

## Chapter 2

# Solving Mutual Exclusion

This chapter is on the implementation of mutual exclusion locks. As announced at the end of the previous chapter, it presents three distinct families of algorithms that solve the mutual exclusion problem. The first is the family of algorithms which are based on atomic read/write registers only. The second is the family of algorithms which are based on specialized hardware operations (which are atomic and stronger than atomic read/write operations). The third is the family of algorithms which are based on read/write registers which are weaker than atomic registers. Each algorithm is first explained and then proved correct. Other properties such as time complexity and space complexity of mutual exclusion algorithms are also discussed.

**Keywords** Atomic read/write register · Lock object · Mutual exclusion · Safe read/write register · Specialized hardware primitive (test&set, fetch&add, compare&swap)

## 2.1 Mutex Based on Atomic Read/Write Registers

### 2.1.1 Atomic Register

The *read/write register* object is one of the most basic objects encountered in computer science. When such an object is accessed only by a single process it is said to be *local* to that process; otherwise, it is a *shared* register. A local register allows a process to store and retrieve data. A shared register allows concurrent processes to also exchange data.

**Definition** A *register*  $R$  can be accessed by two base operations:  $R.read()$ , which returns the value of  $R$  (also denoted  $x \leftarrow R$  where  $x$  is a local variable of the invoking process), and  $R.write(v)$ , which writes a new value into  $R$  (also denoted  $R \leftarrow v$ , where  $v$  is the value to be written into  $R$ ). An *atomic* shared register satisfies the following properties:

- Each invocation  $op$  of a read or write operation:
  - Appears as if it was executed at a single point  $\tau(op)$  of the time line,
  - $\tau(op)$  is such that  $\tau_b(op) \leq \tau(op) \leq \tau_e(op)$ , where  $\tau_b(op)$  and  $\tau_e(op)$  denote the time at which the operation  $op$  started and finished, respectively,
  - For any two operation invocations  $op1$  and  $op2$ :  $(op1 \neq op2) \Rightarrow (\tau(op1) \neq \tau(op2))$ .
- Each read invocation returns the value written by the closest preceding write invocation in the sequence defined by the  $\tau()$  instants associated with the operation invocations (or the initial value of the register if there is no preceding write operation).

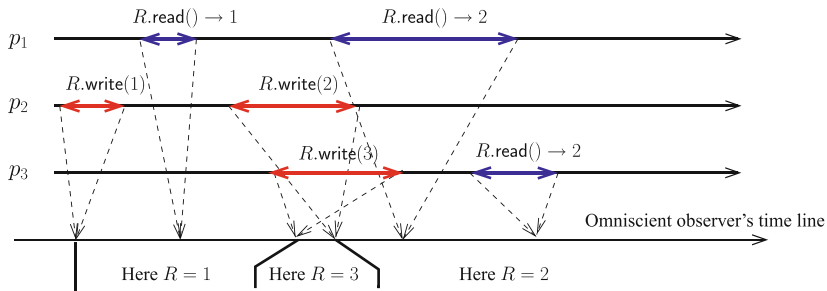
This means that an atomic register is such that all its operation invocations *appear* as if they have been executed sequentially: any invocation  $op1$  that has terminated before an invocation  $op2$  starts appears before  $op2$  in that sequence, and this sequence belongs to the specification of a sequential register.

An atomic register can be single-writer/single-reader (SWSR)—the reader and the writer being distinct processes—or single-writer/multi-reader (SWMR), or multi-writer/multi-reader (MWMR). We assume that a register is able to contain any value. (As each process is sequential, a local register can be seen as a trivial instance of an atomic SWSR register where, additionally, both the writer and the reader are the same process.)

**An example** An execution of a MWMR atomic register accessed by three processes  $p_1$ ,  $p_2$ , and  $p_3$  is depicted in Fig. 2.1 using a classical space-time diagram.  $R.read() \rightarrow v$  means that the corresponding read operation returns the value  $v$ . Consequently, an external observer sees the following sequential execution of the register  $R$  which satisfies the definition of an atomic register:

$R.write(1)$ ,  $R.read() \rightarrow 1$ ,  $R.write(3)$ ,  $R.write(2)$ ,  $R.read() \rightarrow 2$ ,  $R.read() \rightarrow 2$ .

Let us observe that  $R.write(3)$  and  $R.write(2)$  are concurrent, which means that they could appear to an external observer as if  $R.write(2)$  was executed before



**Fig. 2.1** An atomic register execution

$R.write(3)$ . If this was the case, the execution would be correct if the last two read invocations (issued by  $p_1$  and  $p_3$ ) return the value 3; i.e., the external observer should then see the following sequential execution:

$R.write(1), R.read() \rightarrow 1, R.write(2), R.write(3), R.read() \rightarrow 3, R.read() \rightarrow 3.$

Let us also observe that the second read invocation by  $p_1$  is concurrent with both  $R.write(2)$  and  $R.write(3)$ . This means that it could appear as having been executed before these two write operations or even between them. If it appears as having been executed before these two write operations, it should return the value 1 in order for the register behavior be atomic.

As shown by these possible scenarios (and as noticed before) *concurrency* is intimately related to *non-determinism*. It is not possible to predict which execution will be produced; it is only possible to enumerate the set of possible executions that could be produced (we can only predict that the one that is actually produced is one of them).

Examples of non-atomic read and write operations will be presented in Sect. 2.3.

**Why atomicity is important** Atomicity is a fundamental concept because it allows the composition of shared objects for free (i.e., their composition is at no additional cost). This means that, when considering two (or more) atomic registers  $R1$  and  $R2$ , the composite object  $[R1, R2]$  which is made up of  $R1$  and  $R2$  and provides the processes with the four operations  $R1.read()$ ,  $R1.write()$ ,  $R2.read()$ , and  $R2.write()$  is also atomic. Everything appears as if at most one operation at a time was executed, and the sub-sequence including only the operations on  $R1$  is a correct behavior of  $R1$ , and similarly for  $R2$ .

This is very important when one has to reason about a multiprocess program whose processes access atomic registers. More precisely, we can keep *reasoning sequentially* whatever the number of atomic registers involved in a concurrent computation. Atomicity allows us to reason on a set of atomic registers as if they were a single “bigger” atomic object. Hence, we can reason in terms of sequences, not only for each atomic register taken separately, but also on the whole set of registers as if they were a single atomic object.

The composition of atomic objects is formally addressed in Sect. 4.4, where it is shown that, as atomicity is a “local property”, atomic objects compose for free.

### 2.1.2 Mutex for Two Processes: An Incremental Construction

The mutex algorithm for two processes that is presented below is due to G.L. Peterson (1981). This construction, which is fairly simple, is built from an “addition” of two base components. Despite the fact that these components are nearly trivial, they allow us to introduce simple basic principles.

```

operation acquire_mutex1(i) is
    AFTER_YOU ← i; wait (AFTER_YOU ≠ i); return()
end operation.

operation release_mutex1(i) is return() end operation.

```

**Fig. 2.2** Peterson’s algorithm for two processes: first component (code for  $p_i$ )

The processes are denoted  $p_i$  and  $p_j$ . As the algorithm for  $p_j$  is the same as the one for  $p_i$  after having replaced  $i$  by  $j$ , we give only the code for  $p_i$ .

**First component** This component is described in Fig. 2.2 for process  $p_i$ . It is based on a single atomic register denoted *AFTER\_YOU*, the initial value of which is irrelevant (a process writes into this register before reading it). The principle that underlies this algorithm is a “politeness” rule used in current life. When  $p_i$  wants to acquire the critical section, it sets *AFTER\_YOU* to its identity  $i$  and waits until *AFTER\_YOU* ≠  $i$  in order to enter the critical section. Releasing the critical section entails no particular action.

It is easy to see that this algorithm satisfies the mutual exclusion property. When both processes want to acquire the critical section, each assigns its identity to the register *AFTER\_YOU* and waits until this register contains the identity of the other process. As the register is atomic, there is a “last” process, say  $p_j$ , that updated it, and consequently only the other process  $p_i$  can proceed to the critical section.

Unfortunately, this simple algorithm is not deadlock-free. If one process alone wants to enter the critical section, it remains blocked forever in the **wait** statement. Actually, this algorithm ensures that, when both processes want to enter the critical section, the first process that updates the register *AFTER\_YOU* is the one that is allowed to enter it.

**Second component** This component is described in Fig. 2.3. It is based on a simple idea. Each process  $p_i$  manages a flag (denoted *FLAG*[ $i$ ]) the value of which is *down* or *up*. Initially, both flags are down. When a process wants to acquire the critical section, it first raises its flag to indicate that it is interested in the critical section. It is then allowed to proceed only when the flag of the other process is equal to *down*.

To release the critical section, a process  $p_i$  has only to reset *FLAG*[ $i$ ] to its initial value (namely, *down*), thereby indicating that it is no longer interested in the mutual exclusion.

```

operation acquire_mutex2(i) is
    FLAG[i] ← up; wait (FLAG[j] = down); return()
end operation.

operation release_mutex2(i) is FLAG[i] ← down; return() end operation.

```

**Fig. 2.3** Peterson’s algorithm for two processes: second component (code for  $p_i$ )

It is easy to see that, if a single process  $p_i$  wants to repeatedly acquire the critical section while the other process is not interested in the critical section, it can do so (hence this algorithm does not suffer the drawback of the previous one). Moreover, it is also easy to see that this algorithm satisfies the mutual exclusion property. This follows from the fact that each process follows the following pattern: first write its flag and only then read the value of the other flag. Hence, assuming that  $p_i$  has acquired (and not released) the critical section, we had  $(FLAG[i] = up) \wedge (FLAG[j] = down)$  when it was allowed to enter the critical section. It follows that, after  $p_j$  has set  $FLAG[j]$  to the value  $up$ , it reads  $up$  from  $FLAG[i]$  and is delayed until  $p_i$  resets  $FLAG[i]$  to  $down$  when it releases the critical section.

Unfortunately, this algorithm is not deadlock-free. If both processes concurrently raise first their flags and then read the other flag, each process remains blocked until the other flag is set down which will never be done.

**Remark: the notion of a livelock** In order to prevent the previous deadlock situation, one could think replacing **wait** ( $FLAG[j] = down$ ) by the following statement:

```

while ( $FLAG[j] = up$ ) do
     $FLAG[i] \leftarrow down$ ;
     $p_i$  delays itself for an arbitrary period of time;
     $FLAG[i] \leftarrow up$ 
end while.

```

This modification can reduce deadlock situations but cannot eliminate all of them. This occurs, for example when both processes execute “synchronously” (both delay themselves for the same duration and execute the same step—writing their flag and reading the other flag—at the very same time). When it occurs, this situation is sometimes called a *livelock*.

This tentative solution was obtained by playing with asynchrony (modifying the process speed by adding delays). As a correct algorithm has to work despite any asynchrony pattern, playing with asynchrony can eliminate bad scenarios but cannot suppress all of them.

### 2.1.3 A Two-Process Algorithm

**Principles and description** In a very interesting way, a simple “addition” of the two previous “components” provides us with a correct mutex algorithm for two processes (Peterson’s two-process algorithm). This component addition consists in a process  $p_i$  first raising its flag (to indicate that it is competing, as in Fig. 2.3), then assigning its identity to the atomic register *AFTER\_YOU* (as in Fig. 2.2), and finally waiting until any of the progress predicates  $AFTER\_YOU \neq i$  or  $FLAG[j] = down$  is satisfied.

It is easy to see that, when a single process wants to enter the critical section, the flag of the other process allows it to enter. Moreover, when each process sees that

```

operation acquire_mutex( $i$ ) is
   $FLAG[i] \leftarrow up$ ;
   $AFTER\_YOU \leftarrow i$ ;
  wait  $((FLAG[j] = down) \vee (AFTER\_YOU \neq i))$ ;
  return()
end operation.

operation release_mutex( $i$ ) is  $FLAG[i] \leftarrow down$ ; return() end operation.

```

**Fig. 2.4** Peterson's algorithm for two processes (code for  $p_i$ )

the flag of the other one was raised, the current value of the register  $AFTER\_YOU$  allows exactly one of them to progress.

It is important to observe that, in the **wait** statement of Fig. 2.4, the reading of the atomic registers  $FLAG[j]$  and  $AFTER\_YOU$  are asynchronous (they are done at different times and can be done in any order).

**Theorem 1** *The algorithm described in Fig. 2.4 satisfies mutual exclusion and bounded bypass (where the bound is  $f(n) = 1$ ).*

**Preliminary remark for the proof** The reasoning is based on the fact that the three registers  $FLAG[i]$ ,  $FLAG[j]$ , and  $AFTER\_YOU$  are atomic. As we have seen when presenting the atomicity concept (Sect. 2.1.1), this allows us to reason as if at most one read or write operation on any of these registers occurs at a time.

*Proof* Proof of the mutual exclusion property.

Let us assume by contradiction that both  $p_i$  and  $p_j$  are inside the critical section. Hence, both have executed `acquire_mutex()` and we have then  $FLAG[i] = up$ ,  $FLAG[j] = up$  and  $AFTER\_YOU = j$  (if  $AFTER\_YOU = i$ , the reasoning is the same after having exchanged  $i$  and  $j$ ). According to the predicate that allowed  $p_i$  to enter the critical section, there are two cases.

- Process  $p_i$  has terminated `acquire_mutex(i)` because  $FLAG[j] = down$ .

As  $p_i$  has set  $FLAG[i]$  to *up* before reading *down* from  $FLAG[j]$  (and entering the critical section), it follows that  $p_j$  cannot have read *down* from  $FLAG[j]$  before entering the critical section (see Fig. 2.5). Hence,  $p_j$  entered it due to the predicate  $AFTER\_YOU = i$ . But this contradicts the assumption that  $AFTER\_YOU = j$  when both processes are inside the critical section.

- Process  $p_i$  has terminated `acquire_mutex(i)` because  $AFTER\_YOU = j$ .

As (by assumption)  $p_j$  is inside the critical section,  $AFTER\_YOU = j$ , and only  $p_j$  can write  $j$  into  $AFTER\_YOU$ , it follows that  $p_j$  has terminated `acquire_mutex(j)` because it has read *down* from  $FLAG[i]$ . On another side,  $FLAG[i]$  remains continuously equal to *up* from the time at which  $p_i$  has executed the first statement of `acquire_mutex(i)` and the execution of `release_mutex(i)` (Fig. 2.6).

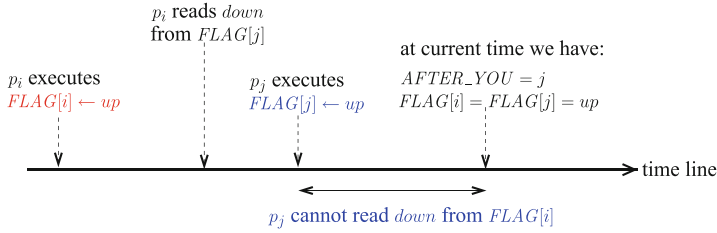


Fig. 2.5 Mutex property of Peterson's two-process algorithm (part 1)

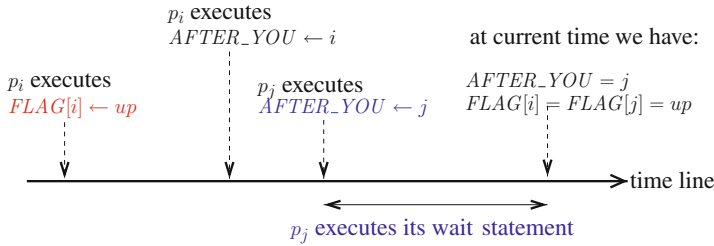


Fig. 2.6 Mutex property of Peterson's two-process algorithm (part 2)

As  $p_j$  executes the **wait** statement after writing  $j$  into  $AFTER\_YOU$  and  $p_i$  read  $j$  from  $AFTER\_YOU$ , it follows that  $p_j$  cannot read down from  $FLAG[i]$  when it executes the **wait** statement. This contradicts the assumption that  $p_j$  is inside the critical section.

Proof of the bounded bypass property.

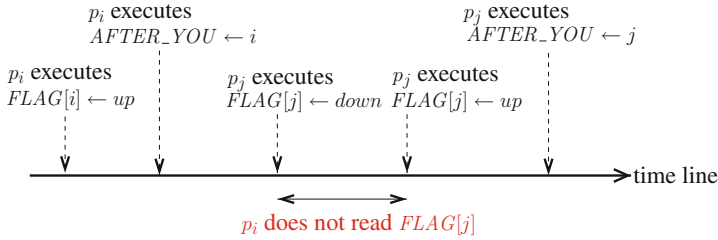
Let  $p_i$  be the process that invokes  $acquire\_mutex(i)$ . If  $FLAG[j] = down$  or  $AFTER\_YOU = j$  when  $p_i$  executes the **wait** statement, it enters the critical section.

Let us consequently assume that  $(FLAG[j] = up) \wedge (AFTER\_YOU = i)$  when  $p_i$  executes the **wait** statement (i.e., the competition is lost by  $p_i$ ). If, after  $p_j$  has executed  $release\_mutex(j)$ , it does not invoke  $acquire\_mutex(j)$  again, we permanently have  $FLAG[j] = down$  and  $p_i$  eventually enters the critical section.

Hence let us assume that  $p_j$  invokes again  $acquire\_mutex(j)$  and sets  $FLAG[j]$  to  $up$  before  $p_i$  reads it. Thus, the next read of  $FLAG[j]$  by  $p_i$  returns  $up$ . We have then  $(FLAG[j] = up) \wedge (AFTER\_YOU = i)$ , and  $p_i$  cannot progress (see Fig. 2.7).

It follows from the code of  $acquire\_mutex(j)$  that  $p_j$  eventually assigns  $j$  to  $AFTER\_YOU$  (and the predicate  $AFTER\_YOU = j$  remains true until the next invocation of  $acquire\_mutex()$  by  $p_i$ ). Hence,  $p_i$  eventually reads  $j$  from  $AFTER\_YOU$  and is allowed to enter the critical section.

It follows that a process loses at most one competition with respect to the other process, from which we conclude that the bounded bypass property is satisfied and we have  $f(n) = 1$ .  $\square$



**Fig. 2.7** Bounded bypass property of Peterson's two-process algorithm

**Space complexity** The space complexity of a mutex algorithm is measured by the number and the size of the atomic registers it uses.

It is easy to see that Peterson's two-process algorithm has a bounded space complexity: there are three atomic registers  $FLAG[i]$ ,  $FLAG[j]$ , and  $AFTER\_YOU$ , and the domain of each of them has two values. Hence three atomic bits are sufficient.

#### 2.1.4 Mutex for $n$ Processes:

##### *Generalizing the Previous Two-Process Algorithm*

**Description** Peterson's mutex algorithm for  $n$  processes is described in Fig. 2.8. This algorithm is a simple generalization of the two-process algorithm described in Fig. 2.4. This generalization, which is based on the notion of level, is as follows.

In the two-process algorithm, a process  $p_i$  uses a simple SWMR flag  $FLAG[i]$  whose value is either *down* (to indicate it is not interested in the critical section) or *up* (to indicate it is interested). Instead of this binary flag, a process  $p_i$  uses now a multi-valued flag that progresses from a flag level to the next one. This flag, denoted  $FLAG\_LEVEL[i]$ , is initialized to 0 (indicating that  $p_i$  is not interested in the critical section). It then increases first to level 1, then to level 2, etc., until the level  $n - 1$ ,

```

operation acquire_mutex( $i$ ) is
(1) for  $\ell$  from 1 to  $(n - 1)$  do
(2)    $FLAG\_LEVEL[i] \leftarrow \ell$ ;
(3)    $AFTER\_YOU[\ell] \leftarrow i$ ;
(4)   wait ( $\forall k \neq i : FLAG\_LEVEL[k] < \ell$ )  $\vee$  ( $AFTER\_YOU[\ell] \neq i$ )
(5) end for;
(6) return();
end operation.

operation release_mutex( $i$ ) is  $FLAG\_LEVEL[i] \leftarrow 0$ ; return() end operation.

```

**Fig. 2.8** Peterson's algorithm for  $n$  processes (code for  $p_i$ )



which allows it to enter the critical section. For  $1 \leq x < n-1$ ,  $FLAG\_LEVEL[i] = x$  means that  $p_i$  is trying to enter level  $x+1$ .

Moreover, to eliminate possible deadlocks at any level  $\ell$ ,  $0 < \ell < n-1$  (such as the deadlock that can occur in the algorithm of Fig. 2.3), the processes use a second array of atomic registers  $AFTER\_YOU[1..(n-1)]$  such that  $AFTER\_YOU[\ell]$  keeps track of the last process that has entered level  $\ell$ .

More precisely, a process  $p_i$  executes a **for** loop to progress from one level to the next one, starting from level 1 and finishing at level  $n-1$ . At each level the two-process solution is used to block a process (if needed). The predicate that allows a process to progress from level  $\ell$ ,  $0 < \ell < n-1$ , to level  $\ell+1$  is similar to the one of the two-process algorithm. More precisely,  $p_i$  is allowed to progress to level  $\ell+1$  if, from its point of view,

- Either all the other processes are at a lower level (i.e.,  $\forall k \neq i: FLAG\_LEVEL[k] < \ell$ ).
- Or it is not the last one that entered level  $\ell$  (i.e.,  $AFTER\_YOU[\ell] \neq i$ ).

Let us notice that the predicate used in the **wait** statement of line 4 involves all but one of the atomic registers  $FLAG\_LEVEL[\cdot]$  plus the atomic register  $AFTER\_YOU[\ell]$ . As these registers cannot be read in a single atomic step, the predicate is repeatedly evaluated asynchronously on each register.

When all processes compete for the critical section, at most  $(n-1)$  processes can concurrently be winners at level 1,  $(n-2)$  processes can concurrently be winners at level 2, and more generally  $(n-\ell)$  processes can concurrently be winners at level  $\ell$ . Hence, there is a single winner at level  $(n-1)$ .

The code of the operation  $release\_mutex(i)$  is similar to the one of the two-process algorithm: a process  $p_i$  resets  $FLAG\_LEVEL[i]$  to its initial value 0 to indicate that it is no longer interested in the critical section.

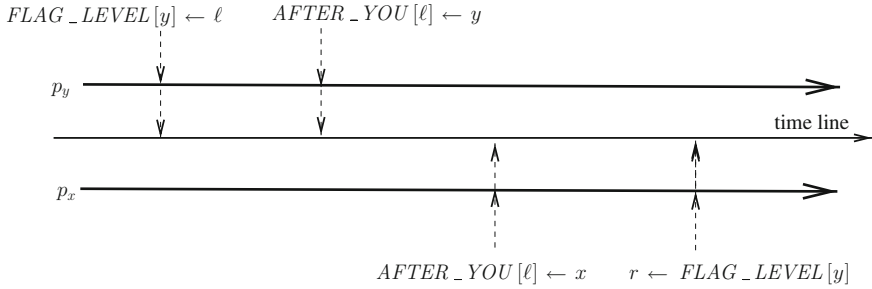
**Theorem 2** *The algorithm described in Fig. 2.8 satisfies mutual exclusion and starvation-freedom.*

*Proof* Initially, a process  $p_i$  is such that  $FLAG\_LEVEL[i] = 0$  and we say that it is at level 0. Let  $\ell \in [1..(n-1)]$ . We say that a process  $p_i$  has “attained” level  $\ell$  (or, from a global state point of view, “is” at level  $\ell$ ) if it has exited the **wait** statement of the  $\ell$ th loop iteration. Let us notice that, after it has set its loop index  $\ell$  to  $\alpha > 0$  and until it exits the **wait** statement of the corresponding iteration, that process is at level  $\alpha-1$ . Moreover, a process that attains level  $\ell$  has also attained the levels  $\ell'$  with  $0 \leq \ell' \leq \ell \leq n-1$  and consequently it is also at these levels  $\ell'$ .

The proof of the mutual exclusion property amounts to showing that at most one process is at level  $(n-1)$ . This is a consequence of the following claim when we consider  $\ell = n-1$ .

**Claim.** For  $\ell$ ,  $0 \leq \ell \leq n-1$ , at most  $n-\ell$  processes are at level  $\ell$ .

The proof of this claim is by induction on the level  $\ell$ . The base case  $\ell = 0$  is trivial. Assuming that the claim is true up to level  $\ell-1$ , i.e., at most  $n-(\ell-1)$



**Fig. 2.9** Total order on read/write operations

processes are simultaneously at level  $\ell - 1$ , we have to show that at least one process does not progress to level  $\ell$ . The proof is by contradiction: let us assume that  $n - \ell + 1$  processes are at level  $\ell$ .

Let  $p_x$  be the last process that wrote its identity into  $AFTER\_YOU[\ell]$  (hence,  $AFTER\_YOU[\ell] = x$ ). When considering the sequence of read and write operations executed by every process, and the fact that these operations are on atomic registers, this means that, for any of the  $n - \ell$  other processes  $p_y$  that are at level  $\ell$ , these operations appear as if they have been executed in the following order where the first two operations are issued by  $p_y$  while the last two operations are issued by  $p_x$  (Fig. 2.9):

1.  $FLAG\_LEVEL[y] \leftarrow \ell$  is executed before  $AFTER\_YOU[\ell] \leftarrow y$  (sequentiality of  $p_y$ )
2.  $AFTER\_YOU[\ell] \leftarrow y$  is executed before  $AFTER\_YOU[\ell] \leftarrow x$  (assumption: definition of  $p_x$ )
3.  $AFTER\_YOU[\ell] \leftarrow x$  is executed before  $r \leftarrow FLAG\_LEVEL[y]$  (sequentiality of  $p_x$ ;  $r$  is  $p_x$ 's local variable storing the last value read from  $FLAG\_LEVEL[y]$  before  $p_x$  exits the **wait** statement at level  $\ell$ ).

It follows from this sequence that  $r = \ell$ . Consequently, as  $AFTER\_YOU[\ell] = x$ ,  $p_x$  exited the **wait** statement of the  $\ell$ th iteration because  $\forall k \neq x : FLAG\_LEVEL[k] < \ell$ . But this is contradicted by the fact that we had then  $FLAG\_LEVEL[y] = \ell$ , which concludes the proof of the claim.

The proof of the starvation-freedom property is by induction on the levels starting from level  $n - 1$  and proceeding until level 1. The base case  $\ell = n - 1$  follows from the previous claim: if there is a process at level  $(n - 1)$ , it is the only process at that level and it can exit the **for** loop. This process eventually enters the critical section (that, by assumption, it will leave later). The induction assumption is the following: each process that attains a level  $\ell'$  such that  $n - 1 \geq \ell' \geq \ell$  eventually enters the critical section.

The rest of the proof is by contradiction. Let us assume that  $\ell$  is such that there is a process (say  $p_x$ ) that remains blocked forever in the **wait** statement during its  $\ell$ th

iteration (hence,  $p_x$  cannot attain level  $\ell$ ). It follows that, each time  $p_x$  evaluates the predicate controlling the **wait** statement, we have

$$(\exists k \neq i : FLAG\_LEVEL[k] \geq \ell) \wedge (AFTER\_YOU[\ell] = x)$$

(let us remember that the atomic registers are read one at a time, asynchronously, and in any order). There are two cases.

- Case 1: There is a process  $p_y$  that eventually executes  $AFTER\_YOU[\ell] \leftarrow y$ .  
As only  $p_x$  can execute  $AFTER\_YOU[\ell] \leftarrow x$ , there is eventually a read of  $AFTER\_YOU[\ell]$  that returns a value different from  $x$ , and this read allows  $p_x$  to progress to level  $\ell$ . This contradicts the assumption that  $p_x$  remains blocked forever in the **wait** statement during its  $\ell$ th iteration.
- Case 2: No process  $p_y$  eventually executes  $AFTER\_YOU[\ell] \leftarrow y$ .

The other processes can be partitioned in two sets: the set  $G$  that contains the processes at a level greater or equal to  $\ell$ , and the set  $L$  that contains the processes at a level smaller than  $\ell$ .

As the predicate  $AFTER\_YOU[\ell] = x$  remains forever true, it follows that no process  $p_y$  in  $L$  enters the  $\ell$ th loop iteration (otherwise  $p_y$  would necessarily execute  $AFTER\_YOU[\ell] \leftarrow y$ , contradicting the case assumption).

On the other side, due to the induction assumption, all processes in  $G$  eventually enter (and later leave) the critical section. When this has occurred, these processes have moved from the set  $G$  to the set  $L$  and then the predicate  $\forall k \neq i : FLAG\_LEVEL[k] < \ell$  becomes true.

When this has happened, the values returned by the asynchronous reading of  $FLAG\_LEVEL[1..n]$  by  $p_x$  allow it to attain level  $\ell$ , which contradicts the assumption that  $p_x$  remains blocked forever in the **wait** statement during its  $\ell$ th iteration.

In both case the assumption that a process remains blocked forever at level  $\ell$  is contradicted which completes the proof of the induction step and concludes the proof of the starvation-freedom property.  $\square$

**Starvation-freedom versus bounded bypass** The two-process Peterson's algorithm satisfies the bounded bypass liveness property while the  $n$ -process algorithm satisfies only starvation-freedom. Actually, starvation-freedom (i.e., finite bypass) is the best liveness property that Peterson's  $n$ -process algorithm (Fig. 2.8) guarantees.

This can be shown with a simple example. Let us consider the case  $n = 3$ . The three processes  $p_1$ ,  $p_2$ , and  $p_3$  invoke simultaneously `acquire_mutex()`, and the run is such that  $p_1$  wins the competition and enters the critical section. Moreover, let us assume that  $AFTER\_YOU[1] = 3$  (i.e.,  $p_3$  is the last process that wrote  $AFTER\_YOU[1]$ ) and  $p_3$  blocked at level 1.

Then, after it has invoked `release_mutex()`, process  $p_1$  invokes `acquire_mutex()` again and we have consequently  $AFTER\_YOU[1] = 1$ . But, from that time,  $p_3$  starts

an arbitrary long “sleeping” period (this is possible as the processes are asynchronous) and consequently does not read  $AFTER\_YOU[1] = 1$  (which would allow it to progress to the second level). Differently,  $p_2$  progresses to the second level and enters the critical section. Later,  $p_2$  first invokes `release_mutex()` and immediately after invokes `acquire_mutex()` and updates  $AFTER\_YOU[1] = 2$ . While  $p_3$  keeps on “sleeping”,  $p_1$  progresses to level 2 and finally enters the critical section. This scenario can be reproduced an arbitrary number of times until  $p_3$  wakes up. When this occurs,  $p_3$  reads from  $AFTER\_YOU[1]$  a value different from 3, and consequently progresses to level 2. Hence:

- Due to asynchrony, a “sleeping period” can be arbitrarily long, and a process can consequently lose an arbitrary number of competitions with respect to the other processes,
- But, as a process does not sleep forever, it eventually progresses to the next level.

It is important to notice that, as shown in the proof of the bounded pass property of Theorem 1, this scenario cannot happen when  $n = 2$ .

**Atomic register: size and number** It is easy to see that the algorithm uses  $2n - 1$  atomic registers. The domain of each of the  $n$  registers  $FLAG\_LEVEL[i]$  is  $[0..(n - 1)]$ , while the domain of each of the  $n - 1$   $AFTER\_YOU[\ell]$  registers is  $[1..n]$ . Hence, in both cases,  $\lceil \log_2 n \rceil$  bits are necessary and sufficient for each atomic register.

**Number of accesses to atomic registers** Let us define the time complexity of a mutex algorithm as the number of accesses to atomic registers for one use of the critical section by a process.

It is easy to see that this cost is finite but not bounded when there is contention (i.e., when several processes simultaneously compete to execute the critical section code).

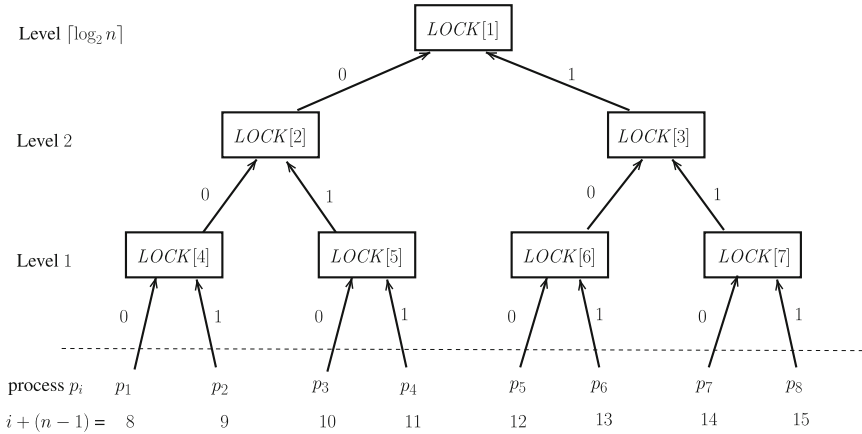
Differently in a contention-free scenario (i.e., when only one process  $p_i$  wants to use the critical section), the number of accesses to atomic registers is  $(n - 1)(n + 2)$  in `acquire_mutex(i)` and one in `release_mutex(i)`.

**The case of  $k$ -exclusion** This is the  $k$ -mutual exclusion problem where the critical section code can be concurrently accessed by up to  $k$  processes (mutual exclusion corresponds to the case where  $k = 1$ ).

Peterson’s  $n$ -process algorithm can easily be modified to solve  $k$ -mutual exclusion. The upper bound of the **for** loop (namely  $(n - 1)$ ) has simply to be replaced by  $(n - k)$ . No other statement modification is required. Moreover, let us observe that the size of the array  $AFTER\_YOU$  can then be reduced to  $[1..(n - k)]$ .

### 2.1.5 Mutex for $n$ Processes: A Tournament-Based Algorithm

**Reducing the number of shared memory accesses** In the previous  $n$ -process mutex algorithm, a process has to compete with the  $(n - 1)$  other processes before



**Fig. 2.10** A tournament tree for  $n$  processes

being able to access the critical section. Said differently, it has to execute  $n - 1$  loop iterations (eliminating another process at each iteration), and consequently, the cost (measured in number of accesses to atomic registers) in a contention-free scenario is  $O(n) \times$  the cost of one loop iteration, i.e.,  $O(n^2)$ . Hence a natural question is the following: Is it possible to reduce this cost and (if so) how?

**Tournament tree** A simple principle to reduce the number of shared memory accesses is to use a tournament tree. Such a tree is a complete binary tree. To simplify the presentation, we consider that the number of processes is a power of 2, i.e.,  $n = 2^k$  (hence  $k = \log_2 n$ ). If  $n$  is not a power of two, it has to be replaced by  $n' = 2^k$  where  $k = \lceil \log_2 n \rceil$  (i.e.,  $n'$  is the smallest power of 2 such that  $n' > n$ ).

Such a tree for  $n = 2^3$  processes  $p_1, \dots, p_8$ , is represented in Fig. 2.10. Each node of the tree is any two-process starvation-free mutex algorithm, e.g., Peterson's two-process algorithm. It is even possible to associate different two-process mutex algorithms with different nodes. The important common feature of these algorithms is that any of them assumes that it is used by two processes whose identities are 0 and 1.

As we have seen previously, any two-process mutex algorithm implements a lock object. Hence, we consider in the following that the tournament tree is a tree of  $(n - 1)$  locks and we accordingly adopt the lock terminology. The locks are kept in an array denoted  $LOCK[1..(n - 1)]$ , and for  $x \neq y$ ,  $LOCK[x]$  and  $LOCK[y]$  are independent objects (the atomic registers used to implement  $LOCK[x]$  and the atomic registers used to implement  $LOCK[y]$  are different).

The lock  $LOCK[1]$  is associated with the root of the tree, and if it is not a leaf, the node associated with the lock  $LOCK[x]$  has two children associated with the locks  $LOCK[2x]$  and  $LOCK[2x + 1]$ .

According to its identity  $i$ , each process  $p_i$  starts competing with a single other process  $p_j$  to obtain a lock that is a leaf of the tree. Then, when it wins, the process

```

operation acquire_mutex( $i$ ) is
(1)   $node\_id \leftarrow i + (n - 1)$ ;
(2)  for level from 1 to  $k$  do    %  $k = \lceil \log_2 n \rceil$  %
(3)     $p\_id[level] \leftarrow node\_id \bmod 2$ ;
(4)     $node\_id \leftarrow \lfloor node\_id / 2 \rfloor$ ;
(5)     $LOCK[node\_id].acquire\_lock(p\_id[level])$ 
(6)  end for;
(7)  return()
end operation.

operation release_mutex( $i$ ) is
(8)   $node\_id \leftarrow 1$ ;
(9)  for level from  $k$  to 1 do
(10)    $LOCK[node\_id].release\_lock(p\_id[level])$ ;
(11)    $node\_id \leftarrow 2 \times node\_id + p\_id[level]$ 
(12) end for;
(13) return()
end operation.

```

**Fig. 2.11** Tournament-based mutex algorithm (code for  $p_i$ )

$p_i$  proceeds to the next level of the tree to acquire the lock associated with the node that is the father of the node currently associated with  $p_i$  (initially the leaf node associated with  $p_i$ ). Hence, a process competes to acquire all the locks on the path from the leaf it is associated with until the root node.

As (a) the length of such a path is  $\lceil \log_2 n \rceil$  and (b) the cost to obtain a lock associated with a node is  $O(1)$  in contention-free scenarios, it is easy to see that the number of accesses to atomic registers in these scenarios is  $O(\log_2 n)$  (it is exactly  $4\log_2 n$  when each lock is implemented with Peterson's two-process algorithm).

**The tournament-based mutex algorithm** This algorithm is described in Fig. 2.11. Each process  $p_i$  manages a local variable  $node\_id$  such that  $LOCK[node\_id]$  is the lock currently addressed by  $p_i$  and a local array  $p\_id[1..k]$  such that  $p\_id[\ell]$  is the identity (0 or 1) used by  $p_i$  to access  $LOCK[node\_id]$  as indicated by the labels on the arrows in Fig. 2.10. (For a process  $p_i$ ,  $p\_id[\ell]$  could be directly computed from the values  $i$  and  $\ell$ ; a local array is used to simplify the presentation.)

When a process  $p_i$  invokes  $acquire\_mutex(i)$  it first considers that it has successfully locked a fictitious lock object  $LOCK[i + (n - 1)]$  that can be accessed only by this process (line 1). Process  $p_i$  then enters a loop to traverse the tree, level by level, from its starting leaf until the root (lines 2–6). The starting leaf of  $p_i$  is associated with the lock  $LOCK[\lfloor (i + (n - 1)) / 2 \rfloor]$  (lines 1 and 4). The identity used by  $p_i$  to access the lock  $LOCK[node\_id]$  (line 5) is computed at line 3 and saved in  $p\_id[level]$ .

When it invokes  $release\_mutex(i)$ , process  $p_i$  releases the  $k$  locks it has locked starting from the lock associated with the root ( $LOCK[1]$ ) until the lock associated

with its starting leaf  $LOCK[\lfloor (i + (n - 1))/2 \rfloor]$ . When it invokes  $LOCK[node\_id].release\_lock(p\_id[level])$  (line 10), the value of the parameter  $p\_id[level]$  is the identity (0 or 1) used by  $p_i$  when it locked that object. This identity is also used by  $p_i$  to compute the index of the next lock object it has to unlock (line 11).

**Theorem 3** *Assuming that each two-process lock object satisfies mutual exclusion and deadlock-freedom (or starvation-freedom), the algorithm described in Fig. 2.11 satisfies mutual exclusion and deadlock-freedom (or starvation-freedom).*

*Proof* The proof of the mutex property is by contradiction. If  $p_i$  and  $p_j$  ( $i \neq j$ ) are simultaneously in the critical section, there is a lock object  $LOCK[node\_id]$  such that  $p_i$  and  $p_j$  have invoked  $acquire\_lock()$  on that object and both have been simultaneously granted the lock. (If there are several such locks, let  $LOCK[node\_id]$  be one at the lowest level in the tree.) Due to the specification of the lock object (that grants the lock to a single process identity, namely 0 or 1), it follows that both  $p_i$  and  $p_j$  have invoked  $LOCK[node\_id].acquire\_lock()$  with the same identity value (0 or 1) kept in their local variable  $p\_id[level]$ . But, due to the binary tree structure of the set of lock objects and the way the processes compute  $p\_id[level]$ , this can only happen if  $i = j$  (on the lowest level on which  $p_i$  and  $p_j$  share a lock), which contradicts our assumption and completes the proof of the mutex property.

The proof of the starvation-freedom (or deadlock-freedom) property follows from the same property of the base lock objects. We consider here only the starvation-freedom property. Let us assume that a process  $p_i$  is blocked forever at the object  $LOCK[node\_id]$ . This means that there is another process  $p_j$  that competes infinitely often with  $p_i$  for the lock granted by  $LOCK[node\_id]$  and wins each time. The proof follows from the fact that, due to the starvation-freedom property of  $LOCK[node\_id]$ , this cannot happen.  $\square$

**Remark** Let us consider the case where each algorithm implementing an underlying two-process lock object uses a bounded number of bounded atomic registers (which is the case for Peterson's two-process algorithm). In that case, as the tournament-based algorithm uses  $(n - 1)$  lock objects, it follows that it uses a bounded number of bounded atomic registers.

Let us observe that this tournament-based algorithm has better time complexity than Peterson's  $n$ -process algorithm.

### 2.1.6 A Concurrency-Abortable Algorithm

When looking at the number of accesses to atomic registers issued by  $acquire\_mutex()$  and  $release\_mutex()$  for a single use of the critical section in a contention-free scenario, the cost of Peterson's  $n$ -process mutual exclusion

algorithm is  $O(n^2)$  while the cost of the tournament tree-based algorithm is  $O(\log_2 n)$ . Hence, a natural question is the following: Is it possible to design a *fast*  $n$ -process mutex algorithm, where *fast* means that the cost of the algorithm is constant in a contention-free scenario?

The next section of this chapter answers this question positively. To that end, an incremental presentation is adopted. A simple one-shot operation is first presented. Each of its invocations returns a value  $r$  to the invoking process, where  $r$  is the value *abort* or the value *commit*. Then, the next section enriches the algorithm implementing this operation to obtain a deadlock-free fast mutual exclusion algorithm due to L. Lamport (1987).

**Concurrency-abortable operation** A *concurrency-abortable* (also named *contention-abortable* and usually abbreviated *abortable*) operation is an operation that is allowed to return the value *abort* in the presence of concurrency. Otherwise, it has to return the value *commit*. More precisely, let `conc_abort_op()` be such an operation. Assuming that each process invokes it at most once (one-shot operation), the set of invocations satisfies the following properties:

- **Obligation.** If the first process which invokes `conc_abort_op()` is such that its invocation occurs in a concurrency-free pattern (i.e., no other process invokes `conc_abort_op()` during its invocation), this process obtains the value *commit*.
- **At most one.** At most one process obtains the value *commit*.

**An  $n$ -process concurrency-abortable algorithm** Such an algorithm is described in Fig. 2.12. As in the previous algorithms, it assumes that all the processes have distinct identities, but differently from them, the number  $n$  of processes can be arbitrary and remains unknown to the processes.

This algorithm uses two MWMR atomic registers denoted  $X$  and  $Y$ . The register  $X$  contains a process identity (its initial value being arbitrary). The register  $Y$  contains a process identity or the default value  $\perp$  (which is its initial value). It is consequently assumed that these atomic registers are made up of  $\lceil \log_2(n + 1) \rceil$  bits.

```

operation conc_abort_op( $i$ ) is
(1)   $X \leftarrow i$ ;
(2)  if ( $Y \neq \perp$ )
(3)    then return( $abort_1$ )
(4)    else  $Y \leftarrow i$ ;
(5)      if ( $X = i$ )
(6)        then return( $commit$ )
(7)        else return( $abort_2$ )
(8)      end if
(9)  end if
end operation.

```

**Fig. 2.12** An  $n$ -process concurrency-abortable operation (code for  $p_i$ )



When it invokes `conc_abort_op()`, a process  $p_i$  first deposits its identity in  $X$  (line 1) and then checks if the current value of  $Y$  is its initial value  $\perp$  (line 2). If  $Y \neq \perp$ , there is (at least) one process  $p_j$  that has written into  $Y$ . In that case,  $p_i$  returns  $abort_1$  (both  $abort_1$  and  $abort_2$  are synonyms of *abort*; they are used only to distinguish the place where the invocation of `conc_abort_op()` is “aborted”). Returning  $abort_1$  means that (from a concurrency point of view)  $p_i$  was late: there is another process that wrote into  $Y$  before  $p_i$  reads it.

If  $Y = \perp$ , process  $p_i$  writes its identity into  $Y$  (line 4) and then checks if  $X$  is still equal to its identity  $i$  (line 5). If this is the case,  $p_i$  returns the value *commit* at line 6 (its invocation of `conc_abort_op(i)` is then successful). If  $X \neq i$ , another process  $p_j$  has written its identity  $j$  into  $X$ , overwriting the identity  $i$  before  $p_i$  reads  $X$  at line 5. Hence, there is contention and the value  $abort_2$  is returned to  $p_i$  (line 7). Returning  $abort_2$  means that, among the competing processes that found  $y = \perp$ ,  $p_i$  was not the last to have written its name into  $X$ .

**Remark** Let us observe that the only test on  $Y$  is  $Y \neq \perp$  (line 2). It follows that  $Y$  could be replaced by a flag with the associated domain  $\{\perp, \top\}$ . Line 4 should then be replaced by  $Y \leftarrow \top$ .

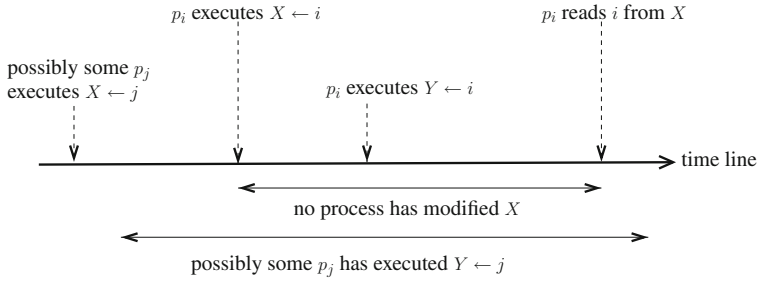
Using such a flag is not considered here because we want to keep the notation consistent with that of the fast mutex algorithm presented below. In the fast mutex algorithm, the value of  $Y$  can be either  $\perp$  or any process identifier.

**Theorem 4** *The algorithm described in Fig. 2.12 guarantees that (a) at most one process obtains the value *commit* and (b) if the first process that invokes `conc_abort_op()` executes it in a concurrency-free pattern, it obtains the value *commit*.*

*Proof* The proof of property (b) stated in the theorem is trivial. If the first process (say  $p_i$ ) that invokes `conc_abort_op()` executes this operation in a concurrency-free context, we have  $Y = \perp$  when it reads  $Y$  at line 2 and  $X = i$  when it reads  $X$  at line 5. It follows that it returns *commit* at line 6.

Let us now prove property (a), i.e., that no two processes can obtain the value *commit*. Let us assume for the sake of contradiction that a process  $p_i$  has invoked `conc_abort_op(i)` and obtained the value *commit*. It follows from the text of the algorithm that the pattern of accesses to the atomic registers  $X$  and  $Y$  issued by  $p_i$  is the one described in Fig. 2.13 (when not considering the accesses by  $p_j$  in that figure). There are two cases.

- Let us first consider the (possibly empty) set  $Q$  of processes  $p_j$  that read  $Y$  at line 2 after this register was written by  $p_i$  or another process (let us notice that, due to the atomicity of the registers  $X$  and  $Y$ , the notion of after/before is well defined). As  $Y$  is never reset to  $\perp$ , it follows that each process  $p_j \in Q$  obtains a non- $\perp$  value from  $Y$  and consequently executes `return(abort1)` at line 3.



**Fig. 2.13** Access pattern to  $X$  and  $Y$  for a successful `conc_abort_op()` invocation by process  $p_i$

- Let us now consider the (possibly empty) set  $Q'$  of processes  $p_j$  distinct from  $p_i$  that read  $\perp$  from  $Y$  at line 2 concurrently with  $p_i$ . Each  $p_j \in Q'$  writes consequently its identity  $j$  into  $Y$  at line 4.

As  $p_i$  has read  $i$  from  $X$  (line 5), it follows that no process  $p_j \in Q'$  has modified  $X$  between the execution of line 1 and line 5 by  $p_i$  (otherwise  $p_i$  would not have read  $i$  from  $X$  at line 5, see Fig. 2.13). Hence any process  $p_j \in Q'$  has written  $X$  (a) either before  $p_i$  writes  $i$  into  $X$  or (b) after  $p_i$  has read  $i$  from  $X$ . But, observe that case (b) cannot happen. This is due to the following observation. A process  $p_k$  that writes  $X$  (at line 1) after  $p_i$  has read  $i$  from this register (at line 5) necessarily finds  $Y \neq \perp$  at line 4 (this is because  $p_i$  has previously written  $i$  into  $Y$  at line 4 before reading  $i$  from  $X$  at line 5). Consequently, such a process  $p_k$  belongs to the set  $Q$  and not to the set  $Q'$ . Hence, the only possible case is that each  $p_j \in Q'$  has written  $j$  into  $X$  before  $p_i$  writes  $i$  into  $X$ . It follows that  $p_i$  is the last process of  $Q' \cup \{p_i\}$  which has written its identity into  $X$ .

We conclude from the previous observation that, when a process  $p_j \in Q'$  reads  $X$  at line 5, it obtains from this register a value different from  $j$  and, consequently, its invocation `conc_abort_op(j)` returns the value  $abort_2$ , which concludes the proof of the theorem.  $\square$

The next corollary follows from the proof of the previous theorem.

**Corollary 1** ( $Y \neq \perp$ )  $\Rightarrow$  a process has obtained the value *commit* or several processes have invoked `conc_abort_op()`.

**Theorem 5** Whatever the number of processes that invoke `conc_abort_op()`, any of these invocations costs at most four accesses to atomic registers.

*Proof* The proof follows from a simple examination of the algorithm.  $\square$

**Remark: splitter object** When we (a) replace the value *commit*,  $abort_1$ , and  $abort_2$  by *stop*, *right*, and *left*, respectively, and (b) rename the operation

$\text{conc\_abort\_op}(i)$  as  $\text{direction}(i)$ , we obtain a one-shot object called a *splitter*. A one-shot object is an object that provides processes with a single operation and each process invokes that operation at most once.

In a run in which a single process invokes  $\text{direction}()$ , it obtains the value *stop*. In any run, if  $m > 1$  processes invoke  $\text{direction}()$ , at most one process obtains the value *stop*, at most  $(m - 1)$  processes obtain *right*, and at most  $(m - 1)$  processes obtain *left*. Such an object is presented in detail in Sect. 5.2.1.

### 2.1.7 A Fast Mutex Algorithm

**Principle and description** This section presents L. Lamport’s fast mutex algorithm, which is built from the previous one-shot concurrency-abortable operation. More specifically, this algorithm behaves similarly to the algorithm of Fig. 2.12 in contention-free scenarios and (instead of returning *abort*) guarantees the deadlock-freedom liveness property when there is contention.

The algorithm is described in Fig. 2.14. The line numbering is the same as in Fig. 2.12: the lines with the same number are the same in both algorithms, line N0 is new, line N3 replaces line 3, lines N7.1–N7.5 replace line 7, and line N10 is new.

To attain its goal (both fast mutex and deadlock-freedom) the algorithm works as follows. First, each process  $p_i$  manages a SWMR flag  $FLAG[i]$  (initialized to *down*)

```

operation acquire_mutex( $i$ ) is
(N0)   $FLAG[i] \leftarrow up$ ;
(1)    $X \leftarrow i$ ;
(2)   if ( $Y \neq \perp$ )
(N3)   then  $FLAG[i] \leftarrow down$ ; wait ( $Y = \perp$ ); restart at line N0
(4)   else  $Y \leftarrow i$ ;
(5)       if ( $X = i$ )
(6)       then return()
(N7.1)       else  $FLAG[i] \leftarrow down$ ;
(N7.2)           for each  $j$  do wait ( $FLAG[j] = down$ ) end for;
(N7.3)           if ( $Y = i$ ) then return()
(N7.4)           else wait ( $Y = \perp$ ); restart at line N0
(N7.5)           end if
(8)       end if
(9)   end if
end operation.

operation release_mutex( $i$ ) is
(N10)  $Y \leftarrow \perp$ ;  $FLAG[i] \leftarrow down$ ; return()
end operation.

```

**Fig. 2.14** Lamport’s fast mutex algorithm (code for  $p_i$ )

that  $p_i$  sets to *up* to indicate that it is interested in the critical section (line N0). This flag is reset to *down* when  $p_i$  exits the critical section (line N10). As we are about to see, it can be reset to *down* also in other parts of the algorithm.

According to the contention scenario in which a process  $p_i$  returns *abort* in the algorithm of Fig. 2.12, there are two cases to consider, which have been differentiated by the values  $abort_1$  and  $abort_2$ .

- Eliminating  $abort_1$  (line N3).

In this case, as we have seen in Fig. 2.12, process  $p_i$  is “late”. As captured by Corollary 1, this is because there are other processes that currently compete for the critical section or there is a process inside the critical section. Line 3 of Fig. 2.12 is consequently replaced by the following statements (new line N3):

- Process  $p_i$  first resets its flag to *down* in order not to prevent other processes from entering the critical section (if no other process is currently inside it).
- According to Corollary 1, it is useless for  $p_i$  to retry entering the critical section while  $Y \neq \perp$ . Hence, process  $p_i$  delays its request for the critical section until  $Y = \perp$ .

- Eliminating  $abort_2$  (lines N7.1–N7.5).

In this case, as we have seen in the base contention-abortable algorithm (Fig. 2.12), several processes are competing for the critical section (or a process is already inside the critical section). Differently from the base algorithm, one of the competing processes has now to be granted the critical section (if no other process is inside it). To that end, in order not to prevent another process from entering the critical section, process  $p_i$  first resets its flag to *down* (line N7.1). Then,  $p_i$  tries to enter the critical section. To that end, it first waits until all flags are down (line N7.2). Then,  $p_i$  checks the value of  $Y$  (line N7.3). There are two cases:

- If  $Y = i$ , process  $p_i$  enters the critical section. This is due to the following reason.

Let us observe that, if  $Y = i$  when  $p_i$  reads it at line N7.3, then no process has modified  $Y$  since  $p_i$  set it to the value  $i$  at line 4 (the write of  $Y$  at line 4 and its reading at line N7.3 follow the same access pattern as the write of  $X$  at line 1 and its reading at line 5). Hence, process  $p_i$  is the last process to have executed line 4. It then follows that, as it has (asynchronously) seen each flag equal to *down* (line 7.2), process  $p_i$  is allowed to enter the critical section (return() statement at line N7.3).

- If  $Y \neq i$ , process  $p_i$  does the same as what is done at line N3. As it has already set its flag to *down*, it has only to wait until the critical section is released before retrying to enter it (line N7.4). (Let us remember that the only place where  $Y$  is reset to  $\perp$  is when a process releases the critical section.)

**Fast path and slow path** The *fast* path to enter the critical section is when  $p_i$  executes only the lines N0, 1, 2, 4, 5, and 6. The fast path is open for a process  $p_i$

if it reads  $i$  from  $X$  at line 5. This is the path that is always taken by a process in contention-free scenarios.

The cost of the fast path is five accesses to atomic registers. As `release_mutex()` requires two accesses to atomic registers, it follows that the cost of a single use of the critical section in a contention-free scenario is seven accesses to atomic registers.

The *slow* path is the path taken by a process which does not take the fast path. Its cost in terms of accesses to atomic registers depends on the current concurrency pattern.

**A few remarks** A register  $FLAG[i]$  is set to *down* when  $p_i$  exits the critical section (line N10) but also at line N3 or N7.1. It is consequently possible for a process  $p_k$  to be inside the critical section while all flags are down. But let us notice that, when this occurs, the value of  $Y$  is different from  $\perp$ , and as already indicated, the only place where  $Y$  is reset to  $\perp$  is when a process releases the critical section.

When executed by a process  $p_i$ , the aim of the **wait** statement at line N3 is to allow any other process  $p_j$  to see that  $p_i$  has set its flag to *down*. Without such a **wait** statement, a process  $p_i$  could loop forever executing the lines N0, 1, 2 and N3 and could thereby favor a livelock by preventing the other processes from seeing  $FLAG[i] = \text{down}$ .

**Theorem 6** *Lamport's fast mutex algorithm satisfies mutual exclusion and deadlock-freedom.*

*Proof* Let us first consider the mutual exclusion property. Let  $p_i$  be a process that is inside the critical section. Trivially, we have then  $Y \neq \perp$  and  $p_i$  returned from `acquire_mutex()` at line 6 or at line N7.3. Hence, there are two cases. Before considering these two cases, let us first observe that each process (if any) that reads  $Y$  after it was written by  $p_i$  (or another process) executes line N3: it resets its flag to *down* and waits until  $Y = \perp$  (i.e., at least until  $p_i$  exits the critical section, line N10). As the processes that have read a non- $\perp$  value from  $Y$  at line 2 cannot enter the critical section, it follows that we have to consider only the processes  $p_j$  that have read  $\perp$  from  $Y$  at line 2.

- Process  $p_i$  has executed `return()` at line 6.

In this case, it follows from a simple examination of the text of the algorithm that  $FLAG[i]$  remains equal to *up* until  $p_i$  exits the critical section and executes line N10.

Let us consider a process  $p_j$  that has read  $\perp$  from  $Y$  at line 2. As process  $p_i$  has executed line 6, it was the last process (among the competing processes which read  $\perp$  from  $Y$ ) to have written its identity into  $X$  (see Fig. 2.13) and consequently  $p_j$  cannot read  $j$  from  $X$ . As  $X \neq j$  when  $p_j$  reads  $X$  at line 5, it follows that process  $p_j$  executes the lines N7.1–N7.5. When it executes line N7.2,  $p_j$  remains blocked until  $p_i$  resets its flag to *down*, but as we have seen,  $p_i$  does so only when it exits the critical section. Hence,  $p_j$  cannot be inside the critical section simultaneously with  $p_i$ . This concludes the proof of the first case.

- Process  $p_i$  has executed `return()` at line N7.3.

In this case, the predicate  $Y = i$  allowed  $p_i$  to enter the critical section. Moreover, the atomic register  $Y$  has not been modified during the period starting when it was assigned the identity  $i$  at line 4 by  $p_i$  and ending at the time at which  $p_i$  read it at line N7.3. It follows that, among the processes that read  $\perp$  from  $Y$  (at line 2),  $p_i$  is the last one to have updated  $Y$ .

Let us observe that  $X \neq j$ , otherwise  $p_j$  would have entered the critical section at line 6, and in that case (as shown in the previous item)  $p_i$  could not have entered the critical section.

As  $Y = i$ , it follows from the test of line N7.3 that  $p_j$  executes line N7.4 and consequently waits until  $Y = \perp$ . As  $Y$  is set to  $\perp$  only when a process exits the critical section (line N10), it follows that  $p_j$  cannot be inside the critical section simultaneously with  $p_i$ , which concludes the proof of the second case.

To prove the deadlock-freedom property, let us assume that there is a non-empty set of processes that compete to enter the critical section and, from then on, no process ever executes `return()` at line 6 or line N 7.3. We show that this is impossible.

As processes have invoked `acquire_mutex()` and none of them executes line 6, it follows that there is among them at least one process  $p_x$  that has executed first line N0 and line 1 (where it assigned its identity  $x$  to  $X$ ) and then line N3. This assignment of  $x$  to  $X$  makes the predicate of line 5 false for the processes that have obtained  $\perp$  from  $Y$ . It follows that the flag of these processes  $p_x$  are eventually reset to *down* and, consequently, these processes cannot entail a permanent blocking of any other process  $p_i$  which executes line N7.2.

When the last process that used the critical section released it, it reset  $Y$  to  $\perp$  (if there is no such process, we initially have  $Y = \perp$ ). Hence, among the processes that have invoked `acquire_mutex()`, at least one of them has read  $\perp$  from  $Y$ . Let  $Q$  be this (non-empty) set of processes. Each process of  $Q$  executes lines N7.1–N7.5 and, consequently, eventually resets its flag to *down* (line N7.1). Hence, the predicate evaluated in the **wait** statement at line N7.2 eventually becomes satisfied and the processes of  $Q$  which execute the lines N7.1–N7.5 eventually check at line N7.3 if the predicate  $Y = i$  is satisfied. (Due to asynchrony, it is possible that the predicate used at N7.2 is never true when evaluated by some processes. This occurs for the processes of  $Q$  which are slow while another process of  $Q$  has entered the critical section and invoked `acquire_mutex()` again, thereby resetting its flag to *up*. The important point is that this can occur only if some process entered the critical section, hence when there is no deadlock.)

As no process is inside the critical section and the number of processes is finite, there is a process  $p_j$  that was the last process to have modified  $Y$  at line 4. As (by assumption)  $p_j$  has not executed `return()` at line 6, it follows that it executes line N7.3 and, finding  $Y = j$ , it executes `return()`, which contradicts our assumption and consequently proves the deadlock-freedom property.  $\square$

### 2.1.8 Mutual Exclusion in a Synchronous System

**Synchronous system** Differently from an asynchronous system (in which there is no time bound), a synchronous system is characterized by assumptions on the speed of processes. More specifically, there is a bound  $\Delta$  on the speed of processes and this bound is known to them (meaning that  $\Delta$  can be used in the code of the algorithms). The meaning of  $\Delta$  is the following: two consecutive accesses to atomic registers by a process are separated by at most  $\Delta$  time units.

Moreover, the system provides the processes with a primitive `delay( $d$ )`, where  $d$  is a positive duration, which stops the invoking process for a finite duration greater than  $d$ . The synchrony assumption applies only to consecutive accesses to atomic registers that are not separated by a `delay()` statement.

**Fischer's algorithm** A very simple mutual exclusion algorithm (due to M. Fischer) is described in Fig. 2.15. This algorithm uses a single atomic register  $X$  (initialized to  $\perp$ ) that, in addition to  $\perp$ , can contain any process identity.

When a process  $p_i$  invokes `acquire_mutex( $i$ )`, it waits until  $X = \perp$ . Then it writes its identity into  $X$  (as before, it is assumed that no two processes have the same identity) and invokes `delay( $\Delta$ )`. When it resumes its execution, it checks if  $X$  contains its identity. If this is the case, its invocation `acquire_mutex( $i$ )` terminates and  $p_i$  enters the critical section. If  $X \neq i$ , it re-executes the loop body.

**Theorem 7** *Let us assume that the number of processes is finite and all have distinct identities. Fischer's mutex algorithm satisfies mutual exclusion and deadlock-freedom.*

*Proof* To simplify the statement of the proof we consider that each access to an atomic register is instantaneous. (Considering that such accesses take bounded duration is straightforward.)

Proof of the mutual exclusion property. Assuming that, at some time, processes invoke `acquire_mutex()`, let  $C$  be the subset of them whose last read of  $X$  returned  $\perp$ . Let us observe that the ones that read a non- $\perp$  value from  $X$  remain looping in the

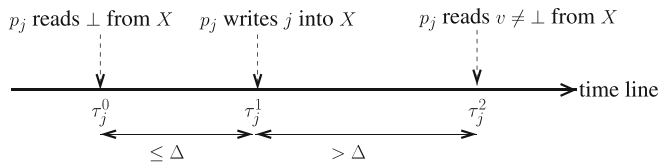
```

operation acquire_mutex( $i$ ) is
(1)  repeat wait ( $X = \perp$ );
(2)     $X \leftarrow i$ ;
(3)    delay( $\Delta$ )
(4)  until ( $X = i$ ) end repeat;
(5)  return()
end operation.

operation release_mutex( $i$ ) is
(6)   $X \leftarrow \perp$ ; return()
end operation.

```

**Fig. 2.15** Fischer's synchronous mutex algorithm (code for  $p_i$ )



**Fig. 2.16** Accesses to  $X$  by a process  $p_j$

**wait** statement at line 1. By assumption,  $C$  is finite. Due to the atomicity of the register  $X$  and the fact that all processes in  $C$  write into  $X$ , there is a last process (say  $p_i$ ) that writes its identity into  $X$ .

Given any process  $p_j$  of  $C$  let us define the following time instants (Fig. 2.16):

- $\tau_j^0$  = time at which  $p_j$  reads the value  $\perp$  from  $X$  (line 1),
- $\tau_j^1$  = time at which  $p_j$  writes its identity  $j$  into  $X$  (line 2), and
- $\tau_j^2$  = time at which  $p_j$  reads  $X$  (line 4) after having executed the  $\text{delay}(\Delta)$  statement (line 3).

Due to the synchrony assumption and the  $\text{delay}()$  statement we have  $\tau_j^1 \leq \tau_j^0 + \Delta$  (P1) and  $\tau_j^2 > \tau_j^1 + \Delta$  (P2). We show that, after  $p_i$  has written  $i$  into  $X$ , this register remains equal to  $i$  until  $p_i$  resets it to  $\perp$  (line 6) and any process  $p_j$  of  $C$  reads  $i$  from  $X$  at line 4 from which follows the mutual exclusion property. This is the consequence of the following observations:

1.  $\tau_j^1 + \Delta < \tau_j^2$  (property P2),
2.  $\tau_i^0 < \tau_j^1$  (otherwise  $p_i$  would not have read  $\perp$  from  $X$  at line 1),
3.  $\tau_i^0 + \Delta < \tau_j^1 + \Delta$  (adding  $\Delta$  to both sides of the previous line),
4.  $\tau_i^1 \leq \tau_i^0 + \Delta < \tau_j^1 + \Delta < \tau_j^2$  (from P1 and the previous items 1 and 3).

It then follows from the fact that  $p_i$  is the last process which wrote into  $X$  and  $\tau_j^2 > \tau_i^1$  that  $p_j$  reads  $i$  from  $X$  at line 4 and consequently does enter the **repeat** loop again and waits until  $X = \perp$ . The mutual exclusion property follows.

**Proof of the deadlock-freedom property.** This is an immediate consequence of the fact that, among the processes that have concurrently invoked the operation  $\text{acquire\_mutex}()$ , the last process that writes  $X$  ( $p_i$  in the previous reasoning) reads its own identity from  $X$  at line 4.  $\square$

**Short discussion** The main property of this algorithm is its simplicity. Moreover, its code is independent of the number of processes.

## 2.2 Mutex Based on Specialized Hardware Primitives

The previous section presented mutual exclusion algorithms based on atomic read/write registers. These algorithms are important because understanding their design and their properties provides us with precise knowledge of the difficulty and subtleties



that have to be addressed when one has to solve synchronization problems. These algorithms capture the essence of synchronization in a read/write shared memory model.

Nearly all shared memory multiprocessors propose built-in primitives (i.e., atomic operations implemented in hardware) specially designed to address synchronization issues. This section presents a few of them (the ones that are the most popular).

### 2.2.1 Test&Set, Swap, and Compare&Swap

**The test&set()/reset() primitives** This pair of primitives, denoted test&set() and reset(), is defined as follows. Let  $X$  be a shared register initialized to 1.

- $X.test\&set()$  sets  $X$  to 0 and returns its previous value.
- $X.reset()$  writes 1 into  $X$  (i.e., resets  $X$  to its initial value).

Given a register  $X$ , the operations  $X.test\&set()$  and  $X.reset()$  are atomic. As we have seen, this means that they appear as if they have been executed sequentially, each one being associated with a point of the time line (that lies between its beginning and its end).

As shown in Fig. 2.17 (where  $r$  is local variable of the invoking process), solving the mutual exclusion problem (or equivalently implementing a lock object), can be easily done with a test&set register. If several processes invoke simultaneously  $X.test\&set()$ , the atomicity property ensures that one and only of them wins (i.e., obtains the value 1 which is required to enter the critical section). Releasing the critical section is done by resetting  $X$  to 1 (its initial value). It is easy to see that this implementation satisfies mutual exclusion and deadlock-freedom.

**The swap() primitive** Let  $X$  be a shared register. The primitive denoted  $X.swap(v)$  atomically assigns  $v$  to  $X$  and returns the previous value of  $X$ .

Mutual exclusion can be easily solved with a swap register  $X$ . Such an algorithm is depicted in Fig. 2.18 where  $X$  is initialized to 1. It is assumed that the invoking process

```

operation acquire_mutex() is
    repeat  $r \leftarrow X.test\&set()$  until  $(r = 1)$  end repeat;
    return()
end operation.

operation release_mutex() is
     $X.reset()$ ; return()
end operation.

```

**Fig. 2.17** Test&set-based mutual exclusion

```

operation acquire_mutex() is
     $r \leftarrow 0$ ;
    repeat  $r \leftarrow X.swap(r)$  until  $(r = 1)$  end repeat;
    return();
end operation.

operation release_mutex() is
     $X.swap(r)$ ; return();
end operation.

```

**Fig. 2.18** Swap-based mutual exclusion

does not modify its local variable  $r$  between `acquire_mutex()` and `release_mutex()` (or, equivalently, that it sets  $r$  to 1 before invoking `release_mutex()`). The test&set-based algorithm and the swap-based algorithm are actually the very same algorithm.

Let  $r_i$  be the local variable used by each process  $p_i$ . Due to the atomicity property and the “exchange of values” semantics of the `swap()` primitive, it is easy to see the swap-based algorithm is characterized by the invariant  $X + \sum_{1 \leq i \leq n} r_i = 1$ .

**The compare&swap() primitive** Let  $X$  be a shared register and *old* and *new* be two values. The semantics of the primitive  $X.compare\&swap(old, new)$ , which returns a Boolean value, is defined by the following code that is assumed to be executed atomically.

```

 $X.compare\&swap(old, new)$  is
    if  $(X = old)$  then  $X \leftarrow new$ ; return(true)
    else return(false)
    end if.

```

The primitive `compare&swap()` is an atomic conditional write; namely, the write of *new* into  $X$  is executed if and only if  $X = old$ . Moreover, a Boolean value is returned that indicates if the write was successful. This primitive (or variants of it) appears in Motorola 680x0, IBM 370, and SPARC architectures. In some variants, the primitive returns the previous value of  $X$  instead of a Boolean.

A compare&swap-based mutual exclusion algorithm is described in Fig. 2.19 in which  $X$  is an atomic compare&swap register initialized to 1. (no-op means “no operation”.) The **repeat** statement is equivalent to **wait** ( $X.compare\&swap(1, 0)$ ); it is used to stress the fact that it is an active waiting. This algorithm is nearly the same as the two previous ones.

### 2.2.2 From Deadlock-Freedom to Starvation-Freedom

**A problem due to asynchrony** The previous primitives allow for the (simple) design of algorithms that ensure mutual exclusion and deadlock-freedom. Said differently, these algorithms do not ensure starvation-freedom.

```

operation acquire_mutex() is
    repeat  $r \leftarrow X.\text{compare\&swap}(1, 0)$  until ( $r$ ) end repeat;
    return()
end operation.

operation release_mutex() is
     $X \leftarrow 1$ ; return()
end operation.

```

**Fig. 2.19** Compare&swap-based mutual exclusion

As an example, let us consider the test&set-based algorithm (Fig. 2.17). It is possible that a process  $p_i$  executes  $X.\text{test\&set}()$  infinitely often and never obtains the winning value 1. This is a simple consequence of asynchrony: if, infinitely often, other processes invoke  $X.\text{test\&set}()$  concurrently with  $p_i$  (some of these processes enter the critical section, release it, and re-enter it, etc.), it is easy to construct a scenario in which the winning value is always obtained by only a subset of processes not containing  $p_i$ . If  $X$  infinitely often switches between 1 to 0, an infinite number of accesses to  $X$  does not ensure that one of these accesses obtains the value 1.

**From deadlock-freedom to starvation-freedom** Considering that we have an underlying lock object that satisfies mutual exclusion and deadlock-freedom, this section presents an algorithm that builds on top of it a lock object that satisfies the starvation-freedom property. Its principle is simple: it consists in implementing a round-robin mechanism that guarantees that no request for the critical section is delayed forever. To that end, the following underlying objects are used:

- The underlying deadlock-free lock is denoted *LOCK*. Its two operations are *LOCK.acquire\_lock(i)* and *LOCK.release\_lock(i)*, where  $i$  is the identity of the invoking process.
- An array of SWMR atomic registers denoted *FLAG*[1.. $n$ ] ( $n$  is the number of processes, hence this number has to be known). For each  $i$ , *FLAG*[ $i$ ] is initialized to *down* and can be written only by  $p_i$ . In a very natural way, process  $p_i$  sets *FLAG*[ $i$ ] to *up* when it wants to enter the critical section and resets it to *down* when it releases it.
- *TURN* is an MWMR atomic register that contains the process which is given priority to enter the critical section. Its initial value is any process identity.

Let us notice that accessing *FLAG*[*TURN*] is not an atomic operation. A process  $p_i$  has first to obtain the value  $v$  of *TURN* and then address *FLAG*[ $v$ ]. Moreover, due to asynchrony, between the read by  $p_i$  first of *TURN* and then of *FLAG*[ $v$ ], the value of *TURN* has possibly been changed by another process  $p_j$ .

The behavior of a process  $p_i$  is described in Fig. 2.20. It is as follows. The processes are considered as defining a logical ring  $p_i, p_{i+1}, \dots, p_n, p_1, \dots, p_i$ . At any time,

```

operation acquire_mutex( $i$ ) is
(1)   $FLAG[i] \leftarrow up$ ;
(2)  wait ( $TURN = i$ )  $\vee$  ( $FLAG[TURN] = down$ ) ;
(3)   $LOCK.acquire\_lock(i)$ ;
(4)  return()
end operation.

operation release_mutex( $i$ ) is
(5)   $FLAG[i] \leftarrow down$ ;
(6)  if ( $FLAG[TURN] = down$ ) then  $TURN \leftarrow (TURN \bmod n) + 1$  end if;
(7)   $LOCK.release\_lock(i)$ ;
(8)  return()
end operation.

```

**Fig. 2.20** From deadlock-freedom to starvation-freedom (code for  $p_i$ )

the process  $p_{TURN}$  is the process that has priority and  $p_{(TURN \bmod n)+1}$  is the next process that will have priority.

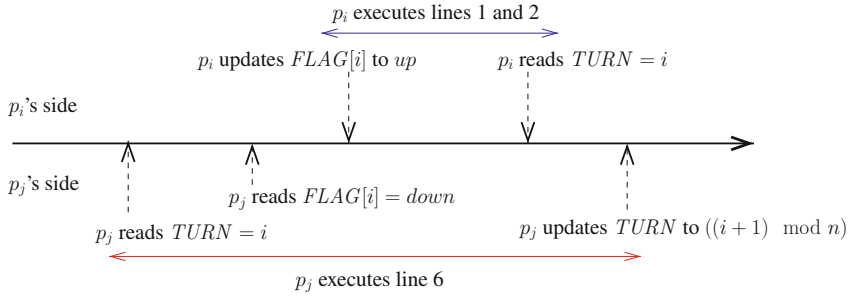
- When a process  $p_i$  invokes  $acquire\_mutex(i)$  it first raises its flag to inform the other processes that it is interested in the critical section (line 1). Then, it waits (repeated checks at line 2) until it has priority (predicate  $TURN = i$ ) or the process that is currently given the priority is not interested (predicate  $FLAG[TURN] = down$ ). Finally, as soon as it can proceed, it invokes  $LOCK.acquire\_lock(i)$  in order to obtain the underlying lock (line 3). (Let us remember that reading  $FLAG[TURN]$  requires two shared memory accesses.)
- When a process  $p_i$  invokes  $release\_mutex(i)$ , it first resets its flag to *down* (line 5). Then, if (from  $p_i$ 's point of view) the process that is currently given priority is not interested in the critical section (i.e., the predicate  $FLAG[TURN] = down$  is satisfied), then  $p_i$  makes  $TURN$  progress to the next process (line 6) on the ring before releasing the underlying lock (line 7).

**Remark 1** Let us observe that the modification of  $TURN$  by a process  $p_i$  is always done in the critical section (line 6). This is due to the fact that  $p_i$  modifies  $TURN$  after it has acquired the underlying mutex lock and before it has released it.

**Remark 2** Let us observe that a process  $p_i$  can stop waiting at line 2 because it finds  $TURN = i$  while another process  $p_j$  increases  $TURN$  to  $((i + 1) \bmod n)$  because it does not see that  $FLAG[i]$  has been set to *up*. This situation is described in Fig. 2.21.

**Theorem 8** Assuming that the underlying mutex lock  $LOCK$  is deadlock-free, the algorithm described in Fig. 2.20 builds a starvation-free mutex lock.

*Proof* We first claim that, if at least one process invokes  $acquire\_mutex()$ , then at least one process invokes  $LOCK.acquire\_lock()$  (line 3) and enters the critical section.



**Fig. 2.21** A possible case when going from deadlock-freedom to starvation-freedom

Proof of the claim. Let us first observe that, if processes invoke  $LOCK.acquire\_lock()$ , one of them enters the critical section (this follows from the fact that the lock is deadlock-free). Hence,  $X$  being the non-empty set of processes that invoke  $acquire\_mutex()$ , let us assume by contradiction that no process of  $X$  terminates the **wait** statement at line 2. It follows from the waiting predicate that  $TURN \notin X$  and  $FLAG[TURN] = up$ . But,  $FLAG[TURN] = up$  implies  $TURN \in X$ , which contradicts the previous waiting predicate and concludes the proof of the claim.

Let  $p_i$  be a process that has invoked  $acquire\_mutex()$ . We have to show that it enters the critical section. Due to the claim, there is a process  $p_k$  that holds the underlying lock. If  $p_k$  is  $p_i$ , the theorem follows, hence let  $p_k \neq p_i$ . When  $p_k$  exits the critical section it executes line 6. Let  $TURN = j$  when  $p_k$  reads it. We consider two cases:

1.  $FLAG[j] = up$ . Let us observe that  $p_j$  is the only process that can write into  $FLAG[j]$  and that it will do so at line 5 when it exits the critical section. Moreover, as  $TURN = j$ ,  $p_j$  is not blocked at line 2 and consequently invokes  $LOCK.acquire\_lock()$  (line 3).

We first show that eventually  $p_j$  enters the critical section. Let us observe that all the processes which invoke  $acquire\_mutex()$  after  $FLAG[j]$  was set to  $up$  and  $TURN$  was set to  $j$  remain blocked at line 2 (Observation OB). Let  $Y$  be the set of processes that compete with  $p_j$  for the lock with  $y = |Y|$ . We have  $0 \leq y \leq n - 1$ . It follows from observation OB and the fact that the lock is deadlock-free that the number of processes that compete with  $p_j$  decreases from  $y$  to  $y - 1$ ,  $y - 2$ , etc., until  $p_j$  obtains the lock and executes line 5 (in the worst case,  $p_j$  is the last of the  $y$  processes to obtain the lock).

If  $p_i$  is  $p_j$  or a process that has obtained the lock before  $p_j$ , the theorem follows from the previous reasoning. Hence, let us assume that  $p_i$  has not obtained the lock. After  $p_j$  has obtained the lock, it eventually executes lines 5 and 6. As  $TURN = j$  and  $p_j$  sets  $FLAG[j]$  to  $down$ , it follows that  $p_j$  updates the register  $TURN$  to  $\ell = (j \bmod n) + 1$ . The previous reasoning, where  $k$  and  $j$  are replaced by  $j$  and  $\ell$ , is then applied again.

2.  $FLAG[j] = \text{down}$ . In this case,  $p_k$  updates  $TURN$  to  $\ell = (j \bmod n) + 1$ . If  $\ell = i$ , the previous reasoning (where  $p_j$  is replaced by  $p_i$ ) applies and it follows that  $p_i$  obtains the lock and enters the critical section.

If  $\ell \neq i$ , let  $p_{k'}$  be the next process that enters the critical section (due to the claim, such a process does exist). Then, the same reasoning as in case 1 applies, where  $k$  is replaced by  $k'$ .

As no process is skipped when  $TURN$  is updated when processes invoke `release_mutex()`, it follows from the combination of case 1 and case 2 that eventually case 1 where  $p_j = p_i$  applies and consequently  $p_i$  obtains the deadlock-free lock.  $\square$

**Fast starvation-free mutual exclusion** Let us consider the case where a process  $p_i$  wants to enter the critical section, while no other process is interested in entering it. We have the following:

- The invocation of `acquire_mutex(i)` requires at most three accesses to the shared memory: one to set the register  $FLAG[i]$  to *up*, one to read  $TURN$  and save it in a local variable *turn*, and one to read  $FLAG[turn]$ .
- Similarly, the invocation by  $p_i$  of `release_mutex(i)` requires at most four accesses to the shared memory: one to reset  $FLAG[i]$  to *down*, one to read  $TURN$  and save it in a local variable *turn*, one to read  $FLAG[turn]$ , and a last one to update  $TURN$ .

It follows from this observation that the stacking of the algorithm of Fig. 2.20 on top of the algorithm described in Fig. 2.14 (Sect. 2.1.7), which implements a deadlock-free fast mutex lock, provides a fast starvation-free mutex algorithm.

### 2.2.3 Fetch&Add

Let  $X$  be a shared register. The primitive  $X.\text{fetch\&add}()$  atomically adds 1 to  $X$  and returns the new value. (In some variants the value that is returned is the previous value of  $X$ . In other variants, a value  $c$  is passed as a parameter and, instead of being increased by 1,  $X$  becomes  $X + c$ .)

Such a primitive allows for the design of a simple starvation-free mutex algorithm. Its principle is to use a fetch&add atomic register to generate tickets with consecutive numbers and to allow a process to enter the critical section when its ticket number is the next one to be served.

An algorithm based on this principle is described in Fig. 2.22. The variable *TICKET* is used to generate consecutive ticket values, and the variable *NEXT* indicates the next winner ticket number. *TICKET* is initialized to 0, while *NEXT* is initialized to 1.

When it invokes `acquire_mutex()`, a process  $p_i$  takes the next ticket, saves it in its local variable *my\_turn*, and waits until its turn occurs, i.e., until (*my\_turn* = *NEXT*). An invocation of `release_mutex()` is a simple increase of the atomic register *NEXT*.

```

operation acquire_mutex() is
     $my\_turn \leftarrow TICKET.fetch\&add();$ 
    repeat no-op until ( $my\_turn = NEXT$ ) end repeat;
    return()
end operation.

operation release_mutex() is
     $NEXT \leftarrow NEXT + 1;$  return()
end operation.

```

**Fig. 2.22** Fetch&add-based mutual exclusion

Let us observe that, while  $NEXT$  is an atomic MWMR register, the operation  $NEXT \leftarrow NEXT + 1$  is not atomic. It is easy to see that no increase of  $NEXT$  can be missed. This follows from the fact that the increase statement  $NEXT \leftarrow NEXT + 1$  appears in the operation `release_mutex()`, which is executed by a single process at a time.

The mutual exclusion property follows from the uniqueness of each ticket number, and the starvation-freedom property follows from the fact that the ticket numbers are defined from a sequence of consecutive known values (here the increasing sequence of positive integers).

## 2.3 Mutex Without Atomicity

This section presents two mutex algorithms which rely on shared read/write registers weaker than read/write atomic registers. In that sense, they implement atomicity without relying on underlying atomic objects.

### 2.3.1 Safe, Regular, and Atomic Registers

The algorithms described in this section rely on *safe* registers. As shown here, safe registers are the weakest type of shared registers that we can imagine while being useful, in the presence of concurrency.

As an atomic register, a safe register (or a regular register)  $R$  provides the processes with a write operation denoted  $R.write(v)$  (or  $R \leftarrow v$ ), where  $v$  is the value that is written and a read operation  $R.read()$  (or  $local \leftarrow R$ , where *local* is a local variable of the invoking process). Safe, regular and atomic registers differ in the value returned by a read operation invoked in the presence of concurrent write operations.

Let us remember that the domain of a register is the set of values that it can contain. As an example, the domain of a binary register is the set  $\{0, 1\}$ .

**SWMR safe register** An SWMR *safe* register is a register whose read operation satisfies the following properties (the notion of an MWMR safe register will be introduced in Sect. 2.3.3):

- A read that is not concurrent with a write operation (i.e., their executions do not overlap) returns the current value of the register.
- A read that is concurrent with one (or several consecutive) write operation(s) (i.e., their executions do overlap) returns *any* value that the register can contain.

It is important to see that, in the presence of concurrent write operations, a read can return a value that has never been written. The returned value has only to belong to the register domain. As an example, let the domain of a safe register  $R$  be  $\{0, 1, 2, 3\}$ . Assuming that  $R = 0$ , let  $R.write(2)$  be concurrent with a read operation. This read can return 0, 1, 2, or 3. It cannot return 4, as this value is not in the domain of  $R$ , but can return the value 3, which has never been written.

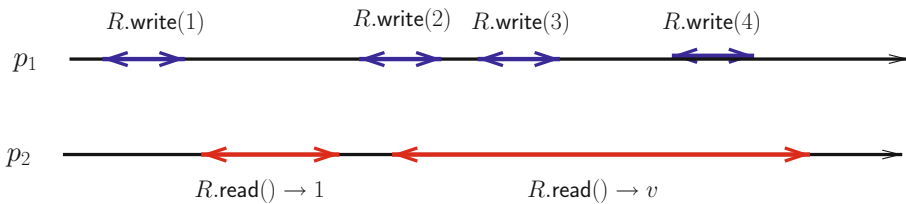
A *binary safe* register can be seen as modeling a flickering bit. Whatever its previous value, the value of the register can flicker during a write operation and stabilizes to its final value only when the write finishes. Hence, a read that overlaps with a write can arbitrarily return either 0 or 1.

**SWMR regular register** An SWMR *regular* register is an SWMR safe register that satisfies the following property. This property addresses read operations in the presence of concurrency. It replaces the second item of the definition of a safe register.

- A read that is concurrent with one or several write operations returns the value of the register before these writes or the value written by any of them.

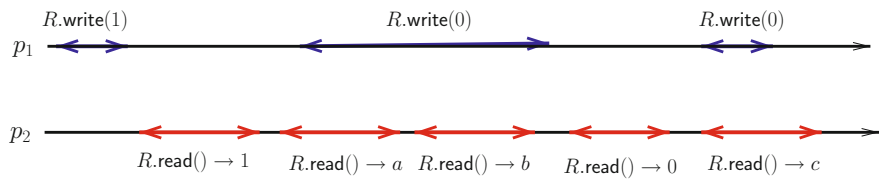
An example of a regular register  $R$  (whose domain is the set  $\{0, 1, 2, 3, 4\}$ ) written by a process  $p_1$  and read by a process  $p_2$  is described in Fig. 2.23. As there is no concurrent write during the first read by  $p_2$ , this read operation returns the current value of the register  $R$ , namely 1. The second read operation is concurrent with three write operations. It can consequently return any value in  $\{1, 2, 3, 4\}$ . If the register was only safe, this second read could return any value in  $\{0, 1, 2, 3, 4\}$ .

**Atomic register** The notion of an atomic register was defined in Sect. 2.1.1. Due to the total order on all its operations, an atomic register is more constrained (i.e., stronger) than a regular register.



**Fig. 2.23** An execution of a regular register





**Fig. 2.24** An execution of a register

**Table 2.1** Values returned by safe, regular and atomic registers

Value returned	$a$	$b$	$c$	Number of correct executions
Safe	1/0	1/0	1/0	8
Regular	1/0	1/0	0	4
Atomic	1	1/0	0	3
Atomic	0	0	0	

To illustrate the differences between safe, regular, and atomic, Fig. 2.24 presents an execution of a binary register  $R$  and Table 2.1 describes the values returned by the read operations when the register is safe, regular, and atomic. The first and third read by  $p_2$  are issued in a concurrency-free context. Hence, whatever the type of the register, the value returned is the current value of the register  $R$ .

- If  $R$  is safe, as the other read operations are concurrent with a write operation, they can return any value (i.e., 0 or 1 as the register is binary). This is denoted 0/1 in Table 2.1.

It follows that there are eight possible correct executions when the register  $R$  is safe for the concurrency pattern depicted in Fig. 2.24.

- If  $R$  is regular, each of the values  $a$  and  $b$  returned by the read operation which is concurrent with  $R.write(0)$  can be 1 (the value of  $R$  before the read operation) or 0 (the value of  $R$  that is written concurrently with the read operation).

Differently, the value  $c$  returned by the last read operation can only be 0 (because the value that is written concurrently does not change the value of  $R$ ).

It follows that there are only four possible correct executions when the register  $R$  is regular.

- If  $R$  is atomic, there are only three possible executions, each corresponding to a correct sequence of read and write invocations (“correct” means that the sequence respects the real-time order of the invocations and is such that each read invocation returns the value written by the immediately preceding write invocation).

### 2.3.2 The Bakery Mutex Algorithm

**Principle of the algorithm** The mutex algorithm presented in this section is due to L. Lamport (1974) who called it the mutex *bakery algorithm*. It was the first algorithm ever designed to solve mutual exclusion on top of non-atomic registers, namely on top of SWMR safe registers. The principle that underlies its design (inspired from bakeries where a customer receives a number upon entering the store, hence the algorithm name) is simple. When a process  $p_i$  wants to acquire the critical section, it acquires a number  $x$  that defines its priority, and the processes enter the critical section according to their current priorities.

As there are no atomic registers, it is possible that two processes obtain the same number. A simple way to establish an order for requests that have the same number consists in using the identities of the corresponding processes. Hence, let a pair  $\langle x, i \rangle$  define the identity of the current request issued by  $p_i$ . A total order is defined for the requests competing for the critical section as follows, where  $\langle x, i \rangle$  and  $\langle y, j \rangle$  are the identities of two competing requests;  $\langle x, i \rangle < \langle y, j \rangle$  means that the request identified by  $\langle x, i \rangle$  has priority over the request identified by  $\langle y, j \rangle$  where “ $<$ ” is defined as the lexicographical ordering on pairs of integers, namely

$$\langle x, i \rangle < \langle y, j \rangle \equiv (x < y) \vee ((x = y) \wedge (i < j)).$$

**Description of the algorithm** Two SWMR safe registers, denoted  $FLAG[i]$  and  $MY\_TURN[i]$ , are associated with each process  $p_i$  (hence these registers can be read by any process but written only by  $p_i$ ).

- $MY\_TURN[i]$  (which is initialized to 0 and reset to that value when  $p_i$  exits the critical section) is used to contain the priority number of  $p_i$  when it wants to use the critical section. The domain of  $MY\_TURN[i]$  is the set of non-negative integers.
- $FLAG[i]$  is a binary control variable whose domain is  $\{down, up\}$ . Initialized to *down*, it is set to *up* by  $p_i$  while it computes the value of its priority number  $MY\_TURN[i]$ .

The sequence of values taken by  $FLAG[i]$  is consequently the regular expression  $down(up, down)^*$ . The reader can verify that a binary safe register whose write operations of *down* and *up* alternate behaves as a regular register.

The algorithm of a process  $p_i$  is described in Fig. 2.25. When it invokes `acquire_mutex()`, process  $p_i$  enters a “doorway” (lines 1–3) in which it computes its turn number  $MY\_TURN[i]$  (line 2). To that end it selects a number greater than all  $MY\_TURN[j]$ ,  $1 \leq j \leq n$ . It is possible that  $p_i$  reads some  $MY\_TURN[j]$  while it is written by  $p_j$ . In that case the value obtained from  $MY\_TURN[j]$  can be any value. Moreover, a process informs the other processes that it is computing its turn value by raising its flag before this computation starts (line 1) and resetting it to *down* when it has finished (line 3). Let us observe that a process is never delayed while in the doorway, which means no process can direct another process to wait in the doorway.

```

operation acquire_mutex( $i$ ) is
(1)   $FLAG[i] \leftarrow up$ ;
(2)   $MY\_TURN[i] \leftarrow \max(MY\_TURN[1], \dots, MY\_TURN[n]) + 1$ ;
(3)   $FLAG[i] \leftarrow down$ ;
(4)  for each  $j \in \{1, \dots, n\} \setminus \{i\}$  do
(5)    wait ( $FLAG[j] = down$ );
(6)    wait ( $(MY\_TURN[j] = 0) \vee \langle MY\_TURN[i], i \rangle < \langle MY\_TURN[j], j \rangle$ )
(7)  end for;
(8)  return()
end operation.

operation release_mutex( $i$ ) is
(9)   $MY\_TURN[i] \leftarrow 0$ ; return()
end operation.

```

**Fig. 2.25** Lamport's bakery mutual exclusion algorithm

After it has computed its turn value, a process  $p_i$  enters a “waiting room” (lines 4–7) which consists of a **for** loop with one loop iteration per process  $p_j$ . There are two cases:

- If  $p_j$  does not want to enter the critical section, we have  $FLAG[j] = down \wedge MY\_TURN[j] = 0$ . In this case,  $p_i$  proceeds to the next iteration without being delayed by  $p_j$ .
- Otherwise,  $p_i$  waits until  $FLAG[j] = down$  (i.e., until  $p_j$  has finished to compute its turn, line 5) and then waits until either  $p_j$  has exited the critical section (predicate  $MY\_TURN[j] = 0$ ) or  $p_i$ 's current request has priority over  $p_j$ 's one (predicate  $(MY\_TURN[i], i) < (MY\_TURN[j], j)$ ).

When  $p_i$  has priority with respect to each other process (these priorities being checked in an arbitrary order, one after the other) it enters the critical section (line 8).

Finally, when it exits the critical section, the only thing a process  $p_i$  has to do is to reset  $MY\_TURN[i]$  to 0 (line 9).

**Remark: process crashes** Let us consider the case where a process may crash (i.e., stop prematurely). It is easy to see that the algorithm works despite this type of failure if, after a process  $p_i$  has crashed, its two registers  $FLAG[i]$  and  $MY\_TURN[i]$  are eventually reset to their initial values. When this occurs, the process  $p_i$  is considered as being no longer interested in the critical section.

**A first in first out (FIFO) order** As already indicated, the priority of a process  $p_i$  over a process  $p_j$  is defined from the identities of their requests, namely the pairs  $\langle MY\_TURN[i], i \rangle$  and  $\langle MY\_TURN[j], j \rangle$ . Moreover, let us observe that it is not possible to predict the values of these pairs when  $p_i$  and  $p_j$  compute concurrently the values of  $MY\_TURN[i]$  and  $MY\_TURN[j]$ .

Let us consider two processes  $p_i$  and  $p_j$  that have invoked `acquire_mutex()` and where  $p_i$  has executed its doorway part (line 2) before  $p_j$  has started executing its doorway part. We will see that the algorithm guarantees a FIFO order property defined as follows:  $p_i$  terminates its invocation of `acquire_mutex()` (and consequently enters the critical section) before  $p_j$ . This FIFO order property is an instance of the bounded bypass liveness property with  $f(n) = n - 1$ .

**Definitions** The following time instant definitions are used in the proof of Theorem 9. Let  $p_x$  be a process. Let us remember that, as the read and write operations on the registers are not atomic, they cannot be abstracted as having been executed instantaneously. Hence, when considering the execution of such an operation, its starting time and its end time are instead considered.

The number that appears in the following definitions corresponds to a line number (i.e., to a register operation). Moreover, “ $b$ ” stands for “beginning” while “ $e$ ” stands for “end”.

1.  $\tau_e^x(1)$  is the time instant at which  $p_x$  terminates the assignment  $FLAG[x] \leftarrow up$  (line 1).
2.  $\tau_e^x(2)$  is the time instant at which  $p_x$  terminates the execution of line 2. Hence, at time  $\tau_e^x(2)$  the non-atomic register  $MY\_TURN[x]$  contains the value used by  $p_x$  to enter the critical section.
3.  $\tau_b^x(3)$  is the time instant at which  $p_x$  starts the execution of line 3. This means that a process that reads  $FLAG[x]$  during the time interval  $[\tau_e^x(1)..\tau_b^x(3)]$  necessarily obtains the value *up*.
4.  $\tau_b^x(5, y)$  is the time instant at which  $p_x$  starts its last evaluation of the waiting predicate (with respect to  $FLAG[y]$ ) at line 5. This means that  $p_x$  has obtained the value *down* from  $FLAG[y]$ .
5. Let us notice that, as it is the only process which writes into  $MY\_TURN[x]$ ,  $p_x$  can save its value in a local variable. This means that the reading of  $MY\_TURN[x]$  entails no access to the shared memory. Moreover, as far as a register  $MY\_TURN[y]$  ( $y \neq x$ ) is concerned, we consider that  $p_x$  reads it once each time it evaluates the predicate of line 6.

$\tau_b^x(6, y)$  is the time instant at which  $p_x$  starts its last reading of  $MY\_TURN[y]$ . Hence, the value *turn* it reads from  $MY\_TURN[y]$  is such that  $(turn = 0) \vee \langle MY\_TURN[x], x \rangle < \langle turn, y \rangle$ .

**Terminology** Let us remember that a process  $p_x$  is “in the doorway” when it executes line 2. We also say that it “is in the bakery” when it executes lines 4–9. Hence, when it is in the bakery,  $p_x$  is in the waiting room, inside the critical section, or executing `release_mutex(x)`.

**Lemma 1** *Let  $p_i$  and  $p_j$  be two processes that are in the bakery and such that  $p_i$  entered the bakery before  $p_j$  enters the doorway. Then  $MY\_TURN[i] < MY\_TURN[j]$ .*

*Proof* Let  $turn_i$  be the value used by  $p_i$  at line 6. As  $p_i$  is in the bakery (i.e., executing lines 4–9) before  $p_j$  enters the doorway (line 2), it follows that  $MY\_TURN[i]$  was assigned the value  $turn_i$  before  $p_j$  reads it at line 2. Hence, when  $p_j$  reads the safe register  $MY\_TURN[i]$ , there is no concurrent write and  $p_j$  consequently obtains the value  $turn_i$ . It follows that the value  $turn_j$  assigned by  $p_j$  to  $MY\_TURN[j]$  is such that  $turn_j \geq turn_i + 1$ , from which the lemma follows.  $\square$

**Lemma 2** *Let  $p_i$  and  $p_j$  be two processes such that  $p_i$  is inside the critical section while  $p_j$  is in the bakery. Then  $\langle MY\_TURN[i], i \rangle < \langle MY\_TURN[j], j \rangle$ .*

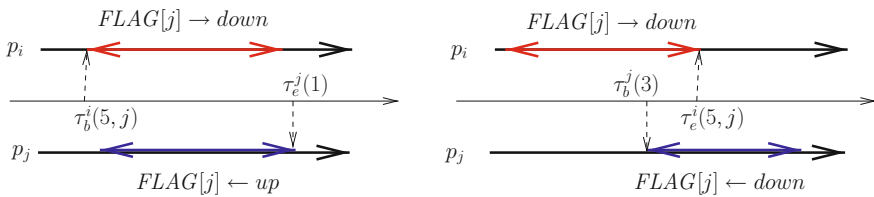
*Proof* Let us notice that, as  $p_j$  is inside the bakery, it can be inside the critical section.

As process  $p_i$  is inside the critical section, it has read *down* from  $FLAG[j]$  at line 5 (and exited the corresponding **wait** statement). It follows that, according to the timing of this read of  $FLAG[j]$  that returned the value *down* to  $p_i$  and the updates of  $FLAG[j]$  by  $p_j$  to *up* at line 1 or *down* at line 3 (the only lines where  $FLAG[j]$  is modified), there are two cases to consider (Fig. 2.26).

As  $p_i$  reads *down* from  $FLAG[j]$ , we have either  $\tau_b^i(5, j) < \tau_e^j(1)$  or  $\tau_e^i(5, j) > \tau_b^j(3)$  (see Fig. 2.26). This is because if we had  $\tau_b^i(5, j) > \tau_e^j(1)$ ,  $p_i$  would necessarily have read *up* from  $FLAG[j]$  (left part of the figure), and, if we had  $\tau_b^i(5, j) < \tau_b^j(3)$ ,  $p_i$  would necessarily have also read *up* from  $FLAG[j]$  (right part of the figure). Let us consider each case:

- Case 1:  $\tau_b^i(5, j) < \tau_e^j(1)$  (left part of Fig. 2.26). In this case process,  $p_i$  has entered the bakery before process  $p_j$  enters the doorway. It then follows from Lemma 1 that  $MY\_TURN[i] < MY\_TURN[j]$ , which proves the lemma for this case.
- Case 2:  $\tau_e^i(5, j) > \tau_b^j(3)$  (right part of Fig. 2.26). As  $p_j$  is sequential, we have  $\tau_e^j(2) < \tau_b^j(3)$  (P1). Similarly, as  $p_i$  is sequential, we also have  $\tau_b^i(5, j) < \tau_b^i(6, j)$  (P2). Combining (P1), (P2), and the case assumption, namely  $\tau_b^j(3) < \tau_b^i(5, j)$ , we obtain

$$\tau_e^j(2) < \tau_b^j(3) < \tau_e^i(5, j) < \tau_b^i(6, j);$$



**Fig. 2.26** The two cases where  $p_j$  updates the safe register  $FLAG[j]$

i.e.,  $\tau_e^j(2) < \tau_b^i(6, j)$  from which we conclude that the last read of  $MY\_TURN[j]$  by  $p_i$  occurred after the safe register  $MY\_TURN[j]$  obtained its value (say  $turn_j$ ).

As  $p_i$  is inside the critical section (lemma assumption), it exited the second **wait** statement because  $(MY\_TURN[j] = 0) \vee \langle MY\_TURN[i], i \rangle < \langle MY\_TURN[j], j \rangle$ .

Moreover, as  $p_j$  was in the bakery before  $p_i$  executed line 6 ( $\tau_e^j(2) < \tau_b^i(6, j)$ ), we have  $MY\_TURN[j] = turn_j \neq 0$ . It follows that we have  $\langle MY\_TURN[i], i \rangle < \langle MY\_TURN[j], j \rangle$ , which terminates the proof of the lemma.  $\square$

**Theorem 9** *Lamport's bakery algorithm satisfies mutual exclusion and the bounded bypass liveness property where  $f(n) = n - 1$ .*

*Proof* Proof of the mutual exclusion property. The proof is by contradiction. Let us assume that  $p_i$  and  $p_j$  ( $i \neq j$ ) are simultaneously inside the critical section. We have the following:

- As  $p_i$  is inside the critical section and  $p_j$  is inside the bakery, we can apply Lemma 2. We then obtain:  $\langle MY\_TURN[i], i \rangle < \langle MY\_TURN[j], j \rangle$ .
- Similarly, as  $p_j$  is inside the critical section and  $p_i$  is inside the bakery, applying Lemma 2, we obtain:  $\langle MY\_TURN[j], j \rangle < \langle MY\_TURN[i], i \rangle$ .

As  $i \neq j$ , the pairs  $\langle MY\_TURN[j], j \rangle$  and  $\langle MY\_TURN[i], i \rangle$  are totally ordered. It follows that each item contradicts the other, from which the mutex property follows.

Proof of the FIFO order liveness property. The proof shows first that the algorithm is deadlock-free. It then shows that the algorithm satisfies the bounded bypass property where  $f(n) = n - 1$  (i.e., the FIFO order as defined on the pairs  $\langle MY\_TURN[x], x \rangle$ ).

The proof that the algorithm is deadlock-free is by contradiction. Let us assume that processes have invoked `acquire_mutex()` and no process exits the waiting room (lines 4–7). Let  $Q$  be this set of processes. (Let us notice that, for any other process  $p_j$ , we have  $FLAG[j] = down$  and  $MY\_TURN[j] = 0$ .) As the number of processes is bounded and no process has to wait in the doorway, there is a time after which we have  $\forall j \in \{1, \dots, n\} : FLAG[j] = down$ , from which we conclude that no process of  $Q$  can be blocked forever in the **wait** statement of line 5.

By construction, the pairs  $\langle MY\_TURN[x], x \rangle$  of the processes  $p_x \in Q$  are totally ordered. Let  $\langle MY\_TURN[i], i \rangle$  be the smallest one. It follows that, eventually, when evaluated by  $p_i$ , the predicate associated with the **wait** statement of line 6 is satisfied for any  $j$ . Process  $p_i$  then enters the critical section, which contradicts the deadlock assumption and proves that the algorithm is deadlock-free.

To show the FIFO order liveness property, let us consider a pair of processes  $p_i$  and  $p_j$  that are competing for the critical section and such that  $p_j$  wins and after exiting the critical section it invokes `acquire_mutex(j)` again, executes its doorway, and enters the bakery. Moreover, let us assume that  $p_i$  is still waiting to enter the critical section. Let us observe that we are then in the context defined in Lemma 1:  $p_i$  and  $p_j$  are in the bakery and  $p_i$  entered the bakery before  $p_j$  enters the doorway.

We then have  $MY\_TURN[i] < MY\_TURN[j]$ , from which we conclude that  $p_j$  cannot bypass again  $p_i$ . As there are  $n$  processes, in the worst case  $p_i$  is competing with all other processes. Due to the previous observation and the fact that there is no deadlock, it can lose at most  $n - 1$  competitions (one with respect to each other process  $p_j$  (which enters the critical section before  $p_i$ ), which proves the bounded bypass liveness property with  $f(n) = n - 1$ .  $\square$

### 2.3.3 A Bounded Mutex Algorithm

This section presents a second mutex algorithm which does not require underlying atomic registers. This algorithm is due to A. Aravind (2011). Its design principles are different from the ones of the bakery algorithm.

**Principle of the algorithm** The idea that underlies the design of this algorithm is to associate a date with each request issued by a process and favor the competing process which has the oldest (smallest) request date. To that end, the algorithm ensures that (a) the dates associated with requests are increasing and (b) no two process requests have the same date.

More precisely, let us consider a process  $p_i$  that exits the critical section. The date of its next request (if any) is computed in advance when, just after  $p_i$  has used the critical section, it executes the corresponding `release_mutex()` operation. In that way, the date of the next request of a process is computed while this process is still “inside the critical section”. As a consequence, the sequence of dates associated with the requests is an increasing sequence of consecutive integers and no two requests (from the same process or different processes) are associated with the same date.

From a liveness point of view, the algorithm can be seen as ensuring a *least recently used* (LRU) priority: the competing process whose previous access to the critical section is the oldest (with respect to request dates) is given priority when it wants to enter the critical section.

**Safe registers associated with each process** The following three SWMR safe registers are associated with each process  $p_i$ :

- $FLAG[i]$ , whose domain is  $\{down, up\}$ . It is initialized to *up* when  $p_i$  wants to enter the critical section and reset to *down* when  $p_i$  exits the critical section.
- If  $p_i$  is not competing for the critical section, the safe register  $DATE[i]$  contains the (logical) date of its next request to enter the critical section. Otherwise, it contains the logical date of its current request.

$DATE[i]$  is initialized to  $i$ . Hence, no two processes start with the same date for their first request. As already indicated,  $p_i$  will compute its next date (the value that will be associated with its next request for the critical section) when it exits the critical section.

- $STAGE[i]$  is a binary control variable whose domain is  $\{0, 1\}$ . Initialized to 0, it is set to 1 by  $p_i$  when  $p_i$  sees  $DATE[i]$  as being the smallest date among the

```

operation acquire_mutex(i) is
(1)  FLAG[i] ← up;
(2)  repeat STAGE[i] ← 0;
(3)      wait ( $\forall j \neq i : (FLAG[j] = down) \vee (DATE[i] < DATE[j])$ );
(4)      STAGE[i] ← 1
(5)  until  $\forall j \neq i : (STAGE[j] = 0)$  end repeat;
(6)  return()
end operation.

operation release_mutex(i) is
(7)  DATE[i] ←  $\max(DATE[1], \dots, DATE[n]) + 1$ ;
(8)  STAGE[i] ← 0;
(9)  FLAG[i] ← down;
(10) return()
end operation.

```

**Fig. 2.27** Aravind's mutual exclusion algorithm

dates currently associated with the processes that it perceives as competing for the critical section. The sequence of successive values taken by *STAGE*[*i*] (including its initial value) is defined by the regular expression  $0((0, 1)^+, 0)^*$ .

**Description of the algorithm** Aravind's algorithm is described in Fig. 2.27. When a process  $p_i$  invokes *acquire\_mutex*(*i*) it first sets its flag *FLAG*[*i*] to *up* (line 1), thereby indicating that it is interested in the critical section. Then, it enters a loop (lines 2–5), at the end of which it will enter the critical section. The loop body is made up of two stages, denoted 0 and 1. Process  $p_i$  first sets *STAGE*[*i*] to 0 (line 2) and waits until the dates of the requests of all the processes that (from its point of view) are competing for the critical section are greater than the date of its own request. This is captured by the predicate  $(\forall j \neq i : (FLAG[j] = down) \vee (DATE[i] < DATE[j]))$ , which is asynchronously evaluated by  $p_i$  at line 3. When, this predicate becomes true,  $p_i$  proceeds to the second stage by setting *STAGE*[*i*] to 1 (line 4).

Unfortunately, having the smallest request date (as asynchronously checked at line 3 by a process  $p_i$ ) is not sufficient to ensure the mutual exclusion property. More precisely, several processes can simultaneously be at the second stage. As an example let us consider an execution in which  $p_i$  and  $p_j$  are the only processes that invoke *acquire\_mutex*() and are such that  $DATE[i] = a < DATE[j] = b$ . Moreover,  $p_j$  executes *acquire\_mutex*() before  $p_i$  does. As all flags (except the one of  $p_j$ ) are equal to *down*,  $p_j$  proceeds to stage 1 and, being alone in stage 1, exits the loop and enters the critical section. Then,  $p_i$  executes *acquire\_mutex*(). As  $a < b$ ,  $p_i$  does not wait at line 3 and is allowed to proceed to the second stage (line 4). This observation motivates the predicate that controls the end of the **repeat** loop (line 5). More precisely, a process  $p_i$  is granted the critical section only if it is the only process which is at the second stage (as captured by the predicate  $\forall j \neq i : (STAGE[j] = 0)$  evaluated by  $p_i$  at line 5).



Finally, when a process  $p_i$  invokes `release_mutex(i)`, it resets its control registers  $STAGE[i]$  and  $FLAG[i]$  to their initial values (0 and *down*, respectively). Before these updates,  $p_i$  benefits from the fact that it is still “inside the critical section” to compute the date of its next request and save it in  $DATE[i]$  (line 7). It is important to see that no process  $p_j$  modifies  $DATE[j]$  while  $p_i$  reads the array  $DATE[1..n]$ . Consequently, despite the fact that the registers are only SWMR safe registers (and not atomic registers), the read of any  $DATE[j]$  at line 7 returns its exact value. Moreover, it also follows from this observation that no two requests have the same date and the sequence of dates used by the algorithm is the sequence of natural integers.

**Theorem 10** *Aravind’s algorithm (described in Fig. 2.27) satisfies mutual exclusion and the bounded bypass liveness property where  $f(n) = n - 1$ .*

*Proof* The proof of the mutual exclusion property is by contradiction. Let us assume that both  $p_i$  and  $p_j$  ( $i \neq j$ ) are in the critical section.

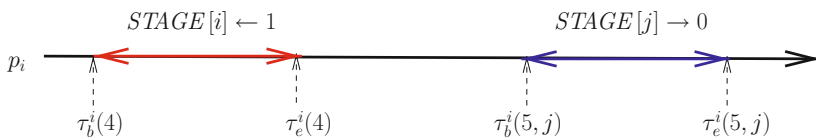
Let  $\tau_b^i(4)$  (or  $\tau_e^i(4)$ ) be the time instant at which  $p_i$  starts (or terminates) writing  $STAGE[i]$  at line 4 and  $\tau_b^i(5, j)$  (or  $\tau_e^i(5, j)$ ) be the time instant at which  $p_i$  starts (or terminates) reading  $STAGE[j]$  for the last time at line 5 (before entering the critical section). These time instants are depicted in Fig. 2.28. By exchanging  $i$  and  $j$  we obtain similar notations for time instants associated with  $p_j$ .

As  $p_i$  is inside the critical section, it has read 0 from  $STAGE[j]$  at line 5 and consequently we have  $\tau_b^i(5, j) < \tau_e^j(4)$  (otherwise,  $p_i$  would necessarily have read 1 from  $STAGE[j]$ ). Moreover, as  $p_i$  is sequential we have  $\tau_e^i(4) < \tau_b^i(5, j)$ , and as  $p_j$  is sequential, we have  $\tau_e^j(4) < \tau_b^j(5, i)$ . Piecing together the inequalities, we obtain

$$\tau_e^i(4) < \tau_b^i(5, j) < \tau_e^j(4) < \tau_b^j(5, i),$$

from which we conclude  $\tau_e^i(4) < \tau_b^j(5, i)$ , i.e., the last read of  $STAGE[i]$  by  $p_j$  at line 5 started after  $p_i$  had written 1 into it. Hence, the last read of  $STAGE[i]$  by  $p_j$  returned 1 which contradicts the fact that it is inside the critical section simultaneously with  $p_i$ . (A similar reasoning shows that, if  $p_j$  is inside the critical section,  $p_i$  cannot be.)

Before proving the liveness property, let us notice that at most one process at a time can modify the array  $DATE[1..n]$ . This follows from the fact that the algorithm satisfies the mutual exclusion property (proved above) and line 7 is executed by a process  $p_i$  before it resets  $STAGE[i]$  to 0 (at line 8), which is necessary to allow



**Fig. 2.28** Relevant time instants in Aravind’s algorithm

another process  $p_j$  to enter the critical section (as the predicate of line 5 has to be true when evaluated by  $p_j$ ). It follows from the initialization of the array  $DATE[1..n]$  and the previous reasoning that no two requests can have the same date and the sequence of dates computed in mutual exclusion at line 7 by the processes is the sequence of natural integers (Observation OB).

As in the proof of Lamport's algorithm, let us first prove that there is no deadlock. Let us assume (by contradiction) that there is a non-empty set of processes  $Q$  that have invoked `acquire_mutex()` and no process succeeds in entering the critical section. Let  $p_i$  be the process of  $Q$  with the smallest date. Due to observation OB, there is a single process  $p_i$ . It then follows that, after some finite time,  $p_i$  is the only process whose predicate at line 3 is satisfied. Hence, after some time,  $p_i$  is the only process such that  $STAGE[i] = 1$ , which allows it to enter the critical section. This contradicts the initial assumption and proves the deadlock-freedom property.

As a single process at a time can modify its entry of the array  $DATE$ , it follows that a process  $p_j$  that exits the critical section updates its register  $DATE[j]$  to a value greater than all the values currently kept in  $DATE[1..n]$ . Consequently, after  $p_j$  has executed line 7, all the other processes  $p_i$  which are currently competing for the critical section are such that  $DATE[i] < DATE[j]$ . Hence, as we now have  $(FLAG[i] = up) \wedge (DATE[i] < DATE[j])$ , the next request (if any) issued by  $p_j$  cannot bypass the current request of  $p_i$ , from which the starvation-freedom property follows.

Moreover, it also follows from the previous reasoning that, if  $p_i$  and  $p_j$  are competing and  $p_j$  wins, then as soon as  $p_j$  has exited the critical section  $p_i$  has priority over  $p_j$  and can no longer be bypassed by it. This is nothing else than the bounded bypass property with  $f(n) = n - 1$  (which defines a FIFO order property).  $\square$

**Bounded mutex algorithm** Each safe register  $MY\_TURN[i]$  of Lamport's algorithm and each safe register  $DATE[i]$  of Aravind's algorithm can take arbitrary large values. It is shown in the following how a simple modification of Aravind's algorithm allows for bounded dates. This modification relies on the notion of an MWMR safe register.

**MWMR safe register** An *MWMR safe* register is a safe register that can be written and read by several processes. When the write operations are sequential, an MWMR safe register behaves as an SWMR safe register. When write operations are concurrent, the value written into the register is any value of its domain (not necessarily a value of a concurrent write).

Said differently, to be meaningful, an algorithm based on MWMR safe registers has to prevent write operations on an MWMR safe register from being concurrent in order for the write operations to be always meaningful. The behavior of an MWMR safe register is then similar to the behavior of an SWMR safe register in which the "single writer" is implemented by several processes that never write at the same time.

**From unbounded dates to bounded dates** Let us now consider that each safe register  $DATE[i]$ ,  $1 \leq i \leq n$ , is an MWMR safe register: any process  $p_i$  can write any register  $DATE[j]$ . MWMR safe registers allow for the design of a (particularly

simple) bounded mutex algorithm. The domain of each register  $DATE[j]$  is now  $[1..N]$  where  $N \geq 2n$ . Hence, all registers are safe and have a bounded domain. In the following we consider  $N = 2n$ . A single bit is needed for each safe register  $FLAG[j]$  and each safe register  $STAGE[j]$ , and only  $\lceil \log_2 N \rceil$  bits are needed for each safe register  $DATE[j]$ .

In a very interesting way, no statement has to be modified to obtain a bounded version of the algorithm. A single new statement has to be added, namely the insertion of the following line 7' between line 7 and line 8:

(7') **if** ( $DATE[i] \geq N$ ) **then for all**  $j \in [1..n]$  **do**  $DATE[j] \leftarrow j$  **end for end if**.

This means that, when a process  $p_i$  exiting the critical section updates its register  $DATE[i]$  and this update is such that  $DATE[i] \geq N$ ,  $p_i$  resets all date registers to their initial values. As for line 7, this new line is executed before  $STAGE[i]$  is reset to 0 (line 8), from which it follows that it is executed in mutual exclusion and consequently no two processes can concurrently write the same MWMR safe register  $DATE[j]$ . Hence, the MWMR safe registers are meaningful.

Moreover, it is easy to see that the date resetting mechanism is such that each date  $d$ ,  $1 \leq d \leq n$ , is used only by process  $p_d$ , while each date  $d$ ,  $n + 1 \leq d \leq 2n$  can be used by any process. Hence,  $\forall d \in \{1, \dots, n\}$  we have  $DATE[d] \in \{d, n + 1, n + 2, \dots, 2n\}$ .

**Theorem 11** *When considering Aravind's mutual exclusion algorithm enriched with line 7' with  $N \geq 2n$ , a process encounters at most one reset of the array  $DATE[1..n]$  while it is executing `acquire_mutex()`.*

*Proof* Let  $p_i$  be a process that executes `acquire_mutex()` while a reset of the array  $DATE[1..n]$  occurs. If  $p_i$  is the next process to enter the critical section, the theorem follows. Otherwise, let  $p_j$  be the next process which enters the critical section. When  $p_j$  exits the critical section,  $DATE[j]$  is updated to  $\max(DATE[1], \dots, DATE[n]) + 1 = n + 1$ . We then have  $FLAG[i] = up$  and  $DATE[i] < DATE[j]$ . It follows that, if there is no new reset,  $p_j$  cannot enter again the critical section before  $p_i$ .

In the worst case, after the reset, all the other processes are competing with  $p_i$  and  $p_i$  is  $p_n$  (hence,  $DATE[i] = n$ , the greatest date value after a reset). Due to line 3 and the previous observation, each other process  $p_j$  enters the critical section before  $p_i$  and  $\max(DATE[1], \dots, DATE[n])$  becomes equal to  $n + (n - 1)$ . As  $2n - 1 < 2n \leq N$ , none of these processes issues a reset. It follows that  $p_i$  enters the critical section before the next reset. (Let us notice that, after the reset, the invocation issued by  $p_i$  can be bypassed only by invocations (pending invocations issued before the reset or new invocations issued after the reset) which have been issued by processes  $p_j$  such that  $j < i$ ).  $\square$

The following corollary is an immediate consequence of the previous theorem.

**Corollary 2** *Let  $N \geq 2n$ . Aravind's mutual exclusion algorithm enriched with line 7' satisfies the starvation-freedom property.*

(Different progress conditions that this algorithm can ensure are investigated in Exercise 6.)

Bounding the domain of the safe registers has a price. More precisely, the addition of line 7' has an impact on the maximal number of bypasses which can now increase up to  $f(n) = 2n - 2$ . This is because, in the worst case where all the processes always compete for the critical section, before it is allowed to access the critical section, a process can be bypassed  $(n - 1)$  times just before a reset of the array *DATE* and, due to the new values of *DATE*[1..*n*], it can again be bypassed  $(n - 1)$  times just after the reset.

## 2.4 Summary

This chapter has presented three families of algorithms that solve the mutual exclusion problem. These algorithms differ in the properties of the base operations they rely on to solve mutual exclusion.

Mutual exclusion is one way to implement atomic objects. Interestingly, it was shown that implementing atomicity does not require the underlying read and write operations to be atomic.

## 2.5 Bibliographic Notes

- The reader will find surveys on mutex algorithms in [24, 231, 262]. Mutex algorithms are also described in [41, 146].
- Peterson's algorithm for two processes and its generalization to  $n$  processes are presented in [224].

The first tournament-based mutex algorithm is due to G.L. Peterson and M.J. Fischer [227].

A variant of Peterson's algorithm in which all atomic registers are SWMR registers due to J.L.W. Kessels is presented in [175].

- The contention-abortable mutex algorithm is inspired from Lamport's fast mutex algorithm [191]. Fischer's synchronous algorithm is described in [191].

Lamport's fast mutex algorithm gave rise to the splitter object as defined in [209].

The notion of fast algorithms has given rise to the notion of *adaptive* algorithms (algorithms whose cost is related to the number of participating processes) [34].

- The general construction from deadlock-freedom to starvation-freedom that was presented in Sect. 2.2.2 is from [262]. It is due to Y. Bar-David.

- The notions of safe, regular, and atomic read/write registers are due to L. Lamport. They are presented and investigated in [188, 189]. The first intuition on these types of registers appears in [184].

It is important to insist on the fact that “non-atomic” does not mean “arbiter-free”. As defined in [193], “An arbiter is a device that makes a discrete decision based on a continuous range of values”. Binary arbiters are the most popular. Actually, the implementation of a safe register requires an arbiter. The notion of arbitration-free synchronization is discussed in [193].

- Lamport’s bakery algorithm is from [183], while Aravind’s algorithm and its bounded version are from [28].
- A methodology based on model-checking for automatic discovery of mutual exclusion algorithms has been proposed by Y. Bar-David and G. Taubenfeld [46]. Interestingly enough, this methodology is both simple and computationally feasible. New algorithms obtained in this way are presented in [46, 262].
- Techniques (and corresponding algorithms) suited to the design of locks for NUMA and CC-NUMA architectures are described in [86, 200]. These techniques take into account non-uniform memories and caching hierarchies.
- A *combiner* is a thread which, using a coarse-grain lock, serves (in addition to its own synchronization request) active requests announced by other threads while they are waiting by performing some form of spinning. Two implementations of such a technique are described in [173]. The first addresses systems that support coherent caches, whereas the second works better in cacheless NUMA architectures.

## 2.6 Exercises and Problems

1. Peterson’s algorithm for two processes uses an atomic register denoted *TURN* that is written and read by both processes. Design a two-process mutual exclusion algorithm (similar to Peterson’s algorithm) in which the register *TURN* is replaced by two SWMR atomic registers *TURN*[*i*](which can be written only by  $p_i$ ) and *TURN*[*j*](which can be written only by  $p_j$ ). The algorithm will be described for  $p_i$  where  $i \in \{0, 1\}$  and  $j = (i + 1) \bmod 2$ .

Solution in [175].

2. Considering the tournament-based mutex algorithm, show that if the base two-process mutex algorithm is deadlock-free then the  $n$ -process algorithm is deadlock-free.

3. Design a mutex starvation-free algorithm whose cost (measured by the number of shared memory accesses) depends on the number of processes which are currently competing for the critical section. (Such an algorithm is called *adaptive*.)

Solutions in [23, 204, 261].

4. Design a fast deadlock-free mutex synchronous algorithm. “Fast” means here that, when no other process is interested in the critical section when a process  $p$  requires it, then process  $p$  does not have to execute the `delay()` statement.

Solution in [262].

5. Assuming that all registers are atomic (instead of safe), modify Lamport’s bakery algorithm in order to obtain a version in which all registers have a bounded domain.

Solutions in [171, 261].

6. Considering Aravind’s algorithm described in Fig. 2.27 enriched with the reset line (line 7’):

- Show that the safety property is independent of  $N$ ; i.e., whatever the value of  $N$  (e.g.,  $N = 1$ ), the enriched algorithm allows at most one process at a time to enter the critical section.
- Let  $x \in \{1, \dots, n - 1\}$ . Which type of liveness property is satisfied when  $N = x + n$  (where  $n$  is the number of processes).
- Let  $I = \{i_1, \dots, i_z\} \subseteq \{1, \dots, n\}$  be a predefined subset of process indexes. Modify Aravind’s algorithm in such a way that starvation-freedom is guaranteed only for the processes  $p_x$  such that  $x \in I$ . (Let us notice that this modification realizes a type of priority for the processes whose index belong to  $I$  in the sense that the algorithm provides now processes with two types of progress condition: the invocations of `acquire_mutex()` issued by any process  $p_x$  with  $x \in I$  are guaranteed to terminate, while they are not if  $x \notin I$ .) Modify Aravind’s algorithm so that the set  $I$  can be dynamically updated (the main issue is the definition of the place where such a modification has to be introduced).



<http://www.springer.com/978-3-642-32026-2>

Concurrent Programming: Algorithms, Principles, and  
Foundations

Raynal, M.

2013, XXXII, 516 p., Hardcover

ISBN: 978-3-642-32026-2