic on the processors-to-memory interconnection network, resulting in poor performance. In local-spin mutual exclusion algorithms, this problem is avoided by requiring all busy-waiting loops to be read-only loops in which only variables cached or stored locally are accessed. In Section 3, we survey work on local-spin algorithms.

Another major research trend originated with the publication of a novel mutual exclusion algorithm by Lamport that requires only seven memory accesses in the absence of contention [56]. This work was driven by the widely accepted belief that "contention for a critical section is rare in well-designed systems" [56]. Through the years, algorithms such as this, in which a process executes a constant-time "fast path" in the absence of contention, have come to be known simply as "fast" mutual exclusion algorithms. In Section 4, an overview of research on such algorithms is presented.

Over the years, work on fast mutual exclusion algorithms evolved into a broader study of "adaptive" algorithms. In many fast algorithms, there is a sudden jump in time complexity between the contention-free and contention-present cases. In an adaptive algorithm, the rise in time complexity as contention increases is more gradual. Research on such algorithms is surveyed in Section 5.

In systems in which some degree of synchrony is maintained, it is reasonable to assume that a bound exists on the time required to execute any statement. A number of interesting "timing-based" mutual exclusion algorithms have been devised in recent years in which such bounds are exploited to reduce overhead. An overview of work on such algorithms is presented in Section 6.

In very early work on the mutual exclusion problem, Lamport noted the circularity inherent in algorithms that require accesses of shared memory to be atomic [52]. In the same paper, he presented an algorithm that is correct even if memory accesses are nonatomic. Since the publication of Raynal's book, a number of other papers have been published on such "nonatomic" algorithms. In Section 7, several such papers are reviewed.

An overriding theme in all of the work described above is that of reducing time complexity. In work on local-spin algorithms, time complexity is measured by counting memory accesses that cause interconnection network traffic. In work on fast mutual exclusion algorithms, time complexity in the absence of contention is the primary concern. In work on adaptive algorithms, the goal is to minimize time complexity as a function of contention. In work on timing-based algorithms, notions of synchrony are exploited to reduce time complexity. Given this emphasis on time, it is not surprising that several researchers began to investigate fundamental limits on time complexity through work on lower bounds. Of related interest is work on space-complexity bounds. Research on such lower bounds constitutes the last major research trend considered in this paper, and is discussed in Section 8.

Any survey paper must reflect the biases of its authors, and this paper is no exception. To say that all research on shared-memory mutual exclusion algorithms conducted over the last seventeen years fits within the categories noted above would be incorrect. Some of the work that does not fit is briefly considered in Section 9. The paper concludes in Section 10.

2 Preliminaries

Before presenting any algorithms, we first define our execution model and also describe notational conventions that will be used in the rest of the paper. A *concurrent program* consists of a set of processes and a set of variables. A *process* is a sequential program consisting of labeled statements. (We sometimes refer to such statements as *operations*.) Each *variable* of a concurrent program is either private or shared. A *private variable* is defined only within the scope of a single process, whereas a *shared variable* is defined globally and may be accessed by more than one process. In the code listings we present, private variables are uncapitalized and shared variables are capitalized. To distinguish private variables of different processes in our algorithm descriptions, we sometimes use the notation $p.v$ to refer to the private variable v of process p. Each process of a concurrent program has a special private variable called its *program counter*: the statement with label k in process p may be executed only when the value of the program counter of p equals k.

A program's semantics is defined by its set of "fair histories." The definition of a fair history, which is given below, formalizes the requirement that each statement of a program is subject to weak fairness. Before giving the definition of a fair history, we introduce a number of other concepts; all of these definitions apply to a given concurrent program.

A *state* is an assignment of values to the variables of the program. One or more states are designated as *initial states*. If state u can be reached from state t via the execution of statement s, then we say that s is *enabled* at state t and we write $t \xrightarrow{s} u$. If statement s is not enabled at state t, then we say that s is *disabled* at t. A *history* is a sequence $t_0 \xrightarrow{s_0} t_1 \xrightarrow{s_1} \cdots$, where t_0 is an initial state. A history may be either finite or infinite; in the former case, it is required that no statement be enabled at the last state of the history. We say that a history satisfies *weak fairness*, or simply, is *fair*, if it is finite or if it is infinite and each statement is either disabled at infinitely many states of the history or is infinitely often executed in the history [57]. Note that this fairness requirement implies that each continuously-enabled statement is eventually executed.

Each algorithm is specified as a concurrent program with N processes, each with the following structure.

```
while true do
    Noncritical Section;
    Entry Section;
    Critical Section;
    Exit Section
od
```

The conditions imposed on these code sections were described earlier in Section 1. The exclusion property is required to hold in any history, while the livelock-freedom and starvation-freedom properties are required to hold only in fair histories. Unless specified otherwise, we will assume that any solution to the mutual exclusion problem is required to be starvation-free.

With regard to complexity, we assess space complexity by counting words of memory (not bits). We use several different time complexity measures, which are introduced as needed.

Some of the algorithms we consider employ read-modify-write synchronization primitives. These primitives execute atomically, *i.e.*, they cause a single state transition. The primitives we consider include *fetch_and_inc*, *fetch_and_dec*, *atomic_add*, *fetch_and_store*, and *compare_and_swap*. These primitives are defined below. In these definitions, *Var* denotes a shared variable, and *val*, *old*, and *new* are used as input and output parameters. In last two definitions, *Var*, *old*, and *new* are assumed to be type-consistent.

```
fetch_and_inc(Var: integer) returns integer
    old := Var;
    Var := Var + 1;
    return(old)

fetch_and_dec(Var: integer) returns integer
    old := Var;
    Var := Var − 1;
```

```
        return(old)

    atomic_add(Var: integer, val: integer constant)
        Var := Var + val

    fetch_and_store(Var, new) returns typeof(Var)
        old := Var;
        Var := new;
        return(old)

    compare_and_swap(Var, old, new) returns boolean
        if Var = old then Var := new; return true
                     else return false
        fi
```

3 Local-spin Algorithms

In local-spin mutual exclusion algorithms, all busy waiting is by means of read-only loops in which one or more "spin variables" are repeatedly tested. Such spin variables must be *locally accessible*, *i.e.*, they can be accessed without causing message traffic on the processors-to-memory interconnection network. Two architectural paradigms have been considered in the literature that allow shared variables to be locally accessed: distributed shared-memory (DSM) machines and cache-coherent (CC) machines. Both are illustrated in Figure 1. In a DSM machine, each processor has its own memory module that can be accessed without accessing the global interconnection network. On such a machine, a shared variable can be made locally accessible by storing it in a local memory module. In a CC machine, each processor has a private cache, and some hardware protocol is used to enforce cache consistency (*i.e.*, to ensure that all copies of the same variable in different local caches are consistent). On such a machine, a shared variable becomes locally accessible by migrating to a local cache line. In this paper, we consider a DSM machine with caches that are kept coherent to be a CC machine. For most of the local-spin algorithms we consider, we assume that there is a unique process executing the algorithm on each processor; we further assume that these processes do not migrate.

In local-spin algorithms for DSM machines, each process must have its own dedicated spin variables (which must be stored in its local memory module). In contrast, in algorithms for CC machines, processes may *share* spin variables, because each process can read a different cached copy. Because dedicated spin variables are required on DSM machines, it is generally more difficult to design correct local-spin algorithms for DSM machines than for CC machines. (Although virtually every modern multiprocessor is cache-coherent, non-cache-coherent DSM systems are still used in embedded applications, where cheaper computing technology often must be used due to cost limitations. Thus, the DSM model is of relevance for reasons other than historical interest.)

In most of the time complexity measures considered in this paper, only certain statement executions are counted. For local-spin algorithms, we usually use the *RMR* (*remote-memory-references*) *time complexity* measure.[2] As its name suggests, only remote memory references that cause an interconnect traversal are counted under this measure. We will assess the RMR time complexity of an algorithm by counting the total number of remote memory references required by one process to enter and then exit its critical section once. An algorithm may have different RMR time complexities under the CC and DSM models because the notion of a remote memory reference differs under these two models. In the CC model, we assume that, once a spin variable has been cached, it remains cached until it is either updated or invalidated as a result of being modified by another process on a different processor. (Effectively, we are assuming an idealized cache of infinite size: a cached variable may be updated or invalidated but it is never replaced by another variable because of associativity or capacity limitations.)

[2]For time complexity measures that simply involve counting only certain statement executions, we use the convention of preceding the term "time complexity" by an adjective that identifies which statement executions are being counted. Thus, under the RMR time complexity measure, only remote memory references are counted.

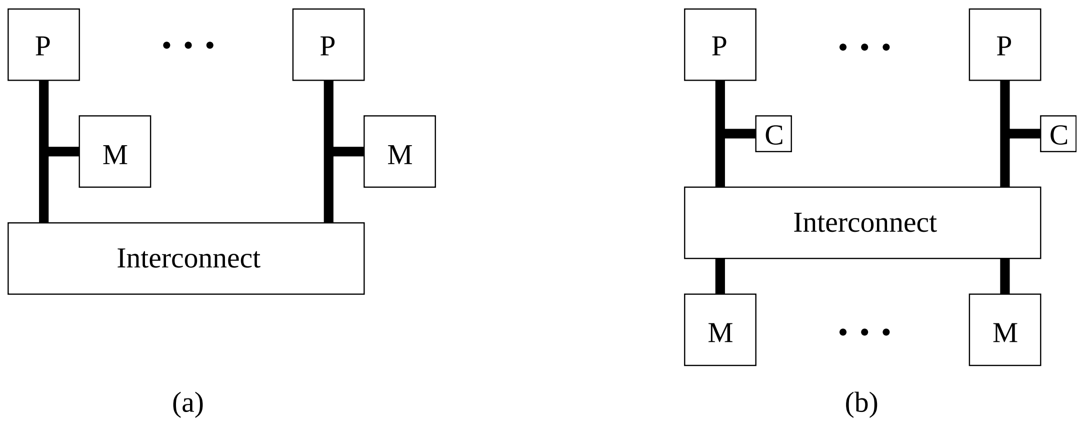


Figure 1: **(a)** DSM model. **(b)** CC model. In both insets, 'P' denotes a processor, 'C' a cache, and 'M' a memory module.

The first local-spin algorithms were "queue-lock" algorithms in which read-modify-write primitives are used to enqueue blocked processes onto the end of a "spin queue" [20, 39, 61]. In each of these algorithms, a process enqueues itself by using a read-modify-write primitive to update a shared "tail" pointer; a process's predecessor (if any) in the queue is indicated by the primitive's return value. A process in the spin queue waits (if necessary) until released by its predecessor. In Section 3.1 below, three queue-lock algorithms are considered in detail and a brief overview of several related algorithms is presented. The three algorithms covered in detail vary in the synchronization primitives used, and whether spinning is local on both CC and DSM systems. In each, a constant number of remote memory references is required per critical-section execution, provided spinning is local.

Yang and Anderson later called into question the necessity of strong synchronization primitives by presenting an algorithm with comparable performance that uses only read and write operations [80]. In terms of RMR time complexity, however, this algorithm is somewhat inferior to the queue locks mentioned above, as it requires $\Theta(\log N)$ remote memory references per critical-section execution. This algorithm and a few other related algorithms are described in Section 3.2.

3.1 Algorithms that Use Read-modify-write Primitives

We begin by considering two queue-lock algorithms, by T. Anderson [20] and by Graunke and Thakkar [39], in which the spin queue is stored in a shared array. Each of these algorithms has $O(1)$ RMR time complexity under the CC model, but unbounded RMR time complexity under the DSM model. In the third algorithm we consider, the spin queue is stored as a shared linked list. This algorithm, which was proposed by Mellor-Crummey and Scott [61], has $O(1)$ RMR time complexity under both the CC and DSM models.

ALGORITHM TA. T. Anderson's algorithm [20], denoted ALGORITHM TA, is shown in Figure 2. (In this and subsequent figures, the statement "**await** B," where B is a boolean expression, is used as a shorthand for the busy-waiting loop "**while** $\neg B$ **do** /$*$ null $*$/ **od**.") ALGORITHM TA uses both *fetch_and_inc* and *atomic_add*. (A *fetch_and_inc* primitive that takes the value to add as input can be used in place of *atomic_add*; in this case, the return value of *fetch_and_inc* is ignored.) The spin queue is defined by an array of "slots." These slots are indexed from 0 to $N-1$. The next free slot at the tail of the queue is indicated by the shared variable *Next_slot*. A process p enqueues itself onto the end of the spin queue by simply using *fetch_and_inc* to increment *Next_slot* (line 2). In addition to updating *Next_slot*, the *fetch_and_inc* operation returns the slot for p to use, which is stored in the private variable *p.my_place*. The main complication to be dealt with occurs when some process increments *Next_slot* beyond slot $N-1$. In this case, the process q that increments *Next_slot* from $N-1$ to N will find *q.my_place* $= N-1$ at line 3, and then execute the *atomic_add* operation

```
const
    $has\_lock = 0$;
    $must\_wait = 1$

shared variable
    $Slots$: array$[0..N-1]$ of $\{has\_lock,\ must\_wait\}$;
    $Next\_slot$: integer initially $0$

initially
    $Slots[0] = has\_lock \ \wedge\ (\forall k : 0 < k < N :: Slots[k] = must\_wait)$

process $p$    /* $0 \le p < N$ */

private variable
    $my\_place$: integer

    while $true$ do
1:      Noncritical Section;
2:      $my\_place := fetch\_and\_inc(Next\_slot)$;
3:      if $my\_place = N - 1$ then
4:          $atomic\_add(Next\_slot,\ -N)$
        fi;
5:      $my\_place := my\_place$ mod $N$;
6:      await $Slot[my\_place] = has\_lock$;    /* spin */
7:      $Slots[my\_place] := must\_wait$;
8:      Critical Section;
9:      $Slot[my\_place + 1$ mod $N] := has\_lock$
    od
```

Figure 2: ALGORITHM TA: Array-based queue lock using *fetch_and_inc* and *atomic_add*.

at line 4 to "correct" the value of *Next_slot*. (Note that *Next_slot* may be incremented at most $N-1$ times by other processes before this correcting step is performed. This is because any such process is enqueued after q and thus is blocked until q finishes its critical section.) Line 5 ensures that the value of $q.my\_place$ ranges over $\{0, \ldots, N-1\}$.

The value of each slot ranges over $\{has\_lock,\ must\_wait\}$. A process in its entry section waits until its slot has the value *has_lock* (line 6). If there is a successor to process p in the spin queue, then its slot is $p.my\_place + 1$ **mod** N. If a successor does exist, then it is granted the lock when p executes line 9. If no successor exists, then line 9 ensures that the lock will be granted to the next process that performs the *fetch_and_inc* operation at line 2. (The initial conditions also ensure this.) Line 7 is executed by p to reinitialize its slot for future use.

The RMR time complexity of ALGORITHM TA is clearly determined by the number of remote memory references generated by line 6. Under the CC model, line 6 generates a constant number of remote memory references. To see this, note that the first read of $Slots[p.my\_place]$ creates a cached copy. If $Slots[p.my\_place] = must\_wait$ holds, then $Slots[p.my\_place]$ will remain cached until it is either invalidated or updated by another process, but this occurs only when p's predecessor in the spin queue executes line 9, which establishes $Slots[p.my\_place] = has\_lock$. At this point, an additional read of $Slots[p.my\_place]$ causes p's waiting to terminate. Under the DSM model, ALGORITHM TA has unbounded RMR time complexity. This is because different processes spin on different memory locations at different times, and hence, these locations cannot be statically allocated so that all spins are local. Thus, we have the following theorem.

Theorem 1 (T. Anderson) *The mutual exclusion problem can be solved with $O(1)$ RMR time complexity using fetch_and_inc under the CC model.* □

```
type
    Tailtype = record last: pointer to boolean;  bit: boolean end    /* stored in one word */

shared variable
    Slots: array[0..N − 1] of boolean initially true;
    Tail: Tailtype initially (&Slots[0], false)

process p     /* 0 ≤ p < N */

private variable
    prev: pointer to boolean;
    bit, temp: boolean

    while true do
1:      Noncritical Section;
2:      (prev, bit) := fetch_and_store(Tail, (&Slots[p], Slots[p]));
3:      await *prev ≠ bit;     /* spin until toggle */
4:      Critical Section;
5:      temp := ¬Slots[p];
6:      Slots[p] := temp
    od
```

Figure 3: Algorithm GT: Array-based queue lock using *fetch_and_store*.

Algorithm GT. Graunke and Thakkar's algorithm [39], denoted Algorithm GT, is shown in Figure 3. (In this and later figures, the notation $*adr$ denotes the contents of the memory location given by address *adr*, and $\&var$ denotes the address of variable *var*.) Like the previous algorithm, Algorithm GT is an array-based queue lock. In this case, however, the enqueue operation is implemented using *fetch_and_store*. As before, an array of *Slots* is used. Recall that in Algorithm TA, the association of slots to processes is not fixed but the ordering of the slots comprising the queue of waiting processes is (*e.g.*, if slot 0 is not at the end of the queue, then slot 1 is the slot following it). Here, the association of slots to processes *is* fixed but the ordering of the slots comprising the queue varies dynamically. In particular, each slot is defined by a boolean value and is "owned" by a unique process. A process enqueues itself by appending its slot to the end of the queue. As we shall see, a waiting process uses its *predecessor*'s slot as a spin variable, *i.e.*, a process p with a predecessor q in the queue waits until q updates the slot owned by q. This is different from Algorithm TA, where each process uses the slot it obtains from the *fetch_and_inc* operation as its spin variable.

In Algorithm GT a shared variable *Tail* is used in addition to the *Slots* array. *Tail* contains two components: the first is a pointer to the last slot in the spin queue; the second is a bit that records the boolean value that was stored in the slot pointed to when the process owning that slot last entered its critical section (see the explanation below). These two components are packed into a single word so that they can be accessed atomically by a single *fetch_and_store* operation. The initial configuration corresponds to a situation where process 0 has just finished executing its critical section, and all processes are now in their noncritical sections.

$Slots[p]$ is the slot owned by process p. In its entry section, process p threads itself onto the end of the spin queue by storing its slot information in *Tail* using a *fetch_and_store* operation (line 2). The *fetch_and_store* operation also returns the slot information for p's predecessor (if it exists) in the spin queue. Before entering its critical section, p waits for the value stored within its predecessor's slot to toggle (line 3). (It is easy to see that p does not wait if it has no predecessor.) In its exit section, p simply toggles the value of its slot (lines 5–6).

An example execution is shown in Figure 4. Initially, the value of *Tail* is $(\&Slots[0],\ false)$. Process 2 performs its *fetch_and_store* operation and obtains $2.prev = \&Slots[0]$ and $2.bit = false$. Note that this implies that process 2 does not wait. Also, the value of *Tail* is now $(\&Slots[2],\ true)$, which corresponds to process 2's slot information. Before process 2 executes its exit section, process 3 performs its *fetch_and_store* operation

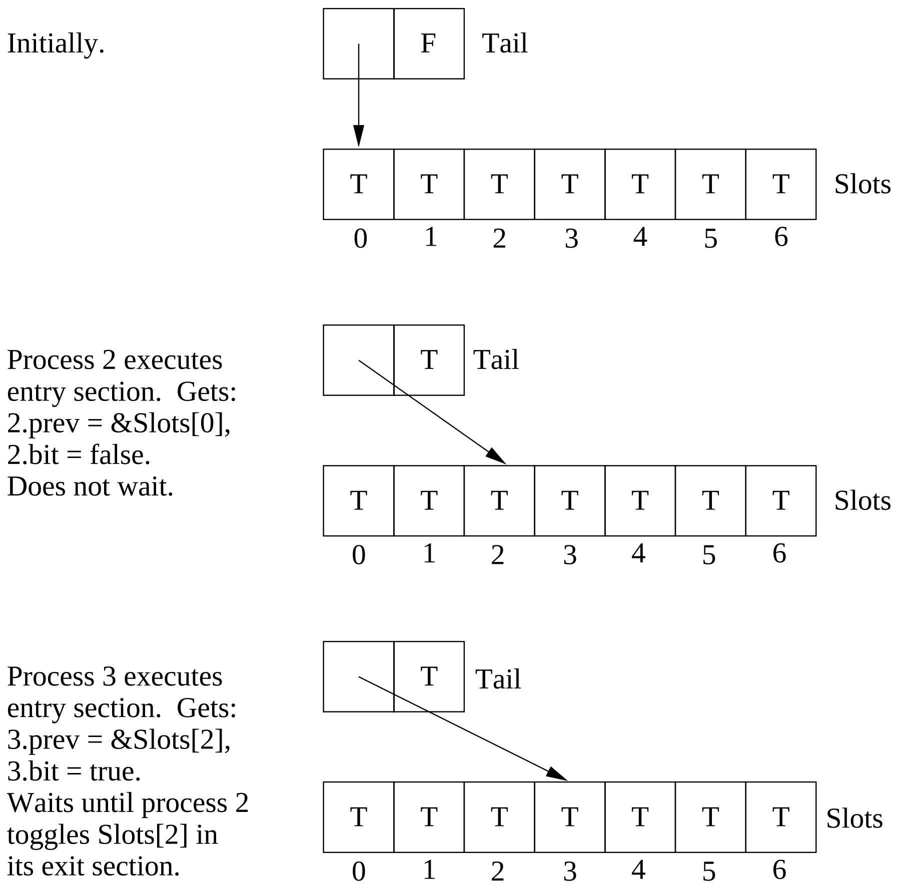

Initially.
F
Tail
T T T T T T T
Slots
0 1 2 3 4 5 6
Process 2 executes entry section. Gets: 2.prev = &Slots[0], 2.bit = false. Does not wait.
T
Tail
T T T T T T T
Slots
0 1 2 3 4 5 6
Process 3 executes entry section. Gets: 3.prev = &Slots[2], 3.bit = true. Waits until process 2 toggles Slots[2] in its exit section.
T
Tail
T T T T T T T
Slots
0 1 2 3 4 5 6

Figure 4: An example execution of ALGORITHM GT.

and changes the value of *Tail* to (&*Slots*[3], *true*), which corresponds to process 3's slot information. From the *fetch_and_store* operation, process 3 also obtains 3.*prev* = &*Slots*[2] and 3.*bit* = *true*. Thus, before entering its critical section, process 3 must wait for process 2 to toggle the value of *Slots*[2] from *true* to *false*.

It is easy to see that the RMR time complexity of ALGORITHM GT is $O(1)$ under the CC model. Under the DSM model, ALGORITHM GT has unbounded RMR time complexity. This is because each process waits on information stored within the slot of its predecessor in the spin queue. This information cannot be statically allocated so that all spins are local. Thus, from ALGORITHM GT, we have the following theorem.

Theorem 2 (Graunke and Thakkar) *The mutual exclusion problem can be solved with $O(1)$ RMR time complexity using fetch_and_store under the CC model.* □

ALGORITHM MCS**.** The final queue-lock algorithm we consider in detail is a linked-list-based algorithm due to Mellor-Crummey and Scott [61]. This algorithm, denoted ALGORITHM MCS, is shown in Figure 5. (In this figure, $a{-}{>}b$ is used as a shorthand for $(*a).b$, where a is a pointer to a record with component b.) ALGORITHM MCS employs both *fetch_and_store* and *compare_and_swap*.

Each entry in the linked list is called a *Qnode*, and each process has its own dedicated *Qnode* (which is assumed to be stored locally, if the algorithm is implemented on a DSM machine). The *Qnode* for each process p has two components: a pointer to p's successor in the spin queue (if any), and a boolean variable *locked*, which is p's spin location. The shared variable *Tail* points to the last *Qnode* in the queue. *Tail* is initially *NIL*.

```
type
    Qnode = record next: pointer to Qnode;  locked: boolean end    /* stored in one word */

shared variable
    Nodes: array[0..N − 1] of Qnode;    /* Nodes[p] is stored locally to process p */
    Tail: pointer to Qnode initially NIL

process p     /* $0 \leq p < N$ */

private constant
    my_node = &Nodes[p]

private variable
    pred: pointer to Qnode

    while true do
1:      Noncritical Section;
2:      my_node−> next := NIL;
3:      pred := fetch_and_store(Tail,  my_node);
4:      if pred ≠ NIL then
5:          my_node−> locked := true;
6:          pred−> next := my_node;
7:          await ¬my_node−> locked      /* spin until granted the lock by predecessor */
        fi;
8:      Critical Section;
9:      if my_node−> next = NIL then
10:         if compare_and_swap(Tail,  my_node,  NIL) = false then
11:             await my_node−> next ≠ NIL;      /* spin until next field is updated */
12:             my_node−> next−> locked := false
            fi
        else
13:         my_node−> next−> locked := false
        fi
    od
```

Figure 5: ALGORITHM MCS: List-based queue lock using *fetch_and_store* and *compare_and_swap*.

A process p threads itself onto the end of the spin queue by performing the *fetch_and_store* operation at line 3. This *fetch_and_store* causes *Tail* to point to p's *Qnode* and also returns to p the previous value of *Tail*. If p threads itself onto a nonempty spin queue, then this previous value gives p's predecessor in the queue. In this case, p initializes its spin location (line 5), updates the *next* pointer of its predecessor (line 6), and then busy-waits until released by its predecessor (line 7).

In its exit section, p must release its successor in the spin queue, if subsequent processes in the queue do indeed exist. If $p.my\_node{-}{>}\, next \neq NIL$ holds when p executes line 9, then p can easily update its successor's spin location (line 13). However, if $p.my\_node{-}{>}\, next = NIL$ holds, then a potential problem arises. In particular, it may be the case that p has no successor, or it may be the case that it does have a successor, but that process has not yet updated p's *next* field. This ambiguity is resolved by the *compare_and_swap* on *Tail* performed at line 10. If p indeed has no successor, then *Tail* must still point to p's *Qnode*, in which case the *compare_and_swap* returns *true*. On the other hand, if p has a successor, then the *compare_and_swap* returns *false*, and p executes lines 11–12. Line 11 causes p to wait until its *next* pointer has been updated by its successor. (This is one of the few mutual exclusion algorithms found in the literature in which a process may wait in its exit section.) Line 12 then updates its successor's spin location.

Figure 6 depicts an example execution. Initially, *Tail* is *NIL*. Process 2 then initializes its *Qnode* and performs the *fetch_and_store* operation at line 3. At this point, *Tail* points to process 2's *Qnode*. Note that, because *Tail* equaled *NIL* prior to process 2's *fetch_and_store*, process 2 does not wait in its entry section. In the third inset, process 3 has initialized its *Qnode* and has performed the *fetch_and_store* on *Tail*. Note that

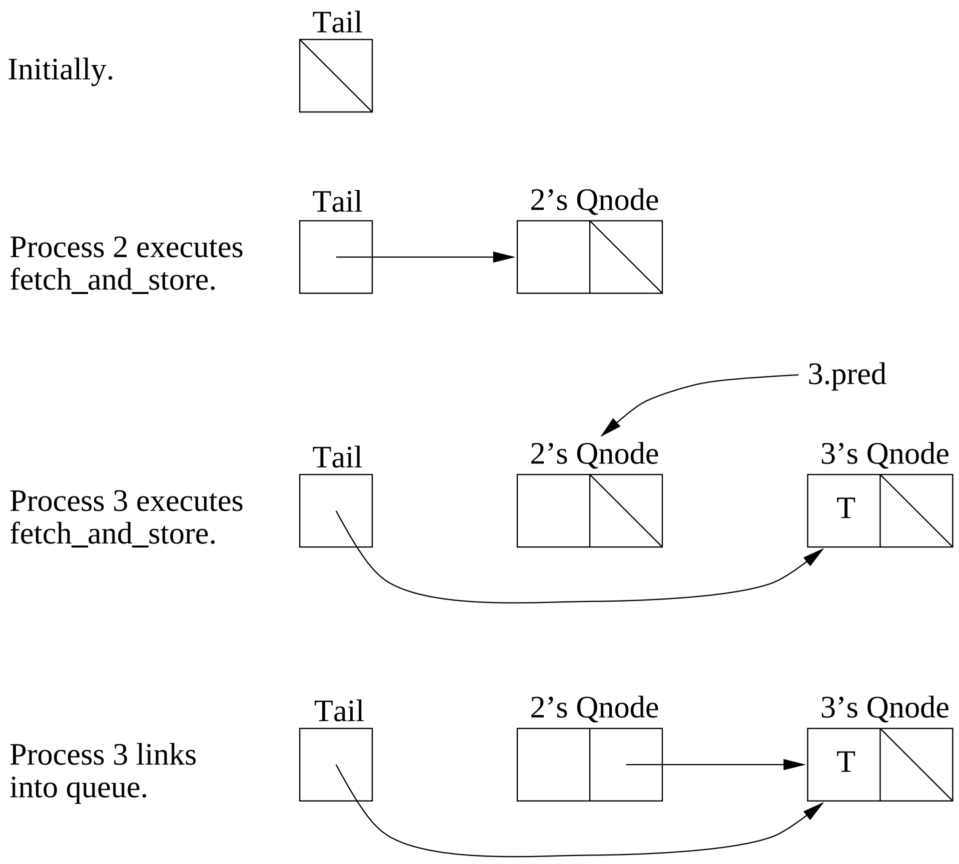

Initially.
Tail
Process 2 executes fetch_and_store.
Tail
2's Qnode
3.pred
Process 3 executes fetch_and_store.
Tail
2's Qnode
3's Qnode
T
Process 3 links into queue.
Tail
2's Qnode
3's Qnode
T

Figure 6: An example execution of ALGORITHM MCS.

process 3's private variable 3.*pred* points to process 2's *Qnode*. In the fourth inset, process 3 has finished linking itself onto the end of the queue by updating the *next* field within process 2's *Qnode*. From this point, process 3 will continue to wait until released by process 2. Note that if we were to "freeze" process 3 in the third inset and allow process 2 to execute until reaching its exit section, then process 2 would find $2.my\_node{-}{>}\,next = NIL$ at line 9, and $Tail \neq 2.my\_node$ at line 10. Thus, process 2 would reach line 11 and wait there until its *next* field is updated by process 3.

Because each process has its own dedicated spin location, it should be clear that the algorithm has $O(1)$ RMR time complexity under either the CC or DSM model. Thus, we have the following.

Theorem 3 (Mellor-Crummey and Scott) *The mutual exclusion problem can be solved with $O(1)$ RMR time complexity using fetch_and_store and compare_and_swap under either the CC or DSM model.* □

One problem with ALGORITHM MCS is that it relies on two synchronization primitives, which may limit its applicability. To circumvent this problem, Mellor-Crummey and Scott also presented a variant that uses only *fetch_and_store* [61]. Unfortunately, this variant is only livelock-free. However, the authors argue that starvation should be very unlikely in practice.

It may be instructive to briefly compare ALGORITHM MCS with ALGORITHM TA and ALGORITHM GT. The latter two algorithms use an array of spin variables and the association of processes to spin variables varies dynamically with time. For this reason, these algorithms are not local-spin algorithms under the DSM model. For an algorithm to spin locally in the CC model, it is sufficient for an exiting process to suitably update a spin variable that *any* successor process must wait on, if one exists. In ALGORITHM TA, any successor must wait for the next slot in the array to be updated. In ALGORITHM GT, any successor must wait for the exiting process's slot information to be updated. In Algorithm MCS, dedicated spin variables are used so that spinning is local even under the DSM model. With dedicated spin variables, an exiting process must determine the *precise identity* of any possible successor in order to update its dedicated spin variable. This is why it is necessary for each enqueuing process to update the *next* pointer of its predecessor.

Other related algorithms. A number of researchers have proposed extensions to the three algorithms covered so far that support process priorities or that tolerate process preemptions [12, 31, 42, 49, 60, 71, 78, 79]. Priorities can be supported either by requiring the spin queue to be priority ordered, or by requiring each process in its exit section to completely scan the queue to find the highest-priority waiting process. In the former case, a process must have the ability to scan the queue and insert its queue record at the position indicated by its priority. Priority-based systems are often multiprogrammed, *i.e.*, there may be multiple processes bound to the same processor. In multiprogrammed systems, preemptions may be common. Preemptions are especially problematic for queue locks, because a preempted process may delay every process after it in the spin queue. Most proposals for dealing with preemptions rely on the kernel to deactivate or remove the queue record of a preempted process.

A restricted form of priority that has been well-studied occurs in algorithms for reader/writer synchronization [30]. Reader/writer synchronization is a generalization of mutual exclusion in which each process is classified as either a reader or a writer. Readers may execute their critical sections simultaneously, but writers require exclusive access. Because readers and writers have different requirements, it is necessary to give readers priority over writers or *vice versa*. After the development of Algorithm MCS, Mellor-Crummey and Scott presented several extensions of that algorithm that support reader/writer synchronization [62].

Other authors have investigated algorithms that use circular waiting lists [37, 40, 41]. Note that, in Algorithm MCS, each process p finds its predecessor in the list by performing the *fetch_and_store* at line 3. However, p cannot identify its successor (if any) until that successor first updates p's *next* field. By using a circular list, this problem can be eliminated, because all nodes can be reached by traversing through the list by reading predecessor pointers.

Fu and Tzeng [37, 41] considered a circular-list algorithm where processes are arranged in a tree, each node of which contains a circular waiting list. Each process in its entry section enqueues itself onto a leaf circular list. A process that is at the head of the circular list at some node ascends the tree, merging the circular list of that node with the circular list of its parent node. It is argued that the tree structure eliminates hot-spot contention, leading to better performance. (*Hot-spot contention* occurs when many processes repeatedly access the same shared variable, or variables stored in the same memory module. Hot-spot contention can lead to degraded performance for all memory accesses, even those that do not target the hot spot [68].)

Despite these potential advantages, we found the algorithms in [37, 40, 41] to be quite difficult to understand, so we cannot assert that they really are an improvement over previous algorithms. Indeed, the algorithm in [37] is not even starvation-free.

In other related work, extensions of some of the queue-based algorithms discussed above were recently proposed in which timeout mechanisms are incorporated [73, 74]. Such mechanisms can be used by a process to abandon its lock request if it has waited too long (*e.g.*, if a deadline has passed).

As noted earlier, Algorithms TA and GT have $O(1)$ RMR time complexity only under the CC model. This raises the question of whether *fetch_and_inc* and *fetch_and_store* primitives are sufficient for constructing constant-time algorithms under the DSM model. Huang [40] presented a circular list algorithm that uses only *fetch_and_store* and that has constant *amortized* RMR time complexity (*i.e.*, the number of remote memory references in a history divided by the number of critical-section entries in that history is constant) in DSM systems. In a recent paper [16], Anderson and Kim showed that constant-time algorithms (without amortization) can be constructed for DSM systems using any of a large class of primitives that includes both *fetch_and_inc* and *fetch_and_store*. Thus, Theorems 1 and 2 can each be strengthened to also apply to such systems.

3.2 Algorithms that Use Only Reads and Writes

Algorithms TA, GT, and MCS were the first local-spin mutual exclusion algorithms to be published, and each requires one or more read-modify-write primitives. This led some researchers to question whether such primitives were in fact *necessary* for local-spin synchronization. In 1993, Anderson showed that this was not the case by presenting an $\Theta(N)$ algorithm that uses only read and write operations [11]. Although this algorithm showed that local-spin synchronization without strong primitives was possible in principle, its RMR time complexity is significantly higher than Algorithms TA, GT, and MCS. In subsequent work, Yang and Anderson narrowed this time-complexity gap by presenting an $\Theta(\log N)$ algorithm based on reads and writes [80]. As explained later in Section 8, it follows from recent work on lower bounds that the RMR time complexity of Yang and Anderson's algorithm is within a factor of $\Theta(\log \log N)$ of optimality for algorithms that use only atomic reads and writes (Theorem 24).

In the rest of this section, an overview is given of Yang and Anderson's algorithm, hereafter denoted Algorithm YA [80]. The earlier algorithm of Anderson [11] happens to be correct even if reads and writes are nonatomic. Therefore, we defer consideration of it to Section 7.

Algorithm YA. In [45], Kessels proposed implementing N-process mutual exclusion by using instances of a two-process solution (which, in Kessels' algorithm, is not a local-spin algorithm) in a binary arbitration tree. Associated with each link in the tree is an entry section and an exit section. The entry and exit sections associated with the two links connecting a given node to its children constitute a two-process mutual exclusion algorithm. Initially, all processes are "located" at the leaves of the tree. To enter its critical section, a process is required to traverse a path from its leaf up to the root, executing the entry section of each link on this path. Upon exiting its critical section, a process traverses this path in reverse, this time executing the exit section of each link.

Algorithm YA is based on the arbitration-tree approach of Kessels. For this approach to work in a DSM system when all busy-waiting is by local spinning, the two-process algorithm being used must provide a mechanism that allows a process to deduce the process (if any) with which it must compete. (If a process incorrectly determines its competitor at some node of the tree, then it may end up writing to a spin variable of a process that has not even accessed that node!) The two-process version of Algorithm YA provides such a mechanism. A slightly-simplified version of the two-process algorithm is shown in Figure 7, and the full N-process algorithm is shown in Figure 8. In Figure 7, the two processes are denoted by the identifiers u and v, which are assumed to be distinct, nonnegative integer values. (Process identifiers range over $\{0, \ldots, N-1\}$ in Figure 8.)

The two-process algorithm employs five shared variables, $C[u]$, $C[v]$, T, $P[u]$, and $P[v]$. Variable $C[u]$ ranges over $\{u, v, \perp\}$ and is used by process u to inform process v of its intent to enter its critical section. Observe that $C[u] = u \neq \perp$ holds while process u is at lines 3–13, and $C[u] = \perp$ holds otherwise. Variable $C[v]$ is used similarly. Variable T ranges over $\{u, v\}$ and is used as a tie-breaker in the event that both processes attempt to enter their critical sections at the same time. The algorithm ensures that the two processes enter their critical sections according to the order in which they update T. Variable $P[u]$ ranges over $\{0, 1, 2\}$ and is used by process u whenever it needs to busy-wait. Variable $P[v]$ is used similarly by process v. (Note that these are statically-allocated spin variables; hence, on DSM machines, they can be stored locally.)

When process u wants to enter its critical section, it informs process v of its intention by establishing $C[u] = u$. Then, process u assigns its identifier u to the tie-breaker variable T, and initializes its spin location $P[u]$. If process v has not shown interest in entering its critical section, *i.e.*, if $C[v] = \perp$ holds when u executes line 5, then process u proceeds directly to its critical section. Otherwise, u reads the tie-breaker variable T. If $T \neq u$, which implies that $T = v$, then u can enter its critical section, as the algorithm prohibits v from entering its critical section when $C[u] = u \ \wedge \ T = v$ holds (recall that ties are broken in favor of the first process to update T). If $T = u$ holds, then either process v executed line 3 before process u, or process v has executed line 2 but not line 3. In the first case, u should wait until v exits its critical section, whereas, in the second case, u should be able to proceed to its critical section. This ambiguity is resolved by having process u execute lines 7–11. Lines 7–8 are executed by process u to release process v in the event that it is

```
shared variables
  $C$: array$[u, v]$ of $\{u, v, \perp\}$ initially $\perp$;
  $P$: array$[u, v]$ of $0..2$ initially $0$;
  $T$: $\{u, v\}$
```

```
process $u$

while true do
  1:  Noncritical Section;
  2:  $C[u] := u$;
  3:  $T := u$;
  4:  $P[u] := 0$;
  5:  if $C[v] \neq \perp$ then
  6:     if $T = u$ then
  7:        if $P[v] = 0$ then
  8:           $P[v] := 1$;
           fi;
  9:        await $P[u] \geq 1$; /* spin */
 10:        if $T = u$ then
 11:          await $P[u] = 2$ /* spin */
           fi
        fi
     fi;
 12:  Critical Section;
 13:  $C[u] := \perp$;
 14:  if $T \neq u$ then
 15:     $P[v] := 2$
     fi
od
```

```
process $v$

while true do
  1:  Noncritical Section;
  2:  $C[v] := v$;
  3:  $T := v$;
  4:  $P[v] := 0$;
  5:  if $C[u] \neq \perp$ then
  6:     if $T = v$ then
  7:        if $P[u] = 0$ then
  8:           $P[u] := 1$
           fi;
  9:        await $P[v] \geq 1$; /* spin */
 10:        if $T = v$ then
 11:           await $P[v] = 2$ /* spin */
           fi
        fi
     fi;
 12:  Critical Section;
 13:  $C[v] := \perp$;
 14:  if $T \neq v$ then
 15:     $P[u] := 2$
     fi
od
```

Figure 7: Algorithm YA: Two-process case.

waiting for u to update the tie-breaker variable (*i.e.*, v is busy-waiting at line 9). Lines 9–11 are executed by u to determine which process updated the tie-breaker variable first. Note that $P[u] \geq 1$ implies that v has already updated the tie-breaker, and $P[u] = 2$ implies that v has finished its critical section. To handle these two cases, process u first waits until $P[u] \geq 1$ (*i.e.*, until v has updated the tie-breaker), re-examines T to see which process updated T last, and finally, if necessary, waits until $P[u] = 2$ (*i.e.*, until process v finishes its critical section).

After executing its critical section, process u informs process v that it is finished by establishing $C[u] = \perp$. If $T = v$, in which case process v is trying to enter its critical section, then process u updates $P[v]$ so that v does not wait.

As discussed above, the N-process case is solved by applying the above two-process algorithm in a binary arbitration tree. When competing within the two-process instance at a node within the tree, a process can determine its competitor by reading the C variable associated with the other side. Some slight modifications to the algorithm are required to implement this. For example, lines 2–4 in Figure 7 are replaced by the following.

```
$C[node][side] := p$;
$T[node] := p$;
$P[h][p] := 0$;
$rival := C[node][1 - side]$
```

In this code fragment, p is the invoking process, and it invokes the algorithm at the tree node at height h indicated by *node* from side *side*, where *side* ranges over $\{0, 1\}$ (*i.e.*, processes from the left subtree beneath this node invoke side 0, and processes from the right subtree invoke side 1). The identity of process p's competitor (if any) is recorded in the private variable *rival*. Notice that the C variables are now indexed

```
const                                   /* for simplicity, we assume $N = 2^L$ */
    $L = \log N$;                       /* height of arbitration tree = $O(\log N)$ */
    $Tsize = 2^L - 1 = N - 1$           /* size of arbitration tree = $O(N)$ */

shared variables
    $C$: array[1..$Tsize$][0, 1] of $(0..N-1, \perp)$ initially $\perp$;
    $P$: array[1..$L$][0..$N-1$] of 0..2 initially 0;
    $T$: array[1..$Tsize$] of 0..$N-1$

private variables
    $h$: 1..$L$;
    $node$: 1..$Tsize$;
    $side$: 0, 1;        /* 0 = left side, 1 = right side */
    $rival$: 0..$N-1$

process $p$                             /* $0 \le p < N$ */
while $true$ do
1:  Noncritical Section;
2:  for $h := 1$ to $L$ do
3:      $node := \lfloor (N+p)/2^h \rfloor$;
4:      $side := \lfloor (N+p)/2^{h-1} \rfloor \bmod 2$;
5:      $C[node][side] := p$;
6:      $T[node] := p$;
7:      $P[h][p] := 0$;
8:      $rival := C[node][1 - side]$;
9:      if $rival \ne \perp \wedge T[node] = p$ then
10:         if $P[h][rival] = 0$ then
11:             $P[h][rival] := 1$;
            fi;
12:         await $P[h][p] \ge 1$;
13:         if $T[node] = p$ then
14:             await $P[h][p] = 2$
            fi
        fi
    od;
15: Critical Section;
16: for $h := L$ downto 1 do
17:     $node := \lfloor (N+p)/2^h \rfloor$;
18:     $side := \lfloor (N+p)/2^{h-1} \rfloor \bmod 2$;
19:     $C[node][side] := \perp$;
20:     $rival := T[node]$;
21:     if $rival \ne p$ then
22:         $P[h][rival] := 2$
        fi
    od
od
```

Figure 8: Algorithm YA: N-process case.

by side, but are assigned values ranging over the process identifiers. The N-process algorithm that uses this modified two-process code is depicted in Figure 8. In this figure, we have assumed that N is a power of two, for the sake of simplicity. Nodes are numbered as follows: 1 (the root node), 2 and 3 (its two children), 4,. . . ,7 (their children), and so on. Process p begins executing in the arbitration tree at the leaf node $N + p$. From this algorithm, we have the following theorem.

Theorem 4 (Yang and Anderson) *The mutual exclusion problem can be solved with $\Theta(\log N)$ RMR time complexity using only reads and writes under either the CC or DSM model.* □

Other related algorithms. In order to guarantee that different nodes in the arbitration tree do not interfere with each other in Algorithm YA, a distinct spin variable is used for each process at each level of the tree. Thus, the algorithm's space complexity $\Theta(N \log N)$. In a recent paper [47], Kim and Anderson presented a simple code transformation that reduces the space complexity of the algorithm to $\Theta(N)$, which is optimal (see Theorem 18). In this new algorithm, each process uses the same spin variable for all levels of the tree. Like Algorithm YA, the RMR time complexity of this new algorithm is $\Theta(\log N)$.

Aside from the algorithms already mentioned, the only other published local-spin mutual exclusion algorithm that uses only reads and writes that we know of is one by Tsay [77]. Tsay's algorithm is very similar to Algorithm YA and also the earlier algorithm of Anderson [11]. Tsay derives his algorithm through a series of transformations from Peterson's algorithm [67]. Although not noted in [11], Anderson's algorithm was obtained in exactly this way. (Actually, Kessels' algorithm [45] represents an important step in the series of transformations.)

Zhang, Yan, and Castañeda have conducted an extensive evaluation of several (read/write)-based mutual exclusion algorithms [81]. Algorithm YA is one of the algorithms tested by them. Their evaluation is based on several metrics that take into account the effects of architectures, systems, and software implementations. In addition, they present three new algorithms, including two tree-based algorithms, that incorporate both local-spin and non-local-spin techniques.

4 Fast Mutual Exclusion

In 1987, Lamport devised a novel mutual exclusion algorithm that requires only seven memory accesses in the absence of contention [56]. Algorithms such as this, in which a process executes a constant-time "fast path" in the absence of contention, are known as "fast" mutual exclusion algorithms. (When determining contention-free time complexities, all memory references are counted, local and remote.) Each of the read-modify-write-based algorithms covered in Section 3.1 clearly has constant time complexity in the absence of contention. Thus, time complexity in the absence of contention is a non-issue if suitable read-modify-write primitives are available. For this reason, the term "fast mutual exclusion algorithm" is usually applied only to algorithms that use only reads and writes. In this section, we present an overview of research to date on such algorithms.

4.1 Algorithm L: Lamport's Fast Mutual Exclusion Algorithm

Lamport's fast mutual exclusion algorithm, hereafter denoted Algorithm L, is shown in Figure 9(a). In this algorithm, a fast-path process reaches its critical section by executing (only) lines 2, 3, 4, 8, and 9. Of these, lines 3, 4, 8, and 9 are of special significance. These lines are shown separately in Figure 9(c), where they are used to define a "black box" element called a *splitter*, which is illustrated in Figure 9(b). (The term "splitter" is not due to Lamport; it was first abstracted as a "black box" by Moir and Anderson [65], and first called a "splitter" by Attiya and Fouren [22].) In the following paragraphs, we consider some properties of the splitter that make it so useful, and then show how these properties ensure the correctness of Algorithm L.

The splitter element. Each process that invokes the splitter code either stops, moves down, or moves right. (The move is defined by the value assigned to the private variable *dir*.) One of the key properties of the splitter that makes it so useful is the following: if several processes invoke a splitter, then at most one of them can stop at that splitter. To see why this property holds, suppose to the contrary that two processes p and q stop. Let p be the process that executed line 4 last. Because p found that $X = p$ held at line 4, X is not written by any process between p's execution of line 1 and p's execution of line 4. Thus, q executed line 4 before p executed line 1. This implies that q executed line 3 before p executed line 2. Thus, p must have read $Y = \mathit{false}$ at line 2 and then assigned "$p.\mathit{dir} := \mathit{right}$," which is a contradiction. Similar arguments can be applied to show that if n processes invoke a splitter, then at most $n-1$ can move right, and at most $n-1$ can move down.

Because of these properties, the splitter element and related mechanisms have proven to be immensely useful in wait-free algorithms for renaming [3, 4, 23, 22, 24, 65, 66]. Renaming algorithms are used to "shrink" the name space from which process identifiers are taken. Such algorithms can be used to speed up concurrent computations with loops that iterate over process identifiers. Because of the splitter's properties, it is possible to solve the renaming problem by interconnecting a collection of splitters in a grid as shown in Figure 10 [66]. A name is associated with each splitter. When a process stops at a splitter, it acquires the name associated with that splitter. If the grid has N rows and N columns, where N is the number of processes, then by induction, every process eventually stops at some splitter. (The original name space may be much larger than N.) To see why, note that at most $N-1$ processes may move to the second row and below, at most $N-2$ to the third row and below, and so on. Hence, at most one process may enter the N^{th} row (name 11 in Figure 10.) Similar arguments can be used to show that at most one process may enter each "leaf node" (names 11–15 in Figure 10), and hence every process eventually stops.

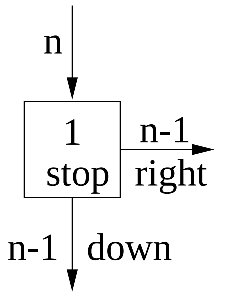
```
shared variable
    B: array[1..N] of boolean initially false;
    X: 1..N;
    Y: 0..N initially 0

process p                          /* 1 ≤ p ≤ N */

private variable
    j: 1..N

while true do
1:  Noncritical Section;
2:  B[p] := true;
3:  X := p;
4:  if Y ≠ 0 then
5:      B[p] := false;
6:      await Y = 0;               /* busy wait */
7:      goto 1
    fi;
8:  Y := p;
9:  if X ≠ p then
10:     B[p] := false;
11:     for j := 1 to N do
12:         await ¬B[j]            /* busy wait */
        od;
13:     if Y ≠ p then
14:         await Y = 0;           /* busy wait */
15:         goto 1
        fi
    fi;
16: Critical Section;
17: Y := 0;
18: B[p] := false
od
```

(a)



(b)

```
/* X and Y are as in part (a) */

process p

private variable
    dir: {stop, right, down}

1:  X := p;
2:  if Y ≠ 0 then dir := right
    else
3:      Y := p;
4:      if X ≠ p then dir := down
                 else dir := stop
        fi
    fi
```

(c)

Figure 9: **(a)** ALGORITHM L: Lamport's fast mutual exclusion algorithm. **(b)** The splitter element and **(c)** its implementation.

Correctness of ALGORITHM L. The splitter's properties also ensure that ALGORITHM L is correct. In particular, because at most one process can stop at a splitter, at most one process at a time can "take the fast path" by reading $Y = 0$ at line 4 and $X = p$ at line 9. (This corresponds to the assignment of $dir := stop$ at Figure 9(c).) Moreover, if no process takes the fast path during a period of contention, then some process must reach lines 10–15. It can be shown that, of these processes, the last to update the variable Y eventually enters its critical section and then reopens the fast path by assigning $Y := 0$ at line 17. Thus, the algorithm is livelock-free, giving us the following theorem.

Theorem 5 (Lamport) *The mutual exclusion problem can be solved by a livelock-free algorithm that requires only seven memory references in the absence of contention.* □

On the other hand, starvation-freedom is not satisfied, because an unfortunate process may repeatedly find either $Y \neq 0$ at line 4 or $Y \neq p$ at line 13, and hence wait forever.

Variations. A number of authors have proposed variants of ALGORITHM L. We only mention variants here that are not covered in detail later. Michael and Scott showed that only two reads and four writes

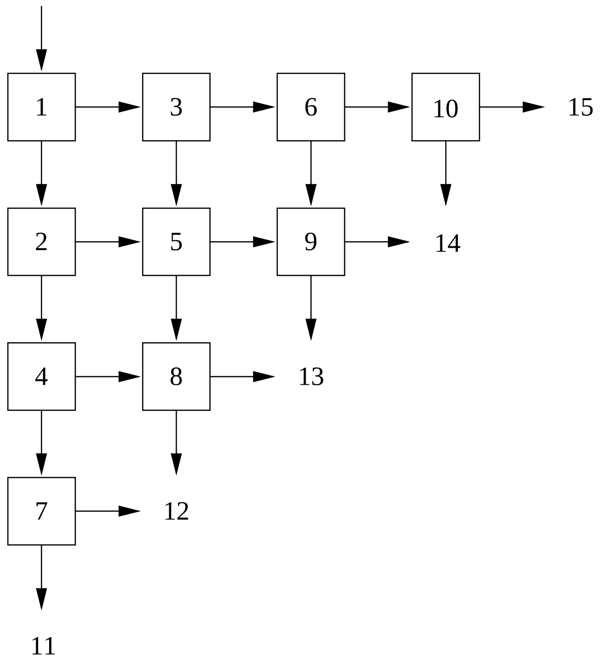


Figure 10: Renaming grid (depicted for $N = 5$).

are required in the absence of contention, if processes have the ability to read and write at both full- and half-word granularities [64].

In contrast, Merritt and Taubenfeld proposed modifications that speed up the algorithm in the *presence* of contention, as compared with Algorithm L [63]. In Algorithm L, each process that reaches line 11 must check the status of every other process. However, in an actual system, the number of processes that actually invoke the algorithm concurrently is likely to be much less than the total number of processes in the system. Merritt and Taubenfeld showed that by using a linked list instead of a simple array scan, the time complexity of this check can be made proportional to the number of contending processes.

4.2 Fast Mutual Exclusion with Local Spinning

All of the fast mutual exclusion algorithms discussed above employ busy-waiting loops in which shared variables are both read and written. Thus, while these algorithms are fast in the *absence* of contention, each has unbounded RMR time complexity *under* contention. In this section, we consider fast mutual exclusion algorithms in which all busy-waiting is by local spinning.

The first such algorithm to be published was actually variant of Algorithm YA considered earlier in Section 3.2 [80]. Unfortunately, in this fast-path variant of Algorithm YA, RMR time complexity under contention is $\Theta(N)$ instead of $\Theta(\log N)$. In later work [15], Anderson and Kim presented an improved fast-path mechanism that results in $O(1)$ time complexity in the absence of contention and $\Theta(\log N)$ RMR time complexity under contention, when used in conjunction with Algorithm YA [15]. The resulting algorithm, denoted Algorithm AK, is described next.

Algorithm AK. The basic idea behind Algorithm AK is to combine the fast path of Algorithm L with the arbitration tree of Algorithm YA, specifically by placing an extra two-process version of Algorithm YA "on top" of the arbitration tree. (The version of Algorithm YA in Figure 8 must be used for the two-process algorithm, because the two processes are not fixed.) This approach is illustrated in Figure 11. The "left" entry section of this extra two-process algorithm (*i.e.*, process 0's code in Figure 8) is

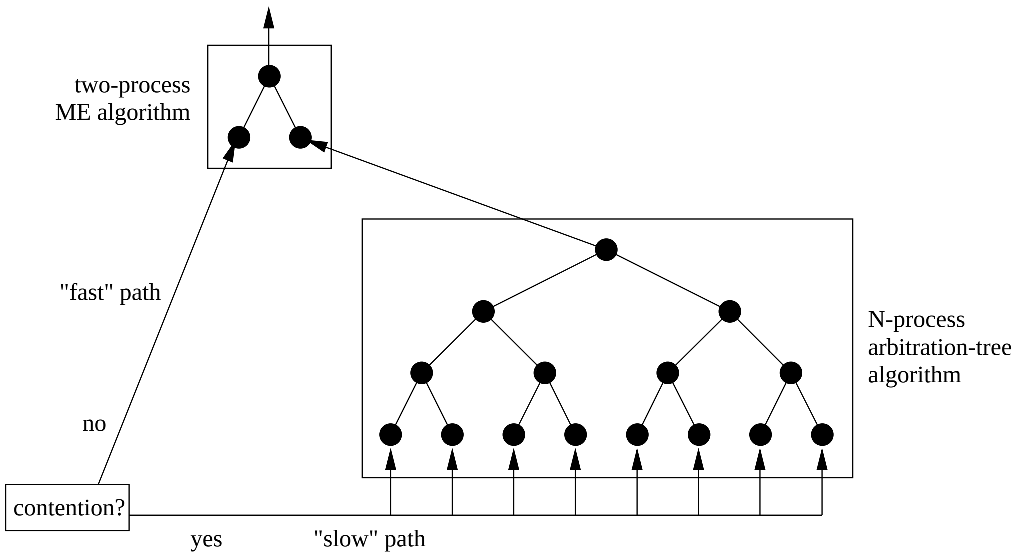


Figure 11: Basic structure of ALGORITHM AK.

executed by a process if that process detects no contention. The "right" entry section of this extra algorithm (*i.e.*, process 1's code in Figure 8) is executed by a process after it has successfully acquired the critical section implemented by the arbitration tree. A process competes within the arbitration tree if it detects any contention.

We now describe the fast-path mechanism of ALGORITHM AK. The version of this mechanism considered here uses unbounded memory. A (slightly more complicated) bounded version is described in [15].

The unbounded version is shown in Figure 12. Before describing how the algorithm works, we first examine its basic structure. Four shared variables and an infinite shared array are used: X, Y, *Reset*, *Name_Taken* (the array), and *Infast*. Variables X and Y are as in Lamport's algorithm, with the exception that Y now has an additional integer *indx* field. As explained below, ALGORITHM AK works by "renaming" any process that acquires the fast path with a new temporary process identifier. The *indx* field of Y is used in renaming processes. The variable *Reset* is used to reinitialize the *indx* field of Y after a period of contention ends. The variable *Name_Taken*$[k]$ is used to indicate whether name k has been assigned to some process. The variable *Infast* is set to record that the fast path has been acquired by some process.

In Figure 12, the entry code for the extra "topmost" two-process algorithm illustrated in Figure 11 is denoted "`ENTRY_2`(0)" for left-invoking processes, and "`ENTRY_2`(1)" for right-invoking processes. The exit code is denoted similarly. The entry and exit code for the N-process arbitration-tree algorithm are denoted "`ENTRY_N`(p)" and "`EXIT_N`(p)," respectively, where p denotes the invoking process.

A process determines if it can access the fast path by executing lines 2–11; of these, lines 2–6 comprise Lamport's fast-path mechanism. Note that the fast path is open if $Y.free = true \wedge Infast = false \wedge Y = Reset$ holds: if a process p begins executing lines 2–11 when this expression holds, and if all other processes remain in their noncritical sections, then p will reach line 12. If a process p detects any other competing process while executing within lines 2–11, then p is deflected out of the fast path and invokes either $SLOW1$ or $SLOW2$. $SLOW1$ is invoked if p has not updated any variables that must be reset in order to reopen the fast path. Otherwise, $SLOW2$ is invoked. A detailed explanation of the deflection mechanism is given below. If a process is not deflected, then it successfully acquires the fast path and executes lines 12–18. A process that either acquires the fast path or is deflected to $SLOW2$ attempts to reopen the fast path by executing lines 15–18 or 27–31, respectively. A detailed explanation of how the fast path is reopened is given below.

One critical property of the algorithm bears mentioning before we consider some of the algorithm's more complicated properties. In particular, note that many of the accesses to shared variables actually occur

```
type Ytype = record free: boolean; indx: $0..\infty$ end          /* stored in one word */

shared variable
    X: $0..N-1$;
    Y, Reset: Ytype initially $(true, 0)$;
    Name_Taken: array[$0..\infty$] of boolean initially false;
    Infast: boolean initially false

process p                    /* $0 \le p < N$ */

private variable  y: Ytype

while true do
1:   Noncritical Section;
2:   $X := p$;
3:   $y := Y$;
4:   if $\neg y.free$ then SLOW1()
     else
5:       $Y := (false, 0)$;
6:       if $(X \ne p \vee$
7:              Infast) then SLOW2()
         else
8:           Name_Taken[y.indx] := true;
9:           if $Reset \ne y$ then
10:              Name_Taken[y.indx] := false;
11:              SLOW2()
             else
12:              Infast := true;
13:              ENTRY_2(0);            /* fast path */
14:                  Critical Section;
15:                  $Reset := (true, y.indx + 1)$;
16:                  $Y := (true, y.indx + 1)$;
17:              EXIT_2(0);
18:              Infast := false
     fi  fi  fi
od

procedure SLOW1()
19: ENTRY_N(p);
20:     ENTRY_2(1);
21:         Critical Section;
22:     EXIT_2(1);
23: EXIT_N(p)

procedure SLOW2()
24: ENTRY_N(p);
25:     ENTRY_2(1);
26:         Critical Section;
27:         $y := Reset$;
28:         $Reset := (false, y.indx)$;
29:         if $\neg$Name_Taken[y.indx] then
30:             $Reset := (true, y.indx + 1)$;
31:             $Y := (true, y.indx + 1)$
            fi;
32:     EXIT_2(1);
33: EXIT_N(p)
```

Figure 12: ALGORITHM AK: $\Theta(\log N)$ fast-path algorithm with unbounded memory.

within code sequences that execute as critical sections (lines 15–16 and 27–31). Thus, when reasoning about the algorithm, we can assume that these code sequences do not interleave with one another. In fact, as we shall see, this assumption is not merely a matter of convenience — the algorithm's correctness relies *crucially* on the fact that these code sequences execute as critical sections.

One of the problems with Lamport's fast-path code is that it is difficult for a slow-path process that is attempting to reopen the fast path (by resetting Y) to know whether some process is "about to" acquire the fast path. In ALGORITHM AK, this problem is solved by including in Y an additional field *indx*, which is an identifier that is used to "rename" any process that acquires the fast path. (The value of $Y.indx$ is meaningful only if the $Y.free = true$ holds. Otherwise, $Y.indx$ is 0, and $Reset.indx$ holds the "last-used" value of $Y.indx$.) This identifier will increase without bound over time, so we will never have to worry about the possibility that two processes are renamed with the same identifier. With this added field, a slow-path process has a way of identifying a process that has taken the fast path. To see how this works, consider what happens when, starting from the initial state, some set of processes execute their entry sections. At least one of these processes will read $Y = (true, 0)$ at line 3 and assign $Y := (false, 0)$ at line 5. By the correctness of Lamport's fast-path code, of the processes that assign Y, at most one will reach line 8. A process that reaches line 8 will either acquire the fast path by reaching line 12, or will be deflected to $SLOW2$ at line 11.

This gives us two cases to analyze: of the processes that read $Y = (true, 0)$ at line 3 and assign Y at line 5, either all are deflected to $SLOW2$, or one, say p, acquires the fast path. In the former case, at least one of the processes that executes $SLOW2$ finds $Name\_Taken[0]$ to be *false* at line 29, and then reopens the fast path by executing lines 30 and 31, which establish $Y.free = true \ \wedge \ Y.indx > 0 \ \wedge \ Y = Reset$. To see why at least one process executes lines 30 and 31, note that each process under consideration reads $Y = (true, 0)$ at line 3, and thus its $y.indx$ variable equals 0 while executing within lines 5–11. Note also that $Name\_Taken[0] = true$ is established only by line 8. Furthermore, each process that is deflected to $SLOW2$ at line 11 first assigns $Name\_Taken[0] := false$. Thus, at least one of the processes deflected to $SLOW2$ finds $Name\_Taken[0]$ to be *false* at line 29.

In the case that some unique process p has acquired the fast path, we must argue that **(i)** the fast-path process p reopens the fast path upon leaving it, and **(ii)** no $SLOW2$ process "prematurely" reopens the fast path before p has left the fast path. Establishing (i) is straightforward. Process p will reopen the fast path by executing lines 15–18, which establish $Y = (true, 1) \ \wedge \ Infast = false \ \wedge \ Y = Reset$. Note that the assignment to *Infast* at line 18 prevents the reopening of the fast path from actually taking effect until after p has finished executing `EXIT_2`(0).

To establish (ii), suppose, to the contrary, that some $SLOW2$ process reopens the fast path by executing line 31 while p is executing within lines 12–18. Let q be the first such $SLOW2$ process to execute line 31. Since we are assuming that the `ENTRY` and `EXIT` calls are correct, q cannot execute line 31 while p is executing within lines 14–16. Moreover, if p is enabled to execute line 17 or 18, then *Infast* is *true*, and hence the fast path is closed. The remaining possibility is that p is enabled to execute line 12. (Note that if p were enabled to execute line 12, and if q were to reopen the fast path, then we could end up with two processes concurrently invoking `ENTRY_2` or `EXIT_2` at lines 13 and 17 with a virtual process identifier of 0! The `ENTRY_2` and `EXIT_2` calls obviously cannot be assumed to work correctly if such a scenario could happen.)

So, assume that q executes line 31 while p is enabled to execute line 12. For this to happen, q must have read $Name\_Taken[0] = false$ at line 29 before p assigned $Name\_Taken[0] := true$ at line 8. (Recall that all the processes under consideration read $Y = (true, 0)$ at line 3. This is why p writes to $Name\_Taken[0]$ instead of some other element of $Name\_Taken$. q reads from $Name\_Taken[0]$ at line 29 because it is the first process to attempt to reset the fast path, which implies that q reads $Reset = (true, 0)$ at line 27.) Because q executes line 29 before p executes line 8, line 28 is executed by q before line 9 is executed by p. Thus, p must have found $Reset \neq y$ at line 9, *i.e.*, it was deflected to $SLOW2$, which is a contradiction. It follows from the explanation given here that after an initial period of contention ends, we must have $Y.free = true \ \wedge \ Y.indx > 0 \ \wedge \ Y = Reset \ \wedge \ Infast = false$. This argument can be applied inductively to show that the fast path is properly reopened after each period of contention ends. When applied inductively, all reasoning is as above, except that, instead of $Y.indx = 0$, we have $Y.indx = k$ for some $k > 0$. (Note that $Reset.indx$ always holds the current or last-used value of $Y.indx$, and hence $Y.indx$ always increases by one when the fast path is reopened.)

ALGORITHM AK requires unbounded memory because the *indx* field of Y that is used for renaming will continue to grow without bound. In [15], a bounded version is presented in which $Y.indx$ is incremented modulo-N. With $Y.indx$ being updated in this way, the following potential problem arises. A process p may reach line 9 in ALGORITHM AK with $p.y.indx = k$ and then stall. While stalled, other processes may repeatedly increment $Y.indx$ (in $SLOW2$) until it "cycles back" to k. At this point, another process q may reach line 9 with $q.y.indx = k$. This is a problem because p and q may interfere with each other in updating $Name\_Taken[k]$. In the bounded version of ALGORITHM AK, an additional mechanism is introduced to prevent $Y.indx$ from prematurely recycling in this way.

The results of this section establish the following theorem.

Theorem 6 (Anderson and Kim) *The mutual exclusion problem can be solved with* $\Theta(\log N)$ *RMR time complexity under contention and* $O(1)$ *time complexity in the absence of contention using only reads and writes under either the CC or DSM model.* □

Algorithm	**System response time**	**Step time complexity**	**RMR/DSM time complexity**	**Space complexity**	**Contention parameter (k)**
S (Styer [75])	$O(\min(N, k \log N))$	$O(\min(N, k \log N))$	∞	$\Theta(N)$	interval
CS (Choy & Singh [29])	$O(k)$	$O(N)$	∞	$\Theta(N)$	interval
AB (Attiya & Bortnikov [21])	$O(\log k)$	$O(k)$	∞	$\Theta(N \log n)$	point
AB-I (Attiya & Bortnikov [21])	$O(\log k)$	$O(k)$	∞	$\Theta(n \log n)$	interval
AST (Afek, *et al.* [5])	$O(k^4)$	$O(k^4)$	∞	$\Theta(N^3 + n^3 N)$	point
AK-RT (Anderson & Kim [13])	$O(\min(k, \log N))$	$O(\min(k, \log N))$	$O(\min(k, \log N))$	$\Theta(N)$	point
AST-RT (Afek, *et al.* [6])	$O(k^2)$	$O(\min(k^2, k \log N))$	$O(\min(k^2, k \log N))$	$\Theta(N^2)$	interval

Table 1: Adaptive algorithms. k denotes either interval or point contention, as indicated. n denotes an upper bound on the maximum number of processes concurrently active in the system (possibly less than N). (Although ALGORITHM AST uses a bounded number of variables, some of these variables are unbounded.)

5 Adaptive Algorithms

Although fast mutual exclusion algorithms perform well in the absence of contention, in most such algorithms, there is a sudden rise in time complexity when contention is present. In this section, we consider adaptive algorithms, which are designed to alleviate this problem. Formally, a mutual exclusion algorithm is *adaptive* if its time complexity (under some measure) is a function of the number of contending processes. As is the case with fast algorithms, adaptivity is a non-issue if appropriate synchronization primitives are available. Thus, the term "adaptive" is usually applied only to algorithms that use only reads and writes.

Two notions of contention have been considered in the literature: "interval contention" and "point contention" [3]. These two notions are defined with respect to a history H. The *interval contention* over H is the number of processes that are active in H, *i.e.*, that execute outside of their noncritical sections in H. The *point contention* over H is the maximum number of processes that are active at the *same state* in H. Note that point contention is always at most interval contention. In this section, unless stated otherwise, we let k denote the contention (point or interval) experienced by an arbitrary process over a history that starts when that process becomes active and ends when it once again becomes inactive. We also let n denote an upper bound on the maximum number of processes concurrently active in the system (possibly less than N). The algorithms considered in this section are summarized in Table 1.

Several different time complexity measures have been applied in work on adaptive algorithms. In defining a meaningful time complexity measure for concurrent algorithms, dealing with potentially unbounded busy-waiting loops is the main difficulty to be faced. Indeed, under the standard sequential-algorithms measure of counting all operations, such a busy-waiting loop has unbounded time complexity, which is not a very interesting or useful statistic. The *step time complexity*[3] of an algorithm is the maximum number of shared-memory operations required by a process to enter and then exit its critical section, assuming that each "**await**" statement is counted as one operation [75]. This measure simply ignores repeated memory references generated by a process while it waits. Another measure, proposed in [29], is the *system response time*, which is the length of time between critical section entries, assuming every active process performs at least one step within some constant time bound, termed a *time unit* [29]. By forcing active processes to take steps, unbounding waiting times are precluded, provided each waiting condition in an algorithm is eventually established by some process within a finite number of its own steps. The *amortized system response time* of an algorithm, a measure also proposed in [29], is defined as the average system response time, provided that all k contending processes start execution simultaneously. The RMR time complexity measure, considered earlier, is also of interest in work on adaptive algorithms. For the most part, we will limit attention to the DSM model when using this measure, because, of the algorithms we consider, only ALGORITHMS AK-RT and AST-RT are local-spin algorithms, and both locally spin on DSM machines.

Before continuing, let us examine the relationship between RMR time complexity and step time com-

[3]Attiya and Bortnikov [21] instead use the term "remote step complexity." This term appears to have been originated by them and not earlier authors. We prefer the term "step time complexity" because the term "remote" may cause this measure to be confused with the RMR measure.

plexity. Under the DSM model, if all **await** statements within an algorithm access only locally-accessible variables, then each statement has $O(1)$ step time complexity and $O(1)$ RMR time complexity.[4] On the other hand, if an **await** statement accesses a non-local variable, then the algorithm's RMR time complexity is obviously unbounded. Under the CC model, any variable accessed in an **await** statement will be brought into a local cache line. However, a single **await** statement might generate a large number of cache misses if the variables it accesses keep changing without satisfying the **await** condition. The step time complexity measure ignores the remote references caused by these misses. Therefore, in general, an algorithm's step time complexity and RMR time complexity need not be the same.

5.1 Early Algorithm L Variants

Algorithm L was a stepping stone leading to some of the earliest adaptive algorithms. One of the problems with Algorithm L is that, when contention is detected, a process must poll *every* other process in order to enter its critical section. As mentioned at the end of Section 4.1, Merritt and Taubenfeld proposed a variant of Algorithm L in which a linked list of active processes is scanned, rather than an array of per-process status variables [63]. However, their algorithm requires an external mechanism for inserting active processes into the list and for removing ones that are no longer active. (These operations must be done in a critical section in order to ensure correctness. Hence, unless we assume an external mechanism, another solution of the mutual exclusion problem is required.) Thus, it is not a true adaptive algorithm in the sense considered here.

Algorithm S. Styer [75] was probably the first to propose using Algorithm L as a building block to create an adaptive algorithm. Styer's algorithm, denoted Algorithm S, is illustrated in Figure 13(a). The algorithm uses an m-ary l-level arbitration tree, where $N \leq m^l$. Each process enters the arbitration tree through a designated leaf node. Associated with each node is an m-process "slow" mutual exclusion algorithm. A process first tries to enter the fast path using Algorithm L, and if it fails, it traverses the arbitration tree from its leaf up to the root, executing the slow algorithm l times along the way.

Each node has an associated variable that marks the last "winner" of that node. If process p takes the fast path, then it also marks itself as the winner of all the nodes from its leaf node up to the root, as shown in Figure 13(b). (Thus, in Algorithm S, the "fast path" takes $\Theta(l)$ time.) If another process q takes the slow path and wins the arbitration tree, then q may also mark itself as the winner of each node in its path, thereby overwriting some of p's markings (Figure 13(c)). After winning the arbitration tree, q determines if a fast-path process (in this case p) exists by traversing the tree recursively, possibly skipping some subtrees (Figure 13(d)). If node i has no winner, then it cannot be an ancestor of a node marked by p, so process q does not have to recursively visit its children. However, if node i has a slow winner, then the marking of p might have been overwritten by this slow process, so process q recurses to its children in order to detect if p is in the subtree. (It is possible that process q checks node i after p has marked some of i's descendants but not i. In this case, q will fail to detect process p. However, if p tries to enter its critical section before q finishes completely, it will detect contention and enter the slow path.)

From the above discussion, it should be clear that the recursion visits a node only if it is an ancestor of some participating process. (We do not consider a node to be "visited" if a process does not recurse into its subtrees. When a node is visited, all of its children are also checked.) Since the tree has l levels, each of the k participating processes has l such ancestors, where k is interval contention. Since visiting each such node takes $O(m)$ remote steps, the entire recursion takes $O(klm)$ remote steps. Since there are $\Theta(N/m)$ non-leaf nodes in total, the entire recursion also takes $O(N)$ remote steps. (A leaf node is not "visited" under our definition, because it does not have any children.) It follows that the step time complexity of the recursion is $O(\min(N, klm))$. Assuming $m = O(1)$, in which case $lm = \Theta(\log N)$, Algorithm S has $O(\min(N, k \log N))$ step time complexity and system response time.

[4]This conclusion is based upon the assumption that, under the step time complexity measure, each **await** statement has constant cost regardless of the number of variables it accesses. It is not clear if this is reasonable if the number of variables accessed is a function of N. In all papers where step time complexity is used, only **await** statements that access a constant number of variables are considered.

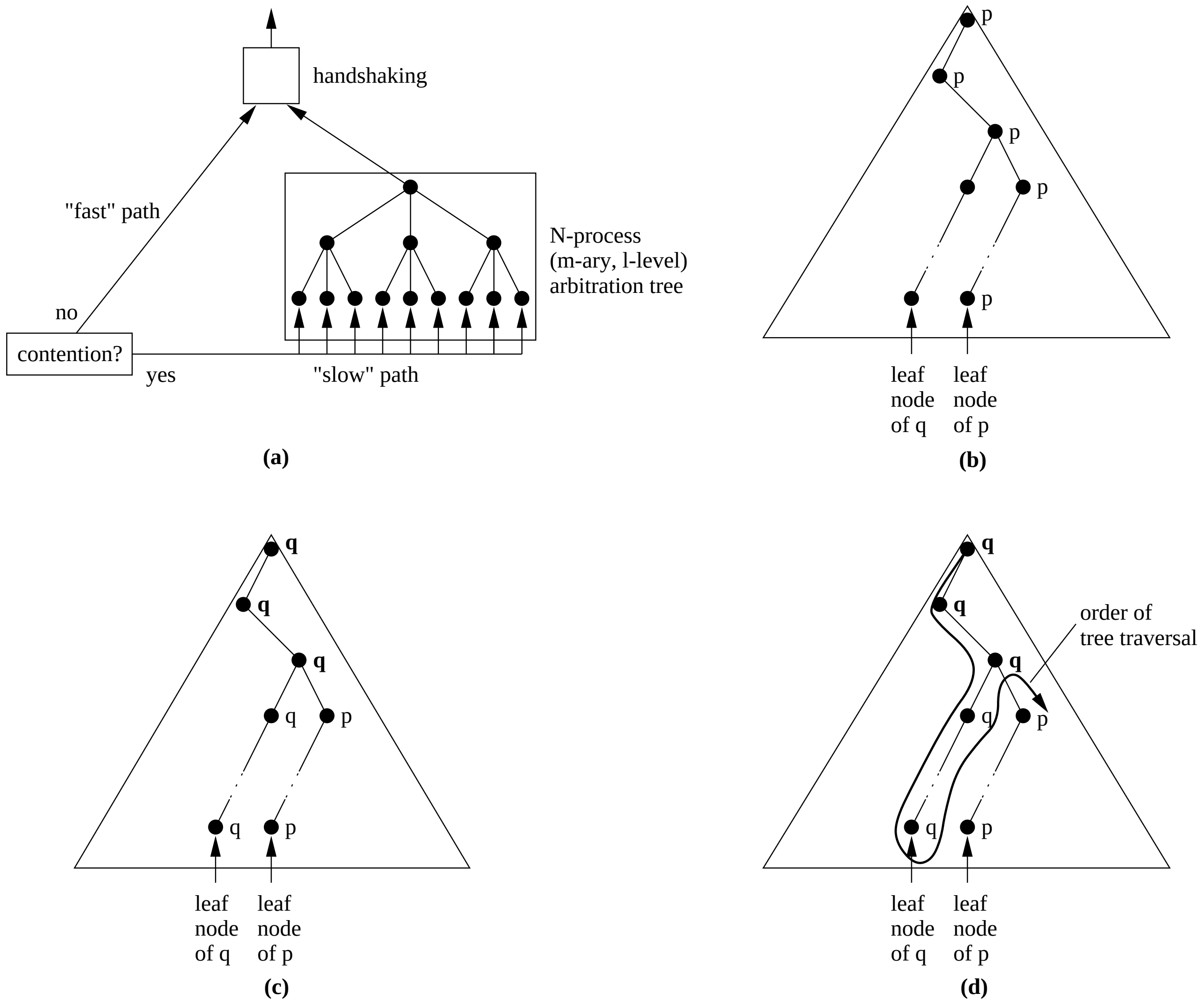


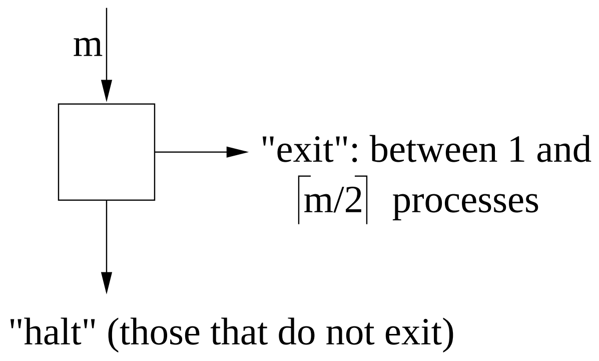
Figure 13: **(a)** Structure of Algorithm S. **(b)** Node marking after p acquires the fast path. **(c)** If q acquires the slow path, then it may partially overwrite p's marking. **(d)** q traverses the tree to check if a fast process is "hidden" by overwriting — only the subtrees with roots marked by slow processes need to be traversed.

5.2 Algorithm CS: Choy and Singh's Filter Algorithm

Choy and Singh [29] devised a novel code fragment called a *filter*, which is shown in Figure 14. A process executing a filter either exits (succeeds) or halts (fails). A filter satisfies the following two properties.

- **Safety:** If m processes enter the filter, then at most $\lceil m/2 \rceil$ processes exit.
- **Progress:** If some processes enter the filter, then at least one of them exits.

To see that the progress property holds, let p be the last process to execute line 1. If no other process exits, then all the halting processes will assign $b := \mathit{false}$ at line 5. Therefore, p will eventually find $b = \mathit{false}$ at line 2 and exit successfully. Now, consider the safety property. Assume that e processes exit. Note that, for any two exiting processes, their executions of lines 1–4 do not overlap. (If an exiting process p executes line 1 before another exiting process q does, then p must execute line 4 before q executes line 1, in order to exit.) While such a process executes lines 1–4, the value of b changes from *false* to *true* at least once.


m
"exit": between 1 and ⌈m/2⌉ processes
"halt" (those that do not exit)

(a)

```
shared variable
    turn: 1..N;
    b: boolean initially false

process p              /* $1 \le p \le N$ */
1:  turn := p;
2:  await ¬b;
3:  b := true;
4:  if turn ≠ p then
5:      b := false;
6:      halt
    else
7:      exit
    fi
```

(b)

Figure 14: **(a)** The filter element and **(b)** its implementation.

Therefore, the value of b must change from *true* to *false* at least $e-1$ times, so there are at least $e-1$ halting processes. The safety property follows from the inequality $e + (e-1) \le m$.

Leader election using filters. A leader-election algorithm can be constructed by concatenating filters so that only those processes that exit from a filter move to the next. The algorithm is shown in Figure 15(a).

We concatenate $\lceil \log_2 N \rceil + 1$ filters, indexed from 0. Since each filter halves the number of exiting processes, at most one process may exit from filter $A[\lceil \log_2 N \rceil]$. When a process p exists from filter $A[i+1]$, it checks filter $A[i]$ in order to determine if p is the only process that has exited from filter $A[i]$. If this is the case, then p elects itself as the leader. (This is why we need one additional filter after $A[\lceil \log_2 N \rceil]$.) The detailed mechanism is explained next.

Each filter has an additional variable c, which is initially *false* and set to *true* when a process fails at that filter. Once c is set to *true*, it remains *true*. Therefore, if process p exits from filter $A[i+1]$ and finds $A[i].c = \mathit{false}$, then no process has (yet) failed at filter $A[i]$. It follows that $A[i].b$ has (so far) been changed from *false* to *true* only once (by process p at line 3). Thus, no process other than p has (yet) exited from filter $A[i]$. Moreover, any process that subsequently exits from $A[i]$ will become blocked at the **await** of filter $A[i+1]$. Thus, p can elect itself as the leader. From the filter safety property, at most one process enters filter $A[\lceil \log_2 N \rceil]$, and hence $\Theta(\log N)$ filters suffice. Thus, the algorithm's space complexity is $\Theta(\log N)$.

Algorithm CS**: Unbounded version.** The election algorithm can be used to create a simple unbounded mutual exclusion algorithm. This algorithm is shown in Figure 15(b). The algorithm uses an unbounded number of filters, $A[0]$, $A[1]$, ... All competing processes participate in the election algorithm using the set of filters $A[\mathit{Entry}]$, $A[\mathit{Entry}+1]$, ... , where *Entry* is a shared variable that "points" to the next unused filter. If a process is elected as the leader, then it enters its critical section. In its exit section, the leader initiates another round of the election algorithm by updating *Entry*. (Since each filter is properly initialized and used only once, a process only has to update *Entry* to point to a new filter.) The algorithm has $O(k)$ system response time. (The analysis of this is rather complicated and will not be presented here. We refer the interested reader to [29].) However, since a process may repeatedly lose in the election algorithm, the algorithm is not starvation-free.

The arbiter mechanism. The algorithm just described has two shortcomings: it uses unbounded memory, and is not starvation-free. Choy and Singh proposed an *arbiter* mechanism as a solution to both problems. In this mechanism, a shared variable *Arbt* is used to ensure that each process in its entry section eventually is accorded a chance to enter its critical section. The *Arbt* pointer is updated each time a critical section

```
shared variable
    A: array[0..⌈log_2 N⌉ + 1] of record
        turn: 1..N;
        b, c: boolean initially false end

process p                    /* 1 ≤ p ≤ N */

private variable
    curr: 0..⌈log_2 N⌉ + 1

curr := 0;
while true do
1:  A[curr].turn := p;
2:  await ¬A[curr].b;
3:  A[curr].b := true;
4:  if A[curr].turn ≠ p then
5:      A[curr].c := true;
6:      A[curr].b := false;
7:      halt
8:  else if curr > 0 ∧ A[curr − 1].c then
9:      elected
    else
10:     curr := curr + 1
    fi
od
```

(a)

```
shared variable
    A: array[0..∞] of record
        turn: 1..N;
        b, c: boolean initially false end;
    Entry: 0..∞ initially 0

process p                    /* 1 ≤ p ≤ N */

private variable
    curr, lentry: 0..∞;
    elected: boolean

while true do
1:  Noncritical Section;
2:  elected := false;
3:  while ¬elected do
4:      lentry, curr := Entry, Entry;
5:      while ¬elected ∧ Entry = lentry do
6:          A[curr].turn := p;
7:          await (¬A[curr].b ∨
8:                     Entry ≠ lentry);
9:          if Entry = lentry then
10:             A[curr].b := true;
11:             if A[curr].turn ≠ p then
12:                 A[curr].c := true;
13:                 A[curr].b := false;
14:                 await Entry ≠ lentry
15:             else if curr > lentry ∧ A[curr − 1].c then
16:                 elected := true
                else
17:                 curr := curr + 1
            fi fi
        od
    od;
18: Critical Section;
19: Entry := curr + 1
od
```

(b)

Figure 15: **(a)** A leader-election algorithm using filters. **(b)** Unbounded version of ALGORITHM CS.

is executed, and cycles through the processes in a round-robin fashion. When a process p exits its critical section, it first checks whether the process pointed to by *Arbt* is in its entry section. If so, process p signals that process to immediately enter its critical section. The next election algorithm starts only after the process pointed to by *Arbt* finishes its critical section.

In this way, a process is guaranteed to enter its critical section after at most N critical-section executions. Since each election round uses $O(\log N)$ filters, and since no process is left in an old election round after N rounds, the number of filters can be limited to $\Theta(N \log N)$. Also, since each election algorithm uses $O(\log k)$ filters, where k is the interval contention over the election period, and since each filter has $O(1)$ step time complexity, the entire algorithm has $O(N \log k)$ step time complexity.

Choy and Singh also presented further optimizations, which result in $O(k)$ system response time, $O(1)$ amortized system response time, and $O(N)$ step time complexity.

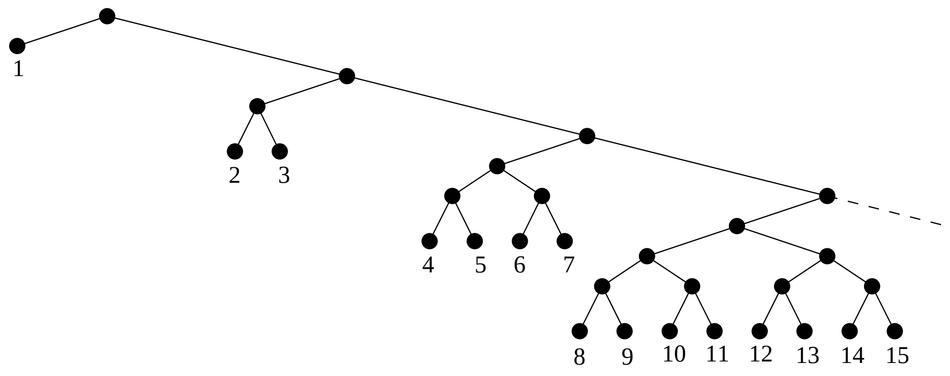


Figure 16: An adaptive tournament tree.

5.3 Algorithm AB: Attiya and Bortnikov's Improved Filter Algorithm

Attiya and Bortnikov [21] devised an improved filter algorithm, which has $O(k)$ step time complexity and $O(\log k)$ system response time. Their algorithm consists of two parts: a *non-wait-free* long-lived renaming algorithm using filters, and an adaptive tournament tree (illustrated in Figure 16). (The *long-lived* renaming problem [66] extends the renaming problem briefly mentioned in Section 4.1 by requiring that processes have the ability to both acquire new names and also release previously-acquired names.) An unbounded version of the renaming algorithm is shown in Figure 17. (Note the similarity between the election algorithm of Figure 15(a) and procedures executeChain and executeFilter.) There has been much prior work on wait-free long-lived renaming algorithms, but no such algorithm has been proposed that is efficient enough to be used in an adaptive mutual exclusion algorithm as done here. The key insight of Attiya and Bortnikov is that a *renaming algorithm used within a mutual exclusion algorithm does not have to be wait-free.* (Recall that Algorithm AK is based on a similar insight.)

In Algorithm AB, a process first obtains a name within a range of size $O(k)$, where k is point contention. This name is then used to enter an adaptive tournament tree for mutual exclusion. Such a tournament tree can be implemented using any two-process mutual exclusion algorithm as a building block. If a constant-time algorithm such as Algorithm YA is used, then a process with a name within a range of size $O(k)$ enters and exits the adaptive tournament tree with a step time complexity of $O(\log k)$. (In fact, if Algorithm YA is used, then the time complexity for entering and exiting the tournament tree is $O(\log k)$ under the stronger RMR measure.) System response time is also $O(\log k)$.

The renaming algorithm uses an array A of n chains, where each chain consists of an unbounded number of filters (numbered 0, 1, ...), and n is an upper bound on the maximum number of processes concurrently active in the system, possibly less than N. (This is the only algorithm we cover that distinguishes between n and N.) In addition, another shared array *StartFilter* is used: *StartFilter*$[l]$ contains the index of the entry filter for the current round of the filter election algorithm at $A[l]$.

A process executes within successive chains, starting with chain 0, until it wins the election algorithm within some chain. The name that is acquired by the process is the index of the chain it wins.

A process p tries to win a chain by calling executeChain, which executes a leader-election algorithm that is very similar to the one shown in Figure 15(a). If p wins within chain l, then it acquires name l and proceeds to the tournament tree. However, if p loses within chain l, then by the mechanism of the election algorithm, it fails in a filter inside the chain. Suppose p fails at the r^{th} filter of chain l. In that case, p skips the next $r-1$ chains and tries to win in chain $l+r$.

Procedure executeFilter is invoked to execute a filter. Unlike the original filter of Choy and Singh depicted in Figure 14(b), this filter has an additional variable d, which is set whenever a process succeeds in the filter. Variable d is checked in lines 16, 19, and 20 in order to ensure that a process promptly fails whenever it detects another process's success. With these (and other) changes, Attiya and Bortnikov's filter ensures not only the safety and the progress properties of the original algorithm, but also the following.

```
constant
    Success = true;
    Fail = false;
    Win = true;
    Lose = false

type
    FilterType: record
        turn : 1..N;
        b, c, d: boolean initially false end

shared variables
    A: array[0..n − 1][0..∞] of FilterType;
    StartFilter: array[0..n − 1] of 0..∞

process p    /* 1 ≤ p ≤ N */

private variables
    lastfilter: 0..∞;
    l, indx, lentry, curr: 0..∞;
    result: {Win, Lose}

procedure getName() returns 0..n − 1
1:  l, indx := 0, −1;
    repeat
2:      l := l + indx + 1;
3:      (result, indx) := executeChain(l)
4:  until result = Win;
5:  lastfilter := StartFilter[l] + indx;
6:  return l

procedure releaseName(name)
7:  StartFilter[name] := lastfilter + 1

procedure executeChain(ch: 0..n − 1)
        returns ({Win, Lose}, 0..n − 1)    /* access chain ch */
8:  lentry := StartFilter[ch];
    curr := lentry;
                        /* move the pointer (curr) to the start */
    while true
9:      if executeFilter(ch, curr) = Success then
10:         if curr > lentry ∧ ¬A[ch][curr − 1].c then
11:             return (Win, curr − lentry)
            else
12:             curr := curr + 1 fi
                                /* proceed to the next filter */
        else
13:         return (Lose, curr − lentry)
        fi
    od

procedure executeFilter(ch: 0..n − 1, curr: 0..∞)
        returns {Success, Fail}    /* access a specific filter */
14: if A[ch][curr].turn ≠ ⊥ then return Fail fi;
15: A[ch][curr].turn := p;
16: if A[ch][curr].d then return Fail fi;
17: await (¬A[ch][curr].b ∨
18:             A[ch][curr].turn ≠ p ∨
19:             A[ch][curr].d);
20: if A[ch][curr].d then return Fail fi;
21: if A[ch][curr].turn = p then
22:     A[ch][curr].b := true fi;
23: if A[ch][curr].turn ≠ p then
24:     A[ch][curr].c := true;
25:     A[ch][curr].b := false;
26:     return Fail
    else
27:     A[ch][curr].d := true;
28:     return Success
    fi
```

Figure 17: Renaming procedures of ALGORITHM AB (unbounded version).

- **Time Complexity:** Some process succeeds in the filter $O(1)$ time units after the first process enters it, where a *time unit* is a constant time bound such that every active process performs at least one step within it.

As in the original election algorithm, the progress and safety properties ensure that if $O(k)$ processes enter chain l, then some process wins the chain within $O(\log k)$ filters. Applying the time complexity property stated above, it follows that some process acquires name l within $O(\log k)$ time units. It can be shown that $l < k$ holds for every winning process. Thus, if a process leaves its critical section while others are waiting, then some process wins the tournament tree and enters its critical section within $O(\log k)$ time units. From these assertions, it is possible to show that the algorithm's system response time is $O(\log k)$. (We refer the interested reader to [21] for details.) The algorithm also has $O(k)$ step time complexity, but the analysis is rather complicated and will be omitted here.

By using a mechanism similar to the "arbiter" of [21], Attiya and Bortnikov constructed a bounded algorithm with $O(N \log n)$ space complexity, while maintaining $O(k)$ step time complexity and $O(\log k)$ system response time. They also presented another algorithm with $O(n \log n)$ space complexity, but at the price of defining k as *interval* contention. This variant is listed as ALGORITHM AB-I in Table 1. ("I" stands

```
shared variables
    $Number$: array[0..$N-1$] of $0..\infty$ initially 0;
    $Choosing$: array[0..$N-1$] of boolean initially $false$
process $p$                                /* $0 \le p < N$ */
private variables
    $q$: $0..N-1$;
    $S$: (a set of processes)
while $true$ do
1:  Noncritical Section;
2:  Join($p$);
3:  $Choosing[p] := true$;
4:  $S :=$ Get_Set();
5:  $Number[p] := 1 + \max_{q \in S} Number[q]$;
6:  $Choosing[p] := false$;
7:  $S :=$ Get_Set();
8:  for each $q \in S - \{p\}$ do
9:      await $\neg Choosing[q]$;
10:     await $(Number[q] = 0) \vee (Number[p], p) < (Number[q], q)$
    od;
11: Critical Section;
12: $Number[p] := 0$;
13: Leave($p$)
od
```

Figure 18: Algorithm AST: An adaptive bakery algorithm.

for "interval.") From these results, we have the following theorem.

Theorem 7 (Attiya and Bortnikov) *The mutual exclusion problem can be solved with $O(\log k)$ system response time and $O(k)$ step time complexity using only reads and writes, where k is point contention.*

5.4 Algorithm AST: An Adaptive Bakery Algorithm

Afek, Stupp, and Touitou [5] constructed an adaptive bakery algorithm, by combining Lamport's bakery algorithm [52] with a wait-free *active set* object. Their algorithm, denoted Algorithm AST, is illustrated in Figure 18. An active set object manages a set of processes, and each process p may execute the following three operations.

- Join(p): adds p to the active set.
- Leave(p): removes p from the active set.
- Get_Set(): returns the current active set S. If a process q's execution of Join(q) or Leave(q) overlaps p's execution of Get_Set, then S is allowed to either contain q or not.

Afek, *et al.* also devised a wait-free implementation of an active set object based on a generic adaptive long-lived renaming algorithm. Several such algorithms are presented in [2, 3], with various tradeoffs among the following parameters: the size of the output name space, step time complexity, space complexity, and whether the algorithm is adaptive to interval or point contention. By using an $O(k^2)$-renaming algorithm presented in [2], the active set algorithm of Afek, *et al.* achieves $O(k^4)$ step time complexity, where k is point contention.

If we remove lines 2 and 13 of Algorithm AST and change lines 4 and 7 to $S := \{0..N-1\}$, then we obtain the original bakery algorithm. Due to these four lines, Algorithm AST only has to check processes that are concurrently active, instead of checking every process in the system. Thus, the **for** loop of lines 8–10 has $O(k)$ step time complexity.

Clearly, overall time complexity is dominated by the active set algorithm, and hence ALGORITHM AST has $O(k^4)$ step time complexity. Since the active set algorithm is wait-free, it can be shown that ALGORITHM AST also has $O(k^4)$ system response time. Unfortunately, as in the original bakery algorithm, each process's *Number* variable in ALGORITHM AST has unbounded range. (The total number of shared variables is $\Theta(N^3 + n^3 N)$.)

5.5 Local-spin Adaptive Algorithms

In a recent paper [13], Anderson and Kim presented a local-spin adaptive algorithm with $O(\min(k, \log N))$ RMR time complexity that is based on ALGORITHM AK. In another paper [6], Afek, Stupp, and Touitou independently devised another local-spin adaptive algorithm with a very similar structure, which is based on a long-lived splitter algorithm [5]. We call these adaptive algorithms ALGORITHM AK-RT and ALGORITHM AST-RT, respectively. ("RT" stands for "renaming tree" — see below.) None of the adaptive algorithms covered above are local-spin algorithms, and thus they have unbounded RMR time complexity under the DSM model. As such, one might question whether these algorithms are truly adaptive. Indeed, Styer's algorithm was shown to perform poorly under contention in a performance study conducted by Yang and Anderson to compare local-spin algorithms with non-local-spin algorithms [80]. It is our belief that for an algorithm to be considered truly adaptive, it must be adaptive under the RMR time complexity measure. After all, the underlying hardware does *not* distinguish between remote memory references generated by **await** statements and remote memory references generated by other statements. ALGORITHMS AK-RT and AST-RT are the only mutual exclusion algorithms (based on reads and writes) known to us that are adaptive according the RMR time complexity measure under the DSM model. Below, we sketch the key ideas of ALGORITHM AK-RT. We then consider ALGORITHM AST-RT.

ALGORITHM AK-RT. As explained in Section 4.2, the main idea behind ALGORITHM AK is to associate a particular name with the fast path; to take the fast path, a process must first acquire this name. Like ALGORITHM AB, ALGORITHM AK exploits the fact much of the code that must be executed to acquire and release the fast-path name can be executed within a process's critical section. ALGORITHM AK-RT extends ALGORITHM AK by using a set of names, each of which is associated with some "path" to the critical section. The length of the path taken by a process is determined by the point contention that it experiences.

In ALGORITHM AK-RT, a long-lived renaming algorithm is used that is based on the grid-of-splitters algorithm of Moir and Anderson [66] depicted earlier in Figure 10. *Long-lived* splitter elements are used. These splitter elements, which have a slightly more complicated implementation than that in Figure 9(c), can be repeatedly acquired and released: a process *acquires* a splitter if it stops at that splitter; a process *releases* a splitter it has acquired by resetting some of the variables that define the splitter so that it can be acquired once again by some process. (For example, the Y variable used in the splitter should be reset to *true*.) In the grid algorithm, a process can release its name by releasing each splitter on the path traversed by it in acquiring its name. For the renaming mechanism to work correctly, it is important that a splitter be released *only* if there are no processes "downstream" from it (*i.e.*, in the sub-grid "rooted" at that splitter). In Moir and Anderson's algorithm, it takes $O(N)$ time to determine if there are "downstream" processes. This is because each process checks every other process individually to determine if it is downstream from a splitter. Clearly, a more efficient release mechanism is needed for an adaptive mutual exclusion algorithm.

The main idea behind ALGORITHM AK-RT is to let an arbitration tree form dynamically within a structure similar to the renaming grid. This tree may not remain balanced, but its height is bounded by a constant multiple of point contention. To simplify the job of integrating the renaming aspects of the algorithm with the arbitration tree, a binary tree of splitters is used instead of a grid, as depicted in Figure 19. (Since we are now working with a tree, we will henceforth refer to the directions associated with a splitter as *stop*, *left*, and *right*.) Note that this results in many more names than before. However, this is not a major concern, because we are really not interested in minimizing the name space. The arbitration tree is defined by associating a three-process mutual exclusion algorithm with each node in the renaming tree. This

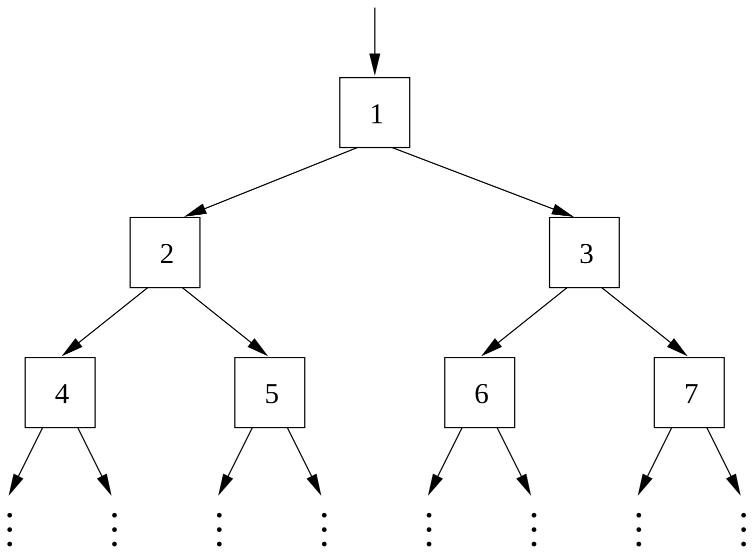


Figure 19: Renaming tree.

three-process algorithm can be implemented with constant RMR time complexity using Algorithm YA [80]. We explain below why a three-process algorithm is needed instead of a two-process algorithm (as one would expect to have in an arbitration tree).

In Algorithm AK-RT, a process p performs the following basic steps.

Step 1 p first acquires a new name by moving down from the root of the renaming tree, until it stops at some node. In the steps that follow, we refer to this node as p's *acquired node*. p's acquired node determines its starting point in the arbitration tree.

Step 2 p then competes within the arbitration tree by executing each of the three-process entry sections on the path from its acquired node to the root. A node's entry section may be invoked by the process that stopped at that node, and one process from each of the left and right subtrees beneath that node. This is why a three-process algorithm is needed.

Step 3 After competing within the arbitration tree, p executes its critical section.

Step 4 Upon completing its critical section, p releases its acquired name by releasing all of the splitters on the path from its acquired node to the root.

Step 5 After releasing its name, p executes each of the three-process exit sections on the path from the root to its acquired node.

In Algorithm AK-RT, the renaming tree's height is defined to be $\lfloor \log_2 N \rfloor$, which results in a tree with $\Theta(N)$ nodes. With a tree of this height, a process could "fall off" the end of the tree without acquiring a name. However, this can happen only if contention is $\Omega(\log N)$. To handle processes that "fall off the end," a second arbitration tree, called the "overflow tree," is used. The overflow tree is implemented using Algorithm YA. The renaming and overflow trees are connected by placing a two-process version of Algorithm YA on top of each tree, as illustrated in Figure 20(a). Figure 20(b) illustrates the steps that might be taken by a process p in acquiring a new name if contention is $O(\log N)$. Figure 20(c) illustrates the steps that might be taken by a process q if contention is $\Omega(\log N)$.

From the discussion above, it follows that the system response time, step time complexity, and RMR time complexity of Algorithm AK-RT are all $O(\min(k, \log N))$, where k is point contention. As shown in [13], space complexity is $\Theta(N)$ if the linear-space variant of Algorithm YA discussed at the end of Section 3.2 is used in implementing the renaming and overflow trees. Thus, we have the following theorem.

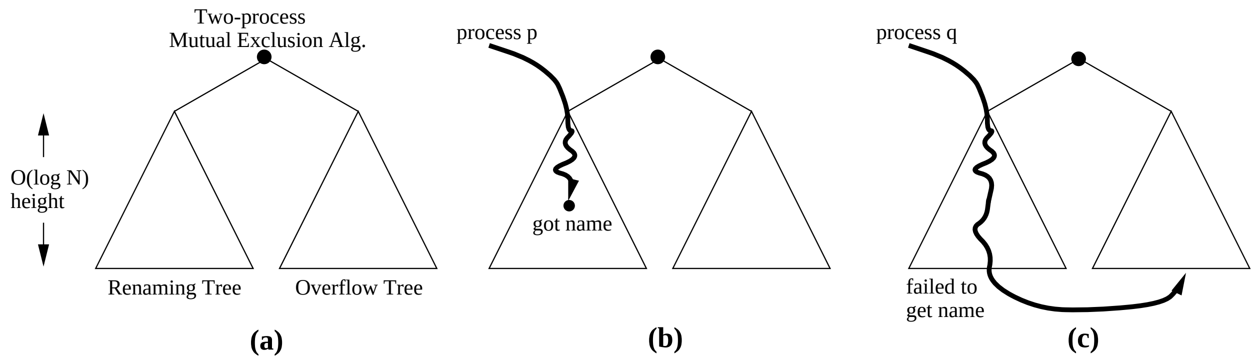


Figure 20: **(a)** Renaming tree and overflow tree. **(b)** Process p gets a name in the renaming tree. **(c)** Process q fails to get a name and must compete within the overflow tree.

Theorem 8 (Anderson and Kim) *The mutual exclusion problem can be solved with $O(\min(k, \log N))$ RMR time complexity (and step time complexity and system response time) and $\Theta(N)$ space complexity using only reads and writes under either the CC or DSM model, where k is point contention.*

It would be interesting to know whether the better features of Algorithms AB and AK-RT could be combined. We leave this as an open problem.

Open Problem: Can the mutual exclusion problem be solved with $O(k)$ RMR time complexity under the DSM model *and* with $O(\log k)$ system response time?

Algorithm AST-RT. As mentioned before, Afek, Stupp, and Touitou [6] devised a local-spin adaptive algorithm with the same tree structure as shown in Figures 19 and 20, independently of Anderson and Kim.

The most challenging problem in using the renaming tree is the releasing of splitters once a process finishes its critical section (Step 4 of Algorithm AK-RT). In Algorithm AK-RT, the problem is solved by exploiting the fact that releasing can be done inside a critical section. In Algorithm AST-RT, the problem is solved by adopting a *long-lived adaptive splitter*, which supports two functions Acquire() and Release() [3]. Function Acquire() returns *stop*, *right* or *down*, as in the splitter of Figure 9(c). If a process acquires a splitter (*i.e.*, invokes Acquire() and gets the return value *stop*), then it must later release the splitter by calling Release(). Processes that fail to acquire a splitter do not have to release it. In addition, a long-lived splitter satisfies the following properties.

- If a process invokes Acquire() in the absence of contention, then it acquires (*i.e.*, stops at) the splitter.
- At most one process may acquire the splitter at any time.
- If n processes invoke Acquire() on an unacquired splitter, and if all of them fail to acquire the splitter, then at most $n - 1$ may move right and at most $n - 1$ may move down.

Because of these properties, Steps 4 and 5 of Algorithm AK-RT can be replaced by the following steps, as done in Algorithm AST-RT.

Step 4 p executes each of the three-process exit sections on the path from the root to its acquired node.

Step 5 If p has stopped inside the renaming tree at step 1, then p releases the single splitter it has acquired.

As shown in [3], by using $3N$ copies of the splitter element, a long-lived adaptive splitter can be constructed that has $O(k)$ step time complexity, $O(k)$ RMR time complexity (under the DSM model), and $\Theta(N)$ space complexity, where k is interval contention. (Interested readers are referred to [3] for details.)

```
shared variable $Y$: $0..N$ initially $0$
process $p$                         /* $1 \le p \le N$ */
while true do
1: Noncritical Section;
     repeat
2:        await $Y = 0$;
3:        $Y := p$;
4:        $delay(\Delta)$
5:   until $Y = p$;
6:   Critical Section;
7:   $Y := 0$
od
```

Figure 21: Algorithm F: A simple timing-based algorithm.

Since each process accesses at most $\min(k, \log N)$ splitters in the renaming tree, Algorithm AST-RT has $O(\min(k^2, k \log N))$ step and RMR time complexity, and $\Theta(N^2)$ space complexity. In can also be shown that Algorithm AST-RT achieves $O(k^2)$ system response time.

6 Timing-based Algorithms

In the preceding sections, we have assumed an *asynchronous* system model: although an enabled statement is eventually executed, no bounds have been stipulated on how long it takes such a statement to complete execution. In practice, statement-execution bounds depend on the manner in which processes are scheduled (and, of course, on various aspects of the hardware on which these processes run). In many systems, characteristics of schedulers can be exploited to deduce reasonably accurate execution bounds. For example, if a scheduling quantum is used, and if a mutual exclusion algorithm is always invoked by a process at the beginning of a new quantum, then it may be possible to deduce that a process will not be preempted by another process while executing the algorithm. In this case, an upper bound on the time required to complete any statement can be fairly accurately deduced. For such systems, it is reasonable to consider whether the availability of timing information can be exploited to design more efficient mutual exclusion algorithms. We refer to algorithms that exploit such information as *timing-based* algorithms.

In work on timing-based algorithms, two system models have been considered. In the *known-delay* model, there is a known upper bound, denoted Δ, on the time required to access (read or write) a shared variable. In the *unknown-delay* model, there still is an upper bound Δ, but its value is unknown *a priori* — thus, the value Δ cannot be directly used in an algorithm. In either model, a process may delay its execution by d time units by executing the statement $delay(d)$.

The earliest timing-based mutual exclusion algorithm was an unpublished algorithm of Fischer, which is described in Lamport's paper on fast mutual exclusion [56]. In the paragraphs that follow, we describe Fischer's algorithm. After this description, an overview of timing-based algorithms published more recently is given.

Algorithm F. Fischer's algorithm, denoted Algorithm F here and shown in Figure 21, is based on the known-delay model. The algorithm is quite simple. A process p waits at line 2 until the "lock" is available, which is indicated by $Y = 0$. If p reads $Y = 0$, then it writes its process identifier to Y at line 3, and delays for Δ time units at line 4. If the value of Y is still p after the delay (line 5), then p enters its critical section.

To see that Algorithm F satisfies the exclusion property, assume to the contrary that two processes p and q execute their critical sections at the same time. Without loss of generality, assume that p executed line 3 before q did. Since p read $Y = p$ at line 5, p's execution of line 5 precedes q's execution of line 3. However, q must have read $Y = 0$ at line 2, and hence q's execution of line 2 precedes p's execution of line 3. Therefore, we have established the following sequence of executions: q executes line 2, p executes lines 3–5,

```
shared variables
      X: 1..N;
      Y: 0..N initially 0;
      Z: boolean initially false
process p                                   /* 1 ≤ p ≤ N */
while true do
1:    Noncritical Section;
Start:
2:    X := p;
3:    await Y = 0;
4:    Y := p;
5:    if X ≠ p then
6:          delay(2 · Δ);
7:          if Y ≠ p then go to Start fi;
8:          await ¬Z
      else
9:          Z := true
      fi;
10:   Critical Section;
11:   Z := false;
12:   if Y = p then
13:         Y := 0
      fi
od
```

Figure 22: ALGORITHM AT: A fast timing-based algorithm.

and q executes line 3. For this to happen, q must have delayed for more than Δ time units before executing line 3, a contradiction.

To see that the algorithm is livelock-free, note that among the processes that execute line 3, the last one to do so will read its own identifier at line 5 and enter its critical section. Unfortunately, ALGORITHM F is not starvation-free: an unfortunate process may fail to exit the loop of lines 2–5 if Y is repeatedly overwritten by other processes.

Another drawback of ALGORITHM F is that a process must delay itself even in the absence of contention. This problem was later addressed by Lamport, who constructed a *fast* timing-based algorithm (*i.e.*, one in which a process performs $O(1)$ shared-memory accesses and executes no delay statements in the absence of contention) [56]. However, his algorithm requires an upper bound not only on the time required for each shared-memory access, but also on the duration of each critical-section execution.

ALGORITHM AT. Alur and Taubenfeld later improved upon Lamport's algorithm by constructing a fast timing-based algorithm in which no bound on critical-section execution times is assumed [10]. Their algorithm, denoted ALGORITHM AT, is shown in Figure 22. By examining Figures 9(c) and 21, it can be seen that ALGORITHM AT was constructed by combining the features of Lamport's splitter and ALGORITHM F.

We now show that ALGORITHM AT satisfies the exclusion property. First, note that there exist two possible "paths" to enter the critical section. We say that a process p takes *Path A* if it enters its critical section by executing lines 6–8; we say that p takes *Path B* if it enters its critical section by executing line 9.

Note that lines 2–5 of ALGORITHM AT constitute the splitter element, except that line 3 is implemented using busy-waiting. (Thus, no process may move "right.") At most one process may "stop" at the splitter and enter its critical section via Path B. On the other hand, line 6 ensures that, if a group of processes read $Y = 0$ at line 3, then every such process executes line 4 before *any* process reads Y at line 7. Therefore, it is straightforward to see that at most one process may enter its critical section via Path A.

It remains to be shown that Paths A and B cannot be used simultaneously. Assume that a process q

enters its critical section via Path B, while another process p tries to enter its critical section via Path A. We consider two cases. If q executes line 4 before p delays itself at line 6, then the delay ensures that q establishes $Z = \mathit{true}$ (line 9) before p resumes execution. Thus, p reads $Z = \mathit{false}$ at line 8 and is blocked until q executes line 11. On the other hand, assume that q does not execute line 4 before p delays itself at line 6. In this case, q must have executed line 3 before p executed line 4 in order to read $Y = 0$. Thus, q must execute line 4 while p is delayed at line 6, and hence p reads $Y \neq p$ at line 7 and fails to enter its critical section.

To see that Algorithm AT is livelock-free, note that if a number of processes execute within lines 2–8 and no such process reaches its critical section, then some of these processes must write Y at line 4, and because of the waiting condition at line 3, after a time, no further writes to Y occur. It is easy to see that the last process to write Y must eventually reach line 8, and thereafter, its critical section, which is a contradiction. Unfortunately, since a process may execute the **go to** statement at line 7 repeatedly, this algorithm is not starvation-free. Therefore, we have the following theorem.

Theorem 9 (Alur and Taubenfeld) *The mutual exclusion problem can be solved in the known-delay model by a livelock-free timing-based algorithm* (*namely,* Algorithm AT) *that requires only eight memory references in the absence of contention.* □

With more careful reasoning, we can also prove the following. (Interested readers are referred to [10].)

Theorem 10 (Alur and Taubenfeld) *The mutual exclusion problem can be solved in the known-delay model by a livelock-free timing-based algorithm* (*namely,* Algorithm AT) *with the following property: after a critical section ends, the next process to enter its critical section does so after performing at most thirteen memory references and explicitly delaying* (*by invoking delay statements*) *for at most* $4 \cdot \Delta$ *time units.* □

Algorithm AAT. One drawback of Algorithm AT is that the bound Δ must be known *a priori*. By combining Algorithm AT with Lamport's fast-path algorithm (Algorithm L), Alur, Attiya, and Taubenfeld constructed a fast timing-based algorithm that works under the unknown-delay model [7]. Their algorithm, denoted Algorithm AAT, is shown in Figure 23.

In Algorithm AAT, the shared variable *Bound* is used to store the current estimate of Δ. The algorithm consists of three parts: **(a)** Algorithm AT, modified to use the value of *Bound* instead of Δ; **(b)** Algorithm L, which is used to ensure correctness even if the estimate of *Bound* is wrong; and **(c)** a mechanism to estimate the value of Δ and increment *Bound* if the current estimate is wrong. (In Figure 23, parts (a)–(c) are distinguished by roman, *italic*, and **boldface** line numbers, respectively.)

Shared variables X, Y, and Z are the same as in Algorithm AT; variables XX, YY, and B correspond to variables X, Y, and B, respectively, of Algorithm L, shown in Figure 9(a). Variable *Update* indicates whether the value of *Bound* is being updated, and *Trying*[p] indicates whether process p is in its entry section.

Lines 1–14 and 42–47 of Figure 23 constitute Algorithm AT. Lines 2–5 ensure that *Update* is *false* (*i.e.*, the variable *Bound* is not being updated). Lines 6–14 and 42–47 are identical to Algorithm AT, with Δ replaced by *Bound*, except for the additional checks of *Update*.

If the current estimate of Δ is correct, then part (a) ensures that the exclusion property holds. However, if the estimate is too small, then multiple processes may enter lines 15–41. In order to cope with such a situation, Algorithm L is executed after Algorithm AT (lines 15–33, 40–41). Note that, if the estimate is correct, then every process enters its critical section via the fast path (*i.e.*, by transiting directly from line 24 to line 34). Therefore, if a process enters its critical section via the slow path (lines 25–33), then it knows that the current estimate is wrong, and sets *Update* to *true* at line 33.

Lines 35–39 update *Bound*, if the current estimate is wrong. To correct *Bound*, a process first waits until every element of the *Trying* array is *false*, in order to ensure that no other process concurrently uses the value of *Bound* while it is updated. Then, it increments *Bound* at line 39.

From the discussion above, we can conclude the following properties of Algorithm AAT.

```
shared variables
    $X, XX$: 1..$N$;
    $Y, YY$: 0..$N$ initially 0;
    $Z$: boolean initially $false$;
    $B$, $Trying$: array[1..$N$] of boolean initially $false$;
    $Bound$: (delay duration) initially 1

process $p$                                   /* $1 \le p \le N$ */
while $true$ do
1:  Noncritical Section;
Start1:
    repeat
2:      if $Update = true$ then
3:          $Trying[p] := false$;
4:          await $Update = false$
        fi;
5:      $Trying[p] := true$;
6:      $X := p$
7:  until $Y = 0$;
8:  $Y := p$;
9:  if $X \neq p$ then
10:     $delay(2 \cdot Bound)$;
11:     if $Y \neq p$ then go to Start1 fi;
12:     await $\neg Z$ $\vee$
13:                 $(Update = true)$
14: else $Z := true$
    fi;
Start2:
15: if $Update = true$ then go to Start1 fi;
16: $B[p] := true$;
17: $XX := p$;
18: if $YY \neq 0$ then
19:     $B[p] := false$;
20:     await $(YY = 0)$ $\vee$
21:                 $(Update = true)$;
22:     go to Start2
    fi;
23: $YY := p$;
24: if $XX \neq p$ then
25:     $B[p] := 0$;
26:     for $j := 0$ to $N - 1$ do
27:         await $\neg B[j]$ $\vee$
28:                 $(Update = true)$
        od;
29:     if $YY \neq p$ then
30:         await $(YY = 0)$ $\vee$
31:                 $(Update = true)$;
32:         go to Start2
        else
33:         $Update := true$
    fi fi;
34: Critical Section;
35: $Trying[p] := false$;
36: if $Update = true$ then
37:     for $j := 0$ to $N - 1$ do
38:         await $\neg Trying[j]$ od;
39:     $Bound := Bound + 1$
    fi;
40: $YY := 0$;
41: $B[p] := false$;
42: $Z := false$;
43: if $Update = true$ then
44:     $Y := 0$;
45:     $Update := false$
46: else if $Y = p$ then
47:     $Y := 0$
    fi
od
```

Figure 23: Algorithm AAT: A fast timing-based algorithm for the unknown-delay model.

- Since both Algorithm AT and Algorithm L are fast, Algorithm AAT is also fast.
- Since both Algorithm AT and Algorithm L are livelock-free, Algorithm AAT is also livelock-free. However, it is not starvation-free.
- When a period of contention begins, Algorithm AAT has a "warm-up" period in which the processes may use the slow path of Algorithm L. During this period, after each critical section ends, the next winning process may subsequently access $\Theta(N)$ shared variables in order to enter and exit its critical section.
- Once the correct estimate of Δ is determined, only the fast path of Algorithm L is used, and hence the algorithm has same timing characteristics as Algorithm AT. In particular, after a critical section ends, the next winning process subsequently performs $O(1)$ memory references and explicitly delays itself for at most $\Theta(\Delta)$ time units in order to enter and exit its critical section.

Theorem 11 (Alur, *et al.*) *The mutual exclusion problem can be solved in the unknown-delay model by a livelock-free timing-based algorithm (namely,* Algorithm ATT*) that requires only $O(1)$ memory references* *in the absence of contention, for which the following property eventually holds: after any critical section ends, the next process to enter its critical section does so after performing $O(1)$ memory references and explicitly delaying (by invoking delay statements) for at most $\Theta(\Delta)$ time units.* □

Related work. By combining the splitter mechanism with Algorithm F, Lynch and Shavit constructed a simple timing-based algorithm with only two shared variables that is livelock-free if timing constraints are met, and that satisfies the exclusion property even if such constraints are violated [59]. In later work, Gafni and Mitzenmacher experimentally evaluated several variants of Lynch and Shavit's algorithm, when process execution speeds follow certain random statistical distributions [38]. In these algorithms, Δ needs to be set to a worst-case value to ensure correctness. However, using a worst-case value may lead to unnecessary delays most of the time. With smaller values of Δ, we gain better performance on average, at the price of occasional "failures" (*i.e.*, all processes may detect other contending processes and restart their entry sections). Gafni and Mitzenmacher studied the behavior of these algorithms while varying execution-time distributions and the number of processes in the system. Their results suggest that some timing-based algorithms exhibit statistically good performance even in very large systems (on the order of 100,000 processes).

7 Mutual Exclusion with Nonatomic Operations

To this point, we have assumed that shared variables are accessed atomically. Requiring atomic variable accesses is tantamount to assuming mutual exclusion at some lower level. Thus, mutual exclusion algorithms requiring this are in some sense circular. Lamport recognized that mutual exclusion can be implemented without requiring operations to be atomic in some early papers [52, 53], and later wrote more extensively on the topic in a two-part work [54, 55]. An overview of this two-part work is presented in Section 7.1. In Section 7.2, we consider nonatomic algorithms in which all busy-waiting is by local spinning.

7.1 Lamport's Nonatomic Algorithms

In the first part of Lamport's two-part paper [54], a formal model of concurrent systems is presented in which no underlying atomicity is assumed. In this model, operation executions have duration and may be concurrent. Only single-writer nonatomic shared variables are considered; Lamport calls such variables *communication variables*. If a write of a communication variable is concurrent with another operation on that same variable, then the other operation must be a read. A read of a shared variable that is concurrent with a write of the same variable is allowed to return any value from the domain of the variable. A read that does not overlap such a write simply returns the variable's current value. The roots of this notion of nonatomic semantics trace back to Lamport's bakery algorithm [52] and it is the standard semantics assumed in most papers on nonatomic algorithms.

To appreciate some of the difficulties that occur when using such nonatomic operations, consider the two scenarios below, which are permitted in this model; in both, the variable x is assumed to be initially 0.

(i) x is updated by a write operation that assigns $x := 0$. A concurrent read of x may obtain any value from the domain of x, including a value different from 0.

(ii) x is updated by a write operation that assigns $x := 1$ that is concurrent with two successive read operations. The first read obtains the value 1 and the second read obtains the value 0.

While general techniques for dealing with nonatomic operations can be quite intricate, two basic ideas are elementary. First, observe that Scenario (i) above can be eliminated by simply skipping redundant writes, *i.e.*, any write of a variable that does not change that variable's value. Second, nondeterminism can be reduced by simply using binary shared variables, in which case a read concurrent with a write of the same variable can only return 0 or 1. Furthermore, if redundant writes are avoided, then a read concurrent with a write returns either the variable's "old" value or the "new" value being written. In all the algorithms considered in this section, we assume that writers avoid redundant writes, even though this is not explicitly

```
shared variable
    $P$: array[0..$N-1$] of boolean initially $false$

process $p$      /* $0 \le p < N$ */

private variable
    $i$: 0..$N-1$;
    $S$: set of 0..$N-1$

    while $true$ do
1:      Noncritical Section;
2:      $P[p] := true$;
3:      $S := \{q \mid 0 \le q < N \ \wedge \ q \neq p\}$;
4:      while $S \neq \emptyset$ do
5:          $i :=$ elementOf($S$); $S := S \setminus \{i\}$;
6:          await $\neg P[i]$
        od;
7:      Critical Section;
8:      $P[p] := false$
od
```

Figure 24: Algorithm L-N1: Basic nonatomic algorithm.

indicated in the code depictions that are given. (This simply requires adding a private variable for each shared variable written by a process that tracks that variable's value; it can be easily determined whether a write is redundant by checking the corresponding private variable.) The situation in Scenario (ii) above is still possible, but it often can be managed.

In the second part of Lamport's two-part paper [55], a series of nonatomic mutual exclusion algorithms is presented. The first and simplest of these algorithms is Algorithm L-N1 shown in Figure 24. (The "N" in this and subsequent designations of Lamport's algorithms stands for "nonatomic.") Each successive algorithm satisfies stronger properties and builds upon the previous one. We cover two of these other algorithms below.

Algorithm L-N1. Algorithm L-N1 is quite simple. Each process in its entry section verifies that every other process is neither in its entry section nor its critical section (selecting processes nondeterministically using "elementOf"). To see that this algorithm satisfies the exclusion property, consider the following operational argument. (A formal proof would require more careful definitions of notions related to event orderings.) Heading for a contradiction, suppose that processes p and q are both in their critical sections. Suppose further that p finished executing line 2 before q did. Then, $P[p] = true$ was stable from the time q began executing line 3 and until q entered its critical section. However, this implies that q could not pass line 6 when $q.i = p$ held, which is a contradiction.

Algorithm L-N2. Unfortunately, Algorithm L-N1 is not even livelock-free. However, it is possible to modify the algorithm so that livelock-freedom holds, without violating the exclusion property. The resulting algorithm is denoted Algorithm L-N2 and is depicted in Figure 25. In Algorithm L-N2, the nondeterministic scan of the P variables is replaced by a two-phase protocol.

In Phase 1, lines 2–7, a process yields to any process of lower index that is in its entry or critical section. In Phase 2, lines 8–9, a process checks each higher-indexed process to determine if it is in its entry section (and has not yielded), and waits if this is the case. Note that a process p may enter its critical section only if it establishes $P[p] = true$ and then finds that $\neg P[q]$ holds, for each $q \neq p$. Thus, the argument for the exclusion property given previously for Algorithm L-N1 applies to Algorithm L-N2 as well.

To see that Algorithm L-N2 is livelock-free, suppose to the contrary that some set of processes remain forever outside of their noncritical sections, yet no process executes its critical section. Then, eventually

```
shared variable
    $P$: array$[0..N-1]$ of boolean initially $false$

process $p$      /* $0 \le p < N$ */

private variable
    $i$: $0..N-1$

    while $true$ do
1:      Noncritical Section;
2:      $P[p] := true$;
3:      for $i := 0$ to $p-1$ do
4:          if $P[i]$ then
5:              while $P[i]$ do
6:                  $P[p] := false$
                od;
7:              go to 2
            fi
        od;
8:      for $i := p+1$ to $N-1$ do
9:          await $\neg P[i]$
        od;
10:     Critical Section;
11:     $P[p] := false$
od
```

Figure 25: Algorithm L-N2: Livelock-free nonatomic algorithm.

a state is reached such that, thereafter, each process remains either forever in its entry section or forever in its noncritical section. Let Q be the set of processes forever executing Phase 2. If q is the process of largest index in Q, then it clearly finds no P variable with the value *true* at line 9. Thus, q eventually enters its critical section, which is a contradiction, and this implies Q is empty. Since a process permanently in the entry section cannot return from Phase 2 to Phase 1, the remaining case is that all processes in the entry section permanently execute Phase 1. Suppose this is the case, and let p be the process of least index executing Phase 1. Since it finds no P variable with value *true* at line 2, p leaves Phase 1 and goes on to Phase 2. This contradicts the assumption that p is permanently executing Phase 1.

Algorithm L-N3**.** Unfortunately, Algorithm L-N2 is not starvation-free, because an unfortunate process can continue to restart Phase 1 in its entry section as processes of lower index repeatedly execute their critical sections. To make the algorithm starvation-free, Lamport introduced a "renumbering" function, whereby processes contending in their entry sections are dynamically ordered so that any such process eventually has highest priority. The modification has two parts. First, another N-bit array E is used to determine the identities of the processes competing in their entry sections. Array E is needed because P is an unreliable indicator of contender identities, due to the yielding mechanism in Phase 1. The second modification is a specialized structure T, also encoded by an array of N bits, that has some queue-like properties enabling the prioritization of competing processes. Because the shared variables P, E, and T result in three bits per process, Lamport called the resulting algorithm the *three-bit algorithm*. Here, we denote it as Algorithm L-N3, as seen in Figure 26.

Instead of iterating through process indices starting from zero, Algorithm L-N3 computes an initial starting index and cyclically iterates through process indices modulo-N. The starting index is returned by the function $\mathsf{low}(T, E)$ (line 4). We defer a complete explanation of this function until later and mention only the following property of it here: if a process p is in its entry section sufficiently long, then eventually $\mathsf{low}(T, E) = p$ holds for any execution of line 4 by any process, at least until p completes its critical section. Evaluations of $\mathsf{low}(T, E)$ by different processes concurrently in their entry sections can return different values because E and T are written as processes enter and leave; however, a precondition for critical-section entry is

```
shared variable
    $P$: array[0..$N-1$] of boolean initially $false$;
    $E$: array[0..$N-1$] of boolean initially $false$;
    $T$: array[0..$N-1$] of boolean

process $p$      /* $0 \leq p < N$ */

private variable
    $i, k$: $0..N-1$;
    $temp$: boolean

    while $true$ do
1:      Noncritical Section;
2:      $P[p] := true$;
3:      $E[p] := true$;
4:      $k := \mathsf{low}(T, E)$;
5:      if $k \neq p$ then
6:          for $i := k$ cyclically to $(p-1)$ mod $N$ do
7:              if $E[i]$ then
8:                  $P[p] := false$;
9:                  go to 3
                fi
            od
        fi;
10:     if $\neg P[p]$ then go to 2 fi;
11:     for $i := p+1$ cyclically to $(k-1)$ mod $N$ do
12:         if $P[i]$ then go to 2 fi
        od;
13:     Critical Section;
14:     $temp := \neg T[p]$;
15:     $T[p] := temp$;
16:     $P[p] := false$;
17:     $E[p] := false$
od
```

Figure 26: Algorithm L-N3: Starvation-free nonatomic algorithm.

the execution of lines 4–12 in sequence without writing P, and this ensures the exclusion property. (Checking $E[i]$ rather than $P[i]$ in line 7 is safe because $E[i]$ remains $true$ so long as process i is in its entry section.)

The argument for livelock-freedom is along similar lines to that given above for Algorithm L-N2. In particular, assuming an execution in which a non-empty set of processes are in their entry sections but never advance to their critical sections, we infer that eventually a state is reached after which no process in its entry section writes any T or E variable, and hence the private k variables of such processes eventually agree and remain constant. Now, for process p satisfying $p.k = \mathsf{low}(T, E)$, it can be argued (by contradiction) that p executes Phase 2 (lines 11–12) only finitely many times before entering its critical section.

The proof that Algorithm L-N3 is starvation-free depends on the precise definition of $\mathsf{low}(T, E)$, which we now give. We start by informally defining a "token ring," with each node of the ring representing a process. Tokens circulate unidirectionally in this ring, and we suppose that each process requires a token to enter its critical section and that a process leaving its critical section passes the token to the next process in the ring. An array A of N bits can implement such a token ring: process p, where $p > 0$, has a token if $A[p] \neq A[p-1]$, and process 0 has a token if $A[0] = A[N-1]$. It is simple to show, by contradiction, that at least one token is present in any state for this definition. When a token holder p no longer needs its token, then the assignment $A[p] := \neg A[p]$ removes its token (provided, of course, that process $q = (p-1)$ mod N does not also change $A[q]$ concurrently).

For Algorithm L-N3, the idea of the token ring is used to ensure starvation-freedom. Rather than a fixed, N-bit array A, the subsequence of array T given by $\langle T[j] : 0 \le j < N \;\wedge\; E[j]\rangle$ is the token ring. Thus, from arrays T and E, a dynamic ring of $|\{j \mid E[j]\}|$ bits is formed. The function $\mathsf{low}(T, E)$ returns the smallest process index p from this dynamic ring such that p is a token holder. (We define $\mathsf{low}(T, E) = 0$ for the case of an empty ring, which is useful later for fault tolerance.)

Having defined $\mathsf{low}(T, E)$, we now show that Algorithm L-N3 is starvation-free (again, by contradiction). Suppose that there exists a nonempty set S of processes that remain forever in their entry sections. Since livelock-freedom has been verified, such a situation is only possible if there exists another nonempty set Q of processes that infinitely often execute their critical sections. We establish starvation-freedom by showing that Q must be empty.

Define d_{ab} for $0 \le a, b < N$ to be the smallest natural number j such that $(a + j) \bmod N = b$. Let (s, q) be such that d_{sq} is minimum for any choice of $s \in S$ and $q \in Q$. A property of the token ring defined by $\langle T[j] : 0 \le j < \;\wedge\; E[j]\rangle$ is that if q has a token prior to executing lines 14–15, then it has no token after executing line 15. Since q executes its critical section infinitely often, eventually there is a state where q has no token and begins its entry section. Now consider the result of q assigning $q.k := \mathsf{low}(T, E)$ at line 4. Because q has no token, $q.k \ne q$ and it executes the loop of lines 6–9. If q's execution of this loop includes s in the loop index range (*i.e.*, $q.k$ to $(q-1)$ **mod** N), then we claim that when q executes lines 6–9, it does not advance beyond line 9 in the entry section. This follows because $E[s]$ remains *true* forever. Note that (s, q) have been chosen so that q cannot regain a token, because s never executes lines 14–15, so $q.k \ne q$ holds after each subsequent execution of line 4. But we assume that q executes the critical section infinitely often, hence s must be outside the loop index range of line 6 some time while q is in the entry section (possibly because $q.k$ is assigned differing values from $\mathsf{low}(T, E)$ at different times). However, in this case, $E[s]$ will be tested in the execution of the loop in lines 11–12, preventing q from entering its critical section. Thus, $q \notin Q$ holds in either case, and therefore Q must be empty. The contradiction proves livelock-freedom.

Ordered critical-section entry. An algorithm satisfies *r-bounded waiting* [72] if the entry section of each process can be partitioned into two components, a *doorway*, which generates a bounded number of memory references (local or remote), followed by a *waiting* part, such that the following holds: once a process p finishes executing its doorway, no process can execute its critical section more than r times before p does so. r-bounded waiting strengthens starvation-freedom. The case $r = 1$ approximates the intuition of first-come first-served (FCFS) ordered entry. In addition to the nonatomic algorithms considered above, Lamport [55] presented two FCFS algorithms (which we do not cover here). One of these algorithms uses N shared variables per process; the second has $O(N!)$ shared variables. (This rather large space requirement also enables the algorithm to have several of the fault-tolerance properties described below.) Interested readers are also referred to a later paper by Lycklama and Hadzilacos [58], where a FCFS nonatomic solution is presented that uses only a constant number of shared variables per process (see also [76]).

Fault tolerance. One of the principal themes of Lamport's paper on nonatomic algorithms [55] is fault tolerance. In some sense, nonatomic operations can be thought of as faulty operations because they violate atomic semantics. Lamport considered several failure models, which are summarized below.

- **Shutdown Safety:** An algorithm is *shutdown safe* if it remains correct in the face of process shutdowns. When a process is shut down, it sets all of its communication variables to default values, returns to the noncritical section, and halts. Note that a process may be shut down while executing its critical section.
- **Abortion Safety:** An algorithm is *abortion safe* if it is resilient to process abortions. When a process aborts, it sets some of its communication variables to default values, and returns to its noncritical section. (The other variables are left unchanged.) The communication variables that are set are those for which initial values have been specified as part of the algorithm's description.
- **Fail Safety:** An algorithm is *fail safe* if it is resilient to process failures. When a process fails, it may arbitrarily change the values of its communication variables for some time; however, it eventually shuts down, setting all of its communication variables to default values, returns to the noncritical section, and thereafter never malfunctions again.

- **Self-stabilization:** An algorithm is *self-stabilizing* [34] if it is resilient to transient failures, *i.e.*, if the algorithm enters some illegitimate state due to transient failures, and if such failures stop happening, then the algorithm eventually converges to a legitimate state. (A state of a program is *legitimate* if it appears in some failure-free history of the program.) A self-stabilizing algorithm can tolerate a malfunctioning process that arbitrarily changes the values of its communication variables as well as its local variables for some finite time, but eventually stops malfunctioning and starts normal behavior, albeit in an arbitrary state, and never malfunctions again.

Technically, each of these failure models is defined with respect to a desired property. For example, an algorithm is self-stabilizing for the exclusion property if, from any illegitimate state, the algorithm eventually converges to a legitimate state, after which exclusion is never violated. According to Lamport, the first two models, shutdown safety and abortion safety, reflect reasonable behaviors that might result from premature process termination (though abortion does not imply halting). A shutdown might be triggered by operator intervention; an abortion represents the case of a process deliberately withdrawing from mutual exclusion. The second two properties are unreasonable behaviors, which might be due to transient hardware errors. Intuitively, fail safety reflects a scenario in which a transient process failure is detected and repaired and the failed process restored to service. Lamport also mentions the problem of dealing with a process that halts silently, with no change to any of its communication variables. Unfortunately, solving this problem requires more synchrony (such as using real-time clocks) than the basic computation model allows.

Self-stabilization. The self-stabilization requirement is in some ways the most challenging of those listed above, so we consider it here in some detail. An in-depth discussion of all of these requirements can be found in [55].

For a mutual exclusion algorithm to be self-stabilizing, it is necessary to require each process that remains in its noncritical section to periodically reset its communication variables to appropriate values. Without this requirement, a process could possibly remain forever in its noncritical section with its communication variables set to values that make it appear to other processes that it is in its critical section. It is also convenient to assume that transient faults only occur initially in any history, and thereafter all state changes are due to correct process operations. Ultimately, algorithms that tolerate additional transient faults are desired. However, the standard approach is to begin by considering only initial faults; after the algorithm recovers, the same approach can be applied to a subsequent transient fault. Earlier, we remarked that an implementation of write to a communication variable can avoid redundantly writing by checking a private variable containing the last value written. For self-stabilization, this technique is not viable, since private variables may be corrupt after a transient fault. Therefore, the write implementation should actually read the communication variable to avoid redundantly writing (optimizations can still use the private variable technique for some fraction of write operations in an execution). With these assumptions, ALGORITHMS L-N1, L-N2, and L-N3 can be shown to be self-stabilizing. We prove this for ALGORITHM L-N3 below. For this algorithm, we require each process in its noncritical section to periodically set its P and E communication variables to *false*.

Theorem 12 (Lamport) *The mutual exclusion problem can be solved by an algorithm* (*namely,* ALGORITHM L-N3) *that uses only nonatomic reads and writes and that is self-stabilizing for the exclusion, livelock-freedom, and starvation-freedom properties.*

Proof: Starting from any initial state (legitimate or illegitimate), each process eventually either resets its P and E communication variables in its noncritical section (by assumption), or executes lines 2–3 for a second time. After all processes have done this, the system is in a state that is reachable from a legitimate initial state. (Note that the array T is initialized arbitrarily.) Once a legitimate state is reached, exclusion, livelock-freedom, and starvation-freedom can be shown to hold, as argued earlier. □

Related work. As noted by Lamport [55], Burns independently discovered ALGORITHM L-N1 [25]. In later work, the idea of using a token ring in shared memory to achieve self-stabilization and starvation-

```
shared variable P, Q, T: array[u, v] of boolean initially true

process u                                   process v

private variable x: boolean                 private variable x: boolean
initially x = T[u]                          initially x = T[v]

while true do                               while true do
    1:  Noncritical Section;                    1:  Noncritical Section;
    2:  P[u] := false;                          2:  P[v] := false;
    3:  Q[u] := false;                          3:  Q[v] := false;
    4:  x := T[v];                              4:  x := ¬T[u];
    5:  T[u] := x;                              5:  T[v] := x;
    6:  if x then                               6:  if x then
    7:      P[u] := true;                       7:      Q[v] := true;
    8:      await P[v]                          8:      await P[u]
        else                                        else
    9:      Q[u] := true;                       9:      P[v] := true;
   10:      await Q[v]                         10:      await Q[u]
        fi;                                         fi;
   11:  Critical Section;                      11:  Critical Section;
   12:  P[u] := true;                          12:  P[v] := true;
   13:  Q[u] := true                           13:  Q[v] := true
od                                          od
```

Figure 27: Algorithm JA: Two-process case.

freedom was extended to self-stabilizing k-exclusion by Abraham, *et al.* [1]. (The k-exclusion problem is defined in Section 9, where another k-exclusion algorithm is considered in more detail.)

7.2 Nonatomic Mutual Exclusion with Local Spinning

One shortcoming of Lamport's algorithms is that they are not local-spin algorithms. In later work, Anderson devised a nonatomic algorithm in which all spins are local [11]. As mentioned previously, this algorithm was actually the first local-spin algorithm that uses only read and write operations. This algorithm, hereafter referred to as Algorithm JA, is described below.

Algorithm JA. Algorithm JA was obtained by first solving the two-process case, and by then using the two-process solution to solve the N-process case. The two-process version of Algorithm JA is shown in Figure 27. The two processes are denoted u and v. With the exception of lines 4, 7, and 9, the two processes are identical. Algorithm JA is similar to earlier two-process solutions given by Peterson [67] and by Kessels [45], but uses only single-reader, single-writer boolean variables.

The two shared variables $T[u]$ and $T[v]$ are used as a tie-breaker in the event that both processes attempt to enter their critical sections at the same time. Process u attempts to establish $T[u] = T[v]$ and process v attempts to establish $T[u] \neq T[v]$. Process u enters its critical section only if $T[u] \neq T[v]$ holds or if process v has not expressed interest in entering its critical section. Similarly, process v enters its critical section only if $T[u] = T[v]$ holds or if process u has not expressed interest in entering its critical section. Thus, in the event of a "tie," process u is favored if $T[u]$ and $T[v]$ differ in value and process v is favored if $T[u]$ and $T[v]$ are equal. This is essentially the idea of Kessels' algorithm. In Algorithm JA, each process checks the condition that is required for it to enter its critical section by waiting on one single-reader, single-writer boolean variable. The manner in which this is accomplished is explained next.

Because $T[u]$ is written only by process u, process u can keep track of its value by using a private variable; variable $u.x$ is used for this purpose. In order for process u to wait until $T[u] \neq T[v]$ holds, it simply tests $u.x$ and then waits for $T[v]$ to have the appropriate value. In particular, if $u.x$ is true (which implies that

$T[u]$ is *true*), then process u waits until $T[v]$ is *false*, and if $u.x$ is *false* (which implies that $T[u]$ is *false*), then process u waits until $T[v]$ is *true*. Process u waits for $T[v]$ to have the appropriate value by waiting on either $P[v]$ or $Q[v]$. As explained next, these variables serve the dual purpose of "signaling" the value of $T[v]$ and "signaling" that process v is in its noncritical section.

Loosely speaking, $P[v]$ is used by process v to signal to process u that $T[v]$ is *false*, and $Q[v]$ is similarly used to signal that $T[v]$ is *true*. While the value of $T[v]$ is being determined in lines 4 and 5 of process v, the appropriate value to signal is not known; thus, to ensure that process u does not enter its critical section when it should not, $P[v]$ and $Q[v]$ are both kept equal to *false* while this value is being determined. When process v is in its noncritical section (where it may halt), process u should not be blocked in its entry section; thus, while v is in its noncritical section, $P[v]$ and $Q[v]$ are both kept equal to *true*. In this way, $P[v]$ and $Q[v]$ serve both purposes mentioned in the previous paragraph. Variables $P[u]$ and $Q[u]$ are similarly used by process u to signal the value of $T[u]$ to process v, except their roles are reversed: $P[u]$ is used to signal that $T[u]$ is *true*, and $Q[u]$ is used to signal that $T[u]$ is *false*.

The N-process version of ALGORITHM JA is obtained by applying the two-process version in the following.

```
process $p$      /* $0 \leq p < N$ */
while $true$ do
    Noncritical Section;
    for $i := 0$ to $N - 1$ do
        if $i \neq p$ then ENTRY($p$, $i$)
    od;
    Critical Section;
    for $i := 0$ to $N - 1$ do
        if $i \neq p$ then EXIT($p$, $i$)
    od
od
```

In this algorithm, `ENTRY`(i, j) denotes the entry section from a two-process solution that process i executes to compete with process j. `EXIT`(i, j) is defined similarly. `ENTRY`(i, j) and `EXIT`(i, j) are assumed to be implemented using the algorithm in Figure 27. It should be straightforward to see that the two-process version of ALGORITHM JA has $O(1)$ RMR time complexity, and the N-process version has $\Theta(N)$ RMR time complexity.

As mentioned above, ALGORITHM JA remains correct even if variable accesses are nonatomic. Since only single-reader, single-writer boolean variables are used in the algorithm, the only potential problem that must be considered is the case of a read of a boolean variable that overlaps a write of that variable. In such a case, it can be shown that it is safe for the read to return either *true* or *false*. Thus, we have the following theorem.

Theorem 13 (Anderson) *The mutual exclusion problem can be solved with $\Theta(N)$ RMR time complexity using only nonatomic reads and writes under either the CC or DSM model.* □

Related work. In recent work [17], Anderson and Kim showed that by applying simple transformations to ALGORITHM YA, covered earlier in Section 3.2, a nonatomic algorithm with the same RMR time complexity can be derived. The key idea is to replace all multi-reader, multi-writer, multi-bit variables by single-reader, single-writer, single-bit variables, so that overlapping operations on the same variable have no adverse impact. Thus, we can strengthen Theorem 13 as follows.

Theorem 14 (Anderson and Kim) *The mutual exclusion problem can be solved with $\Theta(\log N)$ RMR time complexity using only nonatomic reads and writes under either the CC or DSM model.* □

It is easy to see that ALGORITHM JA is shutdown safe, abortion safe, and fail safe: if process u in the two-process algorithm fails, eventually setting its shared variables $P[u]$ and $Q[u]$ to default values, then process v will not subsequently block on u. On the other hand, ALGORITHM YA and its nonatomic variant mentioned above are not tolerant of these failures. This is because the manner in which spin variables are updated in these algorithms is more complicated; hence, it is not generally possible to deduce appropriate reset values for a process to use when it fails. (In fact, it is a little ambiguous as to which variables a failed process should reset: Algorithm YA does not use variables that have a single *statically-associated* writer process, as do the nonatomic algorithms considered in this section.) It is currently unknown whether self-stabilizing local-spin algorithms (atomic or nonatomic) can be devised. We state this as an open problem.

Open Problem: Is it possible to devise a *local-spin* algorithm (atomic or nonatomic) that is self-stabilizing?

Clearly, no local-spin algorithm for DSM systems with read-only spin loops can be self-stabilizing: such an algorithm can be placed in an illegitimate state in which every process is forever blocked, waiting for its dedicated spin variables to be updated by another competing process.

8 Time- and Space-complexity Lower Bounds

In this section, we survey several papers that present time- and space-complexity lower bounds for mutual exclusion. We begin in Section 8.1 by briefly discussing some of the proof techniques used in these papers. Then, in Section 8.2, a summary of results is presented.

8.1 Overview of Proof Methodologies

Most interesting lower bounds pertaining to the mutual exclusion problem are based on arguments that are quite complex. Therefore, we provide only a brief overview of commonly-used proof techniques.

Time-complexity lower bounds for mutual exclusion are often derived by inductively constructing longer and longer histories. Usually, such histories must be constructed in a way that limits "information flow" among competing processes. Information flow occurs when one process reads (from a shared variable) a value that was written by another process. If n processes are in their entry sections, and no information flow among them has occurred, then at least $n - 1$ of them must perform further accesses to shared variables; otherwise, the current history under consideration can be extended to one in which multiple processes are in their critical sections. In other words, if no information flow has occurred among processes in their entry sections, then "most" of these processes have a "next" shared-variable access. By inductively appending such variable accesses while limiting information flow, a lower bound on entry-section execution time can be derived.

One way to prevent information flow is by "erasing" processes in the history under consideration [14, 19, 32, 46]. When a process is erased, its statement executions are completely removed from the history. For example, suppose a history is to be extended by appending the next statement execution of each process in its entry section. If the statement to be appended for process p is a read of some variable x that was previously written, and process q was the last process to write x, then the resulting information flow from q to p can be eliminated by erasing either p or q. Of course, if another process wrote x prior to q, then erasing q does not eliminate all information flow to p. In general, determining which processes to erase so that the induction can continue is a tricky balancing act.

Sometimes, erasing alone does not leave enough processes for the next induction step. In this case, the "roll-forward" strategy can be used: some processes are selected and allowed to "roll forward" until they return to their noncritical sections [14, 32, 46]. For instance, in the example above, information flow among *competing* processes can be eliminated by allowing q to roll forward. In this way, p's read of x obtains a value written by a process in its noncritical section, *i.e.*, a process that p is not competing with. Of course, as q is rolled forward, it may read shared variables that have been written by other processes. Depending on the proof strategy being used, the resulting information flow may need to be dealt with by erasing or rolling forward other processes.

Note that, in the example in the previous paragraph, q's write to x *eliminates* any information flow through x among concurrently-competing processes. This basic proof technique, in which certain processes may be "hidden" by the writes of other processes, dates back to a paper on the space complexity of mutual exclusion by Burns and Lynch [27, 28].

8.2 Lower-bound Results

We now present an overview of research conducted since 1986 on lower bounds. All of the bounds below apply to asynchronous systems, except those that are specifically directed at timing-based systems.

Alur and Taubenfeld proved that a naive definition of "time complexity" (in which *every* shared-variable access is counted) is not meaningful for mutual exclusion [8].

Theorem 15 (Alur and Taubenfeld) *For any N-process mutual exclusion algorithm with $N \geq 2$, the first process to enter its critical section may perform an unbounded number of shared-variable accesses.* $\square$

Anderson and Yang presented several lower bounds that establish trade-offs between the number of remote memory references required for mutual exclusion and write- and access-contention [19]. The *write-contention* (*access-contention*) of a concurrent program is the number of processes that may potentially be simultaneously enabled to write (access) the same shared variable.[5]

Theorem 16 (Anderson and Yang) *For any N-process mutual exclusion algorithm, if write-contention is w, and if each atomic operation accesses at most v remote variables, then there exists a history involving only one process in which that process performs $\Omega(\log_{vw} N)$ remote memory references (under the DSM model) for entry into its critical section. Moreover, among these memory references, $\Omega(\sqrt{\log_{vw} N})$ distinct remote variables are accessed.* $\square$

Theorem 17 (Anderson and Yang) *For any N-process mutual exclusion algorithm, if access-contention is c, and if each atomic operation accesses at most v remote variables, then there exists a history involving only one process in which that process accesses $\Omega(\log_{vc} N)$ distinct remote variables for entry into its critical section.* $\square$

According to Theorem 16, under the DSM model, $\Omega(\log_{vw} N)$ remote memory references are required, even in the absence of contention. In CC machines, the first access of a remote variable must cause a cache miss. Thus, by counting the number of distinct remote variables accessed, a lower bound for the CC model is obtained. Thus, Theorems 16 and 17 imply that $\Omega(\sqrt{\log_{vw} N})$ (or $\Omega(\log_{vc} N)$) remote memory references are required under the CC model in the absence of contention. Note that Theorem 16 implies that fast mutual exclusion algorithms require arbitrarily high write-contention. Thus, the fact that ALGORITHM L uses N-writer shared variables is no accident.

Burns and Lynch considered the space complexity of mutual exclusion algorithms [28].

Theorem 18 (Burns and Lynch) *Any livelock-free N-process mutual exclusion algorithm that uses only reads and writes must use at least N shared variables.* $\square$

They also showed that this bound is tight by presenting a livelock-free algorithm with N boolean shared variables. In related work, Lynch and Shavit investigated the relationship between time and space complexity in timing-based systems [59]. They showed that if a system has $o(N)$ shared variables, then space and time complexity are inversely related in such systems. (The actual formula is too complicated to reproduce here.) Among other result, they showed the following.

Theorem 19 (Lynch and Shavit) *In a timing-based system with unknown delay, any livelock-free N-process mutual exclusion algorithm that uses only reads and writes must use at least N shared variables.* $\square$

[5]The notions of write-contention, access-contention, and contention cost (defined later in this section) are different from the concepts of point and interval contention defined earlier in Section 5.

Note that ALGORITHM F and ALGORITHM AT achieve $O(1)$ space complexity in *known-delay* systems.

Alur and Taubenfeld investigated limitations on variable size in algorithms that use only atomic reads and writes [9].

Theorem 20 (Alur and Taubenfeld) *For any N-process mutual exclusion algorithm using only reads and writes, if each shared variable has l bits, then there exists a history involving only one process in which that process executes at least $\big((\log_2 N)/(l - 2 + 3\log_2\log_2 N)\big)$ operations for entry into its critical section. Moreover, among these operations, at least $\sqrt{(\log_2 N)/(l + \log_2\log_2 N)}$ distinct shared variables are accessed.* $\square$

They also established the following upper bound by constructing a simple variant of ALGORITHM L.

Theorem 21 (Alur and Taubenfeld) *For every l such that $1 \leq l \leq \log N$, there exists a livelock-free N-process mutual exclusion algorithm using only reads and writes, where each shared variable has l bits, such that a process executes at most $7\lceil(\log_2 N)/l\rceil$ operations to enter and exit its critical section in the absence of contention. Moreover, among these operations, at most $3\lceil(\log_2 N)/l\rceil$ different shared variables are accessed.* $\square$

According to Theorem 20, a "fast" algorithm requires variables with $\Omega(\log N)$ bits, *i.e.*, variables large enough to hold process identifiers, or at least some constant fraction of a process identifier. Most modern multiprocessor systems have at least 32-bit memory words, so requiring variables of size $\Omega(\log N)$ usually poses no problem in practice. However, there exist systems and algorithms that exploit the ability to access memory at different granularities. For example, as mentioned in Section 4.1, Michael and Scott presented a variant of ALGORITHM L in which processes access memory at both full- and half-word granularities [64]. Alur and Taubenfeld's results show that there are limitations to this approach.

Dwork, *et al.* questioned the assumption that each access to a variable is equally expensive [35]. Their chief contribution was to introduce a formal complexity model that takes into account the (hardware) contention caused by overlapping accesses to the same variable. Specifically, their model distinguishes between the *invocation* and *response* of each operation. An operation's *contention cost* is defined to be the number of response events from the same variable that occur between the operation's invocation and matching response. Note that the access contention of a program, as defined earlier, is simply the worst-case contention cost of any variable. Dwork, *et al.* applied this model to various classes of algorithms in order to study trade-offs between average/worst-case contention cost and time complexity. Regarding the mutual exclusion problem, they proved the following.

Theorem 22 (Dwork, *et al.*) *For any N-process mutual exclusion algorithm using reads, writes, and read-modify-write operations, if access-contention is c, then there exists a history in which a process executes $\Omega((\log N)/c)$ operations for entry into its critical section.* $\square$

Note that Theorems 20 and 22 do not distinguish between local and remote memory references.

Cypher [32] was the first to present a lower bound on remote memory references under arbitrary access-contention. His lower bound applies to algorithms using reads, writes, and conditional primitives such as *compare-and-swap*, and *test-and-set*. (A *conditional primitive* updates a shared variable after first testing that its value meets some condition.)

Theorem 23 (Cypher) *For any N-process mutual exclusion algorithm using reads, writes, and conditional primitives, there exists a history such that (the total number of remote memory references)/(the number of processes that participates in the history) $= \Omega(\log\log N/\log\log\log N)$.* $\square$

Cypher's lower bound applies to either DSM or CC systems. Given that primitives such as *fetch_and_inc* and *fetch_and_store* can be used to implement mutual exclusion in $O(1)$ time, Cypher's result pointed to an unexpected weakness of *compare-and-swap*, which is still widely regarded as being the most useful of all primitives to provide in hardware. In recent work, Anderson and Kim improved Cypher's result as follows [14].

Theorem 24 (Anderson and Kim) *For any N-process mutual exclusion algorithm using reads, writes, and conditional primitives, there exists a history in which some process performs $\Omega(\log N/\log\log N)$ remote memory references to enter and exit its critical section.* $\Box$

This lower bound also applies to both CC and DSM systems. However, unlike Cypher's result, this bound applies to just a single process and not all competing processes. Thus, it opens up the possibility that the *amortized* number of remote memory references per process is less than $\Theta(\log N/\log\log N)$. We leave this problem for future research.

Open Problem: Can Cypher's amoritzed lower bound be improved?

Given the lower bound in Theorem 24, it follows that the $\Theta(\log N)$ RMR time complexity of ALGORITHM YA is very close to optimal. In fact, we conjecture that it *is* optimal.

Conjecture: For any N-process mutual exclusion algorithm using reads, writes, and conditional primitives, there exists a history in which some process performs $\Omega(\log N)$ remote memory references to enter and exit its critical section.

Note that establishing this conjecture would show that conditional primitives such as *compare-and-swap* are no better than reads and writes from the standpoint of asymptotic RMR time complexity.

In another recent paper [46], Kim and Anderson established a lower bound for adaptive mutual exclusion algorithms, under similar conditions.

Theorem 25 (Kim and Anderson) *For any k, there exists some N such that, for any N-process mutual exclusion algorithm based on reads, writes, or conditional primitives, a history exists involving $\Theta(k)$ processes in which some process performs $\Omega(k)$ remote memory references to enter and exit its critical section.* $\Box$

This result precludes a deterministic algorithm with $O(\log k)$ RMR time complexity, where k is "point contention." However, as shown in [46], expected $O(\log k)$ RMR time complexity *is* possible using randomization.

It is important to point out that Theorem 25 does not establish $\Omega(k)$ as a lower bound. Instead, it shows that $\Omega(k)$ RMR time complexity is required *provided* N is sufficiently large. This leaves open the possibility that an algorithm might have $\Theta(k)$ RMR time complexity for very "low" levels of contention, but $o(k)$ RMR time complexity for "intermediate" levels of contention. However, we find this highly unlikely. Indeed, we conjecture the following.

Conjecture: For any adaptive N-process mutual exclusion algorithm using reads, writes, and conditional primitives, and for any k ranging over $1 \leq k \leq N$, there exists a history in which some process experiences point contention k and generates $\Omega(min(k, \log N))$ remote memory references to enter and exit its critical section.

Note that establishing this conjecture would show that ALGORITHM AK-RT is time-optimal under the RMR time complexity measure for both the CC and DSM models.

The final bounds that we mention pertain to nonatomic algorithms and were recently established by Anderson and Kim [17].

Theorem 26 (Anderson and Kim) *For any N-process mutual exclusion algorithm based on nonatomic reads and writes, there exists a history involving only one process in which that process performs $\Omega(\log N/\log\log N)$ remote memory references (under the DSM model) for entry into its critical section. Moreover, among these memory references, $\Omega(\sqrt{\log N})$ distinct remote variables are accessed.*[6] $\Box$

These bounds show that fast and adaptive algorithms are impossible if variable accesses are nonatomic, even if caching techniques are used to avoid accessing the processors-to-memory interconnection network.

[6]The latter bound requires certain technical conditions involving the semantics of nonatomic writes. See [17] for details.

9 Other Research

A number of important papers on shared-memory mutual exclusion and related topics have been published since 1986 that do not fit within the categories considered until now. We conclude by briefly mentioning several such papers.

Randomized mutual exclusion. In work on randomized mutual exclusion, Kushilevitz and Rabin presented an algorithm that uses a single $\Theta(\log \log N)$-bit shared variable [51]. (This paper is a revision of [69], which was later found to contain errors.) The variable is accessed via a fetch-and-ϕ primitive. It is assumed that processes are scheduled by an "adversary scheduler." An adversary scheduler can examine the "external behavior" of processes (*i.e.*, whether a process is in its noncritical, entry, critical, or exit section), remember the history of external behaviors, and use that information to decide which process will be executed next. However, the scheduler considered by Kushilevitz and Rabin is restricted to not examine the content of any shared or private variables in deciding which process to schedule.

Kushilevitz and Rabin considered a probabilistic notion of progress, defined as follows. Let C_i be the instant at which the i^{th} critical section is started, and let S_i be the set of processes that are in their entry sections between C_{i-1} and C_i and that access the (single) shared variable in that time interval. An algorithm satisfies *linear fairness* if, for each process p in S_i that does not enter its critical section at C_i, p enters its critical section at C_{i+1} with probability $\Omega(1/|S_i|)$ [50]. Informally, linear fairness ensures that, at each round (between C_{i-1} and C_i), every process that participates in the round has a "fair chance" to enter its critical section, thus providing "starvation-freedom" in a statistical sense.

Theorem 27 (Kushilevitz and Rabin) *There exists a mutual exclusion algorithm satisfying linear fairness that uses a single* $\Theta(\log \log N)$*-bit shared variable that is accessed by a fetch-and-*ϕ *primitive.* □

Kushilevitz and Rabin also constructed an algorithm that uses a constant-size variable, at the price of reduced fairness — in this algorithm, the probability of $\Omega(1/|S_i|)$ above in the definition of linear fairness is replaced by $\Omega(1/N)$. It is worth mentioning that, if one is interested in *deterministic* algorithms, there exists a strictly-higher lower bound of $\Omega(\log N)$ bits for starvation-free algorithms, which is also tight [26].

In another related paper, Kushilevitz, *et al.* investigated the relationship between fairness and space complexity in randomized algorithms [50]. The proof technique used by them is interesting in its own right; algorithms are modeled using Markov chains (finite automata in which each move is determined statistically) and lower bounds are deduced from the properties of these chains. In particular, they proved that the space complexity of Kushilevitz and Rabin's algorithm is tight.

Theorem 28 (Kushilevitz, *et al.*) *Any mutual exclusion algorithm that satisfies linear fairness requires a shared variable of* $\Omega(\log \log N)$ *bits.* □

k-Exclusion. The k-exclusion problem was posed by Fischer, *et al.* [36] as a generalization of the mutual exclusion problem in which up to k processes may be in their critical sections at the same time. The starvation-freedom property is modified to guarantee progress in the face of up to $k-1$ undetectable process halting failures. In other words, as long as up to $k-1$ processes "fail" by halting, any nonfailed process in its entry section eventually enters its critical section. (A halting failure is different from "shutdown" discussed above, in that a halting process sets *none* of its variables to default values.) In [18], Anderson and Moir presented the first local-spin algorithms for the k-exclusion problem. One such algorithm is covered here.

On first thought, it may seem that the k-exclusion problem could be efficiently solved by simply modifying one of the queue-lock algorithms in Section 3.1 so that a process waits in the queue only if k other processes are already in their critical sections. However, this approach is problematic. To see why, consider the simple (unrealistic) queue-based k-exclusion algorithm in Figure 28(a). The shared variable X counts the number of processes that may safely enter their critical sections. X is initially k. When $X \leq 0$, a process trying to enter its critical section waits in the queue Q. *Enqueue*$(Q,\ p)$ and *Dequeue*(Q) are the normal queue operations, and *Element*$(Q,\ p)$ is a function that returns *true* if and only if p is in Q. Multi-line atomic

```
shared variable
    X: $(k-N)..k$ initially $k$;
    Q: queue of $0..N-1$ initially $\emptyset$

process $p$    /* $0 \le p < N$ */

while true do
1: Noncritical Section;
2: ⟨if $fetch\_and\_dec(X) \le 0$ then
3:     $Enqueue(Q,\ p)$⟩;
4:     await $\neg Element(Q,\ p)$
   fi;
   Critical Section;
5: ⟨$Dequeue(Q)$;
6: $fetch\_and\_inc(X)$⟩
od
```

(a)

```
shared variable
    X: $(k-N)..k$ initially $k$;
    Q: $0..N-1$

process $p$    /* $0 \le p < N$ */

while true do
1: Noncritical Section;
2: ENTRY_N_k+1($p$);
3:     if $fetch\_and\_dec(X) = 0$ then
4:         $Q := p$;
5:         if $X < 0$ then
6:             await $Q \ne p$
               fi
       fi;
       Critical Section;
7:     $fetch\_and\_inc(X)$;
8:     $Q := p$;
9: EXIT_N_k+1($p$)
od
```

(b)

Figure 28: **(a)** k-exclusion using atomic queue procedures. **(b)** Algorithm AM: a simple local-spin k-exclusion algorithm for CC machines.

statements are enclosed in angle brackets. (Because lines 2 and 3 and lines 5 and 6 execute atomically, $X = |Q|$ is trivially an invariant.)

Aside from the multi-line atomic statements, there are two difficulties involved with implementing this algorithm. First, the queue operations typically require several atomic steps if implemented using only simple primitives. Second, a queue imposes a linear order on the waiting processes. If a process in the queue fails, then other processes in the queue are blocked.

Note, however, that both problems disappear when $N = k+1$, because at most one process ever waits in the queue. This insight is the basis of the algorithms presented in [18]. Specifically, Anderson and Moir concentrate on solving k-exclusion for $k+1$ processes, and then inductively apply such a solution to solve k-exclusion for N processes.

One such algorithm, denoted Algorithm AM, is illustrated in Figure 28(b). In this algorithm, the procedures "ENTRY_N_k+1(p)" and "EXIT_N_k+1(p)" are inductively assumed to implement $k+1$ exclusion for N processes. As shown in [18], by considering different implementations of these procedures, algorithms with different space and time complexities can be obtained (*e.g.*, algorithms that are adaptive or that have a fast path). In Algorithm AM, the idea of having one process in the queue is approximated by using a shared variable Q to store the identifier of the process that is "in the queue." A process can perform the dual functions of enqueueing itself and dequeuing the previously-queued process by simply assigning its own process identifier to Q. The variable X in Algorithm AM is used in the same way as in the queue-based algorithm of Figure 28(a). It is straightforward to see that the busy-waiting loop at line 6 in Algorithm AM generates a constant number of remote memory references on a CC machine. Corresponding algorithms for DSM machines can be found in [18]. All of these algorithms use strong synchronization primitives. This gives rise to the following open problem.

Open Problem: Can the k-exclusion problem be solved by a local-spin algorithm that uses only reads and writes?

Group mutual exclusion. The group mutual exclusion problem, which was proposed only recently [43], is another generalization of the mutual exclusion problem that allows multiple processes to execute their

```
shared variable
    $Session$: integer initally $1$;
    $Num$: integer initially $0$;
    $Q$: queue of $0..N-1$ initially $\emptyset$;
    $Wait$: array$[0..N-1]$ of boolean;
    $Need$: array$[0..N-1]$ of integer

process $p$    /* $0 \le p < N$ */

private variable
    $t,\ v$: integer

while $true$ do
1:      Noncritical Section;
2:      $t :=$ session to attend;
3:      $Wait[p] := false$;
4:      $Need[p] := t$;
5:      ENTRY_N($p$);
6:          if $Session = t \ \wedge\ Q = \emptyset$ then
7:              $Num := Num + 1$
8:          else if $Session \neq t \ \wedge\ Num = 0$ then
9:              $Session := t$;
10:             $Num := 1$
            else
11:             $Wait[p] := true$;
12:             $Enqueue(Q,\ p)$
            fi;
13:     EXIT_N($p$);
14:     await $\neg Wait[p]$;
15:     $\langle\langle$Attend session $t\rangle\rangle$
16:     ENTRY_N($p$);
17:         $Num := Num - 1$;
18:         if $Q \neq \emptyset \ \wedge\ Num = 0$ then
19:             $Session := Need[Head(Q)]$;
20:             for each $v \in Q$ do
21:                 if $Need[v] = Session$ then
22:                     $Delete(Q,\ v)$;
23:                     $Num := Num + 1$;
24:                     $Wait[v] := false$
                    fi
                od
            fi;
25:     EXIT_N($p$)
od
```

Figure 29: Algorithm KM: A simple local-spin group mutual exclusion algorithm.

critical sections simultaneously. In this problem, processes vie to execute within one or more "sessions." Multiple processes may simultaneously execute within the same session, but different sessions cannot be active at the same time. Any process that requests to attend a session must eventually be able to do so. In addition, "if process p has requested session t, and no process is currently attending or requesting a different session, then p can attend session t without waiting for other processes to leave the session" [44]. The group mutual exclusion problem generalizes the readers/writers problem. To see this, note that the latter problem can be solved by allowing readers to request the same session, and by requiring all writers to request different sessions. The group mutual exclusion problem was first defined and solved by Joung [43]. Keane and Moir presented the first local-spin solution [44]. We cover here Keane and Moir's solution, because it is much simpler than Joung's.

Keane and Moir's algorithm, denoted Algorithm KM, is shown in Figure 29. In Algorithm KM, a local-spin mutual exclusion algorithm is used. Its entry and exit sections are denoted "ENTRY_N(p)" and "EXIT_N(p)," as before. Each process p has a dedicated spin location, $Wait[p]$, which it uses to block until the session it has requested can be safely entered. The shared variable $Need[p]$ is used to record the session requested by p. The algorithm also employs a shared queue Q of processes that are blocked. This queue is accessed only within critical sections (lines 6–12 and 17–24), so it is really a sequential data structure. (Indeed, the fact that a *sequential* queue suffices is the main insight behind the algorithm.)

A process p executes lines 6–12 to determine if it must wait. If p requests session t, this is the current session, and no processes are waiting, then it may join session t. The current session is stored in the shared variable $Session$, and the number of processes participating in it is stored in the shared variable Num. If the current session differs from t and no processes are participating in it, then p records that t is now the current session (lines 9–10). The remaining possibility is that the current session differs from t and there exist process processes that are participating in it. In this case, p must block (lines 11–12).

After finishing its session, p decrements *Num* (line 17), and then checks to see if there exist blocked processes that can be released (lines 18–24). This will be the case if $Q \neq \emptyset \ \wedge \ Num = 0$ holds (line 18). If this condition holds, then all processes seeking to join the session requested by the process at the head of Q are unblocked (lines 20–24).

10 Concluding Remarks

The mutual exclusion problem has been a topic of active research from its inception to the present day. We hope that this survey will prove to be a useful supplement to Raynal's book for researchers interested in this problem. We also hope that the open problems mentioned in this paper will fuel continued work on this important topic.

Acknowledgement: We are grateful to Faith Fich for her detailed comments on several earlier drafts of this paper.

References

[1] U. Abraham, S. Dolev, T. Herman, and I. Koll. Self-stabilizing l-exclusion. *Theoretical Computer Science*, 266:653–692, 2001.

[2] Y. Afek, H. Attiya, A. Fouren, G. Stupp, and D. Touitou. Adaptive long-lived renaming using bounded memory. Unpublished manuscript, 1999.

[3] Y. Afek, H. Attiya, A. Fouren, G. Stupp, and D. Touitou. Long-lived renaming made adaptive. In *Proceedings of the 18th Annual ACM Symposium on Principles of Distributed Computing*, pages 91–103. ACM, May 1999.

[4] Y. Afek and M. Merritt. Fast, wait-free $(2k-1)$-renaming. In *Proceedings of the 18th Annual ACM Symposium on Principles of Distributed Computing*, pages 105–112. ACM, May 1999.

[5] Y. Afek, G. Stupp, and D. Touitou. Long-lived adaptive collect with applications. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, pages 262–272. IEEE, October 1999.

[6] Y. Afek, G. Stupp, and D. Touitou. Long-lived adaptive splitter and applications. *Distributed Computing*, 15(2):67–86, 2002.

[7] R. Alur, H. Attiya, and G. Taubenfeld. Time-adaptive algorithms for synchronization. In *Proceedings of the 26th Annual ACM Symposium on Theory of Computing*, pages 800–809. ACM, May 1994.

[8] R. Alur and G. Taubenfeld. Results about fast mutual exclusion. In *Proceedings of the 13th IEEE Real-Time Systems Symposium*, pages 12–21. IEEE, 1992.

[9] R. Alur and G. Taubenfeld. Contention-free complexity of shared memory algorithms. *Information and Computation*, 126(1):62–73, April 1996.

[10] R. Alur and G. Taubenfeld. Fast timing-based algorithms. *Distributed Computing*, 10(1):1–10, 1996.

[11] J. Anderson. A fine-grained solution to the mutual exclusion problem. *Acta Informatica*, 30(3):249–265, May 1993.

[12] J. Anderson, R. Jain, and K. Jeffay. Efficient object sharing in quantum-based real-time systems. In *Proceedings of the 19th IEEE Real-Time Systems Symposium*, pages 346–355. IEEE, December 1998.

[13] J. Anderson and Y.-J. Kim. Adaptive mutual exclusion with local spinning. In *Proceedings of the 14th International Symposium on Distributed Computing*, pages 29–43, October 2000.

[14] J. Anderson and Y.-J. Kim. An improved lower bound for the time complexity of mutual exclusion. In *Proceedings of the 20th Annual ACM Symposium on Principles of Distributed Computing*, pages 90–99, August 2001.

[15] J. Anderson and Y.-J. Kim. A new fast-path mechanism for mutual exclusion. *Distributed Computing*, 14(1):17–29, January 2001.

[16] J. Anderson and Y.-J. Kim. Local-spin mutual exclusion using fetch-and-ϕ primitives. In *Proceedings of the 23rd IEEE International Conference on Distributed Computing Systems*, May 2003 (to appear).

[17] J. Anderson and Y.-J. Kim. Nonatomic mutual exclusion with local spinning. In *Proceedings of the 21st Annual ACM Symposium on Principles of Distributed Computing*, pages 3–12, July 2002.

[18] J. Anderson and M. Moir. Using local-spin k-exclusion algorithms to improve wait-free object implementations. *Distributed Computing*, 11(1):1–20, September 1997.

[19] J. Anderson and J.-H. Yang. Time/contention tradeoffs for multiprocessor synchronization. *Information and Computation*, 124(1):68–84, January 1996.

[20] T. Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, January 1990.

[21] H. Attiya and V. Bortnikov. Adaptive and efficient mutual exclusion. In *Proceedings of the 19th Annual ACM Symposium on Principles of Distributed Computing*, pages 91–100. ACM, July 2000.

[22] H. Attiya and A. Fouren. Adaptive wait-free algorithms for lattice agreement and renaming. In *Proceedings of the 17th Annual ACM Symposium on Principles of Distributed Computing*, pages 277–286. ACM, July 1998.

[23] H. Attiya and A. Fouren. Adaptive long-lived renaming with read and write operations. Technical Report CS0956, Faculty of Computer Science, Technion, Haifa, 1999.

[24] H. Buhrman, J. Garay, J. Hoepman, and M. Moir. Long-lived renaming made fast. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing*, pages 194–203. ACM, August 1995.

[25] J. Burns. Mutual exclusion with linear waiting using binary shared variables. In *ACM SIGACT News*, pages 42–47, Summer 1978.

[26] J. Burns, M. Fischer, P. Jackson, N. Lynch, and G. Peterson. Data requirements for implementation of n-process mutual exclusion using a single shared variable. *Journal of the ACM*, 29(1):183–205, January 1982.

[27] J. Burns and N. Lynch. Mutual exclusion using indivisible reads and writes. In *Proceedings of the 18th Annual Allerton Conference on Communication, Control, and Computing*, pages 833–842, 1980.

[28] J. Burns and N. Lynch. Bounds on shared memory for mutual exclusion. *Information and Computation*, 107(2):171–184, December 1993.

[29] M. Choy and A. Singh. Adaptive solutions to the mutual exclusion problem. *Distributed Computing*, 8(1):1–17, 1994.

[30] P. Courtois, F. Heymans, and D. Parnas. Concurrent control with readers and writers. *Communications of the ACM*, 14(10):667–668, 1971.

[31] T. Craig. Queuing spin lock algorithms to support timing predictability. In *Proceedings of the 14th IEEE Real-Time Systems Symposium*, pages 148–156, December 1993.

[32] R. Cypher. The communication requirements of mutual exclusion. In *Proceedings of the Seventh Annual Symposium on Parallel Algorithms and Architectures*, pages 147–156, June 1995.

[33] E. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, 1965.

[34] E. Dijkstra. Self-stabilizing systems in spite of distributed control, EWD 391. In *Selected Writings on Computing: A Personal Perspective*, pages 41–46. Springer-Verlag, Berlin, 1982.

[35] C. Dwork, M. Herlihy, and O. Waarts. Contention in shared memory algorithms. *Journal of the ACM*, 44(6):779–805, November 1997.

[36] M. Fischer, N. Lynch, J. Burns, and A. Borodin. Resource allocation with immunity to process failure. In *Proceedings of the 20th Annual IEEE Symposium on Foundations of Computer Science*, pages 234–254. IEEE, October 1979.

[37] S. Fu and N.-F. Tzeng. A circular list-based mutual exclusion scheme for large shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 8(6):628–639, June 1997.

[38] E. Gafni and M. Mitzenmacher. Analysis of timing-based mutual exclusion with random times. In *Proceedings of the 19th Annual ACM Symposium on Principles of Distributed Computing*, pages 13–21. ACM, 1999.

[39] G. Graunke and S. Thakkar. Synchronization algorithms for shared-memory multiprocessors. *IEEE Computer*, 23:60–69, June 1990.

[40] T.-L. Huang. Fast and fair mutual exclusion for shared memory systems. In *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems*, pages 224–231, June 1999.

[41] T.-L. Huang and C.-H. Shann. A comment on "A circular list-based mutual exclusion scheme for large shared-memory multiprocessors". *IEEE Transactions on Parallel and Distributed Systems*, 9(4):415–416, April 1998.

[42] T. Johnson and K. Harathi. A prioritized multiprocessor spin lock. *IEEE Transactions on Parallel and Distributed Systems*, 8(9):926–933, September 1997.

[43] Y.-J. Joung. Asynchronous group mutual exclusion. *Distributed Computing*, 13(4):189–206, November 2000.

[44] P. Keane and M. Moir. A simple, local-spin group mutual exclusion algorithm. In *Proceedings of the 18th Annual ACM Symposium on Principles of Distributed Computing*, pages 23– 32, May 1999.

[45] J. Kessels. Arbitration without common modifiable variables. *Acta Informatica*, 17:135–141, 1982.

[46] Y.-J. Kim and J. Anderson. A time complexity bound for adaptive mutual exclusion. In *Proceedings of the 15th International Symposium on Distributed Computing*, pages 1–15, October 2001.

[47] Y.-J. Kim and J. Anderson. A space- and time-efficient local-spin spin lock. *Information Processing Letters*, 84(1):47–55, September 2002.

[48] D. Knuth. Additional comments on a problem in concurrent programming control. *Communications of the ACM*, 9(5):321–322, 1966.

[49] L. Kontothanassis, R. Wisniewski, and M. Scott. Scheduler-conscious synchronization. *ACM Transactions on Computer Systems*, 15(1):3–40, February 1997.

[50] E. Kushilevitz, Y. Mansour, M. Rabin, and D. Zuckerman. Lower bounds for randomized mutual exclusion. *SIAM Journal on Computing*, 27(6):1550–1563, December 1998.

[51] E. Kushilevitz and M. Rabin. Randomized mutual exclusion algorithms revisited. In *Proceedings of the 11th Annual ACM Symposium on Principles of Distributed Computing*, pages 275–283, August 1992.

[52] L. Lamport. A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8):453–455, August 1974.

[53] L. Lamport. A new approach to proving the correctness of multiprocess programs. *ACM Transactions on Programming Languages and Systems*, 1(1):84–97, July 1979.

[54] L. Lamport. The mutual exclusion problem: Part I - A theory of interprocess communication. *Journal of the ACM*, 33(2):313–326, 1986.

[55] L. Lamport. The mutual exclusion problem: Part II - Statement and solutions. *Journal of the ACM*, 33(2):327–348, 1986.

[56] L. Lamport. A fast mutual exclusion algorithm. *ACM Transactions on Computer Systems*, 5(1):1–11, February 1987.

[57] D. Lehman, A. Pnueli, and J. Stavi. Impartiality, justice, and fairness: The ethics of concurrent termination. In *Proceedings of the 8th ICALP*. Lecture Notes in Computer Science, Vol. 115, Springer Verlag, July 1981.

[58] E. Lycklama and V. Hadzilacos. A first-come-first-served mutual-exclusion algorithm with small communication variables. *ACM Transactions on Programming Languages and Systems*, 13(4):558–576, October 1991.

[59] N. Lynch and N. Shavit. Timing based mutual exclusion. In *Proceedings of the 13th IEEE Real-Time Systems Symposium*, pages 2–11. IEEE, December 1992.

[60] E. Markatos. Multiprocessor synchronization primitives with priorities. In *Proceedings of the 1991 IFAC Workshop on Real-Time Programming*, pages 1–7. Pergamon Press, 1991.

[61] J. Mellor-Crummey and M. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, February 1991.

[62] J. Mellor-Crummey and M. Scott. Scalable reader-writer synchronization for shared-memory multiprocessors. In *Proceedings of the Third ACM Symposium on Principles and Practice of Parallel Programming*, pages 106–113. ACM, April 1991.

[63] M. Merritt and G. Taubenfeld. Speeding Lamport's fast mutual exclusion algorithm. *Information Processing Letters*, 45:137–142, 1993.

[64] M. Michael and M. Scott. Fast mutual exclusion, even with contention. Technical Report TR-460, University of Rochester, Rochester, NY, 1993.

[65] M. Moir and J. Anderson. Fast, long-lived renaming. In *Proceedings of the 8th International Workshop on Distributed Algorithms*, pages 141–155, September 1994.

[66] M. Moir and J. Anderson. Wait-free algorithms for fast, long-lived renaming. *Science of Computer Programming*, 25(1):1–39, October 1995.

[67] G. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, 12(3):115–116, June 1981.

[68] G. Pfister and V. Norton. "Hot spot" contention and combining in multistage interconnection networks. *IEEE Transactions on Computers*, C-34(10):943–948, October 1985.

[69] M. Rabin. n-process mutual exclusion with bounded waiting by $4 \log_2 n$-valued shared variable. *Journal of Computer and System Sciences*, 25(1):66–75, August 1982.

[70] M. Raynal. *Algorithms for Mutual Exclusion.* The MIT Press, Cambridge, Massachusetts, 1986.

[71] I. Rhee and C.-Y. Lee. An efficient recovery-based spin lock protocol for preemptive shared-memory, multiprocessors. In *Proceedings of the 15th Annual ACM Symposium on the Principles of Distributed Computing*, pages 77–86, May 1996.

[72] R. Rivest and V. Pratt. The mutual exclusion problem for unreliable processes: Preliminary report. In *Proceedings of the 17th Annual IEEE Symposium on the Foundation of Computer Science*, pages 1–8. IEEE, 1976.

[73] M. Scott. Non-blocking timeout in scalable queue-based spin locks. In *Proceedings of the 21st Annual ACM Symposium on Principles of Distributed Computing*, pages 31–40. ACM, July 2002.

[74] M. Scott and W. Scherer. Scalable queue-based spin locks with timeout. In *Proceedings of the Eighth Annual ACM Symposium on Principles and Practice of Parallel Programming*, pages 44–52. ACM, June 2001.

[75] E. Styer. Improving fast mutual exclusion. In *Proceedings of the 11th Annual ACM Symposium on Principles of Distributed Computing*, pages 159–168. ACM, August 1992.

[76] B. Szymanski. Mutual exclusion revisited. In *Proceedings of the Fifth Jerusalem Conference on Information Technology*, pages 110–117. IEEE, October 1990.

[77] Y.-K. Tsay. Deriving a scalable algorithm for mutual exclusion. In *Proceedings of the 12th International Symposium on Distributed Computing*, pages 393–407. Springer Verlag, September 1998.

[78] R. Wisniewski, L. Kontothanassis, and M. Scott. Scalable spin locks for multiprogrammed systems. In *Proceedings of the Eighth International Parallel Processing Symposium*, pages 26–29, April 1994.

[79] R. Wisniewski, L. Kontothanassis, and M. Scott. High performance synchronization algorithms for multiprogrammed multiprocessors. In *Proceedings of the Fifth ACM Symposium on Principles and Practices of Parallel Programming*, pages 199–206. ACM, July 1995.

[80] J.-H. Yang and J. Anderson. A fast, scalable mutual exclusion algorithm. *Distributed Computing*, 9(1):51–60, August 1995.

[81] X. Zhang, Y. Yan, and R. Castañeda. Evaluating and designing software mutual exclusion algorithms on shared-memory multiprocessors. *IEEE Parallel and Distributed Technology*, pages 25–42, Spring Issue, 1996.