Synchronisation algorithms

- Lecture Notes -

Elad Aigner-Horev

July 2, 2017

Contents

§1. Introduction

We distinguish between two types of synchronisation needs for processes. The first typically arises in a system that has a limited number of resources (like our computers) forcing process to compete for the resources. A situation in which there is competition between the interested parties is said to be a situation with *contention*; otherwise we say it is *contention-free*. The second type of synchronisation arises from processes having to cooperate in order to attain a certain goal. A classical paradigm here is the so called *producer-consumer paradigm* where one process (or a group of processes) produce certain items while another process (or another group of processes) consume the produced items. One should not attempt to come up with a strict classification of synchronisation problems as to have those fit into labels such as "competition-driven synchronisation" and "cooperation-driven synchronisation". Most likely that a given synchronisation problem would include elements of each flavour.

Consider two processes namely `A` and `B` each in charge of loading text files into an array based queue `Q` maintained by, say, a printer. The printer peels off files from the head of the queue for printing. An integer called `last` is maintained and it contains the number of the array slot into which the next file should be inserted at. The processes `A` and `B` share the queue and the variable `last`. In order to load a file to the printer, a process places a file `Q[last]` and then increments `last` by one. The following scenario is then possible which is sufficient in order to motivate the need for synchronisation algorithms.

1. Suppose `last = 10`.
2. Process `A` reads the value of last and records that the value of last is 10.
3. Before `A` puts its file process `B` gets running time places it file in the 10th slot as the value of last has yet to be changed by `A`.
4. A gets running time again recalls that it already read last and that the value is 10 and proceeds to put its file in the 10th slot.

The outcome is that `A` runs over `B`'s file.

Definition 1.1. **(Race conditions)** *We say a program contains* race conditions *if the outcome of the program depends on the order of execution of the processes involved in it.*

The first step towards avoiding race conditions is to detect the parts of the code (involving resources) over which processes compete as these areas will be critical for us.

Definition 1.2. (Critical section) *Parts of a program in which shared resources are accessed.*

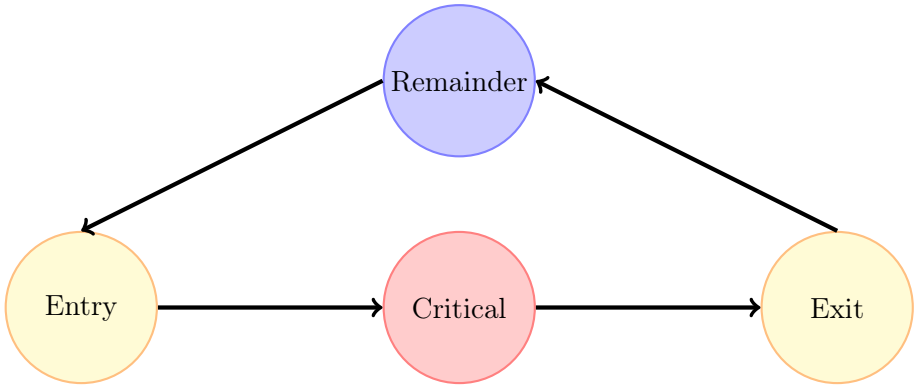
One may to view a synchronisation algorithm is as a type of *locking mechanism* used to coordinate access to critical sections. Our aim then is to design locking mechanisms through which to exert control over critical section. In particular we would like to attain as much as possible form the following.

1. Mutual exclusion. No two processes can reside in a critical section at the same time.
2. Avoid starvation. No process waits forever (i.e., starves) to enter a critical section.

Two matters we cannot assume.

1. A process can be suspended by the OS inside a critical section.
2. There is no control over the speed in which processes operate in. We may not assume that the critical section is "done fast".

§1.1 Synchronisation algorithms: definition. A synchronisation algorithm consists of two parts an *entry code* and an *exit code*. Processes that use a given synchronisation algorithm over a given critical section must consent to executing the entry code of the algorithm every time these want to be given access to the critical section and must execute the exit code of the algorithm every time they leave the critical section. In this way the entry and exit codes of the algorithm in a sense wrap the code of the critical section. We thus partition the code of a program in this way into four parts, the entry code, the critical section, the exist code, and the remainder of the code.



One may regard synchronisation algorithm as *locks* that one puts over critical sections in order to control access to these sections in the code. Throughout, the algorithm we present are unable to prevent programmers from abusing them. For instance we would very much like to attain the property that processes outside a critical section do not effect processes inside a critical section. None of the algorithms we shall can provide such a promise. It is the responsibility of the programmer to use synchronisation algorithm the way these were intended to be used and then rip their benefits.

§1.2 The OS scheduler: the enemy. Throughout we shall design synchronisation algorithms under the assumption that the OS scheduler does everything it can to sabotage our efforts of reaching the goals stated above that we seek to have for critical sections. However, an assumption that an OS scheduler does not let specific processes to run is absurd and in particular does not allow for any algorithm to be proposed. So our working assumption throughout is that

$$\text{every process in the system eventually gets to run.} \tag{1.3}$$

We however make no assumption regarding when it will be allowed to run or for how long.

§1.3 Properties of synchronisation algorithms. A synchronisation algorithm is said to be *safe* if it attains/provides mutual exclusion over the critical section. By *liveness* of a synchronisation algorithm we refer to the manner in which it attains progress. Here are the most basic liveness properties we shall consider.

Definition 1.4. *A synchronisation algorithm is said to be* deadlock-free *if whenever a set of processes of size at least* 2 *attempts to enter the critical*

section using the entry code of the algorithm at least one eventually succeeds to enter the critical section.

Definition 1.5. *A synchronisation algorithm is said to be* starvation-free *if whenever a prescribed process attempts to enter the critical section using the entry code of the algorithm it eventually succeeds in doing so.*

Observation 1.6. *Starvation-freedom implies deadlock-freedom.*

The reason we insist in Definition 1.4 on at least 2 processes and not allow the set to have only one process as in the latter case the definition reduces to that of starvation-freedom and we wish to have these two properties distinguished from one another.

§2. Early Synchronisation patterns

In order to synchronise processes one first needs a piece of code that causes a process to wait for an event to occur before proceeding. The first mechanism we shall use to attain this goal is that of a *spin lock*.

Definition 2.1. *By* spin lock *we mean a piece of code that causes a process to wait for an event (i.e., attaining the lock) by repeatedly checking whether the lock is available.*

Put another way a spin lock is given `while(<condition not met>){}`; we abbreviate this code snippet by writing `await(<condition is met>`. Another name for spin locks used in the literature is *busy waiting*. The main advantage of busy waiting is that they lead to very few context switches with the kernel as no system calls are involved in their implementation.

§2.1 Lock variables. Let us consider the following synchronisation algorithm for two processes namely `A` and `B`; both executing the same code seen in Listing 1.

```
1 Shared: int lock
2 Initially: lock = 0
3
4 await(lock = 0)
5 lock = 1
6 <CS>
7 lock = 0
```

Listing 1: Using lock variables

Claim 2.2. *The algorithm seen in Listing 1 does not provide mutual exclusion.*

Proof. The following scheduling scenario establishes the claim.

1. `A` reads `lock == 0` and is preempted by the OS prior to setting `lock` to one.
2. `B` reads `lock == 0`, sets it to one, enters the CS, and is preempted by the OS while in the CS.
3. `A` continues execution; as `A` already read `lock ==0` it enters the CS.

■

Claim 2.3. *The algorithm seen in Listing 1 is deadlock-free.*

Proof. Assume for the sake of contradiction that both `A` and `B` are stuck in the entry code (i.e., in line 4 in the listing) forever. As there are only two processes and both are in the entry code of the algorithm it follows that `lock` is equal to zero. Any of the two processes that completes a reading of `lock` is able to pass the spin lock and enter the CS. ■

Claim 2.4. *The algorithm seen in Listing 1 is not starvation-free.*

Proof. Starvation for `A`, say, arises in the following scenario.

1. `B` is inside the CS and `A` is stuck in the spin lock.
2. `B` exits the CS sets `lock` to zero, attempts reentry and sets `lock` to one and enters the CS and then preempted by the OS.
3. `A` gets running time and reads that the value of `lock` is one.

Repeating the above scenario leads to the starvation of `A`. ■

For future reference we shall refer to the scenario seen in Claim 2.4 as *one process is faster than the other*.

§2.2 Strict alternation. The following algorithm (see Listings 2 and 3 below) for two contending processes has each process executing a different code. The pattern of synchronisation used is that of *strict alternation*; in that the processes take turns at entering the CS. In the following algorithm the processes share an integer variable called `turn` initialised to zero that is used to keep track of whose turn is it to enter the CS.

```
1 await(turn = 0)
2 <CS>
3 turn = 1
```

Listing 2: Process 0

```
1 await(turn = 1)
2 <CS>
3 turn = 0
```

Listing 3: Process 1

Claim 2.5. *The strict alternation algorithm provides mutual exclusion for two contending processes.*

Proof. If both processes are in the CS at the same point in time then at that point in time `turn` has to assume both values zero and one. ■

Claim 2.6. *The strict alternation algorithm is deadlock-free for two contending processes.*

Proof. If both processes are stuck forever in their entry code then `turn` has to assume both values zero and one. ■

Claim 2.7. *The strict alternation algorithm is not starvation-free for two contending processes.*

Proof. Suppose `turn` is equal to zero yet process 0 is stuck forever in its remainder code (as it is no longer interested in entering the CS). ■

§2.3 Performance degradation due to busy waiting. Assume that two processes are using a synchronisation algorithm that employs busy waiting (i.e., a spin lock) in its entry code (as we have seen so far). Assume that there are two processes namely `L` and `H` using the algorithm. `L` is positioned inside the CS and `H` is attempting to acquire the lock via busy waiting. Assume further that these two processes also have priorities. In particular, let us assume that `H` has the higher priority and `L` has the lower priority. The scheduling policy of the OS is to always allow the process with the highest priority which is ready to run first.

Under these assumptions `H` is always ready and always gets running time from the OS hence `L` never gets to run and so under these extreme as-

sumptions we do in fact reach a deadlock. Such a process scheduling policy negates (1.3) and that under (1.3) one does not reach a deadlock; nevertheless performance most definitely be degraded.

While in the "real world" spin locks are used and are perhaps the flavour of the day for many professional programmers. This "culture" stems from putting trust in the OS scheduling algorithm following the assumption that this algorithm is "reasonable". This is not the approach we take.

§3. Synchronisation using atomic operations

Virtually every OS (of any interest) supports a limited array of more powerful operations for which the OS guarantees that those operations will not be interrupted during their execution.

Definition 3.1. *An operation is said to be* atomic *if it is guaranteed that any process executing the operation will never be preempted during the execution of the operation.*

§3.1 Test and set. The Test and Set operation (`TAS`, hereafter) receives an integer as a parameter; records its value, sets it to one and returns the former value just recorded. The aim of the operation then is to detect that a change to the value of variable occurred.

```
1 TAS(w)
2 {
3     prev:= w
4     w:= 1
5     return pre
6 }
```

Listing 4: Atomic Test and Set operation

The following algorithm uses `TAS` to significantly enhance the algorithm seen in Listing 1 in order to synchronise n processes over a CS where $n > 2$ is possible.

```
1 Shared: int lock
2
3 Initially: lock = 0
4
5 while(TAS(lock) ==1) {}
6 <CS>
7 lock = 0
```

Listing 5: `TAS` synchronisation algorithm

Claim 3.2. *The `TAS` algorithm provides mutual exclusion and is deadlock-free for any number of processes.*

Proof. The atomicity of `TAS` ensures that the first process to invoke it is guaranteed to enter the CS. ■

Nevertheless the atomicity of `TAS` is not strong enough to ensure starvation-freedom.

Claim 3.3. *The `TAS` algorithm is not starvation-free for two processes.*

Proof. Let `A` and `B` be two processes engaging the algorithm. The scenario `B` is faster than `A` establishes the claim. ■

How did we reach the scenario presented in the proof of Claim 3.3? Consider the following exhaustive case analysis. Let `A` spin in the spin lock. We consider the possible locations for process `B`.

1. `B` cannot be stuck forever in its remainder code. For if so the last time `B` left the CS (and this includes the option that it never entered the CS) it would set `lock =0` allowing `A` to pass the spin lock eventually.

2. As we already proved that the algorithm is deadlock-free, `B` cannot be stuck forever in its spin lock as well.

The sole remaining option of the position of `B` is that `B` is faster than `A`.

§3.1.1 ***Java implementation of the TAS algorithm.*** To simulate the TAS command in Java we use the `getAndSet()` method of the `AtomicBoolean` class found at the java.util.concurrent.atomic package which we urge the reader to become familiar with. In addition to this we mark our algorithm (encapsulated as a class here) using the interface `Lock` which can be found at the java.util.concurrent.locks package which we also encourage the reader to become familiar with.

Throughout these notes whenever Java code is offered we forgo the formalities of having to specify properly all packages used; this is left to the reader.

```
1
2 public class TASLock implements Lock {
```

```
3     AtomicBoolean state =
4                       new AtomicBoolean(false);
5
6     public void lock() {
7           while (state.getAndSet(true)){}
8     }
9
10    public void unlock() {
11          state.set(false)
12    }
13  }
```

Listing 6: The TAS lock

§3.1.2 ***The test-and-test-and-set algorithm.*** In § 2.3 we demonstrated how the use of spin locks can lead to performance degradation. The scenario provided in § 2.3 resorted to using priorities for processes to showcase degradation. In this section we revisit this issue only this time we omit the notion of priorities for processes by considering the algorithm presented in Listing 5 and showcase how the use of such an algorithm can lead to performance degradation.

A traditional architecture of a laptop or a desktop computer is that of a *shared bus* architecture. Here, we have a single line of communication called the bus amongst all hardware devices of the machine. In particular, the CPU, the hard drive, and the RAM are all connected to this bus and use it to transfer messages between themselves.

In order to reduce bus traffic and thus to optimise the time required for reading the values of variables (or any memory location for that matter) the CPU maintains a small cache memory separate from the RAM. The values kept in the cache are always valid. This is so mainly because each time a variable is written its cached value is discarded and a RAM write request is sent via the bus. The fact that many CPU caches are not "smart" enough to not discard their values upon write instructions that do not alter the value of a given variable is a major consideration for us.

When a process attempts to read a certain variable, the CPU, prior to engaging the RAM , first checks whether the value of the variable can be found in the cache. If the variable is in the cache we say that the CPU has a *cache hit* allowing it to not check the value of the variable in the RAM. The advantage clearly being the avoidance of having to send a message along the bus to the RAM inquiring regarding the value of the given variable. Otherwise, if the value is not in the cache, we say that the CPU has a *cache miss*.

Let $A_1, \ldots, A_n$ be processes engaging through the algorithm in Listing 5. Suppose that A_1 holds the `lock`. The first read of `lock` by some process after `lock` had been attained by A_1 produces a cache miss. From this point on, as long as A_1 "holds" `lock` and the others keep spinning, attempting to attain `lock` all of the attempts by the other processes produce cache hits and consequently do not cause any bus traffic. The situation changes dramatically when A_1 "releases" `lock`. This act invalidates the values read for `lock` for all spinning processes. This leads to heavy a use of the bus.

Definition 3.4. *A process is said to be* locally spinning *over a given variable if it repeatedly reads cached values of the given variable.*

Based on the above discussion we propose the following.

```
1  Shared: int lock
2
3  Initially: lock = 0
4
5  while(true){
6      while(lock ==1) {}
7      if (TAS(lock) == 0):
8          break
9  }
10 <CS>
11 lock = 0
```

Listing 7: The test-and-test-and-set algorithm

The algorithm seen in Listing 7 is called the *test-and-test-and-set* algorithm. In the `TAS` algorithm (see Listing 5) the `TAS` command is used to simultaneously test whether the `lock` is free and lock it if it is. In the test-and-test-and-set algorithm we seek to increase the "chance" that an invocation of `TAS` would be successful, i.e., that it ends in attaining the `lock`. For any failed attempt leads to invalidating the cached value of `lock` and to increased bus traffic as a result.

To that end the test-and-test-and-set algorithm first spins (in the inner while loop) until the `lock` appears to be free. This spinning involves only reading the value of `lock` and causes no changes to the cached value. Only when the lock appears to be free will an attempt to actually attain it via the `TAS` command.

§3.1.3 *Java implementation of the Test-and-test-and-set algorithm.*

```
1
2  public class TTASLock implements Lock {
```

```
3     AtomicBoolean state =
4                       new AtomicBoolean(false);
5
6     public void lock() {
7           while(true){
8               while (state.get()){}
9               if(!state.getAndSet(true))
10                  return;
11          }
12    }
13
14    public void unlock() {
15          state.set(false)
16    }
17 }
```

Listing 8: The Test-and-test-and-set lock

§3.1.4 ***Backing off.*** One heuristics that is commonly added to the test-and-test-and-set algorithm follows the reasoning that if a process passes line 6 in Listing 7 yet fails at line 7 then that perhaps indicates that there is high contention for the lock. It would be a poor decision to try and attain the lock right after. Instead, the *back off heuristics* suggests that it is better for a process that detects such a failure in line 7 to not try an attain the lock immediately right after. Hence the following adaptation.

```
1  Shared: int lock
2
3  Initially: lock = 0
4
5  while(true){
6      while(lock ==1) {}
7      if (TAS(lock) == 0):
8          break
9      else:
10         backoff(???)
11 }
12 <CS>
13 lock = 0
```

Listing 9: The test-and-test-and-set algorithm with back off

Line 10 requires an explanation. At the current stage of these notes the only way we can cause a process to back off is via a call to some `sleep` system call that is usually available. One approach is to send processes to sleep a certain fixed amount of time. A more sophisticated approach would be to

count the number of failures at line 7 a process encounters and back off an amount of time proportional to that.

§3.2 The swap and CAS operations. Let us mention two other classical atomic operations that can be found on modern systems. The first is the `swap` operation which is just `TAS` with the option of specifying which value will be placed into the variable.

```
1 swap(x,value)
2 {
3     prev := x
4     x := value
5     return pre
6 }
```

Listing 10: The `swap` atomic operation

The next operation is called `CAS` (short for compare and swap) given by the following listing.

```
1 CAS(x,old,new)
2 {
3     if (x == old):
4         x = new
5         return true
6     else:
7         return false
8 }
```

Listing 11: The `CAS` atomic operation

§4. Dekker's algorithm

One of the earliest synchronisation algorithms for two processes is that of Dekker's. This algorithm is often overlooked due to an algorithm by Peterson who achieves the same goals as Dekker does but is much simpler.

In Dekker's algorithm, the processes share a boolean array called `inter` of size two all entries of which are initialised to false. In addition, the two processes share an integer called `turn` initialised to 0. One of the key features of Dekker's algorithm is that if both processes are interested in entering the CS then these engage in strict turn alternation; a pattern considered in § 2.2. In particular, one may verify from the code that 0 never sets `turn` to 0 and 1 never sets `turn` to 1; each has to wait until their counter part sets it to

their identity for them. To prevent starvation that strict alternation can suffer from in the case of one of the parties not being interested (see § 2.2 for details), Dekker uses the `inter` array to break ties.

```
inter[0] = True
while(inter[1] == True){
    if (turn == 1){
        inter[0] = False
        while(turn == 1){}
        inter[0] = True
    }
}

<CS>

turn = 1
inter[0] = False
```

Listing 12: Process 0

```
inter[1] = True
while(inter[0] == True){
    if (turn == 0){
        inter[1] = False
        while(turn == 0){}
        inter[1] = True
    }
}

<CS>

turn = 0
inter[1] = False
```

Listing 13: Process 1

Prior to proving that the algorithm at hand provides mutual exclusion we set up the following notation. We write $\text{Read}_i(\text{register } r)$ in order to denote the point in time in which process i has finished reading register r. We write $\text{Write}_i(\text{register } r)$ to denote the point in time in which process i has finished writing register r. We write CS_i to denote the point in time in which process i entered the CS. Given two events say X and Y we write $X \to Y$ to express that X occurred prior to Y in time.

Claim 4.1. *Dekker's algorithm provides mutual exclusion.*

Proof. We will show that

$$\text{if 0 is in the CS then } \texttt{turn} \text{ must be equal to 0.} \tag{4.2}$$

A similar argument would hold for process 1, i.e.,

$$\text{whenever 1 is in the CS then } \texttt{turn =1}. \tag{4.3}$$

The mutual exclusion property is then implied by (4.2) and (4.3); as if both processes are in the CS then `turn` has to assume two different values at some point in time which is impossible.

It remains to prove (4.2) and (4.3); we prove (4.2) as the argument for (4.3) is similar. There are two paths in the code through which 0 can enter the CS.

1. The first path is for 0 to set `inter[0] =T` at line 1 and then fail the condition in the outer `while` loop found at line 2. In this path then we have that

$$\text{Write}_0(\texttt{inter[0] = T}) \rightarrow \underbrace{\text{Read}_0(\texttt{inter[1] = F})}_{\text{line 2}} \rightarrow CS_0$$

   For 0 to read `inter[1] = F` at line 2 we must have one of two cases hold.

   (a) Either 1 never wrote to `inter[1]` and the value read there by 0 is the initialisation value in which case 1 never even executed its entry and `turn =0` trivially, as this is the initial value of `turn`.

   (b) Or that 1 wrote `inter[1] =F` before 0 read `inter[1] = F`, i.e., $\text{Write}_1(\texttt{inter[1] = F}) \rightarrow \text{Read}_0(\texttt{inter[1]= F})$. Process 1 writes `inter[1] = F` in two places in the code.

      i. If 1 wrote `inter[1] = F` at line 13 and this write preceded 0's reading of this variable then by examining the code we see that

$$\begin{aligned}\underbrace{\text{Write}_1(\texttt{turn = 0})}_{\text{line 12}} \rightarrow \underbrace{\text{Write}_1(\texttt{inter[1] = F})}_{\text{line 13}} \\ \rightarrow \text{Read}_0(\texttt{inter[1] = F}) \\ \rightarrow CS_0\end{aligned}$$

      and (4.2) follows in this case.

      ii. If 1 writes `inter[1] = F` at line 4 then it does so inside the body of the `if` clause (the only one in the code) which it enters only if `turn = 0` when it is at line 3. This coupled with the fact that only process 0 flips the value of `turn` from 0 to 1 at line 12 only means that we can write

$$\begin{aligned}\underbrace{\text{Read}_1(\texttt{turn = 0})}_{\text{line 3}} \rightarrow \underbrace{\text{Write}_1(\texttt{inter[1] = F})}_{\text{line 4}} \\ \rightarrow \text{Read}_0(\texttt{inter[1] = F}) \\ \rightarrow CS_0\end{aligned}$$

      and (4.2) follows in this case as well.

2. The second path through which 0 can enter the CS is to write `inter[0] = T` at line 6 and then repeat the outer loop condition in line 2, fail it

and enter the CS. Note that while in line 2 in this path the exact same invariants that we had in 0's first path hold here as well. Hence the above argument that was offered for the first path holds here as well.

■

We skip the proof that Dekker's algorithm is deadlock-free and prove the following.

Claim 4.4. *Dekker's algorithm is starvation-free.*

Proof. Assume for the sake of contradiction that 0 is stuck in its entry code forever. We consider possible locations for 1 in the code.

1. Process 1 cannot be stuck forever in its remainder code; for if so the last time it exited the CS it sets `inter[1] = F` and thus allowing process 0 to pass the outer while loop.

2. The scenario that we nicknamed "1 is faster than 0" cannot take place here as if both processes are interested we have seen that these engage in strict alternation which does not allow for this scenario to take place.

3. We may assume then that 1 is stuck in its entry code forever as well. As both processes are stuck in their entry codes the value of `turn` is never changed and is now fixed.

   (a) Suppose, first, that `turn = 0` forever. Then after a finite amount of time process 1 (who we assume never leaves its outer while loop as it is stuck) must enter the inner while loop and be stuck there forever. Prior to doing so it sets `inter[1] =F`. This together with the assumption that `turn =0` allows 0 to enter the CS; contradiction.

   (b) Suppose, second, that `turn = 1` forever. A symmetrical argument to the one seen in the previous case (i.e., when `turn = 0` forever) now shows that 1 cannot be stuck forever in its entry code; contradiction.

■

§5. Peterson's algorithm

In this section we consider Peterson's synchronisation algorithm for two processes.

§5.1 First component: making a sacrifice. The first component to Peterson's algorithm is strict alternation; however, unlike the strict alternation we have seen in § 2.2 here each of the two processes do not try to claim their turn but instead waiver it and give the right ahead to the other process.

```
1 turn_to_wait = 0
2 await(turn_to_wait == 1)
3 <CS>
```

Listing 14: Process 0

```
1 turn_to_wait = 1
2 await(turn_to_wait == 0)
3 <CS>
```

Listing 15: Process 1

Here `turn_to_wait` is a shared variable whose initialisation does not matter.

Claim 5.1. *The algorithm in Listings 14 and 15 provides mutual exclusion and is deadloc-free.*

Proof. If two processes are in the CS at the same time then there exists a point in time in which `turn_to_wait` has to assume both values 0 and 1. The same argument applies if both processes are each stuck in their second line forever. Indeed, of so again we have a point in time in which `turn_to_wait` has to assume both values 0 and 1. ■

Claim 5.2. *The algorithm in Listings 14 and 15 is not starvation-free.*

Proof. The failure here is as we have seen with the usual strict alternation algorithm in § 2.2. For suppose process 0 tries to enter and sets `turn_to_wait = 0` while process 1 is stuck in its remainder code forever. ■

§5.2 Second component: expressing interest. In the following algorithm `inter` is a shared boolean array of size two. Initially `inter[0] = inter[1] = False`.

```
1 inter[0] = True
2 await(inter[1] == False)
3 <CS>
4 inter[0] = False
```

Listing 16: Process 0

```
1 inter[1] = True
2 await(inter[0] == False)
3 <CS>
4 inter[1] = False
```

Listing 17: Process 1

Claim 5.3. *The algorithm in Listings 16 and 17 is not deadlock-free.*

Proof. Consider the scheduling scenario in which process 0 sets `inter[0] = True` then process 1 sets `inter[1] = True` which leads to a deadlock. ■

Claim 5.4. *The algorithm in Listings 16 and 17 provides mutual exclusion.*

Proof. Assume for the sake of contradiction that the algorithm does not provide mutual exclusion so that there exists a point in time t_0 in which both process are in the CS. According to the code

$$\text{Read}_0(inter[1] == False) \rightarrow CS_0, \tag{5.5}$$

and

$$\text{Read}_1(inter[0] == False) \rightarrow CS_1. \tag{5.6}$$

Also according to the code

$$\text{Write}_0(inter[0] = T) \rightarrow \text{Read}_0(inter[1] = False), \tag{5.7}$$

and

$$\text{Write}_1(inter[1] = T) \rightarrow \text{Read}_1(inter[0] = False). \tag{5.8}$$

In order for process 0 to read `inter[1] == False` (in the `await`) it must have done so prior to process 1 writing `inter[1] = True`. The same applies to process 1. We thus have

$$\text{Read}_0(inter[1] = False) \rightarrow \text{Write}_1(inter[1] = True), \tag{5.9}$$

and

$$\text{Read}_1(inter[0] = False) \rightarrow \text{Write}_0(inter[0] = True). \tag{5.10}$$

We thus attain the following contradiction.

$$\begin{aligned} \text{Write}_0(inter[0] = T) &\overset{(5.7)}{\rightarrow} \text{Read}_0(inter[1] = F) \\ &\overset{(5.9)}{\rightarrow} \text{Write}_1(inter[1] = T) \\ &\overset{(5.8)}{\rightarrow} \text{Read}_1(inter[0] = F) \\ &\overset{(5.10)}{\rightarrow} \text{Write}_0(inter[0] = T). \end{aligned}$$

■

§5.3 The full formulation. Peterson's algorithm is a combination of the two components seen in the previous two sections. In that the process share a boolean array `inter` of size two initialised `inter[0]= inter[1] = False` and a variable `turn_to_wait` whose initialisation does not matter.

```
1  inter[0] = True
2
3  turn_to_wait = 0
4
5  while((inter[1] == True)
6                AND
7          turn_to_wait == 0)
8          {
9                //busy waiting
10         }
11
12 <CS>
13
14 inter[0] = False
```

Listing 18: Process 0

```
1  inter[1] = True
2
3  turn_to_wait = 1
4
5  while((inter[0] == True)
6                AND
7          turn_to_wait == 1)
8          {
9                //busy waiting
10         }
11
12 <CS>
13
14 inter[1] = False
```

Listing 19: Process 1

The two components of Peterson's algorithm provide mutual exclusion independently of one another (see Claims 5.1 and 5.4) yielding the following.

Claim 5.11. *Peterson's algorithm provides mutual exclusion.*

Peterson's algorithm "inherits" deadlock-freedom from its first component.

Claim 5.12. *Peterson's algorithm is deadlock-free.*

None of the two components of Peterson's algorithm provide starvation-freedom. Nevertheless their combination does.

Claim 5.13. *Peterson's algorithm is starvation-free.*

Proof. Assume for the sake of contradiction that 0 is stuck forever in its entry code. In particular there exists a point in time t_0 such that

$$\texttt{TTW = 0} \text{ after } t_0; \tag{5.14}$$

in addition,

$$0 \text{ always reads } \texttt{inter[1] = T} \text{ after } t_0. \tag{5.15}$$

We consider the possible locations for process 1 after t_0.

1. Process 1 cannot be stuck forever in its remainder code; otherwise there would be a point in time after which (5.15) is invalid; contradiction.

2. As we proved that the algorithm is deadlock-free, process 1 cannot be stuck forever in its entry code.

3. Process 1 cannot attempt any reentry to the CS as that entails setting `TTW = 1` contradicting (5.14).

■

Let us consider the boundedness of Peterson's algorithm.

§5.4 Boundedness of Peterson's algorithm. We may partition the entry code of Peterson's algorithm into two parts. One part is specified by lines 1 to 3 in Listings 18 and 19. This part of the algorithm involves no "waiting"; we refer to this part of the algorithm as the *doorway* where processes are still setting information up in preparation to enter the "waiting" stage. The second part is seen in lines 5-10 in Listings 18 and 19. This is the *waiting area* of Peterson's algorithm.

Let $\mathcal{A}$ be a synchronisation algorithm whose entry code has a doorway and a waiting area properly distinguished, and let `A` and `B` be two processes executing a common synchronisation algorithm. If `A` starts enters the waiting area prior to `B` yet `B` enters the critical section prior to `A` we say that `A` had been *bypassed* by `B`.

Definition 5.16. *Let $r \in \mathbb{N}$. A synchronisation algorithm is said to be* r-bounded *if no process upon entry attempt can be bypassed more than r times*[1].

The *bypass number* of a synchronisation algorithm is the least r such that the algorithm is r-bounded. r-boundedness ensures a finite bypass number which usually implies starvation-freedom. However, the algorithm appearing in Listing 2 and 3 is 0-bounded (i.e., FIFO) and yet allows for starvation to take place. So the claim that r-boundedness implies starvation-freedom is not true. Nevertheless, the spirit of this claim we retain when designing algorithms.

The main problem with the definition given here is that not all algorithms have a clear distinction between doorway and waiting area rendering the above mute.

Claim 5.17. *Peterson's algorithm is* 1*-bounded but is not* 0*-bounded.*

Proof. Once the two processes are interested in entering the CS the order of entry is determined by the order in which `turn_to_wait` is written. Specifi-

[1] Later we shall see that this definition is not so wise; for now we keep it.

cally, the last process to complete a write operating to this variable will lose the competition for the lock. With that said we may have a scenario where:

1. 1 sets `turn_to_wait`;
2. 0 sets `turn_to_wait` and enters wait area
3. 1 bypasses 0;

demonstrating that the algorithm is not 0-bounded. However, the algorithm is 1-bounded. Indeed, once a process is already waiting any process that would attempt reentry (and attempt a second bypass) would fail as the process attempting reentry sets `turn_to_wait` and sacrifices itself for waiting. ■

§5.5 Sensitivity of Peterson's algorithm. In order to establish the properties of Peterson's algorithm we had to relay heavily on the order of the assignments appearing in lines 1 and 3 in Listings 18 and 19. To make this point clearer let us consider the following change to Peterson's algorithm in which we change the order of these lines.

```
1  turn_to_wait = 0
2
3  inter[0] = True
4
5  while((inter[1] == True)
6                AND
7          turn_to_wait == 0)
8          {
9               //busy waiting
10         }
11
12 <CS>
13
14 inter[0] = False
```

Listing 20: Process 0

```
1  turn_to_wait = 1
2
3  inter[1] = True
4
5  while((inter[0] == True)
6                AND
7          turn_to_wait == 1)
8          {
9               //busy waiting
10         }
11
12 <CS>
13
14 inter[1] = False
```

Listing 21: Process 1

This algorithm fails to provide mutual exclusion as the following scenario demonstrates:

1. Process 0 sets `turn = 0` and is suspended.
2. Process 1 sets `turn 1 = 1`, sets `inter[1] = T`, passes the while loop (as currently `inter[0] = F`) enters the CS and is suspended inside the CS.

3. Process 0 resumes; it sets `inter[0] = T`; it reaches the while loop and is able to pass it as now `turn = 1`.

§5.6 Peterson's algorithm fails in the real world. In § 5.5 we have demonstrated how heavily we relied in our proof on the order in which the variables `inter[0],inter[1],` and `turn` are written. Moreover, throughout our proofs we have naively assumed that the order in which a processes writes to different variables takes effect according to program order. On modern computers this is not always true. Two main reasons for that.

1. Compilers often change our code for optimisation reasons and thus can alter the program order we have dictated. In some compilers (mainly for C/C++) one can bypass this behaviour of this compiler by, say, writing sensitive parts of the code in assembler; some compilers supply macros that make the compiler "shut down" code optimisation for some segments. But these solution either result in the programmer having to delve into more primitive languages or to write code that is hard to maintain due to use of macros that interfere with the compilation process making the compilers functionalities not work the way they should leading to run time errors being missed.

2. Modern memory hardware is no necessarily sequential. Indeed, due to optimisation considerations it is quite common to delay writing to variables until it is absolutely necessary. This "lazy" type of maintaining variables can seriously hinder algorithms such as Peterson's algorithm.

§5.7 A Java implementation of Peterson's algorithm. Consider the following implementation of Peterson's algorithm. We should disclose that here we in fact assume that the two threads involved are assigned the IDs 0 and 1. This surely need not be the case; nevertheless we refrain from writing the code that would resolves non-binary IDs of threads.

```
1
2  class PetersonLock implements Lock {
3    private boolean[] interested = new boolean[2];
4    private int turn_to_wait;
5
6    public void lock() {
7
8      int i = ThreadID.get(); // ID 0 or 1
9      int j = 1-i;
10     interested[i] = true;
11     turn_to_wait = i;
```

```
12      while (flag[j] && victim == i) {};
13    }
14
15    public void unlock(){
16      int i = ThreadID.get(); // ID 0 or 1
17      interested[i] = false;
18    }
19  }
```

Listing 22: The Peterson lock

§6. Semaphores

Every synchronisation algorithm prompts the following question: *what should a process do until it attains the lock?* So far we have seen only one approach - spinning. The other approach is to relinquish CPU attention and ask the OS scheduler to schedule a different process; this approach is called *blocking*. In this section we shall define a construct present in every OS called a *semaphore* that will allow us to do that. However, it turns out that the ability to block is relevant to a wide range of synchronisation needs well beyond the mutual exclusion problem.

§6.1 Semaphores: definitions. The classical definition of a semaphore is as follows.

Definition 6.1. *A semaphore is an integer counter coupled with two atomic operations called* `up` *and* `down` *that maintain the counter.*

```
1  DOWN(S)
2  {
3
4    IF S == 0: BLOCK
5
6    ELSE S = S - 1
7
8  }
```

Listing 23: `DOWN` operation

```
1  UP(S)
2  {
3   IF (there are blocked processes)
4   {
5     wake some blocked process
6   }
7   ELSE S= S+1
8  }
```

Listing 24: `UP` operation

In this definition it is easy to see that $S \geq 0$ at all times. Hence we can think of a semaphore as a pile of coins. In order to pass the semaphore blockage a process has to take a coin from the pile. If there are no coins then a process becomes *captive* of the semaphore and to release it a coin (i.e., bail money) is required to set it free which comes in the form of an invocation of `UP`.

Only that if a process is captive then we do not accumulate the coin we associate with the `UP` invocation releasing it; instead the coin is *swallowed* by the semaphore and is not accumulated in its counter.

This analogy prompts the following definitions. A `down` operation leading to blocking is called a *blocking down*. An `up` operation leading to the release of a process blocked on the semaphore is called *releasing up*. The invariant of semaphores is that

$$\text{number of blocking downs} = \text{number of releasing ups}. \tag{6.2}$$

This invariant is guaranteed because the `UP` operation does not increment the counter if there are blocked processes. We refer to (6.2) as the *semaphore invariant*. Below we shall consider alternative formulations for semaphores. A formulation respecting (6.2) is said to be *balanced*.

We distinguish between several types of semaphores. In Definition 6.1 the value of `S` is unbounded. A semaphore is called *bounded* if its counter value is bounded. In particular, a semaphore is called *binary* if its counter is bounded by one. A semaphore that is not binary (i.e., its counter can reach values other than 0 and 1) is called a *counting semaphore*.

§6.2 The mutex algorithm. Strictly speaking in the literature all algorithms that solve the mutual exclusion problem are called *mutex algorithms*. In these notes we do not adhere to this culture; instead we only refer to the following algorithm as the *mutex algorithm*.

```
1 Shared: Semaphore s
2
3 Initially: S = 1
4
5 DOWN(S)
6
7 <CS>
8
9 UP(S)
```

Listing 25: The mutex algorithm

The mutex algorithm is probably the most heavily used algorithm in practice. We explore its properties.

Claim 6.3. *The mutex algorithm provides mutual exclusion and is deadlock-free for any number of processes.*

Proof. As the `DOWN` operation is atomic the first process to will be the only process permitted into the CS. ■

Claim 6.4. *The mutex algorithm is starvation-free for two processes; however it is not starvation-free for three processes. As there is always a first process that invokes* `DOWN` *at least one competing process will enter the CS.*

Proof. Suppose, first, that there are two processes using the `mutex` algorithm. Assume towards a contradiction that process `A` is stuck forever in its entry code. As here there is no busy waiting `A` being stuck forever means that

from a certain point in time `B` never performs `UP(S)`. (6.5)

This in turn limits the possible locations `B` can be in. As the algorithm is deadlock-free `B` is not stuck forever in its entry code. If `B` is stuck forever in its remainder code then `S=1` would be valid at some point in time in particular after `B`'s last exit from the CS. Finally, the scenario of *`B` is faster than `A`* is not possible due to (6.5).

Suppose, second, that at least three processes are using the `mutex` algorithm. The following scenario demonstrates that one can starve.

1. Let `A,B,C` be three process such that `A,B` are stuck in the `DOWN` operation while `C` is in the CS.
2. `C` leaves the CS calls `UP(S)` and releases `B`.
3. `B` enters the CS and upon leaving calls `UP(S)` and releases `C`.

This pattern of `B` and `C` relating one another repeats forever all the while keeping `A` blocked.

■

The proof of Claim 6.4 motivates the following notion.

Definition 6.6. *A semaphore is said to be* fair *if it wakes up blocked processes in the order those were blocked on the semaphore (i.e., FIFO order).*

If there is no assurance regarding the wake-up policy of a semaphore it is called *unfair*. Note that in our terminology the notions of fair and unfair semaphore are not complimentary.

§6.2.1 ***Negative semaphores.*** Let us consider the following proposal for semaphore implementation.

```
1 DOWN(S)
2 {
3
4   IF S == 0: BLOCK
5
6   ELSE S = S - 1
7
8 }
```

Listing 26: Alternative down?

```
1 UP(S)
2 {
3  S += 1
4  IF (there are blocked processes)
5  {
6    wake some blocked process
7  }
8 }
```

Listing 27: Alternative up?

Consider the following scenario.

1. Initially $S = 0$.
2. Process `A` performs a blocking `DOWN(S)` and gets blocked.
3. Process `B` performs `UP(S)` sets `S=1` and releases `A`.
4. Process `C` performs a `DOWN(S)` sets `S = 0` but `C` does not get blocked.
5. Process `A` gets running time and continues running.

Imagine now that these three processes execute the mutex algorithm and that `S` is the semaphore used in the mutex algorithm to regulate entrance to the CS. The scenario above shows that if semaphores are implemented as in Listings 26 and 27 then mutual exclusion is violated as `A` and `C` can reside in the CS at the same time.

In this proposal there is no distinction between releasing ups and non-releasing ups these are all counted using the same counter. This led to the situation we see in the scenario that two `down` calls that were released using a single `up` call.

One way to fix this proposal is to make the code "symmetric" sort of speak. That is, if we do not distinguish between releasing ups and non-releasing ups we should not distinguish between blocking downs and non-blocking downs.

```
1 DOWN(S)
2 {
3
4   S -= 1
5
6   IF S < 0 : BLOCK
7
8 }
```

Listing 28: Negative semaphore

```
1 UP(S)
2 {
3  S += 1
4  IF (there are blocked processes)
5  {
6    wake some blocked process
7  }
8 }
```

Listing 29: Negative semaphore

Let us first examine whether the scenario above still poses a problem.

1. Initially `S= 0`.
2. `A` performs `DOWN(S)`, sets `S= -1` and is blocked.
3. `B` performs `UP(S)`, sets `S=0` and releases `A`.
4. `C` does `DOWN(S)`, sets `S = -1` and is blocked.
5. `A` continues running.

We refer to the implementation appearing is Listings 28 and 29 as a *negative semaphore*; for indeed under this implementation `S < 0` is possible. Note that if `S < 0` then `-S` is the number of processes blocked on the semaphore.

§6.3 Fundamental patterns. The simplest use for semaphore is for sending signals between processes. Suppose two processes `A` and `B` share a semaphore `sem` whose initial value is 0, and both execute the following.

```
1 statement 1
2 up(sem)
```

Listing 30: Process `A`

```
1 down(sem)
2 statement 2
```

Listing 31: Process `B`

Here `B` awaits a signal from `A` prior to executing statement 2. The pattern illustrated by Listings 30 and 31 is called the *signal pattern*.

§6.3.1 *The rendezvous pattern.* The signal pattern can be generalised in order to make it work both ways. That is we seek a synchronisation pattern so the in the following

```
1 statement A1
2 statement A2
```

Listing 32: Process `A`

```
1 statement B1
2 statement B2
```

Listing 33: Process `B`

statement A1 is performed before statement B2 and that statement B1 is performed before A2. These restrictions force the each of two processes to not pass their second statement prior to their counterpart performing its first statement. In a sense they rendezvous (i.e., meet up on the second statements).

An initial attempt might look like this.

```
1 statement A1
2
3 DOWN(semB)
4
5 UP(semA)
6
7 statement A2
```

Listing 34: Process `A`

```
1 statement B1
2
3 DOWN(semA)
4
5 UP(semB)
6
7 statement B2
```

Listing 35: Process `B`

Where here `semA` and `semB` are two shared semaphores both initialised to 0. This attempt however is wrong as clearly both processes can become blocked on their third line. A correct solution has the following form.

```
1 statement A1
2
3 UP(semA)
4
5 DOWN(semB)
6
7 statement A2
```

Listing 36: Process `A`

```
1 statement B1
2
3 UP(semB)
4
5 DOWN(semA)
6
7 statement B2
```

Listing 37: Process `B`

We refer to the pattern illustrated in Listings 36 and 37 as the *rendezvous pattern*.

§6.3.2 ***The multiplex pattern.*** The mutex algorithm allows for at most one process to enter a CS. In the *multiplex pattern* we seek to generalise the mutex algorithm and allow a prescribed number, say k, of processes in the critical section. The solution here depends whether the OS supports bounded semaphores or not. If it does then the solution is quite simple for it simply entails the creation of a bounded semaphore whose bound is k and his initial value is k. If however the OS does not provide this then we have to manage the semaphore counter ourselves. In § 6.6 we shall do so.

§6.3.3 ***The barrier pattern.*** Next we seek to generalise the rendezvous pattern which handles two processes and scale it to fit for a prescribed number of $n \geq 2$ processes. Put another way suppose we have n processes named $p_1, \ldots, p_n$ where each has to execute statements

```
1 Statement i1
2 Statement i2
```

Listing 38: Process p_i

and we seek to make sure that before any process p_i executes its statement `i2` all other processes p_j have executed their statement `i1`.

```
1  Shared:
2      int count
3      BinarySemaphore mutex
4      BinarySemaphore barrier
5
6  Initially:
7      count = 0
8      mutex = 1
9      barrier = 0
10
11 statement i1
12
13 DOWN(mutex)
14     counter+= 1
15 UP(mutex)
16
17 if count == n:
18
19     UP(barrier)
20
21 DOWN(barrier)
22 UP(barrier)
23
24 statement i2
```

Listing 39: Code of process p_i

The pattern presented in Listing 39 is called the *barrier pattern* or *barrier synchronisation*. In this pattern let us focus our attention to lines 21-22 in which one sees that we perform `DOWN` and `UP` over the semaphore `barrier` in rapid succession. Performing `DOWN` and `UP` over a semaphore in rapid succession is a pattern called the *turnstile pattern*. The effect of this pattern is that only one process can pass it at a time. In its initial state the turnstile is locked (i.e., its value is 0). In the barrier pattern only the process to arrive last can open the turnstile which in turn allows all other processes to pass the turnstile; however as `barrier` is binary only one process can pass the turnstile at a time.

§6.4 Implementing counting semaphores. How to implement negative counting semaphores using binary semaphores? Here we assume the following implementation for binary semaphores.

- Invoking `DOWN` on a binary semaphore whose value is 1 is a non-blocking `DOWN` which only reduces the value of the semaphore to 0.

- Invoking `DOWN` on a semaphore of value 0 results in blocking.
- Invoking `UP` on a semaphore of value 1 is not recorded.
- Otherwise `UP` is as in Definition 6.1.

We seek to implement a negative counting semaphore `S`. Throughout we shall denote its value by `S.value`, and we shall use two binary semaphores called `mutex` and `delay` in order to implement `S`. As its name suggests, the semaphore `mutex` will be used for mutual exclusion, while the semaphore `delay` will be used to have the OS implement a queue data structure for us implicitly. Throughout we always assume that `S.value`, `mutex`, and `delay` are shared by all processes using the semaphore `S`.

§6.4.1 ***Failed attempt 1.*** We start with the following failed attempt at an implementation of `S`. This attempt is simply a translation of Listings 28 and 29 that define negative counting semaphores almost verbatim. Initially `mutex = 1,delay = 0`.

```
1  OUR-DOWN(S)
2  {
3    DOWN(mutex)
4    S.value -= 1
5    if (S.value < 0){
6      L1: UP(mutex)
7      L2: DOWN(delay)
8    }
9    else{
10    UP(mutex)
11   }
12
13 }
```

Listing 40: Negative via binary

```
1  OUR-UP(S)
2  {
3
4
5    DOWN(mutex)
6    S.value += 1
7    if (S.value <= 0){
8      UP(delay)
9    }
10   UP(mutex)
11
12
13 }
```

Listing 41: Negative via binary

Prior to analysing this code let us make a few remarks.

1. Note that `OUR-DOWN,OUR-UP` are not atomic. Only the operations on binary semaphores are atomic.
2. Note the labels L1 and L2 which are not a part of the code; we shall use them as anchors into the code to make references.
3. The `if` clause in the `OUR-UP` procedure has `<=` to account for the case that `S.Value = -1` which should indicate a single process is still blocked on `S`.

The following scenario establishes that this implementation fails.

1. Initially `S.value = 0`.
2. Processes P_1, P_2, P_3 all invoke `OUR-DOWN(S)` and are all suspended between the line marked L1 and the line marked L2. Currently `S.value = -3` and none of P_1, P_2, P_3 performed `DOWN(delay)`.
3. Next, processes Q_1, Q_2, Q_3 perform `OUR-UP(S)`.
   - As `S.value = -3` Q_1, Q_2, Q_3 all execute `UP(delay)`; as `delay = 0` prior to these three calls, two of these `UP` calls are not recorded by the underlying binary semaphore and are lost.
   - Currently `delay = 1`.
   - Process Q_1, Q_2, Q_3 exit the `OUR-UP` procedure and terminate.
4. P_1 gets running time, performs `DOWN(delay)`, sets `delay = 0`, leaves the `OUR-DOWN` procedure and terminates.
5. P_2 and P_3 cannot pass `DOWN(delay)`.

The problem with this implementation then is that it does not provide a balanced semaphore. Indeed, we performed three blocking downs; in a correct implementation three invocations of `UP` should have released them. Here we performed three calls to `UP` yet two processes remain blocked.

§6.4.2 ***Failed attempt 2.*** The flaws of the implementation seen at Listings 40 and 40 originated due to the fact that two `UP(delay)` calls made by the processes Q_1, Q_2, Q_3 were lost and not recorded. One way to try and mend this issue is to rewrite the `OUR-UP` code as follows.

```
1  OUR-DOWN(S)
2  {
3    DOWN(mutex)
4    S.value -= 1
5    if (S.value < 0){
6      L1: UP(mutex)
7      L2: DOWN(delay)
8    }
9    else{
10     UP(mutex)
11   }
12 }
```

Listing 42: Negative via binary

```
1  OUR-UP(S)
2  {
3    DOWN(mutex)
4    S.value += 1
5    if (S.value <= 0){
6      UP(delay)
7    }
8    else{
9      UP(mutex)
10   }
11 }
```

Listing 43: Negative via binary

The addition of the `else` clause to the `OUR-UP` code makes sure that in the scenario we had above the `UP` calls of Q_2, Q_3 cannot get lost. In Listing 43 Q_1 leaves the code of `OUR-UP` without a call to `UP(mutex)` forcing both Q_2 and Q_3 to hang at the `DOWN(mutex)` call. This makes sure that their `UP(delay)` calls will not be lost as these are prevented.

Nevertheless, we see that the code in Listing 42 leads to a problem now and has to be changed as well. Continuing the scenario we had so far, note that with Q_1 terminated, and Q_2, Q_3 hanging at `DOWN(delay)` we have the following problem.

1. Suppose process P_1 gets running time now. It is able to pass the `DOWN(delay)` at L2 due to Q_1 call to `UP(delay)`. However P_1 leaves `OUR-UP` without a call to `UP(mutex)`.

2. P_2, P_3 now both get blocked on `DOWN(delay)` at L2.

The situation now is that P_2, P_3 are blocked at L2 while Q_2, Q_3 are blocked at line 3 of `OUR-UP`. This is surely not a balanced implementation of a semaphore.

§6.4.3 ***Failed attempt 3.*** The attempt we had at § 6.4.2 demonstrates that we will not be able to "get away" with mere cosmetic changes. This next attempt will try to handle the issue of lost `UP(delay)` calls by actually counting those. To that end we introduced an additional shared integer variable called `wake` initialised to 0. We will refer to it through `S` by writing `S.wake`.

```
1  OUR-DOWN(S)
2  {
3    DOWN(mutex)
4    S.value -= 1
5    if (S.value < 0){
6      L1: UP(mutex)
7      L2: DOWN(delay)
8          DOWN(mutex)
9          S.wake -= 1
10         if (S.wake > 0){
11           UP(delay)
12         }
13   }
14   UP(mutex)
15 }
```

Listing 44: Negative via binary

```
1  OUR-UP(S)
2  {
3    DOWN(mutex)
4    S.value += 1
5    if (S.value <= 0){
6      S.wake += 1
7      UP(delay)
8    }
9    UP(mutex)
10 }
```

Listing 45: Negative via binary

The implementation proposed in Listings 44 and 45 solves our ongoing scenario.

1. Let P_1, P_2, P_3 be suspended between L1 and L2 so that `S.value = -3`.
2. Next, Q_1, Q_2, Q_3 all perform `OUR-UP` leading to `S.wake = 3`.
3. P_1 is granted running time it passes L2 but sets `delay = 0`. It then sets `S.wake = 2`, executes the `if` clause at line 10 in `OUR-UP` and raises `delay` back to 1.
4. P_2 is now able to pass L2. It goes through the same trajectory as P_1 allowing P_3 to pass L2.

In the scenario above we have allowed the processes P_1, P_2, P_3 to pull one another in some sort of chain out of the `OUR-DOWN` code.

Unfortunately this new implementation is also flawed as the following scenario demonstrates.

1. Let `S.value = 0`.
2. $P_1, \ldots, P_7$ perform `OUR-DOWN(S)` and are all blocked at L2.
3. $P_8, \ldots, P_{11}$ perform `OUR-UP` and their calls to `UP(delay)` releases, say, $P_1, \ldots, P_4$.
4. The situation now is that $P_1, \ldots, P_4$ are all ready, P_5, P_6, P_7 are stuck at L2 and `S.wake = 4`.
5. $P_1, \ldots, P_4$ complete the code of `OUR-DOWN` and can surely release P_5, P_6, P_7.

The problem with this scenario is that we performed 7 blocking down calls on `S` yet were able to release those with merely 4 `OUR-UP(S)` calls. This implementation is not balanced then as well.

§6.4.4 ***Final formulation.*** Our last attempt revealed a certain "chain effect" occurring in Listing 44 as seen in the scenario considered for it above. We now make the following observation.

Observation 6.7. *Suppose that in Listing 44 there are k processes blocked on `DOWN(delay)` at line L2 and suppose further that `wake = k`. In this state performing `UP(delay)` leads to the release of all processes blocked on `delay` at L2.*

Proof. Once we have the initial call for `UP(delay)` which releases the first processes we see that the code in Listing 44 generates precisely $k-1$ additional calls to `UP(delay)` through line 11 in Listing 44 as indeed the last process releaseed will not perform such a call as it cannot pass the if clause found at line 10. ■

Based on this observation we arrive at the following solution to our initial problem.

```
1  OUR-DOWN(S)
2  {
3    DOWN(mutex)
4    S.value -= 1
5    if (S.value < 0){
6      L1: UP(mutex)
7      L2: DOWN(delay)
8          DOWN(mutex)
9          S.wake -= 1
10         if (S.wake > 0){
11           UP(delay)
12         }
13   }
14   UP(mutex)
15 }
```

Listing 46: Negative via binary

```
1  OUR-UP(S)
2  {
3    DOWN(mutex)
4    S.value += 1
5    if (S.value <= 0){
6      S.wake += 1
7      if(S.wake ==1){
8          UP(delay)
9      }
10   }
11   UP(mutex)
12 }
```

Listing 47: Negative via binary

§6.5 The producer-consumer paradigm. A classical scenario of which there are many variants is that of the *producer-consumer paradigm*. In its simplest form it involves two processes one called *producer* and the other *consumer*. These share a bounded buffer of size N. The producer process inserts items into the shared buffer if there is space in the buffer and the consumer removes items from the shared buffer as long as the buffer is nonempty. In this section we consider an implementation of this paradigm using semaphores.

In the proposed implementation the two processes share three semaphores.

- A semaphore called `mutex` initialised to 1.
- A semaphore called `empty` initialised to N.
- A semaphore called `full` initialised to 0.

```
1  int item
2  while(True){
3
4      item = produce()
5      DOWN(empty)
6      DOWN(mutex)
7      insert(item)
8      UP(mutex)
9      UP(full)
10
11 }
```

Listing 48: Producer's code

```
1  int item
2  while(True){
3
4
5      DOWN(full)
6      DOWN(mutex)
7      item = remove()
8      UP(mutex)
9      UP(empty)
10
11 }
```

Listing 49: Consumer's code

The code offered here works no matter how many producer or consumer processes are involved. To see why, note that initially, the semaphores `empty` and `full` satisfy `empty+full = N`. In the code "credits" or "coins" are being passed back and forth between the counting semaphores `empty` and `full`. We distinguish between four types of coins. Those accounted for by `empty`, those accounted for by `code`, those that are in transition from `empty` to `full` due to the code of producer, and those that are in transition from `full` to `empty` due to the code of the consumer. Let t_1 and t_2 denote the latter two quantities, respectively. Throughout the execution of the algorithm we have

$$\texttt{empty} + t_1 + t_2 + \texttt{full} = \mathrm{N}.$$

To a certain extent one may to view this algorithm as a generalisation of the strict alternation pattern met in § 2.2; here the use of counting semaphores relaxes the strictness aspect of the original pattern.

§6.6 The readers-writers problem. Assume a database over which we have two types of processes: readers and writers.

- We pose no limit on the number of readers allowed to read concurrently from the database.
- However, once a writer process is in the process of writing to the database, no other process reader or writer is allowed to interact with the database until the current writer process leaves the database.

In this section we consider how to implement such a system. For our first attempt we maintain the following shared information.

- `int readers` initialised to 0; this will be used to count readers present in the database.

- `BinarySemaphore mutex =1.`
- `BinarySemaphore db = 1.`

```
while(True){

 DOWN(mutex)
 readers+= 1
 if (readers == 1)
   DOWN(db)
 UP(mutex)

 read_db()

 DOWN(mutex)
 readers -= 1
 if (readers == 0)
   UP(db)
 UP(mutex)

}
```

Listing 50: Readers code

```
while(True){
  DOWN(db)
   write_db()
  UP(db)
}
```

Listing 51: Writer code

The pattern utilised here in the readers' code is that of the first reader process capturing the database for all that come after it; similarly, the last reader to leave releases the database. This pattern has the drawback that a steady stream of readers would lead to the starvation of writer processes.

Let us now impose another restriction on the implementation in the form that writer processes have priority over reader processes in the sense that *whenever a writer process W is waiting to enter the database no reader process that comes after W is already waiting can enter the database before W*. To facilitate our attempts at implementing this we introduce the following shared information.

- `int readers = 0, writers = 0.`
- `BinarySemaphore Rmutex = 1, Wmutex = 1.`
- `BinarySemaphore delay =1.`
- `BinarySemapore db =1.`

```
while(True){

 DOWN(delay)
 DOWN(Rmutex)
 readers+= 1
 if (readers == 1)
   DOWN(db)
 UP(Rmutex)
 UP(delay)

 read_db()

 DOWN(Rmutex)
 readers -= 1
 if (readers == 0)
   UP(db)
 UP(Rmutex)

}
```

Listing 52: Readers code

```
while(True){

  DOWN(Wmutex)
  writers+=1
  if (writers == 1)
    DOWN(delay)
  UP(Wmutex)

  DOWN(db)
  write_db()
  UP(db)

  DOWN(Wmutex)
  writers -= 1
  if(writers == 0)
    UP(delay)
  UP(Wmutex)
}
```

Listing 53: Writer code

The writers now mimic the pattern seen in the reader code; that is, the first writer to arrive at the database tries to prevent all readers from entering the database and the last writer to leave the databases releases the database on behalf of all writers. The first writer tries to lock out readers by capturing `delay`.

This code will not resolve the problem. For suppose a reader process has captured `delay`. At this point a writer process W and all readers that come after W had started waiting for entry to the database can all be blocked on the `delay` semaphore at the same time. If `delay` is unfair each call for `UP(delay)` need not release W but some reader process that came in after W already started waiting.

Let us introduce another shared binary semaphore named `X` initialised to 1 and let us add it as follows.

```
while(True){

 DOWN(X)
 DOWN(delay)
 DOWN(Rmutex)
  readers+= 1
  if (readers == 1)
    DOWN(db)
 UP(Rmutex)
 UP(delay)
 UP(X)

  read_db()

 DOWN(Rmutex)
  readers -= 1
  if (readers == 0)
    UP(db)
 UP(Rmutex)

}
```

Listing 54: Readers code

```
while(True){

  DOWN(Wmutex)
   writers+=1
   if (writers == 1)
     DOWN(delay)
   UP(Wmutex)

  DOWN(db)
   write_db()
   UP(db)

  DOWN(Wmutex)
   writers -= 1
   if(writers == 0)
     UP(delay)
   UP(Wmutex)
}
```

Listing 55: Writer code

The main gain of introducing `X` in this fashion is that if `delay` is captured by a reader R_1 then no other reader process R_2 can be blocked on `delay` as these will all be blocked on `X`. This means that in this setting readers and writers cannot be blocked on `delay` at the same time.

What happens though if a writer W is blocked on `delay` that has been captured by R_1 and R_2 is blocked on `X`? As soon as R_1 invokes `UP(delay)` it releases W; however the value of `delay` remains 0. Hence, even if R_2 runs before W it will get blocked on `delay` nonetheless.

§7. The filter algorithm

For two processes the simplest synchronisation algorithm we have seen that provides mutual exclusion and starvation-freedom is that of Peterson[2]. It makes sense then that we should try to generalise Peterson's algorithm to fit $n > 2$ processes while trying to keep all of its traits.

The so called *filter* algorithm does just that. Let n denote the number of processes competing for entry into the CS. The filter algorithm maintains a virtual sieve with n levels.

[2]Dekker's algorithm is more complicated than that of Peterson and the rest of the algorithms we have seen lack at least one of the core properties we seek.

- The top most level is 0 and the bottom most level is $(n-1)$. The $(n-1)$st level represents the CS.
- All processes who wish to enter the CS have to start at level 0 and make their way down the sieve to level $n-1$ corresponding to the CS.
- That is, in order to enter the CS a process must clear all levels from 0 to $n-1$.
- For each level the filter algorithm executes a variant of Peterson's algorithm.

We maintain the following shared information.

1. Array `level[n]` initialised to all 0s. Entry `level[i]` indicates the level to which process i is interested in passing to.
2. Array `TTW[n]` (whose initialisation is irrelevant). For $L \in [1, n]$ the entry `TTW[L]` indicates which process is the victim who has to wait in order to enter level L.

```
1  for (L = 1 to n) {
2      level[i] = L
3
4      TTW[L] = i
5
6      while (exists k != i: level[k] >= L
7
8                      and
9
10               TTW[L] == i ){}
11 }
12
13 <CS>
14
15 level[i] = 0
```

Listing 56: Filter algorithm: code for process i

Consider the condition for suspension in the while loop in the filter algorithm. A process P who wishes to enter level L and is the victim for that level would have to wait until there are no other processes in sieve levels who's number if L and above. Any process that expresses interest in the L-level after P sacrifices itself and thus allowing P to move forward. That is, P can progress to higher levels even if other processes are found in those

level as long as he was not the victim for the first level he needs to progress to.

Theorem 7.1. *The filter algorithm provides mutual exclusion.*

Claim 7.2. *Let P be a set of processes at level $j-1$, let $A \in \mathcal{P}$ be the last process in P to write to `TTW[j]`, and assume that no other processes join the $(j-1)$st level. Then A does not pass to level j until all members of $\mathcal{P} \setminus \{A\}$ set their level to 0 by exiting the CS.*

Proof. By the assumption on A:

$$\text{Write}(TTW[j] = B) \rightarrow \text{Write}_A(TTW[j] = A),\ \forall B \in \mathcal{P} \setminus \{A\}.$$

By examining the code

$$\begin{aligned}\text{Write}_B(level[B] = j) &\rightarrow \text{Write}_B(TTW[j] = B) \\ &\rightarrow \text{Write}_A(TTW[j] = A),\ \forall B \in P \setminus \{A\}.\end{aligned}$$

Also from the code, A first writes to `TTW` and only then reads the array `level`. Hence,

$$\begin{aligned}\text{Write}_B(level[B] = j) &\rightarrow \text{Write}_B(TTW[j] = B) \\ &\rightarrow \text{Write}_A(TTW[j] = A) \\ &\rightarrow \text{Read}_A(level[B]),\ \forall B \in \mathcal{P} \setminus \{A\}.\end{aligned}$$

As long then as there exists a $B \in \mathcal{P} \setminus \{A\}$ that did not exit the CS it has `level[B]` $\in [j, n-1]$. Hence, until B clears the CS process A would read `level[B] >= j-1`. As no other processes are assumed to enter the $(j-1)$st level A will not pass to the jth level until all members of $\mathcal{P} \setminus \{A\}$ pass the CS. ■

Let us remark that in this claim we see that it is crucial to require `>=` in the `while` loop and not `>`. Indeed, here we use that fact that `level[B] >= j-1` for all processes `B` other than `A` to make sure `A` does not proceed. If the condition in the while loop would have been $>$ and not $>=$ having all the B's still in the $j-1$ level would mean A need not be blocked.

To further illustrate the subtlety of having $>$ instead of $>=$ in the while loop consider the following scenario. Suppose there are n processes and let us focus on only two of them namely process 1 and process 2.

1. Let 1 and 2 be at the first level where 2 is the last to write to `TTW` at this level. Suspend 1 by the OS just before it passes to level 2 at the

end of the iteration of the for loop associated with the first level (i.e., line 11 in the listing above).

2. As `level[1] = 1` and the condition in the while loop is with $>$ and not $>=$ process 2 will not be blocked by the while loop as the first condition does not apply to it.

3. This can happen at every level up until the $(n-1)$st level.

4. When this happens at the $(n-1)$st level we get a violation to mutual exclusion.

Claim 7.3. *For every $j \in [0, n-1]$ the number of processes at level j is $\leq n - j$.*

Proof. The proof is by induction on j. For $j = 0$ the claim holds trivially. Assume then that the claim holds up to level $j-1$ and consider the jth level. That is in level $j-1$ there is a set $\mathcal{P}$ of processes satisfying $|\mathcal{P}| \leq n-(j-1)$. If $|\mathcal{P}| < n-(j-1)$ then $|\mathcal{P}| \leq n-j$ and as only processes at level $j-1$ can pass to level j the claim follows trivially in this case. We may assume then that $|\mathcal{P}| = n-(j-1)$. Consequently, no other processes are joining the $(j-1)$st level by the induction hypothesis.

Let $A \in \mathcal{P}$ be the last process in $\mathcal{P}$ to set `TTW[j]`. By Claim 7.2 A does not pass to level j until all of $\mathcal{P} \setminus \{A\}$ clears the CS. The claim follows. ■

Theorem 7.1 now follows by setting $j = n-1$ in Claim 7.3.

Theorem 7.4. *The filter algorithm is starvation-free.*

This theorem follows from the following stronger claim.

Claim 7.5. *For every $j \in [0, n-1]$ if a process reaches level j then it eventually enters the CS.*

Proof. The proof is by induction on j. For $j = n-1$ the claim is trivial. Assume the claim is true for level $j+1$ and above and consider the jth level. Assume towards contradiction that there exists a process A that is stuck forever in level j attempting to pass to level $j+1$. Hence there exists a certain point in time such that beyond it

`level[A] = j+1` and `TTW[j+1] = A` hold forever.

1. As `TTW[j+1] = A` forever no process coming from lower levels tries to bypass A.

2. By the induction hypothesis after a finite amount of time there are no processes in the levels $j+1$ and above.

3. After a finite amount of time there can be no additional processes with A at level j. Indeed, as `TTW[j+1] = A` forever any such process will eventually pass to level $j+1$ and from there enter the CS by the induction hypothesis and in any reentry attempt cannot bypass A.

It now follows that A is alone in its level, no processes are interested in level above its own level. The condition in the while loop now breaks and A passes to level $j+1$; contradiction. ■

One serious drawback to the filter algorithm is that it is quite "unfair" in the following sense. Let us recall Definition defining the notion of *r-bounded synchronisation algorithms*. In particular, for $r \in \mathbb{N}$ a synchronisation algorithm is said to be *r-bounded* if no process upon entry attempt can be bypassed more than r times.

Theorem 7.6. *The filter algorithm is not r-bounded for every $r \in \mathbb{N}$.*

Proof. Fix $r \in \mathbb{N}$. Assume three processes are engaging in the filter algorithm.

1. Initially A is at level 1, B is in the CS, C at level 0, and `TTW[2] = A`.

2. Process C arrives at level 1 and sets `TTW[2] = C`.

3. B clears the CS, attempts reentry and joins A and C at level 1. It sets `TTW[2] = B`.

4. Process C (and not A) gets running time.

In this manner let B and C replace one another at level 1 for at least $r+1$ rounds. (This scenario does not imply starvation of A. As we can keep doing so only a finite amount of time before we must let A run and when it does it will move to the next level).

One should note that for this scenario it is crucial that A is put at level 1 and not level $2 \leq j < n-1$. For indeed, in this case processes B and C cannot catchup to it once they start from level 0. For instance if A is at level 2 say and B starts from level 0 then B cannot pass to level 1 until A clears all levels and exits the CS. ■

§8. Lamport's bakery algorithm

We have seen that the filter algorithm is not r-bounded for any $r \in \mathbb{N}$. We are now interested in developing synchronisation algorithms with the added restrictions that these be fair in some sense. For now we use r-boundedness to measure fairness. In this section we present the so-called *Lamport's bakery algorithm* which makes an attempt to solve this problem by letting the processes mimic a queue is, say, a bakery. That is, each process that attempts entry is assigned a ticket and admission into the CS is with respect to the relative order of the tickets.

For this algorithm we maintain two shared arrays of length n, the number of processes. The first array is `inter[n]` and the second is `number[n]`.

```
1  inter[i] = True
2
3  number[i] = max (number[1],...,number[n])+1
4
5  while( exists k != i s.t.:
6
7           inter[k] == True
8                 and
9           (number[i],i)> (number[k],k)){}
10
11
12 <CS>
13
14 inter[i] = False
```

Listing 57: Lamport's bakery algorithm: code for process i

In line 9 of Listing 57 we can see the definition of tickets in Lamport's algorithm. A ticket is a pair of numbers of the form `(number[i],i)`. The ordering imposed on those is the lexicographic order of the pairs. Indeed, note all n processes may end up with the same number. Lamport's algorithm uses the identities of the processes to break ties.

Lamport's algorithm being starvation-free is a triviality.

Theorem 8.1. *Lamport's bakery algorithm is starvation-free.*

Proof. As there is always a unique process A minimising $(number[A], A)$ such a processes can always enter the CS; hence Lamport's algorithm is deadlock-free. To attain starvation-freedom note that Lamport's algorithm is n-bounded. Indeed, once a process P has a number per line 3 it can be passed by all other processes; each of which will loose to P upon reentry attempt. ∎

Theorem 8.2. *Lamport's bakery algorithm provides mutual exclusion.*

Proof. Assume towards a contraction that process A and B are in the CS at the same time namely t_0. Let n_A and n_B denote the values seen at `number[A]` and `number[B]` when A and B entered the CS, respectively. Assume without loss of generality that

$$(n_A, A) < (n_B, B). \tag{8.3}$$

B entered the CS as at the time of it executing its while loop it noticed at some time $t_1 < t_0$ that

$$\text{either (i) } (n_B, B) < (\texttt{number[A]}, A) \text{ or (ii) } \texttt{inter[A] == False}.$$

Option (i) cannot occur; indeed up until time t_0 $(\texttt{number[A]}, A) \leq (n_A, A)$ holds as we never reset the entries of the `number` array; hence it follows in this case that $(n_B, B) < (n_A, A)$ contradicting (8.3). Hence option (ii) must have occurred. In which case we notice the following chain of events in time:

$$\begin{aligned} n_B \text{ is set} &\to \text{Read}_B(\texttt{inter[A] == False}) \\ &\to \text{Write}_A(\texttt{inter[A] == True}) \\ &\to n_A \text{ is set;} \end{aligned}$$

This then means that $n_B < n_A$ contradicting (8.3). ∎

§8.1 Fairness of Lamport's algorithm. In the literature Lamport's algorithm is celebrated as 0-bounded. Nevertheless we will prove the following.

Proposition 8.4. *Lamport's algorithm is not* 0*-bounded.*

Proof. Consider a situation where the processes read the array `number` at line 3 as follows: all processes read `number[1]`, they they all read `number[2]` and so on. At the end they all end up with the same number. Assume that the process with identity n was the one to start the entry attempt first; nevertheless it would be bypassed $n - 1$ times. ∎

How to bridge this gap between what we have just seen and what is commonly known in the literature? The answer lies in the definition of r-boundedness. In the definition presented thus far we begin counting bypasses over a given process from the moment it started the entry code. A more refined variant

would have us count bypasses only from the point in the entry code in which a process enters a "waiting" state.

In Lamport's algorithm we see that up until line 5 (not including) there is no mechanism amongst these lines to delay a process. We refer to this stretch of code as the *doorway* of the entry code. Lines 5 to 9 is where potentially a process can be delayed; we refer to the part of the entry code where a process can be delayed as the *waiting area*.

Definition 8.5. *Let $r \in \mathbb{N}$. A synchronisation algorithm is said to be* r-waiting-bounded *if once it has finished the doorway area of the synchronisation algorithm it can be bypassed at most r times.*

In the literature the definition just presented is the common definition for r-boundedness. In our presentation we chose to make a distinction in order. Indeed, The scenario presented in the proof of Proposition 8.4 illustrates how meaningless is the definition we gave for r-boundedness as with such a definition no fairness assumption can be imposed. Note now that

Proposition 8.6.

1. *The filter algorithm is not r-waiting-bounded for any $r \in \mathbb{N}$.*
2. *Lamport's bakery algorithm is* 0*-waiting-bounded.*

Proof.

1. The same proof presented Theorem 7.6.
2. Once a process is in the waiting area of Lamport's algorithm all process that join the waiting area after it have a higher number.

■

§8.2 Unbounded tickets. The filter algorithm being exceedingly unfair (see Proposition 8.6) prompted no urge on our part to actually implement it. Lamport's algorithm however, being 0-waiting-bounded (and thus FIFO in a certain sense) does prompt questions regarding its implementation. The first issue one needs to confront here is the fact that Lamport's algorithm is currently employs unbounded tickets. This clearly cannot be implemented on some computers. In this day and age of 64 bit registers we can allow ticket numbers to inflate to roughly 2^{64} on a single register which essentially means we can leave this issue of bounding the ticket numbers on such computers. Nevertheless, not every computer is of this nature so for now we keep

discussing this issue. It turns out that this is not so trivial to attain as the following naive implementation attempt demonstrates quite vividly.

```
number[i] = max (number[1],...,number[n])+1

for each j != i do:
  await((number[j] ==0 )
            or
          (number[j] > number[i]))

<CS>

number[i] = 0
```

Listing 58: Lamport's algorithm: naive implementation

This implementation is not deadlock-free; indeed, if all processes have number 1 assigned to them then no one enters. In Listing 58 we did not employ the lexicographic order over pairs that Lamport's algorithm employs. Let us rewrite the algorithm to include this feature.

```
number[i] = max (number[1],...,number[n])+1

for each j != i do:
  await((number[j] ==0 )
            or
          (number[j],j) > (number[i],i))

<CS>

number[i] = 0
```

Listing 59: Lamport's algorithm: naive implementation

This implementation fails at delivering mutual exclusion as the following scenario demonstrates.

1. Let us consider three process amongst the n process and say that their identities are 1,2,3. Suppose that process 3 is in the CS and that `number[3] = 1` while all other entries in the `number` array are 0.

2. Process 1 reads the `number` array. It reads `number[2] = 0` and is suspended (before reading `number[3]`.

3. Process 2 sets `number[2] = 2`.

4. Process 3 leaves the CS and sets `number[3] = 0`.

5. Process 2 enters the CS.

6. Process 1, after already reading `number[2]` to be 0 continues reading the `number` array and as a result sets `number[1] = 1`.

7. Process 1 enters the CS (while Process 2 is still in the CS).

This scenario demonstrates our need to know when a process is in the process of choosing a number and when it has such a number already set of it. To that end we introduce another shared boolean array `choosing[n]` to be used as follows.

```
1  choosing[i] = True
2
3  number[i] = max (number[1],...,number[n])+1
4
5  choosing[i] = False
6
7  for each j != i do:
8
9    await(choosing[j] == False)
10
11   await((number[j] ==0 )
12               or
13           (number[j],j) > (number[i],i))
14
15 <CS>
16
17 number[i] = 0
```

Listing 60: Lamport's algorithm: introducing choosing

This implementation can handle the scenario above.

1. When process 1 read `number[2]` and is suspended one still has `choosing[1] = True`.

2. Process 2 sets `number[2] = 2`.

3. Process 3 leaves the CS and sets `number[3] = 0`.

4. Process 2 cannot enter the CS as it is delayed by `await(choosing[1] ==False)`.

5. Process 1 finishes reading the `number` array and sets `number[1] = 1`.

6. Process 1 can pass the for loop and it enters the CS prior to Process 2.

While we have been able to resolve the scenario from above note that in this scenario Process 2 started the waiting area before process 1 yet process 1 is able to bypass it and enter the CS before it. So in this implementation we already lost the 0-waiting-boundedness property that Lamport's algorithm is known for. Nevertheless, we prove the following.

Claim 8.7. *The algorithm in Listing 60 provides mutual exclusion.*

Prior to proving this claim we make the following observation.

Observation 8.8. *If process i notices that*

$$\texttt{number[j] == 0} \quad \textit{and} \quad \texttt{choosing[j] == False}$$

for some $j \in [n] \setminus \{i\}$ then process j is between exiting the CS and choosing another number. Whatever this number will be it will satisfy $>$ `number[i]` assuming i did not enter and exit the CS.

Proof of Claim 8.7. At any given moment we may partition the set of process competing for the CS via the algorithm seen in Listing 60 into three sets:

$$\begin{aligned}
\mathcal{P}_1 :=& \{i \in [n] : \texttt{choosing[i] == True}\};\\
\mathcal{P}_2 :=& \{i \in [n] : \texttt{choosing[i] == False} \text{ and } \texttt{number[i] == 0}\};\\
\mathcal{P}_3 :=& \{i \in [n] : \texttt{choosing[i] == False} \text{ and } \texttt{number[i] > 0}\}.
\end{aligned}$$

If there is a violation to mutual exclusion it only involves process in $\mathcal{P}_3$. Process in $\mathcal{P}_1$ are trivially excluded and processes in $\mathcal{P}_2$ are excluded by Observation 8.8. Amongst the members of $\mathcal{P}_3$ only one process can enter the CS; the one with the lexicographic smallest pair. ■

While matters appear promising, before we continue exploring the properties of this algorithm lets us note that this implementation does not solve the problem of having unbounded tickets as well. Moreover, this despite the fact that it resets the entries of the `number` array upon leaving the CS.

Lemma 8.9. *The implementation of Lamport's bakery algorithm in Listing 60 does not guarantee that the tickets are bounded.*

Proof. Consider the following scenario.

1. Initially the array satisfies the property `number[i] = i` for $i \in [n]$.
2. Processs start to enter and leave the CS and attempt reentry.

3. As long as process n does not enter the cs and clears it all processes attempting reentry will be assigned number $n+1, n+2, \ldots$, and so on.

4. Now Process n enters the CS, leaves the CS, and attempts reentry. It is assigned the number $2n$.

5. This pattern repeats itself forever leading for the number to not be bounded.

■

§9. Aravind's algorithm

There are several algorithms that resolve the issue of the unbounded ticket numbers in Lamport's algorithm. In this section we present one of these which is called *Aravind's algorithm*. We start with a formulation of Aravind's algorithm in which the tickets are unbounded. Later on we shall alter the code slightly in order to accommodate bounded tickets.

§9.1 Unbounded dates version. The following is the shared data assumed by the algorithm.

1. A boolean array named `inter[n]` initialised all to false.

2. An integer array named `Date[n]`. `Date[i]` denotes the logical date of process i's next request for the CS. Initially the array satisfies `Date[i] = i` for every $i \in [n]$. That is, we use "dates" instead of "tickets" to control access to the CS.

3. A boolean array `Stage[n]` initialised all to false.

```
1
2  inter[i] = True
3
4  repeat:
5
6      Stage[i] = False
7
8      await(forall j != i:
9            inter[j] == False or Date[i] < Date[j])
10
11     Stage[i] = True (L1)
12
```

```
13 until (forall j!= i: Stage[j] == False)  (L2)
14
15 <CS>
16
17 Date[i] = 1 + max(Date[1],...,Date[n])
18
19 Stage[i] = False  (L3)
20
21 inter[i] = False
```

Listing 61: Aravind's algorithm: code for process i

For each process we observe two stages.

- `Stage[i] = False` - before process i learns it has the smallest date.
- `Stage[i] = True` - after process i learns it has the smallest date amongst the interested processes.

The `until` clause makes sure that the following scenario does not happen.

1. Suppose processes i and j compete for the CS and that
   (a) `Date[i] < Date[j]`,
   (b) `inter[i] = False`,
   (c) `Stage[i] = False`.
2. Let process j enter the CS be halted there.
3. Without the `until` clause process i can enter the CS as well.

This scenario demonstrates that insisting only on having the least date is insufficient in order to ensure mutual exclusion.

Prior to proving that Aravind's algorithm provides mutual exclusion we set up the following notation. Let i and j be two processes.

- We write $W_s(i, L1)$ to denote the time process i starts writing `Stage[i]` at the line marked L1 in Listing 61.
- We write $W_f(i, L1)$ to denote the time process i finishes writing `Stage[i]` at the line marked L1 in Listing 61.
- We write $R_s(i, j, L2)$ to denote the time process i starts reading `Stage[j]` at the line marked L2 in Listing 61.
- We write $R_f(i, j, L2)$ to denote the time process i finishes reading `Stage[j]` at the line marked L2 in Listing 61.

Theorem 9.1. *Aravind's algorithm provides mutual exclusion.*

Proof. Assume for the sake of contradiction that processes i and j are in the CS at the same time. As i is in the CS it had to read `Stage[j] == False` at L2.

$$R_s(i, j, L2) < W_f(j, L1). \tag{9.2}$$

By i's code

$$W_f(i, L1) < R_s(i, j, L2). \tag{9.3}$$

By j's code

$$W_f(j, L1) < R_s(j, i, L2). \tag{9.4}$$

Combining the above inequalities we arrive at

$$W_f(i, L1) \overset{(9.3)}{<} R_s(i, j, L2) \overset{(9.2)}{<} W_f(j, L1) \overset{(9.4)}{<} R_s(j, i, L2).$$

The resulting inequality $W_f(i, L1) < R_s(j, i, L2)$ means that process i finished writing `Stage[i] = True` before process j was able to read it at L2, contradicting the assumption that j was able to clear L2 and enter the CS. ■

Let us examine two scheduling scenarios to better understand this algorithm.

Scenario I. Initially all processes start out as not interested. Now let the all processes but the one whose date is equal to 1 into the entry code one after another in descending order as follows. First let the process whose date is equal to n reach (L1) and execute it and halt it there. Now do the same for the process with date $n - 1$ and so on. Finally do this with the process whose date is 1. Currently all processes have their `stage` flag set to true and all are interested. All now fail the `until` clause and repeat the loop. All but the the process with date equal to 1 get stuck in the await command. The process with date 1 enters the CS.

Scenario II. Initially all processes are not interested. Let the processes with data equal to n enter the CS and halt it there. Now let the rest of the processes execute the entry code. All will fail to pass the `until` clause despite having lower dates than n as `stage[n]` is set to true.

Observation 9.5. *The CS extends until L3.*

Proof. A process is able to clear L2 only if it is the sole process satisfying `Stage[i] = True` at time of executing L2. Hence, setting a `stage` entry to `False` enables processes to enter the CS. In particular once a process is in the CS (it has its `stage` flag set to true) no other process can pass the `until` clause until the process in the CS executes line (L3). ■

We can conclude the following.

- `Date`s are assigned to processes inside the CS.
- This fact together with the fact that the `Date` array is initialised to distinct numbers implies that

  throughout the algorithm the dates are kept as distinct numbers; (9.6)

  a situation that does not occur in Lamport's bakery algorithm.

Theorem 9.7. *Aravind's algorithm is deadlock-free.*

Proof. Assume towards contradiction that there is a set $\mathcal{P}$ of at least two processes that are all stuck forever in their entry code. By (9.6) there exist a $q \in \mathcal{P}$ with a least date. After a finite amount of time q assumes the least date amongst all processes in contention for the CS (these may include processes not in $\mathcal{P}$). Hence after a finite amount of time all processes in contention for the CS cannot pass the `await` command due to `inter[q] = True` and q's date being the least date. However, at this point q can pass the `await` command. Moreover, as all other processes in contention are stuck in their `await` command q is the sole process whose `stage` field is true and thus can pass the `until` clause at (L2) and enter the CS. This is a contradiction. ■

Observing the Aravind's algorithm is $(n-1)$-waiting-bounded together with the fact that it is deadlock-free implies the following.

Corollary 9.8. *Aravind's algorithm is starvation-free.*

§9.2 Bounded dates version. The formulation presented in Listing 61 for Aravind's algorithm does not solve the problem for which we took interest in this algorithm to begin with; that being having bounded tickets. We now address this issue. Let N be a parameter to be chosen later and let us alter the exist code of Listing 61 as follows.

```
1
2  inter[i] = True
3
4  repeat:
5
6      Stage[i] = False
7
8      await(forall j != i:
9           inter[j] == False or Date[i] < Date[j])
10
11     Stage[i] = True (L1)
12
13 until (forall j!= i: Stage[j] == False)  (L2)
14
15 <CS>
16
17 Date[i] = 1 + max(Date[1],...,Date[n])
18
19 If(Date[i] >= N){
20     set Date[j] = j for every j
21 }
22
23 Stage[i] = False  (L3)
24
25 Inter[i] = False
```

Listing 62: Aravind's algorithm: code for process i

By Observation 9.5 the CS extends until (L3). In Listing 62 we reset the entire `Date` array as soon as a process i prior to leaving the CS detects that its date is $\geq N$. The entire `Date` array is reset while a process is in the CS. As all tickets are also issued inside the CS (by Observation 9.5) this is legitimate.

A key issue now is the choice of the value of N. Choosing a value for N that is too low can have serious effect on the performance of the algorithm. Suppose for instance that we set $N = n$. In this case every process that leaves the CS would reset the entire `Date` array. This in particular means that there can be processes in the entry code whose date is reset $\Omega(n)$ times. Moreover, the process whose initial date is n can starve.

Theorem 9.9. *If $N \geq 2n$ then a process in the entry code will experience a reset of its `Date` field at most once.*

Proof. Let P be a process that while in its entry code experiences a rest to its `Date` field. Upon this event it has to compete "from scratch" with all contenders. In the worst case that P's initial date in n, P will be bypassed

$\leq n-1$ times. After all $n-1$ other processes exit the CS the highest ticket number these can reach is $\leq n+(n-1) = 2n-1 < 2n$ so that another reset will not occur until the process P enters the CS and reaches ticket $2n$ which in turn prompts a rest to the dates. ■

§10. The black and white bakery algorithm

Aravind's algorithm resolved the issue of the unbounded ticket numbers in Lamport's algorithm by resorting to issuing ticket numbers within the CS and reseting those in the CS as well. In this section we will consider an algorithm that assigns tickets outside the CS and at the same time is able to bound the ticket numbers.

The algorithm requires the following shared data.

- A bit called `colour` and it can take values in the set $\{B, W\}$ where B stands for black and W for white.
- An array `choosing[n]` which is a boolean array.
- An array `MyColour[n]` whose entries assume values in the set $\{B, W\}$.
- An integer array `number[n]`.

Initially we have

`colour = W`;
`number[i] = 0` for every process i;
`choosing[i] = False` for every process i.

```
1
2  choosing[i] = True
3
4  MyColour[i] = colour
5
6  number[i] =
7     1+ max{number[j]: MyColour[j] == MyColour[i]}
8
9  choosing[i] = F
10
11 For j = 1 to n, j!=i do{
12
13     await(choosing[j] == False)
14
15     if (MyColour[j] == MyColour[i]){
16
17       await(number[j] == 0
18                     or
```

```
19                (number[j],j) > (number[i],i)
20                      or
21                MyColour[j] != MyColour[i])
22        }
23        else{
24          await(number[j] == 0
25                      or
26                MyColour[i] != colour
27                      or
28                MyColour[j] == MyColour[i])
29        }
30  }
31  <CS>
32
33  If (MyColour[i] == B){
34
35        colour = W
36  }
37  else{
38        colour =B
39  }
40
41  number[i] = 0
```

Listing 63: Black and white bakery algorithm: code for process i

Prior to analysing the properties of this algorithm let us first consider the following running example.

1. Initially there are n processes all assigned the colour W.
2. Process 1 enters the CS and upon exit it flips the colour bit to `colour == B`.
3. Process 1 attempts reentry. Upon reentry it assumes the colour B that is now different from the colour of all other processes.
   (a) As a result it falls into the `else` clause in the `for` loop for every process coloured W (which is now every one but it) .
   (b) To beat a given process $j > 1$ process 1 has to wait until:
      i. `number[j] == 0`; or
      ii. `MyColour[1] != colour`; as `MyColour[1] == B` now process 1 has to wait until `colour != B`. Observe that even if all white processes enter and leave the CS they all "flip" the `colour` bit to B which is its current colour. This condition for which process 1 is waiting for can only happen once a process coloured black enters and leaves the CS which will

cause the `colour` to flip to white. Currently 1 is the sole process coloured black.

iii. Or `MyColour[j] == MyColour[i]`; as `MyColour[i]` is black and `MyColour[j]` is initially white such an equality can only happen once j enters the CS, leaves the CS "flips" the `colour` bit to black (which is its current colour) and then attempts reentry upon which j is assigned the colour black the same as process 1. In this scenario this means that if an additional black coloured process arrives we let process 1 continue.

For $i \in [n]$ define $Ticket_i := (MyColour[i], number[i])$; we refer to such pairs as *coloured tickets*. Upon entry attempt processes compete by comparing their coloured tickets. Given $Ticket_i$ and $Ticket_j$ the `if-else` clause in the `for` loop considers two complementary cases; either

$$\texttt{MyColour[i]} = \texttt{MyColour[j]}$$

or

$$\texttt{MyColour[i]} \neq \texttt{MyColour[j]}$$

In each case we have to decide whether it is i or is it j who wins sort of speak.

IF CLAUSE. If $Ticket_i$ and $Ticket_j$ are coloured the same then the one having the lexicographically least pair `(number,id)` pair wins (as in the classical Lamport's bakery algorithm). This vividly indicates that processes of the same colour compete according to the rules of the classic Lamport's bakery algorithm

ELSE CLAUSE. From the `else` clause we learn that if $Ticket_i$ and $Ticket_j$ have different colours then the process whose ticket is coloured *differently* from the `colour` bit wins. More precisely:

> If `MyColour[i] != MyColour[j]` and `MyColour[i] == colour` then $Ticket_j < Ticket_i$ (and we let i get stuck in the `await` command).

Put another way, processes whose colour coincides with the bit `colour` have to wait; that is, if `colour == W` then black process have the right away and if `colour == B` then white processes have the right away.

Scenario. Consider the following.

1. Initially all processes are coloured white.
2. These enter one by one to the CS according to the classical Lamport's bakery algorithm as this is implemented in the `if` clause of the `for` loop.
3. Processes that attempt reentry are coloured black and have to wait until all white processes have been dealt with.
4. Let us observe how black coloured processes are now handled in the following continuation of this scenario. To that end let us mark two processes P and Q.
   (a) Assume P is the first process to be coloured black (i.e., the first to attempt reentry).
   (b) Suppose that P considered Q when Q is still white leading to P getting stuck in the `await` command of the `else` clause.
   (c) Let Q attempt reentry, switch its colour to black and consider the point in time in which Q considers P in its `for` loop. As P and Q are both black Q is led to the `if` clause. There it will lose to P in the `await` command as P already has a number (recall it is in its `else` clause in this scenario). Hence Q is now stuck in the `await` command.
   (d) Meanwhile P cannot be withheld by the `await` command in the `else` clause due to Q anymore. Indeed, Q has now switched to the same colour as P, namely black, and thus releasing P.

This scenario demonstrates the following. For processes coloured in the same colour the black and white bakery algorithm performs the classical Lamport's bakery algorithm. In addition, it performs strict alternation between the colours. First the white processes are processed via Lamport's classical algorithm; processes that attempt reentry switch colour and await until the white processes are handled and then Lamport's algorithm commences once again over the black coloured processes.

Scenario. Assume now that of the n processes competing k are coloured black and $n - k$ are coloured white and `colour = B` so that the white processes have the right away into the CS while the black ones are coloured so due to a re-entry attempt on their part. We observe the black processes.

1. Every black process executes its for loop.

2. Whenever it considers a black process it falls into the `if` clause and whenever it considers a white process it falls into the `else` clause.

3. Let P be a black process.

4. For every white process P is delayed in the `await` of the `else` clause.

5. For every black process the rule of Lamport's algorithm determine whether P will be delayed or not.

THE THIRD CONDITION. In each `await` command there is a third condition. The purpose of this condition is to check whether a re-entry attempt has been carried out by the process j up against i is competing.

1. In the `if` clause i and j are coloured the same. If the third condition occurred in the associated `await` command this means j has changed its colour which is only possible due to a reentry attempt. This means that j has now the 'new' colour while i still holds the 'old' colour. Hence, i has the right away prior to j into the CS.

2. In the `else` clause i and j are coloured differnetly. If the third condition occurred in the associated `await` command this means j has changed its colour which is only possible due to a reentry attempt. Recall however, that i is in the `else` clause due to a re-entry attempt it is carrying out. That is, when i entered the `else` clause its colour was the 'new' colour and j has the 'old' colour. If j had flipped its colour and has joined i's 'new' colour then we would like to let Lamport's rules take over. In this case it is clear that i already holds the lower number as j attempted re-entry after i has already assumed the 'new' colour.

The strict alternation between the colours is not yet clear. We delve into this issue next.

DEFINITION 10.1. *Let t be a point in time in which `colour` $= c \in \{B, W\}$. A point in time t' satisfying:*

1. *$t' \leq t$; and*

2. *at time t' `colour` changed from the complementary colour of c into c; and*

3. *$t - t'$ is minimal*

is called the most recent change of `colour` into c with respect to t.

Lemma 10.2. *Let $t_1 > t_2 > t_3$ be times instances such that the following holds for these:*

- *At time t_1 the bit* `colour` $= c \in \{B, W\}$.
- *t_2 is the most recent change of* `colour` *into c with respect to t_1.*
- *t_3 is the most recent change of* `colour` *into d (complementary colour of c) with respect to t_2.*

Then between t_2 and t_1 any c-coloured process attained its colour after t_2.

Proof. For suppose that between t_2 and t_1 there is a c-coloured process that attained its colour before t_3. Then between t_3 and t_2 this process is c-coloured and every d-coloured process will have to yield to this process implying that the change into c (from d) at time t_2 cannot take place as we are assuming this c-coloured process persists until after t_2. ■

More generally we can prove the following.

Lemma 10.3. *Suppose that at time t:* `colour` $= c \in \{B, W\}$. *Then any process which at time t is in its entry code and whose colour is not c will enter the CS prior to any c-coloured process.*

Proof. Let d denote the complementary colour to c. Let t' be the most recent change of `colour` into c with respect to t. By Lemma 10.2, all c-coloured processes found in their entry code at time t are those processes who attempted (re)entry during the time interval $[t', t]$. All d-coloured processes found in their entry at time t assumed their colour prior to t'. Hence, during the interval $[t', t]$ the colour c is the 'new' colour while d is the 'old' colour.

Let P be a c-coloured process found in its entry at time t and let Q be a d-coloured process found in its entry at time t. Let $t_{P,Q} \geq t'$ denote the time in which P considers Q in its `for` loop after P is c-coloured. If at time $t_{P,Q}$ the process Q is still d-coloured then P enters its `else` clause and loses to Q in the associated `await` command. The claim follows in this case. Otherwise, if at time $t_{P,Q}$ the processes Q is c-coloured then that in particular means that Q already entered the CS as a d-coloured process prior to P (who at time $t_{P,Q}$ is still in its entry code) and the claim holds trivially in this case as well. ■

Corollary 10.4. *Suppose that at time t the bit* `colour` *changes from*

colour $d \in \{B, W\}$ to the other colour namely c. Then all processes found in their entry code at time t are d-coloured.

Proof. By assumption, at time $t - \varepsilon$ for some $\varepsilon > 0$ `colour = d`; hence, by Lemma 10.3, all c-coloured process found in their entry code at time $t - \varepsilon$ must enter the CS prior to all d-coloured processes found in their entry at time $t - \varepsilon$. As at time t a d-coloured process leaves the CS and flips `colour` from d to c together with the fact that ε is arbitrary implies that there exists an $\varepsilon' > 0$ such that at time $t - \varepsilon'$ there are no c-coloured processes in their entry code. ■

The main reason for considering this algorithm was to prove that it solves the issue of having the values of `number[i]` being unbounded. We address this issue next. Prior to doing so let us just remark that the scenario considered in the proof of Lemma 8.9 cannot happen in the B&W-bakery algorithm. The point of that scenario was that of keeping a process with the largest turn in the entry code allowing it to be passed by its peers each then attempting reentry and thus picking numbers exceeding the maximum ticket held by the process in the entry. Here these processes upon re-entry switch their colour and consequently would not consider the ticket held by the process we held in the entry as it is coloured with a different colour.

Proposition 10.5. *`number[i]` $\leq n$ for every $i \in [n]$ throughout the algorithm.*

Proof. Assume towards a contradiction that the claim is false and let $i \in [n]$ be a process that sees `number[i] > n` somewhere throughout the run of the algorithm. Let t_1 denote the time in which this event took place. Let c be the colour of the bit `colour` at time t_1 so that i has chosen its colour according to the set of processes coloured c. Let $t_0 \leq t_1$ denote the most recent change of `colour` into c with respect to t. At time t_0 all processes found in their entry code are all coloured by the complementary colour to c, by Corollary 10.4. Between t_0 and t_1 (not including) the number of c-coloured tickets issued is $\leq n - 1$ (not including i); moreover all these tickets are bounded by $n - 1$ as well as all processes taking turns according to colour c are switching their colour from the complementary colour hence all have zeroed their `number` field upon leaving the CS. ■

§11. Tournament based algorithm

Assume n processes are involved in an algorithm synchronising entrance into a CS. A scenario in which only one is interested to enter the CS is said to be *contention-free*. All synchronisation algorithms considered so far maintained shared data structures whose size is asymptotically equivalent to the number of processes competing for the CS. Upon entry attempt a process had to check the status of all involved processes regardless of whether these are actually competing or not. Put another way, all algorithms considered so far resorted to having each interested process issue $\Omega(n)$ queries in order to determine whether it is allowed to enter the CS. Such a cost makes sense in the presence of contention. In contention-free scenarios this makes little sense. In this section we commence our treatment of the question whether it is possible to design synchronisation algorithms that would render a significantly lower cost in contention-free scenarios.

In this section we shall show how to reduce the number of queries from $\Omega(n)$ to $O(\log_2 n)$. Although this delivers an exponential improvement the algorithms considered in this section will have the cost at $O(\log_2 n)$ queries per process regardless of whether the scenario is contention-free or not. In § 12 and § 13 we shall start addressing contention-detection and lowering the number of queries per process upon detecting that there is no contention for the CS.

A simple principle through which to reduce the number of queries is to have the processes engage in a tournament implemented using a (complete) binary tree. For convenience we shall assume in this section that the number of processes n satisfies $n = 2^k$ for some k. The algorithm starts with all n processes "appearing" at the leaves of a complete binary tree. The internal nodes of the tree represent instances of our favourite 2 process synchronisation algorithm, say Peterson's algorithm. The goal of each process is to travel from its initial leaf, reach the root of the tree, where the root is the final "Peterson Lock" protecting the CS. "Winning" the Peterson Lock found at the root enables a process to enter the CS. The benefit of this approach is that every process as it travels to the root has to compete with $O(k) = O(\log_2 n)$ processes along the way as this is the height of the tree which determines the number of Peterson Locks that a given process has to obtain while travelling up the tree.

We now make this precise. Represent the Peterson Lock with a class having two main methods

`void exit(int id)` and `void enter(int id)`.

If p is an instance of this class then `p.enter(1)` denotes the application of the entry code of Peterson's algorithm where the invoking process assumes the

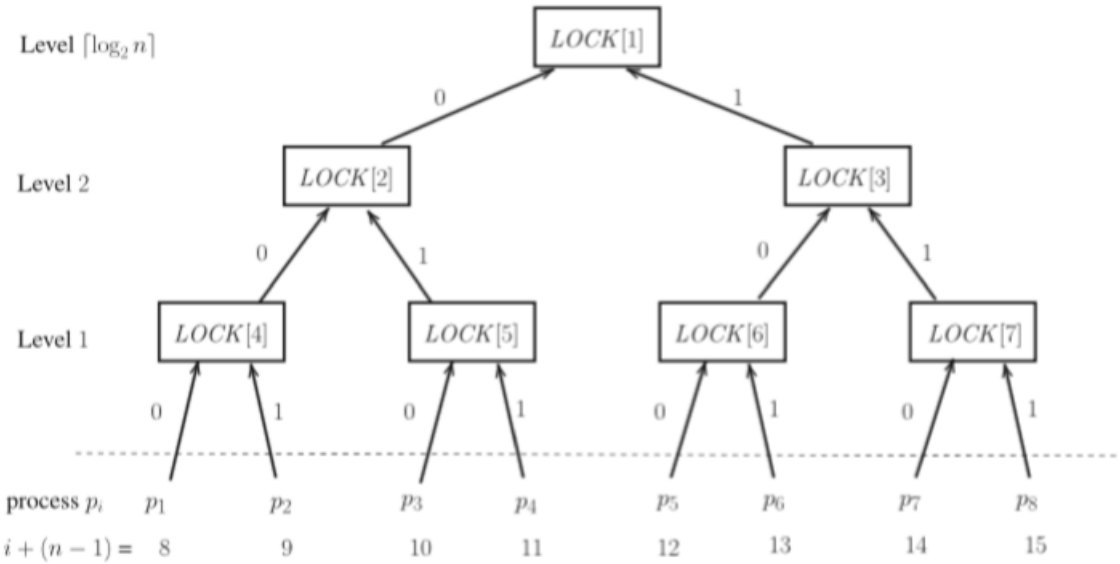


Figure 1: A complete binary tree of Peterson's algorithm for $n = 8$.

rôle of process 1 in that algorithm. `p.enter(0)`, `p.exit(0)`, and `p.exit(1)` are all defined in a similar manner.

The complete binary tree is implemented using an array

```
Peterson LOCK[1,...,n-1]
```

such that each cell of this array houses a different instance of class Peterson. That is array `LOCK` represents the internal nodes of the tournament:`LOCK[1]` is the root, and the children of `LOCK[x]` are `LOCK[2x]` and `LOCK[2x+1]`. The binary tree represented by array `LOCK` has $n-1$ internal nodes and thus has $n/2$ leaves. We create a "virtual" tree from the tree represented by `LOCK` by attaching each leaf of the tree represented by `LOCK` two children which now become leaves themselves. In total we attach n new leaf nodes and in those we "implicitly" house the n processes.

The following listing depicts the entry code for process i in the tournament algorithm.

```
1 enter_cs(i){
2
3     cur-node = i + (n-1)
4
5     for level = 1 to (log_2 n) {
```

```
6
7           my-id[level] = cur-node %2
8
9           cur-node = floor(cur-node/2)
10
11          LOCK[cur-node].enter(my-id[level])
12      }
13
14      return
15
16  }
```

Listing 64: Peterson tournament entry code for process i

The procedure `enter-cs(i)` receives the id of the process. It implicitly places it in leaf number $i + (n - 1)$. To understand this point recall that there are $n - 1$ internal nodes in `LOCK` and and additional n leaves to the tree we had constructed from the tree represented by `LOCK`. Naturally we do not need actual array slots to house the processes in. All we need to start climbing the tree is the index of the leaf in the resulting array-based tree in which we can place the process so it could know who its parent is. This is done through the line 3: `cur-node = i + (n-1)`. In the `for` loop one can see how process i attempts to climb the tree as it keeps updating `cur-node`. Prior to advancing in the tree it has to "win" the Peterson Lock associated with the internal node indicated by `cur-node`.

Each process maintains a local array called `my-id` whose size is equal to the height of the tree. The value of `my-id[level]` is either 0 or 1 and it indicates which rôle process i intends to assume upon engaging the next Peterson Lock pointed to by `cur-node`.

In order to release the CS a process has to backtrack the route it took from its initial leaf to the root. This in order to release all the Peterson Locks it acquired through its climb. This is presented next.

```
1   exit_cs(i){
2
3       cur-node = 1
4
5       for level (log_2 n) to 1 do{
6
7           LOCK[cur-node].exit(my-id[level])
8
9           cur-node = 2 x cur-node + my-id[level]
10      }
11
12      return
13
14  }
```

Listing 65: Peterson tournament exit code for process i

The line `cur-node = 2 x cur-node + my-id[level]` updates the next node to which process i has to descend to depending upon the rôle it assumed (either 0 or 1) when it obtained that Peterson Lock last.

An interesting key feature of the above tournament based algorithm is that unlike all other algorithms that came before here a process has no access to any variable indicating the status of any other process. All is captured through the array `my-id` which is a local array for each process.

§12. Contention detection

Tournament based algorithms have allowed us to reach a logarithmic complexity in terms of the number of queries one process has to make in order to enter the CS. Is it possible to have a process make $O(1)$ queries if it detects that there is no contention for the CS? The answer is yes and in § 13 we shall present an algorithm which delivers such a promise. Prior to this algorithm we first have to devise an algorithm that would detect whether there is contention for the CS or not.

Goal. We seek a procedure to be applied once by each process attempting entry to the CS. The procedure returns one of two possible values: *commit* or *abort*

- If *abort* is returned the algorithm learns that there is contention for the CS and aborts its attempt to enter the CS.
- If *commit* is returned then the process continues its entry attempt.
- Property: The procedure has to have the property that if several processes execute it then at most one of them receives *commit* as a return value.

To implement this procedure we employ two *atomic* shared registers named X and Y. The register X can be initialised arbitrarily. The register Y is initialised with a special symbol @.

```
1 contention-abort(i){
2
3     X = i
4
```

```
5       if (Y != @){ (L1)
6
7           return ABORT-1
8       }
9       else{
10          Y = # (L2)
11          if (X == i){
12
13              return COMMIT
14          }
15          else{
16              return ABORT-2
17          }
18
19      }
20
21  }
```

Listing 66: Contention detection procedure

We currently do not reset the value of Y as we are currently examining the behaviour of this procedure under the assumption that each process performs this procedure precisely once.

- Process i upon invoking this procedure first sets the register X to its identity.
- If i is not the first to invoke the procedure it tries to check it at line L1 by examining the value of Y. For if the value of Y is not @ then i knows some other process already changed the value Y at line L2. Hence it aborts its attempt. That is `ABORT-1` is returned for i if it learns that it is "too late" and that some other process is already trying to enter the CS.
- However, if the value of Y did not change from its initial value then i overwrites it with a special symbol #.
- After this i proceeds to check whether some other process invoked the procedure after it by examining the value of X. If the value of X remained equal to its identity no other process invoked the procedure up until now. If this happens process i receives the value `COMMIT`.
- However, if the value of X at this point changed from i then process i learns that some other process is contending with it for the CS and in some sense i learns that it is "early". In which case it receives `ABORT-2`.

Consider the following scenario.

1. Process i sets `X = i`.
2. Process j sets `X = j`.
3. Both i and j perform L1 one after another and none is aborted.
4. i sets `Y = @`.
5. j sets `Y = @`.
6. i falls into the `else` and gets `ABORT-2`
7. j falls into the `if` and gets `COMMIT`.

Observation 12.1. *If a process performs procedure `contention-abort` in a contention-free scenario it receives commit.*

Lemma 12.2. *If several process perform procedure `contention-abort` then at most one receives COMMIT.*

Proof. Suppose that process P received `COMMIT`. As Y is atomic and is never reset to its initial value @ we can partition the set of involved processes into two sets.

- *Late* processes: those who read Y at L1 *after* P wrote # into Y at L2, and
- *Early* processes: those who read Y at L1 *before* P wrote # into Y at L2.

All *late* processes must receive `ABORT-1`. Every *early* process as well as process P have passed L1 and thus all have set X. The atomicity of X then implies that amongst the set of early process and P there is a process who have set X last. Only this process can receive `COMMIT`. This process is P by assumption. ■

Observation 12.3. *It is possible that no process receives `COMMIT`*

Proof. Here is a scenario with two processes.

1. Process 1 sets `X=1`

2. Process 1 proceeds to L2, set `Y =` # and is suspended.
3. Process 2 sets `X=2`.
4. Process 2 perform L1 and receives `ABORT-1`.
5. Process 1 performs line 11 and sees that the value of X is no longer equal to its identity and it then falls to the `else` clause and receives `ABORT-2`.

■

Observation 12.4. *If $Y =$ # then*

1. *either some process received `COMMIT`,*
2. *or several processes invoked the procedure.*

Observation 12.5. *No matter how many processes have invoked the procedure each of these accesses X and Y four times.*

§13. Lamport's fast mutex algorithm

In this section we present the so called *Lamport's fast mutex algorithm*. This algorithm carries the promise that in contention-free scenarios processes will only perform $O(1)$ queries prior to entering the CS. This algorithm employs the `contention-abort` procedure yet not in a black doc manner. The algorithm has the following shared data.

- Two atomic registers X and Y where $Y = @$ initially.
- Array `inter[n]`.

```
1  fast-enter(i){
2
3      inter[i] = T (L1)
4      X = i
5      if (Y != @){
6          inter[i] = F
7          await(Y = @)
8          GOTO line (L1)
9      }
10     else{
```

```
11          Y = i
12          if (X == i){
13              return
14          }
15          else{
16              inter[i] = F
17              for j = 1 to n do{
18                  await(inter[j] == F)
19              }
20              if (Y==i){
21                  return
22              }
23              else{
24                  GOTO line (L1)
25              }
26          }
27
28      }
29
30  }
```

Listing 67: Lamport's fast mutex algorithm: entry code

```
1  fast-exit(i){
2      Y = @
3      inter[i] = F
4  }
```

Listing 68: Lamport's fast mutex algorithm: exit code

The entry code of the fast mutex algorithm of Lamport is simply a rewrite of the `contention-abort` procedure. In this procedure we abort the attempt of a process in two cases:

`ABORT-1`: A process is "too late" in the sense that other process have already tried to enter before it. The process learns of this by examining the value of Y in the first `if`. If it notices that Y is not equal to @ then, by Observation 12.4 that means either some process got commit or there is contention. Hence in this case we see in lines 6-8 that the algorithm reroutes the process to try entry from scratch again.

`ABORT-2`: This type of abort was issued by `contention-abort` upon detecting that some other process entered after the current process started its attempt. This type of abort allowed for the case that no process gets `COMMIT`. Unlike the procedure `contention-abort` here we seek a different behaviour in this respect. Here we would like to make sure that one of the competing

process does in fact get `COMMIT`.

When process i detects that it should receive `ABORT-2` in the "base algorithm" it proceeds to do the following:

- Declares no-interest: `inter[i]= F`;
- awaits all other processes are not interested.
- If after all this it sees that $Y \neq i$ it waits until the CS is released and tries again.
- However, if upon finishing the `for` loop it detects that $Y = i$ still then it receives `COMMIT`.
  - For indeed, here i have just detected that it was the last to write to Y and since his writing to Y no other process is interested in the CS. As only processes who write to Y are in attempt to enter we allow i to enter the CS.

A process that enters the CS via the `COMMIT` in line 12 is said to enter via the *fast path*. Observe that in the fast path an algorithm only perform $O(1)$ queries.

Observation 13.1. *In a contention-free scenario a process enters via the fast path.*

Unlike all previous algorithms here a process can be inside the CS while heaving its `inter` flag set to false as we allow entry via line 21. This does not bother us as if this happen $Y \neq @$ and Y is reset only in the exit code.

Theorem 13.2. *Lamport's fast mutex algorithm is not deadlock-free.*

Proof. Let P be the process that is last to write to X and as a result all other $n-1$ process are currently at line 18 all awaiting for its flag to be set to false. However, P is fast to enter the CS leave and reacquire it so that all other processes never get to see its `inter` flag with the false value. This can happen as follows.

1. $n-1$ processes are at line 18 awaiting.
2. P enters via line 13.
3. It leaves the CS, resets Y, attempts reentry and is able to pass the first if.

4. Now let all other processes run, but these see the `inter` flag of P set to true and remain in awaiting status.

This scenario repeats itself forever. ■

While the algorithm is not deadlock-free, we are still able to ensure some type of progress of the algorithm,

Theorem 13.3. *Amongst the n processes at least one is able to enter.*

Proof. Assume for the sake of contradiction that all n processes are stuck forever in their entry code. Partition the set of n process into two sets: $\mathcal{Q}_1$ and $\mathcal{Q}_2$. The former consists of all processes stuck due to lines 3-8 while $\mathcal{Q}_2$ consists of all processes who are stuck due to the first `else` clause. The set $\mathcal{Q}_2$ is non-empty as it must contain the first process to read $Y = @$ at line 5.

- No member of $\mathcal{Q}_2$ enters the CS via line 13 (as by assumption $\mathcal{Q}_2$ is blocked).
- Hence all members of $\mathcal{Q}_2$ are stuck in the 2nd `else` clause starting at line 15.
- No process external to $\mathcal{Q}_2$ can bypass $\mathcal{Q}_2$ as all these are in $\mathcal{Q}_1$ and are infinitely looping in line 3-8.
- Let $P \in \mathcal{Q}_2$ be the last process in $\mathcal{Q}_2$ to write to Y at line 11.
- Process P is able to enter the CS. Surely it is able to clear the member of $\mathcal{Q}_2$ in the `for` loop. The reason it can also clear the members of $\mathcal{Q}_1$ is after a finite amount of time $Y \neq @$ forever and thus all member of $\mathcal{Q}_1$ are stuck at line 7 awaiting Y to be reset. Hence, by line 6 these have all set their `inter` flags to false allowing P to clear those in the `for` loop.

■

Theorem 13.4. *Lamport's fast mutex algorithm provides mutual exclusion.*

Proof. Suppose process P is currently in the CS. Then $Y \neq @$ and process P entered the CS either by returning through line 13 or line 21 of the procedure `fast-enter`. Let t_0 denote the time P wrote to Y at line 11 during its current entry attempt to the CS. Processes who at their entry attempt read Y at

line 5 after time t_0 cannot enter the CS until P leaves the CS. We thus focus on the following set of processes

$$\mathcal{Q} := \{Q : \ Q \text{ reads } Y = @ \text{ before } t_0 \text{ at line } 5 \ \}.$$

As all members of $\mathcal{Q} \cup \{P\}$ all read $Y = @$ at line 5 these all compete for the CS through the `else` clause. We consider two cases.

1. Suppose P entered the CS through line 13. Then P was the last to write to X amongst $\mathcal{Q} \cup \{P\}$. Suppose now that some process Q enters the CS and resides there together with P.

   (a) If Q entered via line 13, then

   $$Q \text{ writes } X \text{ at l.4} \ \rightarrow \ Q \text{ enters CS via l.13} \ \rightarrow \ P \text{ writes } X \text{ at l.4.}$$

   However, as

   $$Q \text{ writes } Y \ \rightarrow \ Q \text{ enters CS}$$

   it follows that

   $$Q \text{ writes } Y \ \rightarrow \ P \text{ writes } X \text{ at l.4} \ \rightarrow \ P \text{ reads } Y \text{ at l.5;}$$

   hence by the time P reads Y at line 5 it fails to pass the `if` as it sees that $Y \neq @$. This contradicts the assumption that P was able to enter the CS together with Q.

   (b) Suppose then that Q had entered via line 21. As process P enters via line 13 it never sets its `inter` flag to false at line 16. Leading to Q never being able to pass the `for` loop on account of the `inter` flag of P being true.

2. Consider the complimentary case that P entered via line 21. Assume towards contradictions hat some process $Q \in \mathcal{Q}$ entered the CS together with P. We consider two cases.

   (a) Suppose, first, that Q entered via line 13. This is case 1(b) above which we have excluded already.

   (b) Suppose, second, that Q entered via line 21. Both P and Q then enter via line 21 and these both do so as these detect that these are last to change Y at line 11. As Y is atomic there could only be one of those who is last to change Y.

■