17.3 STABILIZING MUTUAL EXCLUSION

Dijkstra's work [D74] initiated the field of self-stabilization in distributed systems. He first demonstrated its feasibility by proposing stabilizing solutions to the problem of mutual exclusion on three different types of networks. In this section, we present the solutions to two of these versions.

17.3.1 Mutual Exclusion on a Unidirectional Ring

Consider a *unidirectional ring* of n processes 0, 1, 2, …, $n - 1$ (Figure 17.2). Each process can remain in one of the k possible states 0, 1, 2, …, $k - 1$. We consider the shared-memory model of computation: A process i, in addition to reading its own state $s[i]$, can read the state $s[i - 1 \bmod n]$ of its *predecessor* process $i - 1 \bmod n$. Depending on whether a predefined guard (which is a Boolean function of these two states) is true, process i may choose to modify its own state.

We will call a process with an enabled guard a *privileged process* or a process *holding a token*. This is because a privileged process is one that can take an action, just as in a token ring network, a process holding the token is eligible to transmit or receive data. A legal configuration of the ring is characterized by the following two conditions:

Safety: The number of processes with an enabled guard is exactly one.

Liveness: During an infinite behavior, the guard of each process is enabled infinitely often.

A privileged process executes its critical section. A process that has an enabled guard but does not want to execute its critical section simply executes an action to pass the privilege to its neighbor. Transient failures may transform the system to an illegal configuration. The problem is to design a protocol, so that starting from an arbitrary initial state, the system eventually converges to a legal configuration and remains in that configuration thereafter.

Dijkstra's solution assumed process 0 to be a *distinguished process* that behaves differently from the remaining processes in the ring. All the other processes run identical programs. There is a central scheduler for the entire system. In addition, the condition $k > n$ holds. The programs are as follows:

```
Program ring;
{program for process 0}
do s[0]=s[n−1]→s[0]:=s[0]+1 mod k   od
{program for process i≠0}
do s[i]≠s[i−1]→s[i]:=s[i−1] od
```

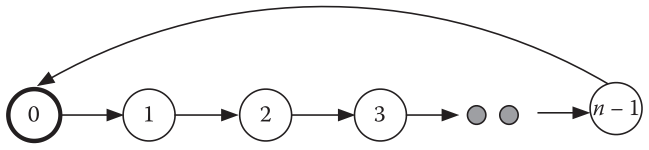

FIGURE 17.2 A unidirectional ring of n processes.

Before studying the proof of correctness of the aforementioned algorithm, we *strongly* urge the reader to study a few sample runs of the aforementioned protocol and observe the convergence and closure properties. A configuration in which $\forall i, j : s[i] = s[j]$ is an example of a legal configuration.

Proof of correctness

Lemma 17.1

[No deadlock] In any configuration, at least one process must have an enabled guard.

Proof: If every process 1 through $n - 1$ has a disabled guard, then $\forall i > 0, s[i] = s[i - 1]$. But this implies that $s[0] = s[n - 1]$, so process 0 must have an enabled guard. ■

Lemma 17.2

[Closure] The legal configuration satisfies the closure property.

Proof: If only process 0 has an enabled guard, then $\forall i, j : 0 \leq i, j \leq n - 1 : s[i] = s[j]$. A move by process 0 will disable its own guard and enable the guard for process 1. If only process i $(0 < i < n - 1)$ has an enabled guard, then

- $\forall j < i : s[j] = s[i - 1]$
- $\forall k > i : s[k] = s[i]$
- $s[i] \neq s[i - 1]$

Accordingly, a move by process i will disable its own guard and enable the guard for process $(i + 1) \bmod n$. Similar arguments hold when only process $(n - 1)$ has an enabled guard. ■

As a consequence of Lemmas 17.1 and 17.2, in an infinite computation, the guard of each process will be true infinitely often.

Lemma 17.3

[Convergence] Starting from any illegal configuration, the ring eventually converges to a legal configuration.

Proof: Observe that every action by a process disables its own guard and enables *at most* one new guard in a different process—so the number of enabled guards never increases. Now, assume that the claim is false, and the number of enabled guards remains constant during an infinite suffix of a behavior. This is possible if every action that disables an existing guard enables exactly one new guard.

There are n processes with $k(k > n)$ states per process. By the pigeonhole principle, in any initial configuration, at least one element $j \in \{0, 1, 2, \ldots, k - 1\}$ *must not* be the initial state of any process. Each action by process ($i > 0$) essentially copies the state of its predecessor, so if j is not the state of any process in the initial configuration, no process can be in state j until $s[0]$ becomes equal to j. However, it is guaranteed that at some point, $s[0]$ will be equal to j, since process 0 executes actions infinitely often, and every action increments $s[0]$ (mod k). Once $s[0] = j$, eventually every process attains the state j, and the system reaches a legal configuration. ■

The property of stabilization follows from Lemmas 17.2 and 17.3.

17.3.2 Mutual Exclusion on a Bidirectional Array

The second protocol operates on an array of processes 0 through $n - 1$ (Figure 17.3). We present here a modified version of Dijkstra's protocol, taken from [G93]

In this system, $\forall i : s[i] \in \{0,1,2,3\}$ and is independent of the size of the array. The two processes 0 and $n - 1$ behave differently from the rest—they have two states each. By definition, $s[0] \in \{1,3\}$ and $s[n-1] \in \{0,2\}$. Let $N(i)$ designate the set of neighbors of process i. The program is as follows:

```
program four-state;
{program for process i, i = 0 or n - 1}
do ∃j ∈N(i):s[j]=s[i]+1mod4→s[i]:=s[i]+2mod4 od
{program for process i, 0 < i < n - 1}
do ∃j ∈N(i):s[j]=s[i]+1mod4→s[i]:=s[i]+1mod4 od
```

Proof of correctness

The *absence of deadlock* can be trivially demonstrated using arguments similar to those used in Lemma 17.1. We focus on convergence only.

For a process i, call the processes $i + 1$ and $i - 1$ to be the *right* and the *left* neighbors, respectively. Define two predicates $L \cdot i$ and $R \cdot i$ as follows:

$$L \cdot i \equiv s[i-1] := s[i] + 1 \bmod 4$$

$$R \cdot i \equiv s[i+1] := s[i] + 1 \bmod 4$$

We will represent $L \cdot i$ by drawing a $\rightarrow$ from process $i - 1$ to process i and $R \cdot i$ by drawing a $\leftarrow$ from process $i + 1$ to process i. Thus, any process with an enabled guard has at least one arrow pointing toward it.

For a process i ($0 < i < n - 1$), the possible moves fall exactly into one of the seven cases in part (a) of Table 17.1. Each entry in the columns *precondition* and *postcondition* represents

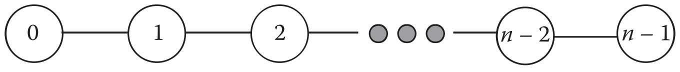

FIGURE 17.3 An array of n processes.