

NOTES ON MUTUAL EXCLUSION — Part 2

1. The bakery algorithm

The model of computation assumed in Dijkstra’s algorithm and its variants is somewhat unsatisfactory in that it assumes mutual exclusion at a low level (between read and write operations) to get mutual exclusion at a higher level (between critical section operations). Even though it is often realistic to assume that read and write operations exclude each other (e.g., because they must happen at different clock cycles of the computer) it is at least theoretically interesting to ask whether this assumption can be removed. Could we solve the mutual exclusion problem *without* assuming mutual exclusion between operations at *any* level? The answer is, perhaps surprisingly, affirmative. Lamport was the first to pursue this question and his investigations have led to many interesting (and some possibly practical) ideas going well beyond the mutual exclusion problem. The new, weaker assumptions that we’ll make about the model of computation are:

- (1) Processes can communicate only by reading or writing shared variables.
- (2) Reading or writing a shared variable is *not* an atomic (i.e., indivisible) action. Thus a process may be reading a shared variable while another is writing into it and, in that event, the value returned to the reader is an arbitrary but legitimate value for that variable. Furthermore, the reader does not “know” that it interfered with a writer. However, a read operation which is not concurrent with any write operation of the same shared variable will return the “correct” value, i.e., the value last written into that variable. To prevent concurrent write operations on the same shared variable (which would raise the issue of defining the “correct” value of a variable after such concurrent writing) we assume that:
 - Each shared variable is *owned* by a unique process; any process can read a variable but *only its owner can write into it*.
 - No process ever issues two writes (for the same variable) concurrently.
- (3) Processes are live but not infinitely fast. I.e., in an infinite amount of time each process takes infinitely many steps; while in a finite amount of time each process can only take finitely many steps. This is the only way in which the rate of computation by different processes is constrained.

Lamport devised the “bakery algorithm”, shown below, which solves the ME problem without requiring that read and write operations be mutually exclusive. Furthermore, it was the first algorithm to guarantee FCFS access to the critical section. The bakery algorithm is so called because it mimics the protocol sometimes used in bakeries (and other stores) to enforce fair service to customers: Each customer takes a number upon entry to the store and customers are serviced in the order of the numbers they have picked. Provided that each customer picks a number greater than that taken by any customer who picked a number earlier but is still awaiting service, fairness is guaranteed.

Lines 2–8 represent the trying protocol, line 10 is the exit protocol, and lines 2–4 represent the “doorway” portion of the trying protocol. The “bakery” consists of lines 4–9; note that it is defined to include the critical section. The idea is this: in the doorway (of the bakery), a process picks a number higher than that of any other process. The rest of the protocol is essentially checking to see if the process’ turn has come up — i.e., if its own number is less than that of others waiting to enter the critical section. To break ties between processes that enter the doorway simultaneously and choose the same number, we actually compare the pair consisting of the chosen number and

communication variables:

$num[1..N]$: array of integer, initially all 0

$dw[1..N]$: array of Boolean, initially all false

Process i owns num_i and dw_i .

program for process i **local var:**

j : $1..N$

repeat

1. NCS % noncritical section %
2. $dw_i := \text{true}$
3. $num_i := 1 + \max\{num_j : 1 \leq j \leq N\}$
4. $dw_i := \text{false}$
5. **for** $j := 1$ **to** N **do begin**
6. **while** dw_j **do skip**
7. **while** $num_j \neq 0$ **and** $\langle num_j, j \rangle < \langle num_i, i \rangle$ **do skip**
8. **end**
9. CS % critical section %
10. $num_i := 0$

forever

the process' own number, so that in such a situation the lower numbered process gains access to the critical section first.

2. Lamport's theory of nonatomic operations

To establish various properties of algorithms in a model of computation where read and write operations are nonatomic, we need a formal theory. This is especially important because, as experience shows, informal arguments about computations involving nonatomic operations are extremely error-prone. Lamport proposed such a theory that has proved very successful. Here we shall present a simplified version of his theory. The simplification is due to the fact that we adopt the so-called "global time model", in which we think of the operations of all processes as taking place in a single time frame (a Newtonian conception of time). In Lamport's original formulation of the theory, each process has its own time frame, in which its operations are taking place, and the different time frames are only related through communication events (an Einsteinian conception of time).[†]

Formally speaking an *execution* is a tuple $\Gamma = (O_1, \dots, O_N, \longrightarrow, - - \triangleright)$, where

- O_i is a countable set of *elementary operations* of process i .
- \longrightarrow and $- - \triangleright$ are binary relations on $\bigcup_{1 \leq i \leq N} O_i$. These relations satisfy certain axioms listed below.

We assume that among the elementary operations we have $\text{Read}(x)$ and $\text{Write}(x)$ operations for each communication (shared) variable x . Furthermore, for each x all $\text{Write}(x)$ operations belong to the same process, the owner of x : if W, W' are $\text{Write}(x)$ operations and $W \in O_i, W' \in O_j$ then $i = j$.

[†] The effects of this choice on the theory are reflected in the axioms A1–A9 described below. More specifically, in Lamport's original formulation, Axiom A3 is weaker (it is missing the "if" direction), while Axiom A9 is stronger than the version described here.

Informally, \longrightarrow and $- - \triangleright$ indicate the temporal ordering of elementary operations. The intention is that $a \longrightarrow b$ iff a precedes b (i.e., a ends before b begins); and that $a - - \triangleright b$ iff a can causally affect b (i.e., a precedes or is concurrent with b or, said differently, a starts no later than b ends). Formally, the properties of these relations are expressed by the following axioms (for any elementary operations a, b, c, d):

General Axioms:

- A1 \longrightarrow is an irreflexive partial order.
- A2 If $a \longrightarrow b$ then $a - - \triangleright b$.
- A3 $a \longrightarrow b$ iff $b - - \not\triangleright a$.
- A4 If $a \longrightarrow b - - \triangleright c$ or $a - - \triangleright b \longrightarrow c$ then $a - - \triangleright c$.
- A5 If $a \longrightarrow b - - \triangleright c \longrightarrow d$ then $a \longrightarrow d$.
- A6 $\{e : e - - \triangleright a\}$ is finite
- A7 If a, b belong to the same process then $a \longrightarrow b$ or $b \longrightarrow a$.

Communication Axioms:

In addition, there are two axioms that pertain specifically to read and write operations (assumed to be elementary operations). A8 states that for each communication (shared) variable there is a write operation that produces the variable's initial value. A9 asserts that a $\text{Read}(x)$ operation that is not concurrent with any $\text{Write}(x)$ operation must return the value last written into x . More precisely, for each communication variable x :

- A8 There is a $\text{Write}(x)$ operation W such that for any $\text{Read}(x)$ operation R , $W \longrightarrow R$.
- A9 Let R be a $\text{Read}(x)$ operation such that for any $\text{Write}(x)$ operation W , either $W \longrightarrow R$ or $R \longrightarrow W$. Let W^* be the \longrightarrow -maximum element of the set

$$X = \{W : W \text{ is a } \text{Write}(x) \text{ operation and } W \longrightarrow R\}.$$

Then R returns the value written into x by W^* .[†]

Fix an execution $\Gamma = (O_1, \dots, O_N, \longrightarrow, - - \triangleright)$. An *operation* of process i is a nonempty $A \subseteq O_i$ with the following property: If $a, b \in A$ and there is some $c \in O_i$ such that $a \longrightarrow c \longrightarrow b$ then $c \in A$. A is a *terminating* operation if it is finite; otherwise, it is *nonterminating*.

For convenience, and by slight abuse of notation, we shall not distinguish an elementary operation a from the operation $\{a\}$. Hence, any elementary operation is a terminating operation. We now extend the relations \longrightarrow and $- - \triangleright$ to arbitrary operations (not just elementary ones) A, B as follows:

$$\begin{aligned} A \longrightarrow B & \text{ iff } \forall a \in A, b \in B \ a \longrightarrow b \\ A - - \triangleright B & \text{ iff } \exists a \in A, b \in B \ a - - \triangleright b. \end{aligned}$$

With these definitions, it is straightforward to check that A1–A5 hold for arbitrary operations, not just elementary ones. In the sequel we shall refer to these axioms even when speaking of (nonelementary) operations. (A6 and A7 do *not* hold for arbitrary operations; a weaker form of A6 does — cf. Lemma 4 below.)

We now state some simple but useful results. Two $\text{Write}(x)$ operations W_1 and W_2 are called *successive* if $W_1 \longrightarrow W_2$ and there is no $\text{Write}(x)$ operation W so that $W_1 \longrightarrow W \longrightarrow W_2$.

[†] Note that the W^* of A9 is well-defined: By A6 and A2, X is finite. By A8, it is nonempty. By the fact that all $\text{Write}(x)$ operations belong to the process that owns x , A7 and A1, \longrightarrow is a total order for X . Any nonempty, totally ordered, finite set has a maximum element.

Lemma 1. *If W_1, W_2 are successive $\text{Write}(x)$ operations and R is a $\text{Read}(x)$ operation such that $W_1 \longrightarrow R \longrightarrow W_2$ then R returns the value written by W_1 .* \square

Lemma 2. *If W is the last $\text{Write}(x)$ operation and R is a $\text{Read}(x)$ operation such that $W \longrightarrow R$ then R returns the value written by W .* \square

Lemma 3. *If an execution has finitely many $\text{Write}(x)$ operations and infinitely many $\text{Read}(x)$ operations then the set of $\text{Read}(x)$ operations that return a value other than that written by the last $\text{Write}(x)$ is finite.* \square

Lemma 4. *For any operation A (not necessarily terminating) the set $\{B : B \longrightarrow A\}$ is finite.* \square

3. Correctness of the bakery algorithm

Notation: We need some notation that will allow us to identify the operations that correspond to the execution of particular pieces of code of a given algorithm — such as the bakery algorithm.

$i.l$ denotes the operation of executing the statement in line l of process i 's algorithm.

$i.l[R(x, v)]$ denotes the operation of reading variable x in line l of process i 's algorithm, where the value returned by the read operations is v .

$i.l[W(x, v)]$ denotes the operation of writing value v into variable x in line l of process i 's algorithm.

This is not completely unambiguous, since a line may be executed several times. This is certainly the case in the bakery algorithm, which consists of a repeat-forever loop. We could extend this notation (e.g. by including the iteration number through the loop) to resolve the ambiguity. To keep the notation reasonable, we shall not do so. Rather we will tacitly assume that each operation identified as above occurs at the “current” iteration of the repeat-forever loop.

Lemma 5. *Suppose that some process i is in the CS while some process $k \neq i$ is in the bakery; more precisely, $k.4 - - \succ i.9 - - \succ k.9$. For $p \in \{i, k\}$, let v_p be the value that p writes into num_p (on line 3). Then $\langle v_i, i \rangle < \langle v_k, k \rangle$.*

PROOF: Since i is in CS, it has completed $i.6$. Thus, it has executed $i.6[R(dw_k, false)]$, and so it cannot be that $k.2 \longrightarrow i.6[R(dw_k, false)] \longrightarrow k.4$ (for, if that were the case, by Lemma 1, i would have found $dw_k = true$ in line 6 — i.e., the operation would have been $i.6[R(dw_k, true)]$, not $i.6[R(dw_k, false)]$). Hence, by A3, either $i.6 - - \succ k.2$ or $k.4 - - \succ i.6$.

Case 1: Suppose $k.4 - - \succ i.6[R(dw_k, false)]$. By A7 and A5, we have

$$k.3[W(num_k, v_k)] \longrightarrow i.7[R(num_k, v)]$$

for some v . But we also have

$$i.7[R(num_k, v)] \longrightarrow i.9 - - \succ k.9 \longrightarrow k.10.$$

So by A5, we get $i.7 \longrightarrow k.10$ and therefore

$$k.3[W(num_k, v_k)] \longrightarrow i.7[R(num_k, v)] \longrightarrow k.10[W(num_k, 0)].$$

By Lemma 1, it must be that $v = v_k$. That is, i read the value written into num_k by k in $k.3$.

Note that $v_k > 0$ (because k always writes a positive number into num_k). Since i is in CS, it must have executed $i.7[R(num_k, v)]$ for some v s.t. $v = 0$ or $\langle v_i, i \rangle < \langle v, k \rangle$. But, as we just showed, $v = v_k \neq 0$, so it must be that $\langle v_i, i \rangle < \langle v, k \rangle = \langle v_k, k \rangle$, as wanted.

Case 2: Suppose $i.6[R(dw_k, false)] - - \triangleright k.2$. By A7 and A5, we have

$$i.3[W(num_i, v_i)] \longrightarrow k.3[R(num_i, v)]$$

for some v . Also,

$$k.3[R(num_i, v)] \longrightarrow k.4 - - \triangleright i.9 \longrightarrow i.10[W(num_i, 0)].$$

So by A5, we get $k.3[R(num_i, v)] \longrightarrow i.10[W(num_i, 0)]$. So we have

$$i.3[W(num_i, v_i)] \longrightarrow k.3[R(num_i, v)] \longrightarrow i.10[W(num_i, 0)].$$

By Lemma 1 then, $v = v_i$. So, k read v_i in $k.3$; therefore $v_k > v_i$, and $\langle v_i, i \rangle < \langle v_k, k \rangle$, as wanted. \square

Theorem 1. *The bakery algorithm satisfies mutual exclusion (ME), deadlock freedom (DF) and first-come-first-served (FCFS).*

PROOF: (As in Lemma 5, v_p denotes the value written by p into num_p in line 3.)

(ME) ME follows directly from Lemma 5; for if two distinct processes i and k are both in CS, then $\langle v_i, i \rangle < \langle v_k, k \rangle$ and $\langle v_k, k \rangle < \langle v_i, i \rangle$, which is a contradiction.

(FCFS) Suppose, by way of contradiction, that k finishes the doorway before i starts the doorway but i enters the CS before k does; i.e., $k.4 \longrightarrow i.2$ but $i.9 \longrightarrow k.9$.

We have that for some value v (the value of num_k that i reads on line 3):

$$k.3[W(num_k, v_k)] \longrightarrow k.4 \longrightarrow i.2 \longrightarrow i.3[R(num_k, v)] \longrightarrow i.3[Write(num_i, v_i)] \longrightarrow i.9$$

Furthermore,

$$i.9 \longrightarrow k.9 \longrightarrow k.10 : W(num_k, 0)$$

By A1 and A5,

$$k.3[W(num_k, v_k)] \longrightarrow i.3[R(num_k, v)] \longrightarrow k.10[W(num_k, 0)]$$

and therefore, by Lemma 1, $v = v_k$. Since the value v_i that i writes into num_i on line 3 is larger than any of the values it reads on line 3, it follows that $v_i > v$ and therefore $v_i > v_k$. By our hypotheses we have $k.4 \longrightarrow i.2 \longrightarrow i.9 \longrightarrow k.9$, and by A1 and A2 this implies $k.4 - - \triangleright i.9 - - \triangleright k.9$. By Lemma 5 this, in turn, implies $\langle v_i, i \rangle < \langle v_k, k \rangle$, which contradicts that $v_i > v_k$.

(DF): Suppose, by way of contradiction, that there is a deadlock. Thus, eventually there is a set T of processes executing nonterminating trying protocols, while the remaining processes \bar{T} are executing nonterminating NCSs.

We claim that for each process i , there is (only) a finite number of $Write(num_i)$ and $Write(dw_i)$ operations. To see this, first consider a process $i \in \bar{T}$. Since there are no write operations to num_i or dw_i in NCS, we have (by A7) that $W \longrightarrow NCS$, where W is any write operation of i into num_i or dw_i and NCS is the nonterminating NCS operation of i . By Lemma 4 the set $\{W : W \longrightarrow NCS\}$ is finite. Regarding process $i \in T$ the claim follows similarly if we observe that there are no write

operations for num_i and dw_i in lines $i.6$ and $i.7$, which are the only potentially nonterminating operations inside the trying protocol.

Next we claim that no $i \in T$ can be executing a nonterminating $i.6$. For, as was just argued, for all j there are (only) finitely many write operations for dw_j and the last such operation sets $dw_j := false$. Therefore, by Lemma 3, there can't be infinitely many $Read(dw_j)$ operations that return true — which would have to be the case if some $i \in T$ were executing a nonterminating line 6. Therefore, every $i \in T$ must be executing a nonterminating $i.7$.

Now pick $i \in T$ so that for all $k \in T \setminus \{i\}$, $\langle v_i, i \rangle < \langle v_k, k \rangle$. As just argued i must be executing a nonterminating $i.7$. Thus, for some j , there are infinitely many $i.7[R(num_j, v)]$ operations where $v \neq 0$ (*) and $\langle v, j \rangle < \langle v_i, i \rangle$ (**). As argued before there is (only) a finite number of assignments to num_j .

Case 1: $j \notin T$. Then the last assignment to num_j is in line $j.10$ and sets $num_j := 0$. Therefore, by Lemma 3, there can't be infinitely many $i.7[R(num_j, v)]$ operations with $v \neq 0$, contradicting (*).

Case 2: $j \in T$. Then the last assignment to num_j is in line $j.3$ and sets $num_j := v_j$. Therefore, by Lemma 3, we must have $v = v_j$. But by choice of i (and the fact that $j \in T$), $\langle v_i, i \rangle \leq \langle v_j, j \rangle = \langle v, j \rangle$, contradicting (**).

Therefore there can't be a deadlock. □

Note: Since the bakery algorithm satisfies DF and FCFS it also satisfies LF.

A problem with the bakery algorithm is that the shared variables num_i are of unbounded size. Even if we assume that a read must return a value that was previously written or is being written concurrently with the read, in an execution in which the bakery is never empty, the value of num_i chosen by each process i will keep increasing forever.

A number of mutual exclusion algorithms that work in the model of nonatomic read and write operations have been developed. The goal is to achieve the maximum possible degree of fairness (FCFS, being the ideal) with as few shared bits per process as possible. The bakery algorithm achieves the first goal but requires shared variables that can hold unboundedly large integers. Among these algorithms are:

- An algorithm independently discovered by Burns and Lamport, which uses 1 shared bit per process and achieves ME and DF, but not LF.
- Algorithms by Lamport and Peterson that use 3 and 2 shared bits per process, respectively, and achieve ME and LF (and therefore DF).
- An algorithm by Katseff that requires N shared bits per process and satisfies ME, DF and FCFS (and therefore LF as well).
- An algorithm by Lycklama and Hadzilacos that requires 5 shared bits per process and satisfies ME, DF and FCFS. (A simple modification results in a 4-bit per process algorithm.)

4. The Burns-Lamport algorithm

communication variables:
 $x[1..N]$: array of Boolean, initially all false
 x_i is owned by process i

program for process i

local var:
 j : $1..N$

repeat

1. NCS % noncritical section %
2. L: $x_i := \text{true}$
3. **for** $j := 1$ **to** $i - 1$ **do**
4. **if** x_j **then begin**
5. $x_i := \text{false}$
6. **while** x_j **do skip**
7. **goto** L
8. **end**
9. **for** $j := i + 1$ **to** N **do**
10. **while** x_j **do skip**
11. CS % critical section %
12. $x_i := \text{false}$

forever

This algorithm uses just one bit of shared variable per process. The basic idea is this: Each process i sets its bit x_i to indicate its interest in entering the critical section and tests the bits of lower-numbered processes. If i finds any of them interested in the critical section, it gives up and restarts. If, on the other hand, none of the lower-numbered processes has its bit set, i proceeds to the next phase where it tests the bits of higher-numbered processes, waiting for each one of them that it finds interested in entering the critical section — but now merely waiting, without giving up and starting over. This asymmetry in the treatment of lower- and higher-numbered processes makes the algorithm deadlock-free: lower-numbered processes get priority. The same feature makes the algorithm subject to lockout: a “fast” process with small number can enter the CS arbitrarily many times while higher-numbered processes repeatedly give deference to it.

Notation: To distinguish the execution of operations in different iterations through lines 2–7 we must extend our notation for identifying operations. If a is an operation that occurs in these lines then a^* denotes the last execution of that operation before the executing process enters its CS.

Theorem 2. *The Burns-Lamport algorithm satisfies ME and DF.*

PROOF: (ME) Suppose, by way of contradiction, that i and k are both in their CS. Formally speaking, $i.11 \not\rightarrow k.11$ and $k.11 \not\rightarrow i.11$. Without loss of generality, assume $i < k$.

By A3, $k.11 - - \triangleright i.11$ and $i.11 - - \triangleright k.11$. Since i has entered the CS, it must have executed $i.10[R(x_k, \text{false})]$. Then we claim that

$$i.10[R(x_k, \text{false})] - - \triangleright k.2[W(x_k, \text{true})]^* \tag{1}$$

For, otherwise, by A3 and the fact that $i.11 - - \triangleright k.11$ we would have

$$k.2[W(x_k, \text{true})]^* \longrightarrow i.10[R(x_k, \text{false})] \longrightarrow i.11 - - \triangleright k.11 \longrightarrow k.12[W(x_k, \text{false})]$$

and by A5, we would get

$$k.2[W(x_k, true)]^* \longrightarrow i.10[R(x_k, false)] \longrightarrow k.12[W(x_k, false)]$$

which contradicts Lemma 1.

By a similar argument

$$k.4[R(x_i, false)]^* - - \succ i.2[W(x_i, true)]^* \tag{2}$$

From (1), A7 and A5 we get $i.2^* \longrightarrow k.4^*$ and therefore, by A3, $k.4^* - \not\geq i.2^*$, contradicting (2). Thus ME is satisfied.

(DF) Suppose, by way of contradiction, that there is a deadlock. Let T be a set of processes executing a nonterminating trying protocol.

Claim 1. *For all $k \notin T$ the set of $Write(x_k)$ operations is finite and the last such operation sets $x_k := false$.*

PROOF: Since there is a deadlock, if $k \notin T$ then k must eventually execute a nonterminating NCS, say NCS_k . Thus for every $Write(x_k)$ operation W we must have $W \longrightarrow NCS_k$ (because there are no $Write(x_k)$ operations in NCS) and by Lemma 4, the set of $Write(x_k)$ operations is finite. The last $Write(x_k)$ operation before k enters NCS_k is $k.12$, which sets $x_k := false$. \square

Let $i = \min(T)$.

Claim 2. *i executes a nonterminating $i.10$.*

PROOF: Since $i \in T$, i executes a nonterminating $i.10$ or a nonterminating $i.2 - i.7$. By way of contradiction suppose the latter. So for some $j < i$, i executes an infinite set of $i.4[R(x_j, true)]$ or $i.6[R(x_j, true)]$ operations. By choice of i , $j \notin T$ and, therefore, in view of Claim 1, we get a contradiction to Lemma 3. \square

Corollary. *There is a finite number of $Write(x_i)$ operations and the last one sets $x_i := true$.* \square

Claim 3. *For any $k \in T$ that executes a nonterminating $k.2 - k.7$, the number of $Write(x_k)$ operations is finite and the last such operation sets $x_k := false$.*

PROOF: By Claim 2, $k \neq i$, so $k > i$. We claim that k must execute a nonterminating $k.6$. If not, k must execute infinitely many $k.6[R(x_i, false)]$ (because, by assumption, it is executing a nonterminating $k.2 - k.7$ operation) which, in view of the Corollary, contradicts Lemma 3. The rest of the proof is similar to the proof of Claim 1, with the nonterminating $k.6$ playing the role of the nonterminating CS_k . \square

Claim 4. *No $k \in T$ can be executing a nonterminating $k.10$.*

PROOF: Suppose the contrary and let m be the maximum process executing a nonterminating $k.10$. So m executes infinitely many $m.10[R(x_j, true)]$ for some $j > m$. But for all $j > m$, either $j \notin T$ or j is executing a nonterminating $j.2 - j.7$. By Claim 1 or Claim 3, respectively, we get a contradiction to Lemma 3. \square

But now Claim 4 contradicts Claim 2. \square

5. Types of registers

The seminal paper for the material very briefly sketched here is Lamport, “On inter-process communication — Parts I and II”, *Distributed Computing* 1(1986):77–101.

Informally, a register is a shared variable that can hold a number of values and can be accessed by a fixed set of readers and a fixed set of writers.

A register is characterised by:

1. The number of values it can hold.
2. The number of readers.
3. The number of writers.
4. The “synchronisation strength” of the register: safe, regular or atomic (these are explained below).

Read and write operations of a register may overlap (they are not atomic). A write operation W is *most recent* with respect to (wrt) a read operation R if $W \longrightarrow R$ and there is no W' such that $W \longrightarrow W' \longrightarrow R$. If each process (reader or writer) that uses the register issues its operations one-at-a-time (nonoverlapping) and there is one writer for each register (the register’s “owner”), then there is at most one write operation which is most recent wrt a given read operation. (This is the model we assumed in our discussion of mutual exclusion algorithms that work with nonatomic read and write operations.) However, if concurrent write operations are allowed then there could be several (concurrent) writes, each of which is “most recent” wrt a given read operation.

By *synchronisation strength* of a register we mean the strength of the guarantees that the register can make about the values it returns to read operations that occur concurrently with writes. There are three degrees of synchronisation strength, each making stronger guarantees than the previous:

- **Safe:** A read operation R which is not concurrent with any write operation, is guaranteed to return the value written by a write operation W which is most recent wrt R .
- **Regular:** A read operation R is guaranteed to return a value which is written by one of the writes it is concurrent with, or a most recent write wrt it.

Note that a regular register can exhibit the following sort of behaviour: Two read operations R_1 and R_2 with $R_1 \longrightarrow R_2$ return, respectively, the values written by write operations W_1 and W_2 , but $W_2 \longrightarrow W_1$. In particular, this can happen, if R_1 is concurrent with both W_2 and W_1 , and returns the value written by W_1 ; while R_2 is concurrent with only W_1 , and returns the value written by the most recent write wrt it, namely W_2 . In other words, it is possible for a more recent read operation to return an older value than a previous read. This sort of “anomalous” behaviour (which can be exhibited by regular registers) is called a “new-old inversion”.

- **Atomic:** A regular register where no new-old inversions can occur.

The goal is to devise algorithms that construct stronger types of registers using weaker ones. Ultimately, we want to construct multireader, multiwriter, multivalued, atomic registers, starting with single-reader, single-writer, 2-valued (Boolean), safe registers. This can be achieved easily by using mutual exclusion algorithms: We think of the strong register (which we want to construct) as a critical section. To read or write the register, a reader or writer respectively must enter the critical section. The algorithms we have discussed previously solve mutual exclusion with multireader, single-writer, Boolean, safe registers. As we shall see shortly, it is very easy to go from single-reader to multireader safe registers, so we can accomplish the desired construction by using mutual exclusion algorithms.

An important disadvantage of this construction is that it is not fault-tolerant. If a process fails (stops) while it is accessing the constructed register, it will prevent all other processes from ever accessing the register. This is because to access the register a process must be in the critical section. To achieve more robust solutions, we require that the algorithms used in the construction be *strongly wait-free*. That is, the reading and writing of the constructed register should be *bounded* operations. (The solution using mutual exclusion is not, because the trying protocol in a mutual exclusion algorithm cannot be bounded.) Lamport, in his seminal paper, gave five constructions which partially accomplish the goal stated previously. Starting with single-reader, single-writer, Boolean, safe registers he was able to construct single-reader, single-writer, multivalued, atomic registers. All constructions involve single-writer registers.

Construction 1. *From safe (resp. regular), 1-reader, m -valued registers, construct a safe (resp. regular), multireader, m -valued register.*

In this construction the writer maintains a separate copy of the register for each reader. To write a value, the writer writes into all copies of the register; to read the register, the reader reads its own copy. Note that this construction does *not* produce an atomic multireader register if we start with atomic single-reader registers: Consider read operations R_i and R_j of reader, i and j respectively such that R_i precedes R_j but both overlap the same write operation. It is possible that R_i returns the new value from i 's (atomic) register while R_j returns the old value from j 's (atomic) register, resulting a new-old inversion in the (simulated) multireader register.

Construction 2. *From safe, m -reader, 2-valued registers, construct a safe, m -reader, multivalued register.*

This construction implements an n -bit safe register from n 1-bit safe registers. The writer simply treats the value it wishes to write as a binary number and writes it into the n 1-bit registers. A reader reads the n bits and interprets them as a binary number. The constructed register is safe since, by definition, a read does not overlap a write if one operation has accessed all n bits before the other has accessed any. Note that the constructed register is not regular, even if the 1-bit registers we are starting with are regular.

Construction 3. *From safe, m -reader, 2-valued registers, construct a regular, m -reader, 2-valued register.*

In a safe register, a read that overlaps a write may return any value, while in a regular register it must return either the new or the old value. Notice that a read of a safe 2-valued register must return either 0 or 1; thus it will necessarily return either the new or the old value of the register if it overlaps a read operation that changes the register's value. The only way a safe 2-valued register may exhibit nonregular behaviour is if a read overlaps a write that does not change the value of the register (i.e. re-writes the old value). Thus to make a safe, 2-valued register into a regular, 2-valued register we change the writer's algorithm so that the writer does nothing if the value it is about to write is the same as the last value it has written into the register. (Since there is just one writer, this can be accomplished very easily by having the writer remember the last value it wrote, in a variable local to it.)

Construction 4. *From regular, m -reader, 2-valued registers, construct a regular, m -reader, multivalued register.*

This construction employs unary encoding of the value being written. Thus we use n regular 2-valued registers to construct a regular n -valued register. Value k ($0 \leq k < n$) is represented by the

following bit pattern: 0's in bits 0 through $k - 1$ and 1 in bit k (the rest of the bits can have any values; thus k is represented by 2^{n-k+1} different bit patterns). To write value k , the writer sets bit k to 1 and then sets bits $k - 1, k - 2, \dots, 0$ (in that order) to 0. To read the register, a reader reads bits $0, 1, \dots, k$ (in that order) until it encounters the first bit set to 1 and returns the position of that bit. Note that the reader and writer scan the bits in opposite directions.

Construction 5. *From a regular, 1-reader, multivalued register and a regular, 1-reader, 2-valued register, construct an atomic, 1-reader, multivalued register.*

This construction is pretty complicated and involves two-way communication between the reader and writer. The multivalued register is written by the writer and read by the reader, while the 2-valued register is written by the reader and read by the writer. The details of the reader's and writer's algorithms are fairly convoluted and the correctness of the construction highly nonobvious. It is interesting that the need for two-way communication is not fortuitous, as the following fact shows:

Theorem 3. *It is impossible to construct an atomic register out of finitely many regular registers if the writer can only write to them and the reader can only read them (i.e., such a construction requires two-way communication).*

Subsequent to Lamport's work, other researchers pursued the program of register constructions. By now there are several ways of composing different constructions to obtain multireader, multiwriter, multivalued atomic registers starting with single-reader, single-writer, 2-valued safe registers. Some of the steps in these constructions are quite intricate.