

Lamport on Mutual Exclusion: 27 Years of Planting Seeds

James H. Anderson
Department of Computer Science
University of North Carolina at Chapel Hill

Abstract

Mutual exclusion is a topic that Leslie Lamport has returned to many times throughout his career. This article, which is being written in celebration of Lamport's sixtieth birthday, is an attempt to survey some of his many contributions to research on this topic.

1 Introduction

Leslie Lamport is certainly responsible for more groundbreaking results on the mutual exclusion problem than any other single researcher. In this survey article, I describe some of his major contributions to research on mutual exclusion. Looking back on these contributions now, one is struck by the many “seeds” that Lamport ended up planting along the way — seeds that have grown into research topics of independent interest that many other researchers have actively pursued.

For example, Lamport was the first to notice the circular reasoning inherent in shared-memory mutual exclusion algorithms that require atomic instructions [22, 31, 32]. He showed that this circularity can be eliminated, not only in mutual exclusion algorithms, but in general [34]. He established the latter by showing that atomic reads and writes can be implemented from nonatomic reads and writes without mutual exclusion. This work sparked 15 years of subsequent research within the distributed algorithms community on wait-free and lock-free synchronization.

Lamport also initiated the study of mutual exclusion algorithms that are “fast” in the absence of contention [35]. This work led to a flurry of research by others on fast and adaptive algorithms for both mutual exclusion and other problems, such as renaming. Lamport's fast-path code sequence is used directly in many of these other algorithms.

Lamport is also responsible for the first fully distributed solution to the mutual exclusion problem. This algorithm is presented in a paper that also introduces logical clocks and state machines [25]. Logical clocks and state machines are among the most fundamental of all principles in distributed computing. This partially explains why this paper was selected last year as the first recipient of the PODC Influential Paper Award.

In the rest of this article, these and other contributions arising from Lamport's work on mutual exclusion are explained in greater detail.

2 In the Beginning

Mutual exclusion algorithms are used to resolve conflicting accesses to shared resources by asynchronous, concurrent processes. The problem of designing such an algorithm is widely regarded as the preeminent “classic” problem 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.

- **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.

Most of Lamport's work on mutual exclusion has focused on “user-level” algorithms for shared-memory systems, based on either atomic or nonatomic reads and writes. For this reason, attention is limited in most of this article to shared-memory algorithms that do not use strong synchronization primitives or operating-system services. Message-passing systems are considered as well, but only in Sec. 7.

The mutual exclusion problem has been studied for many years. The first N -process algorithm was due to Dijkstra [15]. Dijkstra's algorithm, which is based on an earlier two-process algorithm by Dekker, is livelock-free but not starvation-free. A related algorithm by Knuth was the first starvation-free solution [19]. Dijkstra's and Knuth's algorithms are both quite difficult to understand. In each algorithm, contention is resolved by means of a complicated busy-waiting loop in which many shared variables are read and written. To be convinced that contention is properly resolved, numerous cases must be considered. In two independent papers, Lamport [22] and Peterson [45] proposed simpler solutions, each of which is based upon a more structured approach. Lamport's “more structured” solution is his famous bakery algorithm. As we shall see, the bakery algorithm has implications that stretch well beyond the mutual exclusion problem.

```

shared variable
  Choosing: array[1..N] of 0..1 initially 0;
  Number: array[1..N] of 0..∞ initially 0

process p:                                     /* 1 ≤ p ≤ N */

private variable
  q: 1..N

while true do
  0: Noncritical Section;
  1: Choosing[p] := 1;
  2: Number[p] := 1 + max{Number[1], ..., Number[N]};
  3: Choosing[p] := 0;
  4: for q := 1 to N skip p do
  5:   await Choosing[q] = 0;                                     /* busy wait */
  6:   await Number[q] = 0 ∨ (Number[p], p) < (Number[q], q) /* busy wait */
  od;
  7: Critical Section;
  8: Number[p] := 0
od

```

Figure 1: The bakery algorithm. (In line 6, the notation $(a, b) < (c, d)$ is a shorthand for $(a < c) \vee (a = c \wedge b < d)$.)

3 Mutual Exclusion Made Simple

The bakery algorithm, which is shown in Fig. 1, is based upon a simple scheme often used in bakeries: a customer entering the bakery chooses a number; within the bakery, customers are served in order by their chosen numbers.

3.1 Algorithm Description and Basic Correctness

In the bakery algorithm, process p 's number is given by $Number[p]$. The shared variable $Choosing[p]$ is updated before and after p chooses its number (lines 1 and 3), to allow other processes to detect when p is in the process of choosing. p chooses its number (line 2) by reading each process's number, taking the maximum value, and adding one. Before p can enter its critical section, it must check the status of each other process q (lines 5 and 6). If p detects that q is in the process of choosing its number, then p busy-waits until q has finished choosing (line 5). p then busy-waits by repeatedly reading $Number[q]$ until it finds either $Number[q] = 0$ or $(Number[p], p) < (Number[q], q)$. In the former case, process q is either in its noncritical section or about to choose a new number (which must be larger than p 's, since p blocked until $Choosing[q] = 0$ held). In the latter case, p 's number is at most q 's. A tie here is broken in favor of the process with the smaller process identifier.

To establish the Exclusion property for this program, we define a process to be “in the doorway” if executing within lines 1-3, and “in the bakery” if executing within lines 4-7. Exclusion follows from the following property.

If process p is in its critical section, and if another process q is in the bakery, then $(Number[p], p) < (Number[q], q)$.

This property is easily established. Prior to entering its critical section, process p waits until $Number[q] = 0 \vee (Number[p], p) < (Number[q], q)$ holds. If, at this point, q has already chosen its number, then $Number[q] \neq 0$ holds, and hence $(Number[p], p) < (Number[q], q)$. If q chooses its number later, then it chooses a larger number than p .

The algorithm also satisfies Starvation-freedom. Informally, no process can wait forever, because it will eventually have the smallest (nonzero) number.

3.2 Other Interesting Properties

The bakery algorithm has several other interesting properties, which were the basis of a number of important themes that Lamport revisited many times in subsequent publications. One interesting property of the bakery algorithm is that it remains correct even if reads and writes are nonatomic. Requiring atomic statement execution is tantamount to assuming mutual exclusion in hardware. Thus, mutual exclusion algorithms requiring this are in some sense circular.

With atomic variable accesses, reads and writes can be viewed as taking place instantaneously. With nonatomic accesses, reads and writes take place over intervals of time, and hence may overlap one another. In the bakery algorithm, each shared variable is written by only one process, so overlapping writes of the same variable are not a concern. However, it is possible for a process to read a variable while it is being written. We assume that such a read may return *any* value from the value domain of the variable in question.

Consider the busy-waiting statements at lines 5 and 6. If process p reads $Choosing[q]$ at line 5 while it is being assigned the value 1 by process q at line 1, then p 's read returns either 0 or 1. If it returns 0 (respectively, 1), then it's as if p reads $Choosing[q]$ atomically before (respectively, after) it is written. A similar argument applies if p reads $Choosing[q]$ while it is being assigned the value 0 by q at line 3. Thus, allowing reads and writes of $Choosing[q]$ to be nonatomic causes no problems.

In [22], line 6 is dealt with by an argument similar to the following. Because $Choosing[q] = 1$ holds while q is executing line 2, when p transits from line 5 to line 6, q cannot be executing line 2. Thus, if p reads $Number[q]$ at line 6 while it is being written by q at line 2, then q must be writing a number that is larger than p 's. If p 's read of $Number[q]$ returns a value that is either equal to or larger than that concurrently being written, then p 's busy-waiting loop at

line 6 terminates, as it should. If p reads a smaller value, then it may continue to busy-wait, but p cannot busy-wait here forever, because eventually q must finish writing its number, establishing $(Number[p], p) < (Number[q], q)$. Although the argument given here ostensibly seems correct, as Lamport discovered some years later, it is actually somewhat flawed, because it is based on some implicit (though mild) assumptions regarding how nonatomic writes of $Number[q]$ are implemented; see [37] for details.

Another interesting property of the bakery algorithm is that it is resilient to the premature termination of processes. When a process p terminates prematurely, it is required to set both $Choosing[p]$ and $Number[p]$ to 0 and then return to its noncritical section and halt. It is straightforward to see that the Exclusion and Starvation-freedom properties continue to hold if processes are allowed to terminate prematurely in this way.

The bakery algorithm is also of interest because it satisfies the following rather strong progress property.

- **First-come, First-served (FCFS) Priority:** If process p enters the bakery before q enters the doorway, then p executes its critical section before q .

To see why this property holds, note that if p enters the bakery before q enters the doorway, then q must choose a larger number than p .

In a later paper, Lamport reconsidered the issue of priority [30]. He argued that, in the context of synchronization problems, priority is impossible to specify using known methodologies. He illustrated the point by considering FCFS priority in detail.

Lamport argued that FCFS priorities implicitly require a *request* action, which may be nonatomic (*i.e.*, it may be comprised of a sequence of atomic actions). For the bakery algorithm, the *request* action consists of the statements comprising the doorway. Under FCFS priority ordering, if process p executes its *request* action before process q executes its *request* action, then p executes its critical section before q . The problem with this specification is that there is great leeway in defining what constitutes the *request* action. If *request* is defined to consist of the atomic action performed by a process to transit from its noncritical section to its entry section, then it is impossible for different processes to determine the order in which their *request* actions are executed. Thus, *request* surely must be defined to consist of a sequence of atomic actions. But then an implementor is free to define a process's entire entry section to be the *request* action. With this, the FCFS requirement becomes vacuous. Of course, it is possible to define the *request* action for a particular algorithm, as we did above with the bakery algorithm. The point being made here is that is difficult to *specify* FCFS priority in a way that is generally meaningful.

The bakery algorithm was the impetus for much subsequent work by Lamport on verification issues. At the time of the bakery algorithm's publication, he felt that he did not have adequate formal tools to establish its correctness [46]. As Lamport writes in [24], "[I] have written several multiprocessor algorithms to solve synchronization problems, and given informal proofs of their correctness. Although the proofs were simple and convincing, they were ultimately based on the method of considering all possible execution sequences. This is not well-suited for formal proofs."

Because of this dissatisfaction, Lamport became an early and continuing advocate for assertional proof techniques based on temporal logic. In the first of his many papers

on this topic, he independently proposed the well-known Owicki-Gries method [44] and introduced the terms *safety* and *liveness* [24]. Much of his recent work has focused on the development and application of a formalism called TLA (the temporal logic of actions) [38]. Along the way, he wrote many other important and widely read papers on verification and specification issues (there are too many such papers to mention them all here). Of particular relevance to this survey article is the work he did on verifying programs with nonatomic statements [27, 28, 31, 33, 37]. Some of this work is considered in the next section.

4 Atomicity Questioned

The bakery algorithm showed that the circularity caused by assuming that statements execute atomically can be eliminated, at the price of using unbounded memory. This gives rise to several questions. Is it possible to solve the mutual exclusion problem with nonatomic statements and with bounded memory? Is it possible to do this for other synchronization problems as well? How does one formally reason about programs with nonatomic instructions?

These questions were answered in two two-part papers, [31, 32] and [33, 34], published by Lamport in 1986. In the first pair of papers, a formal model of concurrent systems is presented in which no underlying atomicity is assumed (Part I), and several mutual exclusion algorithms are presented and proved correct based on this model (Part II). In the second pair of papers, the formalism is extended by defining what it means for a low-level system to correctly implement a high-level one (Part I). Several constructions of atomic variables from nonatomic variables are then presented and proved correct using this extended formalism (Part II). In this case, reads and writes of the atomic variable being constructed are operations of the high-level system, and reads and writes of the nonatomic variables used in the construction are operations of the low-level system. As explained below, these constructions are "wait-free." Thus, they can be applied to eliminate the need for hardware-based mutual exclusion mechanisms, not only in solutions to the mutual exclusion problem, but other problems as well.

4.1 A New Formalism

In the formalism of [31, 33], a *process* is defined to consist of a set of operation executions. Informally, an operation execution is a single instance of some action, such as sending a message or reading a shared variable. Formally, an *operation execution* consists of a set of events. Thus, operation executions have *duration*. Two relations are of fundamental importance in the formalism: $A \rightarrow B$ means that operation execution A precedes operation execution B ; $A \dashrightarrow B$ means that A can causally affect B . These relations actually come from an earlier paper of Lamport's, where a similar formalism is used to prove the correctness of a nonatomic variant of the bakery algorithm [27]. The arrow relations are used in [31, 33] to state general axioms that define the effects of concurrent read and write operations, and to define what it means for a low-level system to correctly implement a high-level one. (Some similar axioms are stated in [27] in the context of the bakery algorithm.)

4.2 Mutual Exclusion without Atomicity, Revisited

Four mutual exclusion algorithms are presented in [32]. Due to space limitations, these algorithms are not considered in

detail here. The four algorithms differ in the progress and fault-tolerance properties they satisfy. Each algorithm is constructed using *communication variables*, which are simply single-writer, multi-reader nonatomic boolean variables. (The shared variable *Choosing*[*p*] in the bakery algorithm is an example of a communication variable.) The following fault-tolerance properties are considered.

- **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 and halts.
- **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.
- **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 aborts, setting all of its communication variables to default values.
- **Self-stabilization:** An algorithm is *self-stabilizing* [16] 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.

The simplest algorithm presented in [32] requires only one communication variable per process, and is shutdown safe and fail safe. The most sophisticated algorithm requires $N!$ communication variables per process, and is shutdown safe, abortion safe, fail safe, and self-stabilizing. The main lesson to be drawn from these algorithms is that virtually any variant of the mutual exclusion problem is solvable in systems without hardware support for mutual exclusion.

4.3 Wait-free Register Constructions

As stated above, the problem of implementing atomic variables from nonatomic variables in a wait-free manner is considered in [34]. These variables are formalized as shared objects called *registers*. Only single-writer registers are considered in [34]. Three classes of such registers are defined: *safe*, *regular*, and *atomic*.

- **Safe Registers:** Reads and writes of a safe register must satisfy the following: if a read does not overlap a write, then the read returns the most-recently written value (or the register's initial value if it has not been written). A read that does overlap a write may return any arbitrary value from the value domain of the register. (This is the same notion of nonatomic statement execution considered in [22, 32].)
- **Regular Registers:** A regular register is a safe register that also satisfies the following: if a read overlaps some sequence of writes, then it must return the value written by one of these writes, or the value of the register before the first such write.
- **Atomic Registers:** An atomic register is a regular register that satisfies the following additional property: if a read operation R precedes another read operation S , then S cannot return an "older" value than R . This implies that a valid linearization order can be defined

for concurrent reads and writes. (The notion of linearizability was formally defined a few years later by Herlihy and Wing [17].) Intuitively, each read and write operation must "appear" as if it occurs instantaneously at a single instant of time.

The main contribution of [34] was to show that a single-writer, single-reader, multi-bit atomic register can be constructed in a wait-free manner from a collection of single-writer, single-reader, single-bit safe registers; registers of the latter type essentially correspond to nonatomic flip-flops. This result was established by means of a sequence of register constructions, each showing that some "more powerful" register can be implemented from "less powerful" ones. The *wait-freedom* requirement means that operations of the constructed register are implemented without busy-waiting loops or blocking synchronization constructs. A similar requirement called *lock-freedom* has also been considered in the literature. The difference between the two is that wait-freedom requires starvation freedom for individual processes, while lock-freedom does not.

Lamport's work on atomic registers left two interesting open problems: constructing a multi-reader atomic register, and constructing a multi-writer atomic register. These open problems were just too tempting for some of us to resist. As a result, in the years following the publication of [34], a number of papers on atomic-register constructions appeared [9, 11, 12, 18, 43, 48, 52, 53]. These papers were just the first of a great many papers to be published on wait-free and lock-free synchronization (a complete list of such papers simply can't be included here, because the bibliography alone would exceed the space limitations of this article). In retrospect, the publication of Lamport's work on atomic registers can be seen as a watershed event, because it was *the* catalyst that led to the explosive interest in wait-free and lock-free synchronization in recent years.

4.4 Earlier Roots

Having said that, it is important to note that the notions of wait-free and lock-free synchronization were actually first introduced in earlier papers. The first lock-free algorithm was a concurrent read/write buffer algorithm presented by Lamport in 1977 [23]. The first wait-free algorithms were a collection of read/write buffer algorithms presented by Peterson in 1983 [45].

Interestingly, Sorensen and Hemacher began investigating nonblocking buffering mechanisms for real-time systems even before Lamport's lock-free buffer algorithm was published [49, 50]. Nonblocking algorithms are of interest in real-time systems because they are not susceptible to priority inversions. (A *priority inversion* occurs when a high-priority process is forced to wait on a lower-priority process. Priority inversions are a problem because, under most real-time scheduling disciplines, higher-priority processes have more stringent deadline constraints.) However, Sorensen and Hemacher's buffering mechanisms are not true lock-free or wait-free algorithms because they are implemented within the operating system.

In Lamport's lock-free read/write buffer, the buffer consists of a sequence of "digits," which are read and written atomically. Only one writer is assumed, so there is no need to consider concurrent write operations. The writer is wait-free, but it may interfere with concurrent read operations. If a read operation is interfered with, then it must be retried. Repeated retries may be needed before a read can successfully complete. This is why the algorithm is lock-free but

not wait-free.

The algorithm's correctness depends on several results that are proved in the paper concerning read and write operations of multi-digit buffers. It is shown that if such digits are read and written in opposite directions, then certain conclusions can be drawn that relate the value returned by a read operation and values written to the buffer by write operations.

In Lamport's buffer algorithm, two version numbers $V1$ and $V2$ are associated with the buffer. These version numbers allow readers to detect when they have been interfered with. Each version number is a multi-digit value. To perform a write operation, the writer increments $V1$, writes the buffer, and then increments $V2$. To perform a read operation, a reader reads $V2$, reads the buffer, and then reads $V1$. If the values read from $V2$ and $V1$ are the same, then a consistent value was read from the buffer. Note that the order in which the sequence numbers and the buffer are read is the opposite of the order in which they are written. In addition, the digits comprising each version number are read and written in opposite directions. These read- and write-ordering properties are fundamental in showing that read operations always return consistent values.

Related work includes a later paper by Lamport on the concurrent reading and writing of clocks [36]. A *clock* is defined by a set of regular registers. The algorithms presented in [36] for reading and writing a clock are based on the ideas of [23]. The following interesting comment appears in [36]: "The version of [reference [23] of this article] submitted for publication assumed only regular registers, but the editor was afraid that the concept of nonatomic operations on individual digits might be considered heretical and insisted that it be removed from the paper."

In 1993, Kopetz and Reisinger applied Lamport's lock-free buffer algorithm in a real-time control system [20]. They were apparently unaware that Lamport had invented this algorithm 16 years earlier, because they did not credit him for it. However, their main focus was not the algorithm itself, but scheduling analysis techniques for accounting for synchronization overheads when the algorithm is used.

4.5 Hardware Connections

A concept quite similar to "non-wait-freedom" arises in the study of asynchronous arbiter circuits. I refer to the well-known glitch phenomenon, which is captured by the following principle.

For any device making a decision among a finite number of possible outcomes, based upon a continuum of possible inputs, there will be inputs for which the device takes arbitrarily long to reach its decision. [39]

The connection between the glitch phenomenon and mutual exclusion is explained quite well by Lamport in [21].

When I wrote [reference [22] of this article], a colleague at Massachusetts Computer Associates pointed out that the concurrent reading and writing of a single register, assumed in the bakery algorithm, requires an arbiter — a device for making a binary decision based on inputs that may be changing. In the early 70s, computer designers rediscovered that it's impossible to build an arbiter that is guaranteed to reach a decision in a bounded length of time. (This had been realized in the 50s but had been forgotten.) My

colleague's observation led to my interest in the arbiter problem — or "glitch" problem, as it was sometimes called.

Lamport wrote two papers on the glitch phenomenon. The first of these presents a proof of the glitch principle [39]. The second paper considers several (rather amusing) examples of "glitches" from everyday life. For example, the concept is motivated by considering the "problem of Buridan's Ass" [29]. In this problem, the ass starves to death after being placed equidistant between two sources of food that are both equally preferable.

5 Fast Mutual Exclusion

In 1987, Lamport devised a novel mutual exclusion algorithm that requires only seven memory accesses in the absence of contention [35]. This work was driven by the widely accepted belief that "contention for a critical section is rare in well-designed systems" [35]. 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.

Lamport's fast mutual exclusion algorithm is shown in Fig. 2(a). In this algorithm, the fast path consists of lines 1, 2, 3, 7, 8, 16, and 17. Of these, lines 2, 3, 7, and 8 are of special significance. These lines are shown separately in Fig. 2(c), where they are used to define a "black box" element called a *splitter*, which is illustrated in Fig. 2(b). (The term "splitter" is not due to Lamport; it was first abstracted as a "black box" by Moir and Anderson [41], and first called a "splitter" by Attiya and Fouren [7].) In the following subsection, we consider some properties of the splitter that make it so useful, and then show how these properties ensure the correctness of Lamport's fast mutual exclusion algorithm.

5.1 The Ever-useful 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 3 last. Because p found that $X = p$ held at line 3, X is not written by any process between p 's execution of line 0 and p 's execution of line 3. Thus, q executed line 3 before p executed line 0. This implies that q executed line 2 before p executed line 1. Thus, p must have read $Y = false$ at line 1 and then assigned " $dir := right$," 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 [1, 2, 8, 7, 10, 41, 42]. 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 Fig. 3 [42]. A name is associated with each splitter. If the grid has N rows and N columns, where N is the number of

```

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

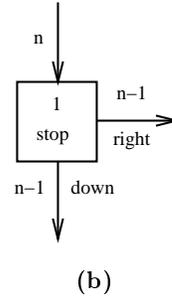
process p:                                /* 1 ≤ p ≤ N */

private variable
  j: 1..N

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

```

(a)



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

```

process p:

private variable
  dir: {stop, right, down}

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

```

(c)

Figure 2: (a) Lamport’s fast mutual exclusion algorithm. (b) The splitter element and (c) its implementation.

processes, then by induction, every process eventually stops at some splitter. (The original name space is assumed to be much larger than N .) When a process stops at a splitter, it acquires the name associated with that splitter.

The splitter’s properties also ensure that the algorithm in Fig. 2(a) 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.” Moreover, if no process takes the fast path during a period of contention, then some process must reach lines 9-14. The algorithm ensures that, of these processes, the last to update the variable Y eventually gets to its critical section and then reopens the fast path by assigning $Y := 0$ at line 16. Thus, the algorithm satisfies Exclusion and Livelock-freedom. On the other hand, Starvation-freedom is not satisfied, because an unlucky process may repeatedly find either $Y \neq 0$ at line 3 or $Y \neq p$ at line 12, and hence wait forever.

5.2 Research on Fast and Adaptive Mutual Exclusion (Or, Other Applications of the Splitter and Related Mechanisms)

Lamport’s fast mutual exclusion algorithm sparked a wave of research on fast and adaptive mutual exclusion algorithms that has continued to this day. Much of this work has

focused on algorithms that are fast both in the absence and presence of contention. In 1993, Yang and Anderson presented an N -process “local-spin” mutual exclusion algorithm that has $\Theta(\log N)$ time complexity, regardless of contention, where “time” is measured by counting only remote memory references that cause a traversal of the interconnect between processors and memory [54]. They also presented a fast-path variant that has $O(1)$ time complexity in the absence of contention. This variant directly uses the splitter code sequence. Unfortunately, Yang and Anderson’s fast-path variant has $\Theta(N)$ worst-case time complexity under contention. This is because of a “polling loop” that is executed to determine if the fast-path can be reopened after a period of contention ends. Recently, Anderson and Kim presented a new fast mechanism, which also directly incorporates the splitter code, that results in $O(1)$ time complexity in the absence of contention and $\Theta(\log N)$ time complexity under contention.

In all of the fast-path algorithms considered until now, there is a sudden jump in time complexity between the contention-free and contention-present cases. Other researchers have considered algorithms where the rise in time complexity as contention increases is more gradual. Such algorithms are called *adaptive*. Research on adaptive algorithms has also been heavily influenced by Lamport’s fast mutual

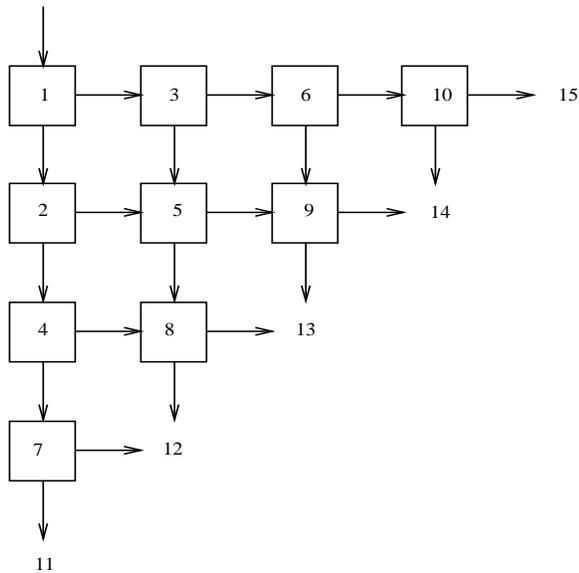


Figure 3: Renaming grid (depicted for $N = 5$).

exclusion algorithm.

In work on adaptive algorithms, two notions of contention have been considered: “interval contention” and “point contention” [1]. 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.

Two time complexity measures have been considered in work on adaptive algorithms in addition to the remote-memory-references measure considered above: “remote step complexity” and “system response time.” The *remote step complexity* 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 [51]. The *system response time* is the length of time between critical section entries, assuming each enabled read or write operation is executed within some constant time bound [13]. Several algorithms have been presented that are adaptive to some degree under these time complexity measures [51, 13, 6, 3]. In each of the papers cited here, either Lamport’s splitter element or a closely related mechanism is used.

Lamport’s fast-path code sequence also has implications for research on time complexity lower bounds [4, 5, 14]. In particular, for many synchronization problems, this code sequence implies that one cannot simply focus on contention-free program executions when trying to establish a nontrivial (*i.e.*, nonconstant) lower bound on time.

6 Sequential Consistency and True vs. Virtual Mutual Exclusion

Lamport was the first to observe that the assumptions regarding memory accesses implicit within most proof frameworks may be violated in actual multiprocessor systems. He proposed a condition called *sequential consistency* that ensures that multiprocessor programs are executed correctly

[26]. Sequential consistency is defined as follows.

[T]he result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program. [26]

The main contribution of [26] was to *formally* articulate the memory model that most people had previously assumed implicitly. Sequential consistency has since become the “benchmark” against which all other memory consistency policies are compared.

In recent joint work with Perl and Weihl, Lamport showed that mutual exclusion algorithms designed assuming sequentially consistent memory may not guarantee true mutual exclusion in practice [40]. To see why this is so, note that sequential consistency does not require the actual execution order of operations to be consistent with any *particular* sequential order. In particular, operations of the same processor may be performed in an order that differs from program order as long as both orders give the same result. For example, some operations within a process’s exit section may actually be performed before certain operations within its critical section. Also, operations of different processors that commute can be ordered quite arbitrarily.

The following motivating example is given in [40].

[C]onsider the following basic mutual exclusion algorithm that is used by several standard algorithms.

Processor A:

```
a := 1;
if b = 0 then critical section; a := 0 fi
```

Processor B:

```
b := 1;
if a = 0 then critical section; b := 0 fi
```

Suppose neither of the critical sections accesses a , b , or any variable accessed by the other. A multiprocessor could execute A ’s assignment $a := 1$, its *if* test, and its assignment $a := 0$, and then arbitrarily interleave the execution of its critical section with the execution of processor B ’s protocol. Because the memory operations of processor A ’s critical section commute with all the memory operations of processor B ’s protocol, the resulting execution is equivalent to executing the two protocols sequentially (in either order). Thus, the definition of sequential consistency, which says nothing about the order in which operations are actually executed, is satisfied. Hence virtual but not real mutual exclusion is satisfied.

With *virtual* mutual exclusion, operations are executed in a way that makes it appear as if one critical section precedes another. If only memory accesses are performed during critical sections, then virtual mutual exclusion is sufficient. However, if I/O operations are performed, then true mutual exclusion is needed.

Perl *et al.* posit a special statement called *Lull* that prevents lookahead. Specifically, if a process performs a *Lull* statement, then none of its instructions after the *Lull* can be performed until all of its instructions prior to the *Lull* have completed. The main contribution of [40] is the following result.

If two processors execute critical sections each containing a *Lull* statement, then critical-section instructions that precede the *Lull* statements cannot be concurrently executed by the two processors.

Thus, the *Lull* statement provides a simple means for ensuring true mutual exclusion.

7 Last But Definitely Not Least

The last of Lamport's papers on mutual exclusion covered in this article is his seminal paper "Time, Clocks, and the Ordering of Events in a Distributed System" [25]. This is the only paper surveyed in this article in which a message-passing model is explicitly assumed. Although this paper is being covered last, it is probably Lamport's most-cited paper. In fact, it is among the most frequently-cited papers in all of distributed computing.

There is a common misconception that the "Time, Clocks" paper is primarily a paper about mutual exclusion. This is incorrect. The main contribution of this paper is a new method for synchronizing distributed processes, based on two concepts: logical clocks and state machines. A mutual exclusion algorithm is presented merely to illustrate the method. This algorithm just happened to be the first fully distributed solution to the mutual exclusion problem.

7.1 Logical Clocks and "Happens Before"

The most fundamental contribution of [25] is the notion of a *logical clock*, which is an inexpensive mechanism that can be used order events in an asynchronous distributed system in a meaningful way. In the model of this paper, each process is defined to be a linear sequence of events. Each event is either an internal event, a message-send event, or a message-receive event. In any distributed system, there is an implicit ordering of events. This ordering is captured by a fundamental partial-order relation introduced in this paper called the "happens-before" relation, which is denoted " \rightarrow ." This relation is defined as follows.

- If a and b are events of the same process and a occurs before b , then $a \rightarrow b$.
- If a is the sending of a message by one process and b is the receipt of that message by another process, then $a \rightarrow b$.
- If $a \rightarrow b \wedge b \rightarrow c$ then $a \rightarrow c$.

The happens-before relation is among the most fundamental of all notions in distributed computing. Indeed, it is so ubiquitous, it would simply be impossible to compile a complete list of papers in which it is used.

For an ordering of events in a distributed system to be "meaningful," it should be consistent with the happens-before relation. Logical clocks provide a means for constructing such an ordering. Using the notation of [25], a *logical clock of a process* P_i is a function C_i that assigns an integer value to each event in P_i . The *logical clock for an entire system of processes* is a function C that assigns to any event b of process P_j the number $C_j(b)$. Logical clocks are required to satisfy the following.

Clock Condition: For any events a and b , if $a \rightarrow b$, then $C(a) < C(b)$.

Logical clocks are easy to implement. A single counter K_i is assigned to each process P_i . This counter is updated as follows.

- Each internal event b is replaced by the single event $\langle b; K_i := K_i + 1 \rangle$.
- Each message-send event $send(m)$ is replaced by the single event $\langle send(m, K_i + 1); K_i := K_i + 1 \rangle$.
- Each message-receive event $receive(m, K)$ is replaced by the single event $\langle receive(m, K); K_i := \max(K, K_i) + 1 \rangle$.

Process P_i 's logical clock is defined by requiring $C_i(b)$ to equal the value of K_i immediately after the occurrence of the event b . A useful total ordering on events \Rightarrow can now be defined as follows.

$a \Rightarrow b$ if and only if

- $C(a) < C(b)$, or
- $C(a) = C(b)$ and a is in P_i and b is in P_j and $i < j$.

7.2 State Machines and Mutual Exclusion

The synchronization method of [25] couples logical clocks with the concept of a state machine. A *state machine* is defined by a set of states and a set of commands. Each command causes a deterministic state transition. State machines are fundamental to the study of active replication techniques for ensuring fault tolerance [47]. Each replica can be defined by a state machine that responds to client requests.

In [25], Lamport illustrated the utility of state machines and logical clocks by presenting a new distributed mutual exclusion algorithm that incorporates these concepts. This algorithm is sketched in Fig. 4. The main idea behind the algorithm is that of distributing a queue. Each process is defined by a state machine that responds to timestamped *request* and *release* commands. Each process's state is defined by a local queue of timestamped *request* messages (commands), which it has received from processes that are waiting to enter their critical sections. The condition for critical-section entry easily implies that the Exclusion property is satisfied by this algorithm. It is also easy to see that each process's *request* eventually must have the lowest timestamp. Hence, the algorithm satisfies Starvation-freedom.

8 Concluding Remarks

It is difficult to fully describe the impact of Leslie Lamport's work on the mutual exclusion problem in a limited amount of space. Nonetheless, I hope this article has done an adequate job of it. In assessing Lamport's track record on the mutual exclusion problem, it is quite evident that he has been *the* foremost figure for many years in defining the course of research on this problem.

At the time this article was written, a complete listing of Lamport's papers was available on-line at [21]. Electronic copies of many of his papers can be obtained there.

Acknowledgements: I am grateful to Cynthia Dwork, Yong-Jik Kim, Keith Marzullo, and Fred Schneider for their comments on an earlier draft of this paper.

<p>Protocol for P_i:</p> <p>To request critical section:</p> <ul style="list-style-type: none"> • Send a timestamped <i>request</i> message to every other process. • Put that message on a local request queue. <p>When <i>request</i> message is received:</p> <ul style="list-style-type: none"> • Put it on local request queue. • Send back a timestamped <i>acknowledgement</i>. <p>To exit critical section:</p> <ul style="list-style-type: none"> • Remove P_i's <i>request</i> message from local request queue. • Send a <i>release</i> message to all other processes. 	<p>When <i>release</i> message is received:</p> <ul style="list-style-type: none"> • Remove corresponding <i>request</i> message from local request queue. <p>P_i enters its critical section if both of the following hold:</p> <ul style="list-style-type: none"> • Its own <i>request</i> has the lowest timestamp (according to \Rightarrow) among all <i>requests</i> in its queue. • P_i has received a message from every other process timestamped later than its own <i>request</i>.
--	--

Figure 4: Timestamp algorithm. Note: FIFO message delivery is assumed.

References

- [1] 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*, pp. 91–103. May 1999.
- [2] Y. Afek and M. Merritt. Fast, wait-free $(2k - 1)$ -renaming. In *Proceedings of the 18th Annual ACM Symposium on Principles of Distributed Computing*, pp. 105–112. May 1999.
- [3] J. Anderson and Y.-J. Kim. Adaptive mutual exclusion with local spinning. In *Proceedings of the 14th International Symposium on Distributed Computing*, pp. 29–43, October 2000.
- [4] 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* (this proceedings), August 2001.
- [5] J. Anderson and J.-H. Yang. Time/contention tradeoffs for multiprocessor synchronization. *Information and Computation*, 124(1):68–84, January 1996.
- [6] H. Attiya and V. Bortnikov. Adaptive and efficient mutual exclusion. In *Proceedings of the 19th Annual ACM Symposium on Principles of Distributed Computing*, pp. 91–100. July 2000.
- [7] 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*, pp. 277–286. July 1998.
- [8] H. Attiya and A. Fouren. Adaptive long-lived renaming with read and write operations. Technical Report CS0956, Faculty of Computer Science, Technion, Haifa, 1999.
- [9] B. Bloom. Constructing two-writer atomic registers. *IEEE Transactions on Computer Systems*, 37(12):1506–1514, December 1988.
- [10] 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*, pp. 194–203. August 1995.
- [11] J. Burns and G. Peterson. Constructing multi-reader atomic values from non-atomic values. In *Proceedings of the Eighth Annual ACM Symposium on Principles of Distributed Computing*, pp. 222–231, August 1987.
- [12] J. Burns and G. Peterson. Pure buffers for concurrent reading while writing. Technical Report GIT-ICS-87/17, School of Information and Computer Science, Georgia Institute of Technology, 1987.
- [13] M. Choy and A. Singh. Adaptive solutions to the mutual exclusion problem. *Distributed Computing*, 8(1):1–17, 1994.
- [14] R. Cypher. The communication requirements of mutual exclusion. In *Proceedings of the 17th Annual Symposium on Parallel Algorithms and Architectures*, pp. 147–156, 1995.
- [15] E. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, 1965.
- [16] E. Dijkstra. Self-stabilizing systems in spite of distributed control, EWD 391. In *Selected Writings on Computing: A Personal Perspective*, pp. 41–46. Springer-Verlag, Berlin, 1982.
- [17] M. Herlihy and J. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.
- [18] L. Kirousis, E. Kranakis, and P. Vitanyi. Atomic multireader register. In *Proceedings of the Second International Workshop on Distributed Algorithms*, pp. 278–296, October 1987.

- [19] D. Knuth. Additional comments on a problem in concurrent programming control. *Communications of the ACM*, 9(5):321–322, 1966.
- [20] H. Kopetz and J. Reisinger. The non-blocking write protocol nbw: A solution to a real-time synchronization problem. In *Proceedings of the 14th IEEE Symposium on Real-Time Systems*, pp. 131–137. December 1993.
- [21] L. Lamport. My writings. <http://www.research.compaq.com/SRC/personal/lamport/pubs/pubs.html>.
- [22] L. Lamport. A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8):453–455, August 1974.
- [23] L. Lamport. Concurrent reading and writing. *Communications of the ACM*, 20(11):806–811, November 1977.
- [24] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, SE-3(2):125–143, March 1977.
- [25] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.
- [26] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.
- [27] 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.
- [28] L. Lamport. Reasoning about nonatomic operations. In *Proceedings of the Tenth Annual ACM Symposium on Principles of Programming Languages*, pp. 28–37. 1983.
- [29] L. Lamport. Buridan's principle. Technical report, SRI Technical Report, October 1984.
- [30] L. Lamport. What it means for a concurrent program to satisfy a specification: Why no one has specified priority. In *Proceedings of the 12th Annual ACM Symposium on Principles of Programming Languages*, pp. 78–83. January 1985.
- [31] L. Lamport. The mutual exclusion problem: Part I - A theory of interprocess communication. *Journal of the ACM*, 33(2):313–326, 1986.
- [32] L. Lamport. The mutual exclusion problem: Part II - Statement and solutions. *Journal of the ACM*, 33(2):327–348, 1986.
- [33] L. Lamport. On interprocess communication: Part I - Basic formalism. *Distributed Computing*, 1:77–85, 1986.
- [34] L. Lamport. On interprocess communication: Part II - Algorithms. *Distributed Computing*, 1:86–101, 1986.
- [35] L. Lamport. A fast mutual exclusion algorithm. *ACM Transactions on Computer Systems*, 5(1):1–11, February 1987.
- [36] L. Lamport. Concurrent reading and writing of clocks. *ACM Transactions on Computer Systems*, 8(4):305–310, November 1990.
- [37] L. Lamport. *win* and *sin*: Predicate transformers for concurrency. *ACM Transactions on Programming Languages and Systems*, 12(3):396–428, July 1990.
- [38] L. Lamport. Introduction to TLA. Technical Report 1994-001, Digital Systems Research Center, Palo Alto, CA, 1994.
- [39] L. Lamport and R. Palais. On the glitch phenomenon. Technical Report CA-7611-0811, Massachusetts Computer Associates, Wakefield, Massachusetts, November 1976.
- [40] L. Lamport, S. Perl, and W. Weihl. When does a correct mutual exclusion algorithm guarantee mutual exclusion? *Information Processing Letters*, 76(3):131–134, 2000.
- [41] M. Moir and J. Anderson. Fast, long-lived renaming. In *Proceedings of the Eighth International Workshop on Distributed Algorithms*, pp. 141–155, September 1994.
- [42] M. Moir and J. Anderson. Wait-free algorithms for fast, long-lived renaming. *Science of Computer Programming*, 25(1):1–39, October 1995.
- [43] R. Newman-Wolfe. A protocol for wait-free, atomic, multi-reader shared variables. In *Proceedings of the Sixth Annual Symposium on Principles of Distributed Computing*, pp. 232–248, 1987.
- [44] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 6:319–340, 1976.
- [45] G. Peterson. Concurrent reading while writing. *ACM Transactions on Programming Languages and Systems*, 5(1):46–55, 1983.
- [46] F. Schneider. Private communication.
- [47] F. Schneider. Implementing fault-tolerant services using the state-machine approach. *ACM Computing Surveys*, 22, December 1990.
- [48] A. Singh, J. Anderson, and M. Gouda. The elusive atomic register. *Journal of the ACM*, 41(2):311–339, 1994.
- [49] P. Sorensen. *A Methodology for Real-Time System Development*. PhD thesis, University of Toronto, Toronto, Canada, 1974.
- [50] P. Sorensen and V. Hemachar. A real-time system design methodology. *INFOR*, 13(1):1–18, 1975.
- [51] E. Styer. Improving fast mutual exclusion. In *Proceedings of the 11th Annual ACM Symposium on Principles of Distributed Computing*, pp. 159–168. August 1992.
- [52] J. Tromp. How to construct an atomic variable. In *Proceedings of the Third International Workshop on Distributed Algorithms*, pp. 292–302. Lecture Notes in Computer Science 392, Springer-Verlag, 1989.
- [53] P. Vitanyi and B. Awerbuch. Atomic shared register access by asynchronous hardware. In *Proceedings of the 27th IEEE Symposium on the Foundations of Computer Science*, pp. 233–243, 1986.
- [54] J.-H. Yang and J. Anderson. Fast, scalable synchronization with minimal hardware support. In *Proceedings of the 12th Annual ACM Symposium on Principles of Distributed Computing*, pp. 171–182. August 1993.