2.2. PETERSON’S ALGORITHM

```java
class Attempt2 implements Lock {
    boolean wantCS[] = {false, false};
    public void requestCS(int i) {
        // entry protocol
        wantCS[i] = true;  // declare intent
        while (wantCS[1 - i]);  // busy wait
    }
    public void releaseCS(int i) {
        wantCS[i] = false;
    }
}
```

Figure 2.4: An attempt that can deadlock

Yet another attempt to fix the problem is shown in Figure 2.5. This attempt is based on evaluating the value of a variable `turn`. A process waits for its turn to enter the critical section. On exiting the critical section, it sets `turn` to `1-i`.

```java
class Attempt3 implements Lock {
    int turn = 0;
    public void requestCS(int i) {
        while (turn == 1 - i);
    }
    public void releaseCS(int i) {
        turn = 1 - i;
    }
}
```

Figure 2.5: An attempt with strict alternation

This protocol does guarantee mutual exclusion. It also guarantees that if both processes are trying to enter the critical section, then one of them will succeed. However, it suffers from another problem. In this protocol, both processes have to alternate with each other for getting the critical section. Thus, after process $P_0$ exits from the critical section it cannot enter the critical section again until process $P_1$ has entered the critical section. If process $P_1$ is not interested in the critical section, then process $P_0$ is simply stuck waiting for process $P_1$. This is not desirable.

By combining the previous two approaches, however, we get Peterson’s algorithm for the mutual exclusion problem in a two-process system. In this protocol, shown in Figure 2.6, we maintain two flags, `wantCS[0]` and `wantCS[1]`, as in Attempt2, and the `turn` variable as in Attempt3. To request the critical section, process $P_i$
sets its `wantCS` flag to true at line 6 and then sets the `turn` to the other process $P_j$ at line 7. After that, it waits at line 8 so long as the following condition is true:

$$(\text{wantCS}[j] \&\& (\text{turn} == j))$$

Thus a process enters the critical section only if either it is its turn to do so or if the other process is not interested in the critical section.

To release the critical section, $P_i$ simply resets the flag `wantCS[i]` at line 11. This allows $P_j$ to enter the critical section by making the condition for its `while` loop false.

```java
1 class PetersonAlgorithm implements Lock {
2     boolean wantCS[] = {false, false};
3     int turn = 1;
4     public void requestCS(int i) {
5         int j = 1 - i;
6         wantCS[i] = true;
7         turn = j;
8         while (wantCS[j] \&\& (turn == j)) ;
9     }
10    public void releaseCS(int i) {
11        wantCS[i] = false;
12    }
13 }
```

Figure 2.6: Peterson’s algorithm for mutual exclusion

We show that Peterson’s algorithm satisfies the following desirable properties:

1. **Mutual exclusion**: Two processes cannot be in the critical section at the same time.

2. **Progress**: If one or more processes are trying to enter the critical section and there is no process inside the critical section, then at least one of the processes succeeds in entering the critical section.

3. **Starvation-freedom**: If a process is trying to enter the critical section, then it eventually succeeds in doing so.

We first show that mutual exclusion is satisfied by Peterson’s algorithm. For the purposes of the proof, we introduce auxiliary variables `trying[0]` and `trying[1]`. Whenever $P_0$ reaches line 8, `trying[0]` becomes true. Whenever $P_0$ reaches line 9, i.e., it has acquired permission to enter the critical section, `trying[0]` becomes false.
2.3 LAMPORT’S BAKERY ALGORITHM

Consider the predicate $H(0)$ defined as

$$H(0) \equiv wantCS[0] \land ((turn = 1) \lor ((turn = 0) \land trying[1]))$$

Assuming that there is no interference from $P_1$, it is clear that $P_0$ makes this predicate true after executing $turn = 1$ at line 7. Similarly, the predicate

$$H(1) \equiv wantCS[1] \land ((turn = 0) \lor ((turn = 1) \land trying[0]))$$

is true for $P_1$ after it executes line 7.

We now take care of interference between processes. It is sufficient to show that $P_0$ cannot falsify $H(1)$. From symmetry, it will follow that $P_1$ cannot falsify $H(0)$.

The first conjunct of $H(1)$ is not falsified by $P_0$ because it never updates the variable $wantCS[1]$. It only reads the value of $wantCS[1]$. Now, let us look at the second conjunct. Whenever $P_0$ falsifies $(turn = 0)$ by setting $turn = 1$, it makes $(turn = 1) \land trying[0]$ true. So, the only case left is falsification of $(turn = 1) \land trying[0]$. Since $wantCS[1]$ is also true, we look at falsification of $(turn = 1) \land trying[0] \land wantCS[1]$. $P_0$ can falsify this only by setting $trying[0]$ to false (i.e., by acquiring the permission to enter the critical section). But, $(turn = 1) \land wantCS[1]$ implies that the condition for the while statement at line 8 is true, so $P_0$ cannot exit the while loop.

Now, it is easy to show mutual exclusion. If $P_0$ and $P_1$ are in critical section, we get $\neg trying[0] \land H(0) \land \neg trying[1] \land H(1)$, which implies $(turn = 0) \land (turn = 1)$, a contradiction.

It is easy to see that the algorithm satisfies the progress property. If both the processes are forever checking the entry protocol in the while loop, then we get

$$wantCS[1] \land (turn = 1) \land wantsCS[0] \land (turn = 0)$$

which is clearly false because $(turn = 1) \land (turn = 0)$ is false.

The proof of freedom from starvation is left as an exercise. The reader can also verify that Peterson’s algorithm does not require strict alternation of the critical sections—a process can repeatedly use the critical section if the other process is not interested in it.

2.3 Lamport’s Bakery Algorithm

Although Peterson’s algorithm satisfies all the properties that we initially required from the protocol, it works only for two processes. Although the algorithm can be extended to $N$ processes by repeated invocation of the entry protocol, the resulting algorithm is more complex.