19. Mutual Exclusion

So far our consideration of resource management has made a fundamental assumption. That is: the processes competing for a shared resource (e.g., CPU, disk, or network) do not intend to cooperate, or more in general to change the state of the resource they share. They merely need to use that resource. In computing systems, it is tempting to think of resources as devices. But in general, a resource could also be a data structure or a piece of software.

19.1 Data Races

Consider two of processes that are each incrementing by 1 a shared variable that has value $C = 15$ at some point in time. How is it possible that in some cases C' after the update is $C' = C + 1 + 1 = 16$?

To understand that, consider what could happen when multiple concurrent processes attempt to "access" that variable by adding one to it. To do so, a process will need to (1) read the memory location in which that integer variable is stored, (2) add one to that value, and (3) then store that value back in memory. This is not a problem if only one process has access to that variable. Indeed, if we only have one such process, then clearly "adding one" to the variable will always yield the desirable result of incrementing the value of the counter by one. Now consider what could happen if we allow multiple (say two) processes to access the shared counter data structure in an unrestricted manner.

For the sake of discussion, let's consider the initial value of the variable $C = 15$. We note that, due to the fact the two processes are executing concurrently on the same processor, the execution of their individual instructions could be interleaved. That is because we cannot assume anything about the choices taken but the underlying processor scheduling strategy. Consider the interleaved execution of the set of three identical instructions for process P1 and for process P2 as depicted in Table 19.1.

Step	Process P1	Process P2
1	Read C into local variable X	
2		Read C into local variable X
3		$X = X + 1$
4		Store in C the value of X
5	$X = X + 1$	
6	Store in C the value of X	

Table 19.1: Example execution of two concurrent processes accessing a shared variable C resulting in an incorrect result.

Note that X is a local variable and that P1 and P2 have hold their own local value of X. What would be the value stored in the shared variable C? Clearly, the intention was for P1 to increment by 1 and for P2 to increment by 1, which should yield a result of 17. The above interleaved execution of P1 and P2 does not yield that answer, but rather it yields an answer of 16.

This example is meant to motivate the fact that uncontrolled access to a data structure could compromise the "correctness" of that data structure. Clearly, one need not worry only about integers and counters. For instance, concurrent processes may want to share more elaborate data structures, such as buffers, queues, linked lists, trees, graphs, files, database records, etc. The uncontrolled concurrent access to such data structures could result in unintended consequences. For example, adding an element to a linked list, only to discover that it was never added, or having two processes de-queue an element twice from the same queue. Or worse yet, having the consistency of the linked list corrupted with pointer-to-next that point to invalid memory (dangling pointers).

What discussed so far exemplifies the existence of a *data race*—also called a *race condition*. A data race (or race condition) is defined as an uncontrolled access by two or more concurrent processes to a shared variable (or resource, or data structure, etc.) where the final state of the data depends on the specific interleaving of instructions executed by the competing processes. Clearly this is problematic because the result of the execution of concurrent processes does not depend on the programmed semantics, but rather on the relative progress that processes make throughout their execution.

19.2 Critical Sections of Code

When a process executes code that manipulates shared data (or in general changes the state of a resource), we say that the process is in a critical section for that shared data. In the motivating example shown in Table 19.1, the portion of the code in which the shared data structure (the variable C) is accessed constitutes the critical section. The problems we encountered in that example were due to the fact that we did not "coordinate" or "synchronize" the execution of the code in the critical section. Specifically, it was the interleaving of instructions from the two critical sections that resulted in the undesirable outcomes.

How could we have avoided the incorrect interleaving of instructions from the two processes?

Had we disallowed any interleaving of instructions between the two independent executions of the critical sections, the problem would have not arisen in the first place. Specifically, we could guarantee correctness if we made access to the critical section code *mutually exclusive* in the sense that either one process or the other is allowed to be in the critical section—but never both. Indeed, guaranteeing that a critical section is executed in a mutually-exclusive fashion amongst multiple processes is the first (and as we will see, in many ways the most fundamental) problem in synchronization.

19.3 The Mutual Exclusion Problem

Consider a critical segment of code (such as the one we used in the example of Table 19.1) which we would like to ensure is never executed by more than one process at the same time. More precisely, we want to ensure that if any process is executing any instruction of a critical section, then no other process would be executing any instruction from that same critical section—i.e., at most one process is *inside* that critical section. How could we guarantee/enforce such a requirement? Ensuring mutual exclusion is a fundamental problem in computer science.

To answer this, we need a more concrete statement of the problem, including our assumptions about the processes and the processing environment in which these processes will execute. Also, we will need to define what properties we would like to have in an acceptable solution.

19.3.1 Assumptions about the Processing Environment

We make the following assumptions about the processes that may want to execute the critical section:

1. **Progress:** Each process executes at non-zero speed. In other words, no process is dead. This assumption is important because it allows us to ignore situations in which a process may get into the critical section but never leave it.
2. **Unknown rate of progress:** Processes execute at some arbitrary speeds. Moreover, we know nothing about the relative speeds of the various processes, and we cannot assume anything.
3. **Concurrency exists:** The concurrency of process execution may be the result of instruction-level interleaving due to multi-programming on a single processor, or due to multi-processing where multiple processors execute different processes simultaneously.
4. **Unknown Interleaving:** When processes are interleaved on a single processor, we cannot assume that any particular dispatching technique (or scheduler) is in use. In other words, processes may be interleaved in arbitrary ways.
5. **Serialized Memory:** We assume that any multi-processor system allows for shared memory access, but that the shared memory architecture prevents simultaneous access to the same memory location.

All these assumptions are quite normal in computing system as we want to decouple the correctness of parallel code from the specific execution environment (e.g. which specific OS, or hardware platform) in which it will end up executing.

19.3.2 Critical Section Handling Protocol

We assume that the critical section is some arbitrary piece of code that a number of processes may want to execute. This could range from a few lines of code to a long segment of code with function calls, etc. Since the code is arbitrary, we cannot assume that we can change it. Thus, the only thing we can do is to specify a *protocol* for the processes to follow when accessing that piece of code. Specifically, we would require a process to go through an *entry protocol* before being allowed into the critical section, and an *exit protocol* after completing the critical section.

Generally speaking, a process may wish to use the critical section many times in its lifetime. For example, in reference to the example given in Table 19.1, P1 (or P2) may need to increment the counter repeatedly—e.g., as part of an iteration. Thus, we could represent any process that may be interested in accessing the critical section as going through the above three sections or else going through some other code, which we will call a *remainder section*.

The solutions we will seek cannot change the critical section, but could specify specific steps to be followed in either the entry section, exit section, or both. Thus any process interested in accessing the critical section must go through the following ordered steps:

1. Some *remainder section*;
2. An *entry section*;
3. The *critical section*;
4. An *exit section*;
5. Some *remainder section*.

We can then produce the pseudo-code of the generic structure of a process, say P, that wants to access a critical section. This is provided in Listing 19.1.

```
Process P:
   repeat:
      /* remainder section */
      entry section;
      /* critical section */
      exit section;
      /* remainder section */
   forever
```

Listing 19.1: General structure of a process attempting to access a critical section. The entry and exit sections are what a solution to the mutual exclusion problem must specify.

19.3.3 Desirable Properties in a Solution for Mutual Exclusion

We require that any solution of the mutual exclusion problem satisfy the following four criteria:

1. **Mutual Exclusion:** By definition of the problem, at any time, at most one process can be in the critical section of code for which we need to guarantee mutual exclusion.
2. **Isolation (a.k.a. jurisdiction):** Processes not in the critical section nor trying to get into the critical section should not participate in the protocol used to guarantee mutual exclusion and

should not be affected by it. Another way to say this is that: the only processes that should be involved in the mutual exclusion protocol are those competing for access to the critical section. Other processes should be *isolated* from this protocol. This property could be seen as specifying whether or not a process has jurisdiction over the decision of who gets into the critical section and whether a process should be impacted by such a decision.

3. **Progress:** The decision of which process may access the critical section cannot be postponed indefinitely. This property ensures that the solution to the mutual exclusion problem is not simply to disallow all processes from entering. Said differently, this property ensures that processes competing for entry into the critical section will make progress. This property also implies that any solution to the problem must not allow for a deadlock or a livelock.
4. **Bounded Waiting:** After a process has made a request to enter its critical section, there is a bound on the number of times that other processes are allowed to enter their critical sections. This property ensures fairness by bounding the number of times a process may be "bypassed" by other processes bidding for the critical section. This property guarantees that processes will not suffer from starvation—which is defined as unbounded waiting.

19.4 Solution Attempts to Achieve Mutual Exclusion

We are now ready to consider various solutions to the mutual exclusion problem. Let us first consider a very simple solution where processes can simply disable interrupts and hence scheduling altogether.

19.4.1 Attempt 0: Disable/Enable Interrupts Around Critical Sections

Consider the following straightforward solution. In the entry section, a process would disable interrupts (and hence scheduling), and in the exit section, the process would re-enable interrupts. The idea is that by disabling interrupts, it should not be possible for the execution of the critical section to be interrupted; hence mutual exclusion would be guaranteed. Does this work?

Recall that in our presentation of the processing environment assumptions, we have stated that concurrency among processes may be due to multi-processing. This means that, given two processes P1 and P2 running on two different processors, if P1 disables interrupts on the first processor and proceeds to the critical section, and similarly process P2 disables interrupts on the second processor and proceeds to the critical section, we can see that the mutual exclusion property is violated. Even disabling interrupts on all the processors would not work because P1 and P2 might be already running in parallel. Hence, even without any interrupt being handled by the platform, P1 and P2 might execute the critical section in parallel.

We thus conclude that disabling hardware will not work in multi-processing environments. But, does it work for uni-processor environments? First, we observe that a solution relying on enabling/disabling interrupts cannot guarantee fairness. Indeed, one cannot bound the delay that a process wishing to access the critical section must endure. This is a violation of our bounded waiting requirement.

Another point to observe is that the hardware solution for uni-processors also violates the

isolation property. In particular, by disabling interrupts, a process is in effect impacting not only processes attempting to get into the critical section, but also those who are in their remainder section (i.e., not interested in competing for the critical section). Indeed a high priority task released while interrupts are disabled might not be able to run even if it has no business with the critical section under analysis.

Finally, from a practical point of view, disabling interrupts is a fairly *privileged* operation that we may not want to allow general processes the freedom to perform. It is an operation that we may want to keep for processes we can trust, such as those that are part of the Operating System, as opposed to an application.

To summarize, the hardware solution for mutual exclusion is not possible for multi-processing environment; it is also not adequate, and thus not acceptable for uni-processor environments as well.

19.4.2 Attempt 1: Taking Strict Turns

If playing with interrupts and directly trying to influence the scheduler does not work what can we do purely in software? In other words, what entry/exit protocols that are comprised of *regular instructions* could we propose to solve this problem? To make our problem even simpler, we will initially restrict our consideration to the problem of ensuring mutual exclusion for exactly two processes. What can we do now? In what follows, we will go through a number of attempts to solve this problem, eventually coming up with the "correct" solution. More importantly, these attempts will show us various subtleties related to solving this problem.

Let us talk about a first attempt. Consider the following approach to ensuring mutual exclusion between two processes Pi and Pj such that $i \neq j$. Let the two processes access the critical section in strict order, i.e., Pi, Pj, Pi, Pj, ... Thus the generic process Pi can only access the critical section if it is its turn to do so—otherwise it must wait. One can implement this approach simply by introducing a shared variable, say `turn`, and by checking in the entry section for process Pi that `(turn == i)`. If it is not, Pi will simply wait until that variable `turn` changes value to be equal to i. Upon exit from the critical section, a process Pi will simply set the value of the variable `(turn = j)`—i.e., make it j if it was i and i if it was j).

```
1  /* Global, shared variable */
2  var turn = i;
3
4  Process Pi:
5     repeat:
6        /* remainder section */
7        while(turn!=i);
8        /* critical section */
9        turn = j;
10       /* remainder section */
11    forever
```

Listing 19.2: Taking strict turns as a solution to the two-process mutual exclusion problem.

Listing 19.2 provides the code of the generic process Pi that implements the logic described above. The question is: is our *taking strict turns* solution a good one for the mutual exclusion

problem?

First, we observe that mutual exclusion is indeed satisfied. There could only be one process in the critical section—namely process P*i*, where `(turn==i)`. If the other process is interested in entering the critical section, it will get stuck in the busy waiting loop at Line 7.

What about the other correctness criteria that we require of the solution? Consider the progress correctness criterion. Does the *taking strict turns* approach satisfy that requirement? The answer is no. In particular, consider the case when P*i* needs to use the critical section when `turn==j`. Clearly, it can't. Now assume that P*j* is not interested in the critical section—in other words, P*j* is executing its (potentially very long) remainder section. We now have a situation in which the critical section is not in use, yet a process that needs it is unable to use it. In other words, the property of isolation is violated. As a side effect, the lack of progress on P*i* could be for any arbitrary period of time. Indeed, there is no telling how long it will be before P*j* decides to use the critical section. Only then P*j* will eventually, upon exit, switch the turn variable to *i*, allowing P*i* to enter the critical section.

Thus, the *taking strict turns* approach is not an acceptable solution to our problem. Let's try something else.

19.4.3 Attempt 2: Signal Use of Critical Section

Another possible approach to ensuring mutual exclusion is for whichever process happens to be in the critical section to indicate that it is there by setting a flag. This solution is akin to having a *lock* that a process must find open in order to get into the critical section, and which it should close upon entry (and open upon exit). Let us then introduce a shared variable called `locked` that is initialized to `false` (i.e., the lock is *open*) indicating that the critical section is initially available. This approach is illustrated in Listing 19.3.

```
1  /* Global, shared variable */
2  var locked = false;
3
4  Process Pi:
5     repeat:
6        /* remainder section */
7        while (locked == true);
8        locked = true;
9        /* critical section */
10       locked = false;
11       /* remainder section */
12    forever
```

Listing 19.3: Signaling use of critical section by setting a common lock variable.

So, is the solution shown in Listing 19.3 acceptable? The answer is no. It does not even guarantee mutual exclusion!

To understand this, consider the interleaved execution shown in Table 19.2. In this code execution instance, P*i* starts by fetching the value of the shared `locked` variable and checking to see if it were `true` or `false`. This variable is initially set to `false` indicating that the critical section is

Step	Process P*i*	Process P *j*
1	`while (locked == true);`	
2		`while (locked == true);`
3		`locked = true;`
4		**[P *j* in critical section!]**
5	`locked = true`	
6	**[P*i* in critical section!]**	

Table 19.2: Violation of mutual exclusion due to a race condition on a shared lock variable. Example of interleaved execution of two processes leading to the violation.

not in use. Thus, P*i* will break out of its busy waiting loop and complete the execution of Line 7 in Listing 19.3. Now assume that right after this point, a context switching happens and P *j* now starts executing. P *j* (following the same procedure) will be able to get into the critical section. At some point in the future, P*i* will resume its execution and it will also end up into the critical section!

One may think that the solution to the above race problem is due to the fact that we shared the use of the `locked` variable. Thus, a possible fix would be to use one such *flag* per process. We would require that whichever process happens to be in the critical section indicates that it is there by setting (write only) its own flag—a flag that the other process would check (read only) before going into the critical section. Thus, when a process P*i* needs to get into the critical section, all it has to do is to check if the flag for process P *j* is set. If it is, P*i* must wait in its entry section. If not, then P*i* sets its own flag and proceeds into the critical section. This strategy is sketched in the code given in Listing 19.4.

```
1  /* Global, shared variables */
2  var flag[i] = false;
3  var flag[j] = false;
4
5  Process Pi:
6     repeat:
7        /* remainder section */
8        while (flag[j] == true);
9        flag[i] = true;
10       /* critical section */
11       flag[i] = false;
12       /* remainder section */
13    forever
```

Listing 19.4: Signaling use of critical section by setting own flag.

Can we conclude that Listing 19.4 represents a solution that is any better than Listing 19.3? The answer is no. Indeed, one can see that the two processes could both end up in the critical section if they each read each other's flag before they have a chance to change their own flag, as shown in the trace in Table 19.3.

Indeed the problem we are encountering with both solutions sketched in Listing 19.4 and Listing 19.3 is the fact that we are not regulating access to the shared variables `locked` and `flag[]`. However, to do so, we need to solve the mutual exclusion problem. In other words, we are

Step	Process P*i*	Process P*j*
1	`while (flag[j] == true);`	
2		`while (flag[i] == true);`
3	`flag[i] = true`	
4		`flag[j] = true;`
5		**[P*j* in critical section!]**
6	**[P*i* in critical section!]**	

Table 19.3: Violation of mutual exclusion due to race conditions on the flag variables. Example of interleaved execution of two processes leading to the violation.

solving the mutual exclusion problem using an approach that requires mutual exclusion—a bit of a Catch-22 situation.

19.4.4 Attempt 3: Signal Intention to Use Critical Section

One observation we can make from our previous attempt is that the reason for violating the mutual exclusion was the fact that a process checks the flag of the other process before it sets its own flag, allowing for the race conditions shown in Table 19.2 and Table 19.3 to materialize. Could setting one's flag before checking the other process' flag solve our problem? Notice that by setting one's flag early the process is signaling not just the use of the critical section, but also the intention to do so. Listing 19.5 shows the resulting solution. Does it work, though?

```
/* Global, shared variables */
var flag[i] = false;
var flag[j] = false;

Process Pi:
   repeat:
      /* remainder section */
      flag[i] = true;
      while (flag[j] == true);
      /* critical section */
      flag[i] = false;
      /* remainder section */
   forever
```

Listing 19.5: Signaling intention to use critical section by setting own flag.

As far as mutual exclusion is concerned, the solution given in Listing 19.5 does indeed guarantee that no more than one process will ever be in the critical section. One can prove this by contradiction. Namely, assume that both P*i* and P*j* ended up in the critical section. One can see that this could only have happened if (say) process P*i* had observed P*j*'s flag being set `false`. This in turn guarantees that P*i* could not have seen P*j*'s flag being anything but `true` which would have prevented it from getting into the critical section. A contradiction!

What about the other criteria? By signaling intentions, it is possible for the two processes to end up preventing each other from entering the critical section. We can see this in the trace of execution

Step	Process P*i*	Process P *j*
1	`flag[i] = true`	
2		`flag[j] = true;`
3	`while (flag[j] == true);`	
4		`while (flag[i] == true);`

Table 19.4: Developing of a livelock when each process pre-asserts intention to enter the critical section. Example of interleaved execution of two processes leading to the livelock.

shown in Table 19.4.

19.4.5 Attempt 4: Intermittently, Signal Intention to Use Critical Section

The problem with our solution shown in Listing 19.5 is that once a process signals its intention, it prohibits the other process from entering the critical section—creating the livelock condition we observed in the trace shown in Table 19.4. To get out of this livelock situation, a process should not keep its flag set to `true` if it realizes that it cannot proceed into the critical section. Thus, one solution may be to do the following. If a process realizes that its attempt to get through to the critical section is not successful, then such a process should turn its flag to `false` for a certain amount of time before *trying again*.

```
1  /* Global, shared variables */
2  var flag[i] = false;
3  var flag[j] = false;
4
5  Process Pi:
6     repeat:
7        /* remainder section */
8        flag[i] = true;
9        while (flag[j] == true) {
10          flag[i] = false;
11          delay();
12          flag[i] = true;
13       }
14       /* critical section */
15       flag[i] = false;
16       /* remainder section */
17    forever
```

Listing 19.6: Intermittently signaling intention to use critical section by setting own flag.

This idea is fleshed out in the solution shown in Listing 19.6. Here, it can be noted that while the generic process P*i* has detected that the flag of process P*j* is set to `true` (Line 9), P*i* sets its own flag to `false` (Line 10). The idea is that by doing so, P*i* is trying to solve the case where both processes have remained stuck with both flags set to `true`. Next, P*i* proceeds to sleep for some time by invoking some `delay()` function (Line 11). Hoping that enough progress was made by P*j*, process P*i* then tries again to re-assert its intention to enter the critical section (Line 12).

As with the previous solution, the protocol spelled out in Listing 19.6 guarantees mutual exclu-

Step	Process Pi	Process Pj
1	`flag[i] = true`	
2		`flag[j] = true;`
3	`while (flag[j] == true);`	
4		`while (flag[i] == true);`
5	`flag[i] = false`	
6		`flag[j] = false;`
7	`delay()`	
8		`delay()`
...	...	...
$k+1$	`flag[i] = true`	
$k+2$		`flag[j] = true;`
$k+3$	`while (flag[j] == true);`	
$k+4$		`while (flag[i] == true);`
$k+5$	`flag[i] = false`	
$k+6$		`flag[j] = false;`
$k+7$	`delay()`	
$k+8$		`delay()`

Table 19.5: Developing of a livelock when each process pre-asserts intention to enter the critical section and then sleeps to solve a conflict. Example of interleaved execution of two processes leading to the livelock.

sion. But, does it solve the issue with livelock—namely, the violation of the progress requirement? The answer is not really. Notice that in the solution given in Listing 19.6, it is still possible for two processes to execute in lock steps as shown in the trace in Table 19.5.

Of course one may argue that this execution in lock steps is highly unlikely in a real system. This may be true and, indeed, it may well be the case that the livelock situation may not persist for a very long period of time, especially if the `delay()` between setting the flag to `false` and back to `true` is made random. However, even then, one cannot prove that this situation will never occur. Also, recall that our expectation that this execution in lock-steps is highly unlikely violates our assumption that nothing could be assumed about the relative speeds of the various processes in the system.

19.4.6 Attempt 5: Intermittently and Assertively, Signal Intention to Use Critical Section

Our previous attempt was almost correct, except for the possibility (albeit very unlikely) that a livelock could still occur. We note that this livelock situation results from the symmetry between the two processes. If we can break this symmetry in such a way that only one of the two processes will reset its flag, we would be all set. One possibility is to assume (say) that Pj has a higher priority than Pi. As such, if Pi finds out that it cannot get into the critical section, it resets its flag, but Pj does not. This would ensure that Pj will go through, breaking the livelock problem we encountered before. In other words, we could ensure that Pi will always back out of the contention to use the critical section in favor of Pj (in case both of them desire to use it). This solution will certainly work, but is it fair?

The answer is no. Indeed, if we always allow the tie between Pi and Pj to be broken in favor of one of them (say Pj), we could end up with a scenario in which Pi is indefinitely locked out of using the critical section—a violation of the bounded waiting correctness criterion.

19.4.7 Dekker's Algorithm: Intermittently but Fairly, Signal Intention to Use Critical Section

Let us try something else. First, note that to some extend, the first solution we tried, i.e. what we called *taking strict turns* (Section 19.4.2) did guarantee the basic alternating fairness in the *assignment* of the critical section. We can then try to use that approach as a starting point and try to combine that with the concept of the two process-owned `flag[]` variables that would allow us to enforce mutual exclusion.

In a sense, the idea of breaking the symmetry is the right idea, except that we do not want the symmetry to be broken always in favor of one process over the other. What we need is a method to break the symmetry that would alternate in giving priority between the two processes. This brings us to the solution shown in Listing 19.7, which is also known as the Dekker's algorithm for mutual exclusion.

```
/* Global, shared variables */
var flag[i] = false;
var flag[j] = false;
var turn = i;

Process Pi:
   repeat:
      /* remainder section */
      flag[i] = true;
      while (flag[j] == true) {
         if (turn == j) {
            flag[i] = false;
            while (turn == j);
            flag[i] = true;
         }
      }
      /* critical section */
      turn = j;
      flag[i] = false;
      /* remainder section */
   forever
```

Listing 19.7: Dekker's algorithm for mutual exclusion.

Dekker's algorithm uses a variable `turn` to break the tie between the two processes if both of them get entangled in the entry section while attempting to access the critical section. In particular, the `turn` variable acts as an arbiter, ensuring that one process will persist in asserting its intention to use the critical section, while requiring the other to temporarily *back up*.

As with the previous solution, Dekker's algorithm guarantees mutual exclusion. Moreover, using this algorithm, progress is ensured because the livelock condition we had before is broken fairly using the *let's take turns* strategy we studied in Section 19.4.2.

19.4.8 Peterson's Algorithm: "Intermittently but Politely, Signal Intention to Use Critical Section"

A more elegant (albeit slightly harder to comprehend) algorithm that solves the mutual exclusion problem is the one shown in Listing 19.8, known as Peterson's algorithm. As can be seen, this algorithm is quite similar to Dekker's in that a flag is used to register a process' interest in entering the critical section. The difference is that a process willingly gives up the tie breaking opportunity to the other process before checking on whether it can get to the critical section. This ensures that in case both processes end up going through the entry section concurrently, the one that goes through the `turn = j` first will prevail.

```
/* Global, shared variables */
var flag[i] = false;
var flag[j] = false;
var turn = i;

Process Pi:
   repeat:
      /* remainder section */
      flag[i] = true;
      turn = j;
      while (flag[j] == true && turn == j);
      /* critical section */
      flag[i] = false;
      /* remainder section */
   forever
```

Listing 19.8: Peterson's algorithm for mutual exclusion.