It was impossible to get a conversation going; everybody was talking too much.

—YOGI BERRA

A.1 MUTUAL EXCLUSION: SOFTWARE APPROACHES

Software approaches can be implemented for concurrent processes that execute on a single processor or a multiprocessor machine with shared main memory. These approaches usually assume elementary mutual exclusion at the memory access level ([LAMP91], but see Problem A.3). That is, simultaneous accesses (reading and/or writing) to the same location in main memory are serialized by some sort of memory arbiter, although the order of access granting is not specified ahead of time. Beyond this, no support in the hardware, operating system, or programming language is assumed.

Dekker's Algorithm

Dijkstra [DIJK65] reported an algorithm for mutual exclusion for two processes, designed by the Dutch mathematician Dekker. Following Dijkstra, we develop the solution in stages. This approach has the advantage of illustrating many of the common bugs encountered in developing concurrent programs.

FIRST ATTEMPT As mentioned earlier, any attempt at mutual exclusion must rely on some fundamental exclusion mechanism in the hardware. The most common of these is the constraint that only one access to a memory location can be made at a time. Using this constraint, we reserve a global memory location labeled `turn`. A process (P0 or P1) wishing to execute its critical section first examines the contents of `turn`. If the value of `turn` is equal to the number of the process, then the process may proceed to its critical section. Otherwise, it is forced to wait. Our waiting process repeatedly reads the value of `turn` until it is allowed to enter its critical section. This procedure is known as **busy waiting**, or **spin waiting**, because the thwarted process can do nothing productive until it gets permission to enter its critical section. Instead, it must linger and periodically check the variable; thus it consumes processor time (busy) while waiting for its chance.

After a process has gained access to its critical section and after it has completed that section, it must update the value of `turn` to that of the other process.

In formal terms, there is a shared global variable:

```
int  turn = 0;
```

Figure A.1a shows the program for the two processes. This solution guarantees the mutual exclusion property but has two drawbacks. First, processes must strictly alternate in their use of their critical section; therefore, the pace of execution is dictated by the slower of the two processes. If P0 uses its critical section only once per hour but P1 would like to use its critical section at a rate of 1,000 times per hour, P1 is forced to adopt the pace of P0. A much more serious problem is that if one process fails, the other process is permanently blocked. This is true whether a process fails in its critical section or outside of it.

```
   /* PROCESS 0 /*                    /* PROCESS 1 */
.                                  .
.                                  .
while (turn != 0)                  while (turn != 1)
   /* do nothing */ ;                 /* do nothing */;
/* critical section*/;              /* critical section*/;
turn = 1;                          turn = 0;
.                                  .
```

(a) First attempt

```
   /* PROCESS 0 */                    /* PROCESS 1 */
.                                  .
.                                  .
while (flag[1])                    while (flag[0])
   /* do nothing */;                  /* do nothing */;
flag[0] = true;                    flag[1] = true;
/*critical section*/;              /* critical section*/;
flag[0] = false;                   flag[1] = false;
.                                  .
```

(b) Second attempt

```
   /* PROCESS 0 */                    /* PROCESS 1 */
.                                  .
.                                  .
flag[0] = true;                    flag[1] = true;
while (flag[1])                    while (flag[0])
   /* do nothing */;                  /* do nothing */;
/* critical section*/;              /* critical section*/;
flag[0] = false;                   flag[1] = false;
.                                  .
```

(c) Third attempt

```
   /* PROCESS 0 */                    /* PROCESS 1 */
.                                  .
.                                  .
flag[0] = true;                    flag[1] = true;
while (flag[1]) {                  while (flag[0]) {
   flag[0] = false;                   flag[1] = false;
   /*delay */;                        /*delay */;
   flag[0] = true;                    flag[1] = true;
}                                  }
/*critical section*/;              /* critical section*/;
flag[0] = false;                   flag[1] = false;
.
```

(d) Fourth attempt

Figure A.1 Mutual Exclusion Attempts

The foregoing construction is that of a **coroutine**. Coroutines are designed to be able to pass execution control back and forth between themselves (see Problem 5.2). While this is a useful structuring technique for a single process, it is inadequate to support concurrent processing.

SECOND ATTEMPT The flaw in the first attempt is that it stores the name of the process that may enter its critical section, when in fact we need state information about both processes. In effect, each process should have its own key to the critical section so that if one fails, the other can still access its critical section. To meet this requirement a Boolean vector `flag` is defined, with `flag[0]` corresponding to P0 and `flag[1]` corresponding to P1. Each process may examine the other's flag but may not alter it. When a process wishes to enter its critical section, it periodically checks the other's flag until that flag has the value `false`, indicating that the other process is not in its critical section. The checking process immediately sets its own flag to `true` and proceeds to its critical section. When it leaves its critical section, it sets its flag to `false`.

The shared global variable[1] now is

```
enum        boolean (false = 0; true = 1);
boolean     flag[2] = {0, 0}
```

Figure A.1b shows the algorithm. If one process fails outside the critical section, including the flag-setting code, then the other process is not blocked. In fact, the other process can enter its critical section as often as it likes, because the flag of the other process is always `false`. However, if a process fails inside its critical section or after setting its flag to `true` just before entering its critical section, then the other process is permanently blocked.

This solution is, if anything, worse than the first attempt because it does not even guarantee mutual exclusion. Consider the following sequence:

P0 executes the **while** statement and finds `flag[1]` set to `false`

P1 executes the **while** statement and finds `flag[0]` set to `false`

P0 sets `flag[0]` to `true` and enters its critical section

P1 sets `flag[1]` to `true` and enters its critical section

Because both processes are now in their critical sections, the program is incorrect. The problem is that the proposed solution is not independent of relative process execution speeds.

THIRD ATTEMPT Because a process can change its state after the other process has checked it but before the other process can enter its critical section, the second attempt failed. Perhaps we can fix this problem with a simple interchange of two statements, as shown in Figure A.1c.

As before, if one process fails inside its critical section, including the flag-setting code controlling the critical section, then the other process is blocked, and if a process fails outside its critical section, then the other process is not blocked.

---

[1]The **enum** declaration is used here to declare a data type (`boolean`) and to assign its values.

Next, let us check that mutual exclusion is guaranteed, using the point of view of process P0. Once P0 has set `flag[0]` to `true`, P1 cannot enter its critical section until after P0 has entered and left its critical section. It could be that P1 is already in its critical section when P0 sets its flag. In that case, P0 will be blocked by the **while** statement until P1 has left its critical section. The same reasoning applies from the point of view of P1.

This guarantees mutual exclusion but creates yet another problem. If both processes set their flags to `true` before either has executed the **while** statement, then each will think that the other has entered its critical section, causing deadlock.

Fourth Attempt In the third attempt, a process sets its state without knowing the state of the other process. Deadlock occurs because each process can insist on its right to enter its critical section; there is no opportunity to back off from this position. We can try to fix this in a way that makes each process more deferential: Each process sets its flag to indicate its desire to enter its critical section but is prepared to reset the flag to defer to the other process, as shown in Figure A.1d.

This is close to a correct solution but is still flawed. Mutual exclusion is still guaranteed, using similar reasoning to that followed in the discussion of the third attempt. However, consider the following sequence of events:

P0 sets `flag[0]` to `true`.

P1 sets `flag[1]` to `true`.

P0 checks `flag[1]`.

P1 checks `flag[0]`.

P0 sets `flag[0]` to `false`.

P1 sets `flag[1]` to `false`.

P0 sets `flag[0]` to `true`.

P1 sets `flag[1]` to `true`.

This sequence could be extended indefinitely, and neither process could enter its critical section. Strictly speaking, this is not deadlock, because any alteration in the relative speed of the two processes will break this cycle and allow one to enter the critical section. This condition is referred to as **livelock**. Recall that deadlock occurs when a set of processes wishes to enter their critical sections but no process can succeed. With livelock, there are possible sequences of executions that succeed, but it is also possible to describe one or more execution sequences in which no process ever enters its critical section.

Although the scenario just described is not likely to be sustained for very long, it is nevertheless a possible scenario. Thus we reject the fourth attempt.

A Correct Solution We need to be able to observe the state of both processes, which is provided by the array variable `flag`. But, as the fourth attempt shows, this is not enough. We must impose an order on the activities of the two processes to avoid the problem of "mutual courtesy" that we have just observed. The variable `turn` from the first attempt can be used for this purpose; in this case

the variable indicates which process has the right to insist on entering its critical region.

We can describe this solution, referred to as Dekker's algorithm, as follows. When P0 wants to enter its critical section, it sets its flag to `true`. It then checks the flag of P1. If that is `false`, P0 may immediately enter its critical section. Otherwise, P0 consults `turn`. If it finds that `turn` = 0, then it knows that it is its turn to insist and periodically checks P1's flag. P1 will at some point note that it is its turn to defer and set its to flag `false`, allowing P0 to proceed. After P0 has used its critical section, it sets its flag to `false` to free the critical section and sets `turn` to 1 to transfer the right to insist to P1.

```
boolean flag [2];
int turn;
void P0()
{
     while (true) {
          flag [0] = true;
          while (flag [1]) {
               if (turn == 1) {
                    flag [0] = false;
                    while (turn == 1) /* do nothing */;
                    flag [0] = true;
               }
          }
          /* critical section */;
          turn = 1;
          flag [0] = false;
          /* remainder */;
     }
}
void P1( )
{
     while (true) {
          flag [1] = true;
          while (flag [0]) {
               if (turn == 0) {
                    flag [1] = false;
                    while (turn == 0) /* do nothing */;
                    flag [1] = true;
               }
          }
          /* critical section */;
          turn = 0;
          flag [1] = false;
          /* remainder */;
     }
}
void main ()
{
     flag [0] = false;
     flag [1] = false;
     turn = 1;
     parbegin (P0, P1);
}
```

Figure A.2 **Dekker's Algorithm**

Figure A.2 provides a specification of Dekker's algorithm. The construct **parbegin** (P1, P2, . . . , P*n*) means the following: suspend the execution of the main program; initiate concurrent execution of procedures P1, P2, . . . , P*n*; when all of P1, P2, . . . , P*n* have terminated, resume the main program. A verification of Dekker's algorithm is left as an exercise (see Problem A.1).

Peterson's Algorithm

Dekker's algorithm solves the mutual exclusion problem but with a rather complex program that is difficult to follow and whose correctness is tricky to prove. Peterson [PETE81] has provided a simple, elegant solution. As before, the global array variable `flag` indicates the position of each process with respect to mutual exclusion, and the global variable `turn` resolves simultaneity conflicts. The algorithm is presented in Figure A.3.

That mutual exclusion is preserved is easily shown. Consider process P0. Once it has set `flag[0]` to `true`, P1 cannot enter its critical section. If P1 already is in its critical section, then `flag[1] = true` and P0 is blocked from entering its critical section. On the other hand, mutual blocking is prevented. Suppose that P0 is blocked in its **while** loop. This means that `flag[1]` is `true` and `turn` = 1. P0 can

```
boolean flag [2];
int turn;
void P0()
{
     while (true) {
          flag [0] = true;
          turn = 1;
          while (flag [1] && turn == 1) /* do nothing */;
          /* critical section */;
          flag [0] = false;
          /* remainder */;
     }
}
void P1()
{
     while (true) {
          flag [1] = true;
          turn = 0;
          while (flag [0] && turn == 0) /* do nothing */;
          /* critical section */;
          flag [1] = false;
          /* remainder */
     }
}
void main()
{
     flag [0] = false;
     flag [1] = false;
     parbegin (P0, P1);
}
```

Figure A.3 Peterson's Algorithm for Two Processes

enter its critical section when either `flag[1]` becomes `false` or `turn` becomes 0. Now consider three exhaustive cases:

1. P1 has no interest in its critical section. This case is impossible, because it implies `flag[1] = false`.
2. P1 is waiting for its critical section. This case is also impossible, because if `turn` = 1, P1 is able to enter its critical section.
3. P1 is using its critical section repeatedly and therefore monopolizing access to it. This cannot happen, because P1 is obliged to give P0 an opportunity by setting `turn` to 0 before each attempt to enter its critical section.

Thus we have a simple solution to the mutual exclusion problem for two processes. Furthermore, Peterson's algorithm is easily generalized to the case of *n* processes [HOFR90].